



One Postgres WAL slot, four destinations

Noémi Ványi (@kvch)

21/10/2026, PGConf.EU 2026, Valencia, Spain

Welcome everyone!

I am going to talk about sending WAL events using a single replication slot to four different destinations.

Earlier this year, I became the maintainer of pgstream an open source CDC tool.

I had to ramp up quickly on the codebase and dive into PostgreSQL replication internals. This talk is the story of my journey.

I started it with a simple question: could I replicate changes from a single WAL slot to four different destinations?

Contents

- 1 **Replication to PostgreSQL**
- 2 **Replication to OpenSearch**
- 3 **Replication to Kafka**
- 4 **Replication using webhooks**
- 5 **Four targets demo**

Then I will show you the demo of my experiment.

I am using pgstream as a tool to demonstrate the process and problems in replication to other systems. However, I am not going to dive into how pgstream works. If any of you are interested you can ask me at the end of the presentation or after the presentation.

Noémi Ványi (@kvch)

I have a decade of experience in open source software development.

I worked on **syslog-ng**, Elastic **beats**, was a maintainer of **searx**, the privacy respecting metasearch engine for 6 years. I also maintained **pgroll**, a zero downtime PostgreSQL schema migration CLI.

Now I am focusing on **pgstream**, a PostgreSQL CDC tool.

Blog: <https://kvch.me>



Before we jump in, let me introduce myself. I am Noemi, I have a decade of experience in open source software development. I worked on syslog-ng, Elastic Beats. I was a maintainer of searx, the privacy respecting metasearch engine for 6 years. Right now I work at Xata, where we are building a hosted PostgreSQL service. I have integrated PgBouncer into our infra, developed an SQL proxy, worked on pgroll. And now I am focusing on pgstream and moving terrabytes of data in and out of PostgreSQL.

PostgreSQL

Relational
database for
structured data
with ACID
transactions

OpenSearch

Search tool for
unstructured
data at scale

Kafka

Distributed
event store and
streaming
platform

Webhook

Automatic event
driven HTTP
request sent
from one service
to another

Pgstream supports 4 targets: PostgreSQL, OpenSearch, Kafka, Web services.

Target 1.

PostgreSQL to PostgreSQL

Replication basics

Let's familiarize ourselves with the basics of replication.

Replication basics

Write Ahead Log:

- append only file
- stores internal identifiers
- data in binary format
- a transaction is only completed when the WAL file is flushed to disk

Log Sequence Number:

- byte offset in the WAL file

PostgreSQL writes changes to its Write Ahead Log. It's an append only file which stores internal identifiers and the data in binary format. A transaction is only completed when the WAL file is flushed to disk.

Another important concept is the Log Sequence Number. It is the byte offset in the WAL file.

Replication basics

Replication:

- Propagating changes from one system to another

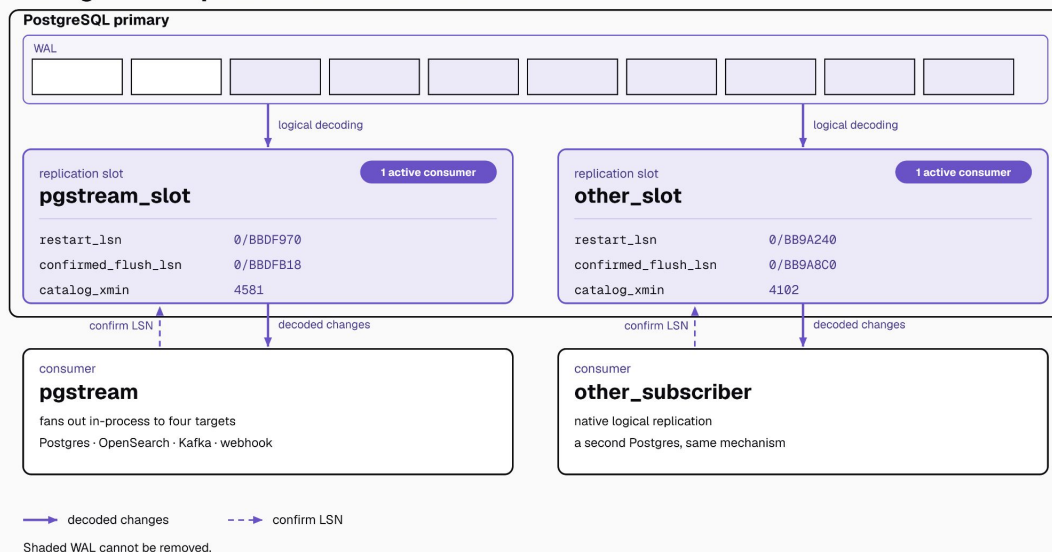
Replication slot:

- One consumer
- Ensures that WAL is not removed until the consumer has finished

We are propagating changes from one system to another. Processes create replication slots in PostgreSQL.

One slot serves one consumer. It ensures that a WAL file is only removed after the consumer finished processing it.

PostgreSQL replication slots and their consumers



In this diagram we have two replication slots. One is created by Pgstream, the other one belongs to some other process.

The important thing here is that WAL files cannot be removed until all consumers have finished reading. So the shaded WAL files can't be removed, only the white ones.

Now we know the basics. Let's discuss the not so obvious parts of replication. Like how do we replicate data schema changes using logical replication?

Demo: ALTER TABLE on source database

First I will show you what plain logical decoding gives you on ALTER TABLE statements. Then I turn on pgstream's event triggers and run the same statements.

```
— the target —  
TARGET the other database · psql on :7654
```

```
— the source —  
█
```

On the top you can see the schema of the target. On the bottom you will see the statements we run on the source.

First let's add a new column without the event triggers of pgstream. And watch what happens.

Nothing changes on the target.

Now let's enable the event triggers.

The new column is added, and it shows up on the target as well.

Replication gotchas

Missing DDL during logical replication:

- DDL changes are captured in the catalog
- `pg_logical_emit_message` on DDL change
 - Commits and rolls back with the transaction
- Target replays the DDL statement

In the demo I have shown that pgstream captures DDL changes like ALTER TABLE statements. However, PostgreSQL does not include these statements in WAL events out of the box. These are only captured in the catalog.

So pgstream installs a trigger so it can run `pg_logical_emit_message` on any DDL changes. Then the target can replay the message.

Replication gotchas

```
CREATE EVENT TRIGGER pgstream_emit_ddl_command_end
ON ddl_command_end

CREATE EVENT TRIGGER pgstream_emit_ddl_drop
ON sql_drop

pgstream.emit_ddl()
  Ddl_command := current_query();
  PERFORM pg_logical_emit_message(true, ...);
```

Pgstream creates two event triggers.

Replication gotchas

```
CREATE EVENT TRIGGER pgstream_emit_ddl_command_end  
ON ddl_command_end
```

```
CREATE EVENT TRIGGER pgstream_emit_ddl_drop  
ON sql_drop
```

```
pgstream.emit_ddl()  
  Ddl_command := current_query();  
  PERFORM pg_logical_emit_message(true, ...);
```

First one is triggered on DDL commands

Replication gotchas

```
CREATE EVENT TRIGGER pgstream_emit_ddl_command_end  
ON ddl_command_end
```

```
CREATE EVENT TRIGGER pgstream_emit_ddl_drop  
ON sql_drop
```

```
pgstream.emit_ddl()  
  Ddl_command := current_query();  
  PERFORM pg_logical_emit_message(true, ...);
```

The second one is on DROP commands.

Replication gotchas

```
CREATE EVENT TRIGGER pgstream_emit_ddl_command_end  
ON ddl_command_end
```

```
CREATE EVENT TRIGGER pgstream_emit_ddl_drop  
ON sql_drop
```

```
pgstream.emit_ddl()  
  Ddl_command := current_query();  
  PERFORM pg_logical_emit_message(true, ...);
```

The emit_ddl function itself is simple. It gets the DDL query the user has run, and emits a WAL message.

Replication gotchas

```
{ "command_tag": "ALTER TABLE",  
  "schema_name": "my_schema",  
  "ddl": "ALTER TABLE my_schema.t1  
        ADD COLUMN email text NOT NULL;",  
  "objects": [{ "type": "table", "identity": "t1.id",  
                "pgstream_id": "{id}",  
                "primary_key_columns": ["id"],  
                "columns": [{ "name": "id",  
                             "type": "integer", ... } ... ] }
```

This is the message.

Replication gotchas

```
{ "command_tag": "ALTER TABLE",  
  "schema_name": "my_schema",  
  "ddl": "ALTER TABLE my_schema.t1  
        ADD COLUMN email text NOT NULL;",  
  "objects": [{ "type": "table", "identity": "t1.id",  
                "pgstream_id": "{id}",  
                "primary_key_columns": ["id"],  
                "columns": [{ "name": "id",  
                             "type": "integer", ... } ... ] }
```

17

The ddl portion contains the DDL statement you can replay on the target.

So now we know how to capture and replay DDL statements on the target, let's look into the other tricky part of replication. TOAST values.

Replication gotchas

the row, as it lives in the table page

column	type	value	size
id	integer	1	4 B
label	text	'nothing'	8 B
blob	text	► pointer	18 B
30 bytes in the page. The value is not in here.			

va_valueid
= 42868

pg_toast.pg_toast_42859

chunk 0	1996 B
chunk 1	1996 B
chunk 2	1996 B
chunk 3	412 B
4 chunks	6400 B

What is TOAST? The Oversized-Attribute Storage Technique.

It is not possible to store too big values directly, because tuples are not allowed to span multiple pages.

What is too big? A single PostgreSQL tuple size limit is set by `TOAST_TUPLE_THRESHOLD`. It is usually 2 kilobytes. When the row is bigger than that, Postgres compresses the widest values and moves them out of line until the row fits.

If we look at the row in the slide, we can see 2 columns that are stored in the table's page. And a 3rd one named `blob` which stores a pointer to the TOAST table. The pointer stores data about the TOAST table, `chunk_id`, and other metadata for retrieving the TOASTED value.

Why do we need to care about TOAST values during replication?

Replication gotchas

the row, as stored

column	type	value
id	integer	1
label	text	'before'
blob	text	out of line

TOAST table

blob · 6400 bytes

the UPDATE

```
UPDATE toasty
SET label = 'after';
blob is not touched
```

the WAL record

columns
id, label
blob is absent
what is not rewritten is not logged

identity — the old row

REPLICA IDENTITY DEFAULT
id
nothing to merge from

REPLICA IDENTITY FULL
id, label, blob (6400)
the value is here, in the old row

Let's say we update a row with a tuple that does not fit a single page. The WAL event emitted does not contain the blob column by default when replica identity is set. The TOASTed value is only included when it is updated.

When you set replica identity to full, Postgres puts the old row in the identity, so we can merge the blob column.

Demo: TOAST + REPLICATION IDENTITY

But let's see how TOAST values work with different REPLICATION IDENTITY settings.

In the demo we are going to update the label of the row. The value in the TOAST table remains untouched.

DEFAULT: The id of the updated row show up.

FULL: The whole old record is in the identity.

NOTHING: Update is empty. Updates are not replicated at all.

Now let's add a publication using `CREATE PUBLICATION` in the background. Nothing subscribes to it. We just update the record to show you what happens.

We get an error. PostgreSQL does not let you update such rows because the updates can't be replicated to the subscriber.

Notice when a publication is created, PostgreSQL makes sure that the subscribers are not going to drift. However, if you have a replication slot, there are no such protections in place.

Summary

- WAL file + LSN
- Replicating DDL statements using event triggers
- TOAST tables and REPLICA IDENTITY settings

That was part one: the PostgreSQL features you need to build your own CDC tool.

The WAL, the append only file and the LSN, the position in it.

DDL which logical decoding does not provide. Pgstream send it using event triggers.

And TOAST tables with proper replica identity settings.

Target 2.

PostgreSQL to OpenSearch

Document versioning, type mapping, schema changes

Next target is OpenSearch. The WAL events go to a search index. We face a new set of problems.

OpenSearch documents

Document version

- LSN
- Out of order messages and duplicates are rejected
- Idempotency token

```
{
  "_index": "{schema-name}-1",
  "_id": "{table-id}_{rec-id}",
  "_version": {lsn},
  "_source": {
    "_table": "{table-id}",
    "{table-id}-{col-id}": "espresso",
    "{table-id}-{col-id}": 2
  }
}
```

OpenSearch documents are stored in indices. Each one has an ID and a version number.

OpenSearch documents

Document version

- LSN
- Out of order messages and duplicates are rejected
- Idempotency token

```
{
  "_index": "{schema-name}-1",
  "_id": "{table-id}_{rec-id}",
  "_version": {lsn},
  "_source": {
    "_table": "{table-id}",
    "{table-id}-{col-id}": "espresso",
    "{table-id}-{col-id}": 2
  }
}
```

The document version is going to be the LSN. So we can reject out of order messages and duplicates. Essentially, LSN becomes the idempotency token.

OpenSearch documents

Type mapping is challenging

- target has fewer data types than PostgreSQL
 - numeric -> float
 - microsecond timestamps -> millisecond
- mappings are immutable

```
{
  "_index": "{schema-name}-1",
  "_id": "{table-id}_{rec-id}",
  "_version": {lsn},
  "_source": {
    "_table": "{table-id}",
    "{table-id}-{col-id}": "espresso",
    "{table-id}-{col-id}": 2
  }
}
```

When replicating to another target we need to consider the data type mapping. It can be problematic if the target does not have the same data types as PostgreSQL.

OpenSearch has fewer data types, so in some cases we lose precision. For example numeric types are converted to floats. Also, microsecond timestamps become millisecond timestamps.

Also, mappings are immutable. So you need to decide in the beginning which data types you need.

OpenSearch documents

DDL statements could trigger expensive reindexing

- table, column renames are expensive
- rename adds an alias, no need for reindexing documents

```
{
  "_index": "{schema-name}-1",
  "_id": "{table-id}_{rec-id}",
  "_version": {lsn},
  "_source": {
    "_table": "{table-id}",
    "{table-id}-{col-id}": "espresso",
    "{table-id}-{col-id}": 2
  }
}
```

Furthermore, while DDL changes can be cheap in one system, they can be expensive in another one.

In case of OpenSearch renaming a table or column requires reindexing all documents in the index. Even though in PostgreSQL these operation are almost free.

OpenSearch documents

DDL statements could trigger expensive reindexing

- table, column renames are expensive
- rename adds an alias, no need for reindexing documents

```
{
  "_index": "{schema-name}-1",
  "_id": "{table-id}_{rec-id}",
  "_version": {lsn},
  "_source": {
    "_table": "{table-id}",
    "{table-id}-{col-id}": "espresso",
    "{table-id}-{col-id}": 2
  }
}
```

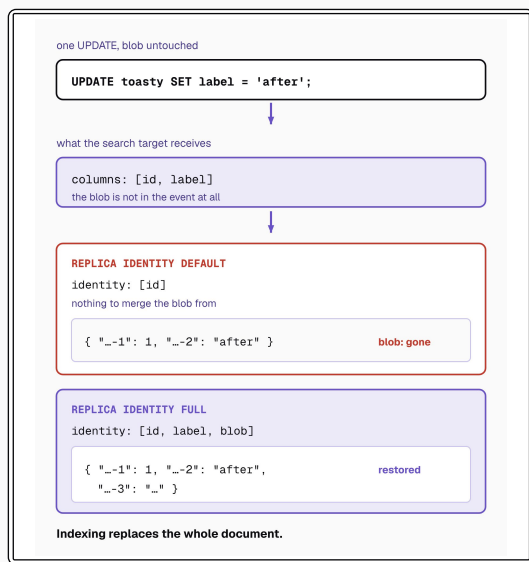
Pgstream avoids this problem by adding an alias to the index.

Notice that pgstream sets IDs for the table name and column names. When a table or a column is renamed, pgstream just adds a new alias.

OpenSearch documents

TOAST values

- LSN is an external version
- On update, we create a new document
- **REPLICA IDENTITY FULL** is required on update if the TOASTed value stays the same



Remember what I have said about replica identity. When you set it to default, the values from the TOAST table are omitted from the WAL event on update unless that value is changed.

Because we set the version to LSN, we cannot use the Update API of Opensearch.

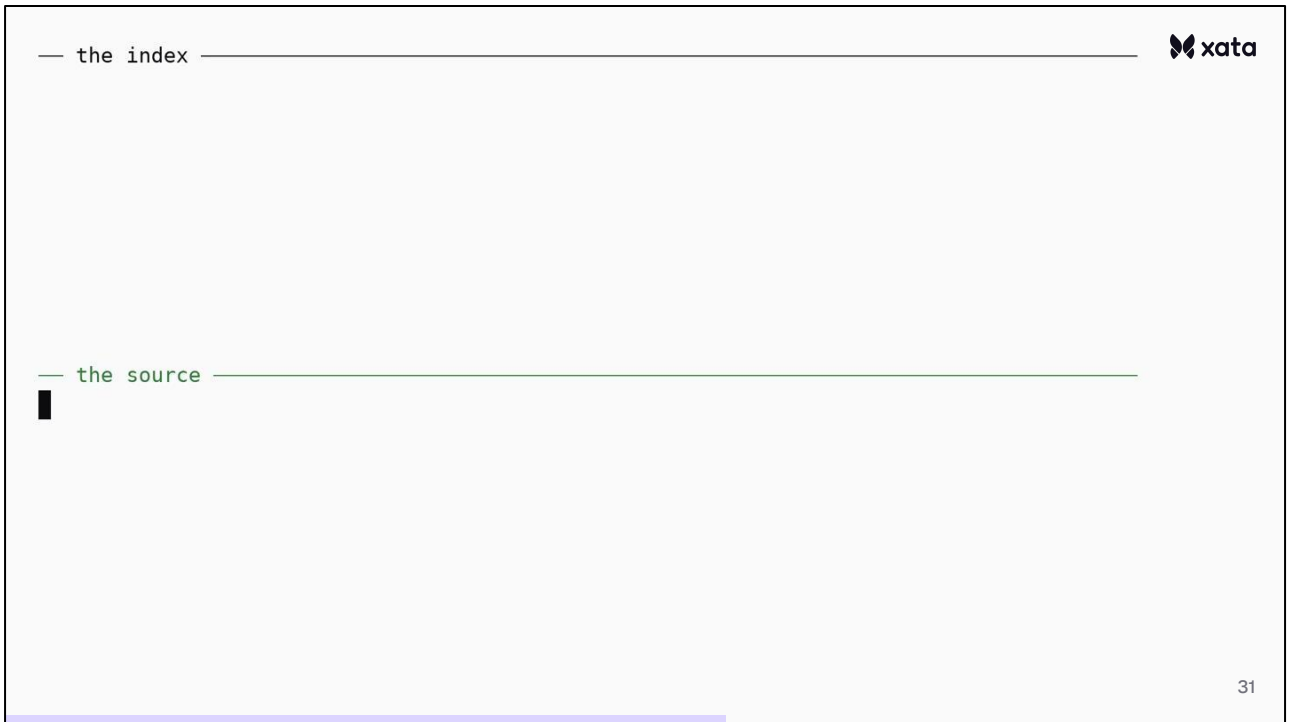
So you need to create a new document in Opensearch.

Thus, you need all the values during inserting the new updated document.

So you must set **REPLICA IDENTITY** to full. Otherwise, big values are not going to be part of the document.

Demo: Opensearch

Let's see the demo.



On the top we see the current LSN.

In the second portion, we can see the contents of the shop index. At the moment it is empty.

Let's add 2 new documents.

Now we are going to attempt to index the same document with an existing LSN.

This is rejected.

Let's rename the column product to item.

This creates an field alias in OpenSearch, so there is no need to reindex document. We can retrieve documents with both names.

Summary

- LSN as idempotency key
- Type mapping challenges: precision loss, immutability
- DDL statement can be free in PostgreSQL but extensive on the target
- Unchanged values in TOAST tables are only forwarded when REPLICATION IDENTITY is set to FULL

This was the second part. We've discussed the challenges around out of order and duplicated events. Also, we talked about the challenges of type mapping and the nature of schema changes in different systems. Finally, we covered values in TOAST tables.

Target 3.

PostgreSQL to Kafka

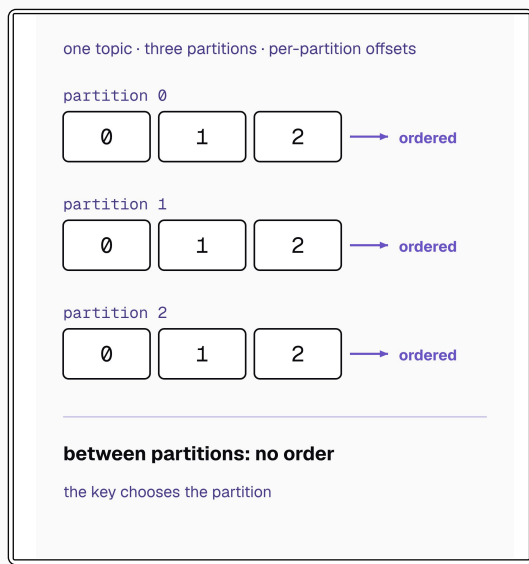
Ordering vs throughput

So far we've covered PostgreSQL and OpenSearch. Now we are diving into Kafka.

Unlike PostgreSQL, it does not store records, it stores the events.

Kafka partitioning

- One topic consists of multiple partitions
- Event key determines which partition stores the event
- Partitions are ordered



Kafka stores events in topics. Topics are usually stored on multiple partitions. The partition key of the event determines which partition the event is going to land.

Ordering is only guaranteed within a partition.

Kafka partition keys

	ordered	parallelism
schema	whole schema	# schemas
table	rows of a table	# tables
primary_key	one row	# partitions

We can use different partition keys for events: schema, table, primary key.

If you select schema, every event in that schema will land in the same partition in the same topic. Thus, every update to your database schemas will be ordered. You will get a hot partition.

If you choose table, events that belong to the same table will be in the same partition. This helps you distribute load among the partitions.

However, if you have a hot table, you might want to consider partitioning based on primary keys. This distributes every event evenly among the available partitions.

Demo schema

schema: shop

shop.products

id	serial	PK
name	text	
price	numeric(6,2)	
in_stock	boolean	added

shop.orders

id	serial	PK
product	text	
qty	int	

schema: analytics

analytics.events

id	serial	PK
kind	text	
payload	text	

```
ALTER TABLE shop.products ADD COLUMN in_stock boolean NOT NULL DEFAULT true;
```

We created two schemas and 3 tables: analytics and shop.

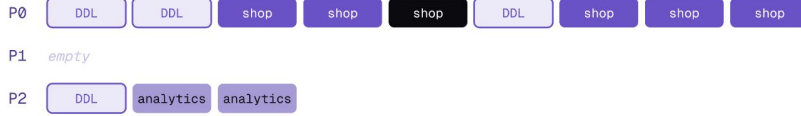
the same twelve changes, one topic, three partitions — only the key changes



partition_key: schema

orders everything in a schema, DDL included

ceiling: one partition per schema



partition_key: table

orders the rows of one table

ceiling: one partition per table



partition_key: primary_key

orders the writes to one row

ceiling: the partition count



Debezium comparison

	pgstream	Debezium (Postgres)
Topics	one, whole database	one per table
Default key	schema	primary key
Output plugins	wal2json	pgoutput
Schema changes	supported	no ALTER TABLE msg

Pgstream's approach differs from Debezium. Pgstream puts the whole database in the same topic. Debezium puts tables in different topics.

Because the table is already the topic, Debezium uses the primary key as the partition key. So every insert, update and deletion event for the same row go to the same partition. As I mentioned before pgstream support multiple partitioning keys: schema, table and primary key.

Another significant difference is that pgstream requires wal2json plugin. Which needs to be installed on your PostgreSQL instance. Debezium relies on pgoutput which is the builtin method.

I would say consider your requirements when choosing between the two. Pgstream is a specialist, Debezium supports multiple database types with a rich connector ecosystem.

the same twelve changes, one database, two tools reading it at once

pgstream · one topic · key: primary_key

P0 shop DDL CREATE TABLE

P0 shop DDL CREATE TABLE

P2 analytics DDL CREATE TABLE

P0 shop.products:1 INSERT shop.products

P0 analytics.events:1 INSERT analytics.events

P2 shop.products:2 INSERT shop.products

P0 shop.orders:1 INSERT shop.orders

P0 shop DDL ALTER TABLE

P0 shop.products:3 INSERT shop.products

P1 analytics.events:2 INSERT analytics.events

P0 shop.products:1 UPDATE shop.products

P0 shop.products:1 UPDATE shop.products

Debezium · one topic per table

(no event)

(no event)

(no event)

P0 dbz.shop.products INSERT {"id":1}

P0 dbz.analytics.events INSERT {"id":1}

P1 dbz.shop.products INSERT {"id":2}

P0 dbz.shop.orders INSERT {"id":1}

(no event)

P0 dbz.shop.products INSERT {"id":3}

P1 dbz.analytics.events INSERT {"id":2}

P0 dbz.shop.products UPDATE {"id":1}

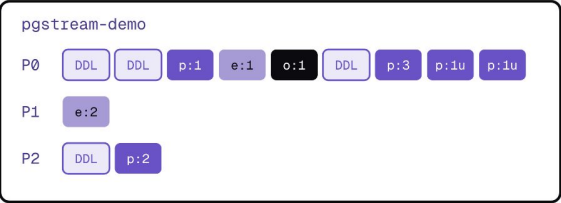
P0 dbz.shop.products UPDATE {"id":1}

Debezium's Postgres connector emits no schema change of its own.

Both key by row here. Debezium's key needs no table name — the topic already carries it.

the same writes, keyed by row in both

pgstream · one topic



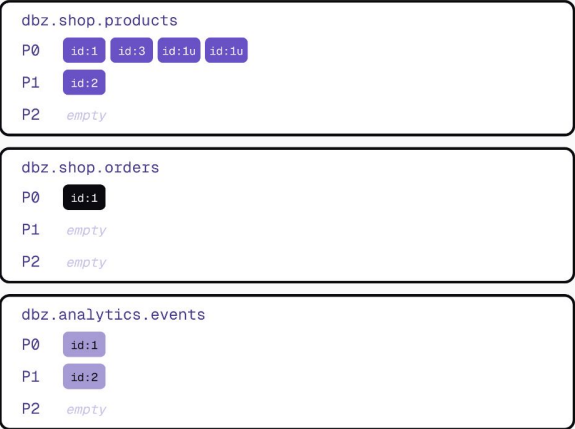
1 topic · 3 partitions · 12 events — schema changes in the same stream

what the chips mean

- p:1** shop.products
- o:1** shop.orders
- e:1** analytics.events
- DDL** CREATE / ALTER TABLE

the number is the primary key · u marks an update

Debezium · one topic per table



3 topics · 9 partitions · 8 events — no schema changes at all

Debezium spends the table on the topic name. pgstream spends it on the key.

Summary

- PostgreSQL sends WAL events in order
- Some targets cannot guarantee ordering for every configuration

In this section I talked about ordering the changes and how to configure the throughput when dealing with Kafka.

Target 4.

PostgreSQL to Webhook

LSN as an idempotency key

45

Now comes the last target: webhooks.

We can send HTTP requests to web services in case of data or schema changes.

In this part of the presentation I am going to talk about idempotency and retries.

HTTP request

- Usually a trigger fires an HTTP call
 - http extension
 - LISTEN + NOTIFY
- Or implement outbox pattern in PostgreSQL
 - needs a worker to poll PostgreSQL
 - duplicates the changes captured in the WAL

There are several ways to send HTTP requests based on PostgreSQL rows inserts/updates.

On a trigger you can either send an HTTP request using the http extension. However, it runs in the insert transaction. So a row cannot be committed until the web server accepts the request. Alternatively, you can use LISTEN and NOTIFY to read records in your application. But if there is no listener, the rows will get lost.

A more durable solution is to implement the outbox pattern. But in this case you need to write the worker and run it yourself. Not to mention that you are keeping a record of changes in a different table now. We already have these changes captured in the WAL file. So why not use them?

HTTP request

Idempotency key

- Eliminates duplicates
- LSN in **X-Pgstream-Lsn**
- Globally monotonic

Retries:

- On HTTP 5xx/HTTP 429/network issues -> request is retried
- HTTP 4xx -> no retry

```
POST /capture HTTP/1.1
Idempotency-Key: 0/BC5E3F0
X-Pgstream-Lsn: 0/BC5E3F0

{"Data":{"action":"I","lsn":"0/BC5E3F0","schema":"shop","table":"orders","columns":[{"name":"product","type":"text","value":"doppio"}, ...], "identity":null}}
```

47

When sending events to an HTTP endpoint we need to determine the idempotency of the endpoint. POST is usually not idempotent. But some services let you set an idempotency key in your request. Based on that key the service can deduplicate incoming requests.

We can use the LSN to distinguish events. It is globally monotonic, so it is not going to collide with another key. Also, if any of the requests arrive out of order, you can use this to ensure proper ordering.

Error handling is vital here. You might get 500s or 429, those are retried. 400 are not.

A retryable failure that runs out of retries keeps the event and stops the pipeline. A permanent failure drops the event and keeps going.

Demo: webhook

— the three subscribers —

WEBHOOK SUBSCRIBERS one event -> three subscribers

subscriber	answers	class	reqs	delivered	dupes
analytics	200	success	0	0	0
billing	200	success	0	0	0
legacy	200	success	0	0	0

SLOT confirmed_flush 0/CEF8A1A retained 0 bytes advancing

— 200 / 400 / 503 —

█

Summary

- Targets might be temporarily unavailable
- Some requests can be retried, others might not
- Decide if you want to drop events or stop the processing pipeline

This was the last target I wanted to talk about. In this section I mentioned that targets might be temporarily unavailable. I also mentioned different errors, ones that can be retried and ones that cannot. It is up to you how you want to handle these errors. You can either drop events or stop the processing pipeline, or wait until it recovers.

The experiment

Using a single WAL slot

To feed data into 4 destinations

51

Normally pgstream is used to send to one target. I attempted to transform pgstream into a tool that can serve 4 targets at the same time.

Demo: One slot, four targets

52

The previous demos showcased one target at a time to introduce replication concepts and problems.

Now comes the demo of my experiment with pgstream. I created a prototype to send events to multiple targets at the same time from the same pgstream process.

— the four sinks —

POSTGRES **SELECT on the target**

(nothing yet)

OPENSEARCH **GET shop/_search**

(nothing yet)

KAFKA

(nothing yet)

WEBHOOK

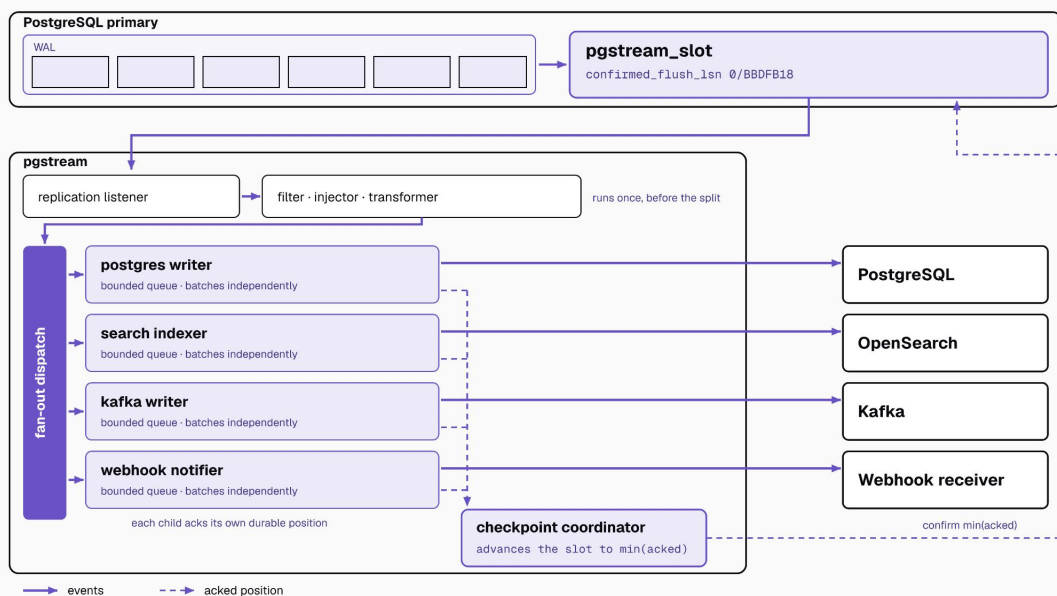
(nothing yet)

— the source —

█

So let's see the demo.

This is the happy path. All targets are healthy and can process the incoming events, requests in time.



What did we see in the demo?

Pgstream read the changes in PostgreSQL. The replication events went through the usual pgstream pipeline (filtering, transformers, etc.). Then each event was sent to all four targets.

Once each target processed the events, pgstream acknowledged the changes, saved the latest checkpoint.

But now let's see what happens when any of the targets fail to process events.

Demo: Let's break it.

It's time to break the pipeline to discover the weaknesses of the approach.

```
— the slot —  
TIME      RETAINED    FLUSHED    CAT_XMIN  DEAD  
t+000s    53 kB      0/CF7BF0C  5803     390
```

```
— break it —  
█
```

On the top we can see the replication slot status, catalog xmin and number of dead tuples.

On the bottom we are going to start all targets and stop Opensearch.

Notice that the retained WAL size is increasing, LSN stays the same, catalog xmin, stays the same, and the number of dead tuples are increasing.

Catalog vacuum cannot remove the dead tuples because the consumer has not finished processing the WAL file.

Now let's restart OpenSearch and watch how WAL size goes down and dead tuples are removed, once vacuum runs again.

Coupling

-  If one target slows down, replication slot cannot progress
-  If the pgstream process dies, all targets stop receiving
-  On restart, all targets get the duplicated events based on the slowest target

What problems did we see?

First of all, targets are tightly coupled. If one of them slows down or stops, the replication slot cannot progress. Or even worse, if pgstream dies, all targets stop receiving events.

When pgstream comes back online, it might send duplicated events that targets need to handle.

Backpressure

-  You cannot eliminate backpressure. But you can choose where to handle it.
-  In PostgreSQL it means growing WAL, increasing slot lag, catalog_xmin
-  If you want to move backpressure to a different location, you can try Kafka.

Takeaways

- PostgreSQL comes with powerful primitives
 - WAL
 - LSN
 - REPLICA IDENTITY
- Consider dropping events and retries
- You choose where backpressure lives during replication
- Monitor PostgreSQL
 - Watch retained WAL

So that was a lot of information and demos about PostgreSQL, Opensearch, Kafka and webhooks.

The main takeaway of my talk is that PostgreSQL comes with powerful primitives to build your own CDC tool. You can leverage WAL files, LSN and the proper replica identity settings.

You also need to consider failure modes because as we all know systems sometimes misbehave. You might lose events or your processing pipeline can get stuck. During high load, you might encounter backpressure. You need to choose which piece in your architecture is going to handle it.

Always monitor PostgreSQL to avoid nasty surprises like running out of disk.

Resources

- Repo with experimental multi target support
 - <https://github.com/kvch/pgconfeu26-pgstream/tree/fanout-experiment>

If you'd like to dig deeper, everything from this talk (the slides, the scripts, and the **pgstream** experiment) is available on GitHub.

And one last recommendation: if it's been a while, go home and watch *Ratatouille*. It's still just as inspiring as the first time.



Branch any PostgreSQL database in seconds

Thank you!

xata.io

[X](#) @xata

How pgstream consumes WAL

WAL Storage
example

WAL Storage

Stores internal
identifiers and data in
binary format

Logical Decoding
example

Logical Decoding

Looks up table metadata,
assembles transactions
in commit order

Plugin specific
formatting example

Plugin specific format

`wal2json` plugin turns
the output into JSON
format

the same nine writes, in write order, under two keys

partition_key: schema

P0	shop	INSERT shop.products
P2	analytics	INSERT analytics.events
P0	shop	INSERT shop.products
P0	shop	INSERT shop.orders
P0	shop	ALTER TABLE shop.products
P0	shop	INSERT shop.products
P2	analytics	INSERT analytics.events
P0	shop	UPDATE shop.products
P0	shop	UPDATE shop.products

partition_key: primary_key

P0	shop.products:1	INSERT shop.products
P0	analytics.events:1	INSERT analytics.events
P2	shop.products:2	INSERT shop.products
P0	shop.orders:1	INSERT shop.orders
P0	shop	ALTER TABLE shop.products
P0	shop.products:3	INSERT shop.products
P1	analytics.events:2	INSERT analytics.events
P0	shop.products:1	UPDATE shop.products
P0	shop.products:1	UPDATE shop.products

Every key changed except the DDL's.

DDL is keyed by its schema in every mode.

the same twelve changes, one database, two tools reading it at once

pgstream · one topic

P0 shop DDL CREATE TABLE
P0 shop DDL CREATE TABLE
P2 analytics DDL CREATE TABLE
P0 shop INSERT shop.products
P2 analytics INSERT analytics.events
P0 shop INSERT shop.products
P0 shop INSERT shop.orders
P0 shop DDL ALTER TABLE
P0 shop INSERT shop.products
P2 analytics INSERT analytics.events
P0 shop UPDATE shop.products
P0 shop UPDATE shop.products

Debezium · one topic per table

(no event)
(no event)
(no event)
P0 dbz.shop.products INSERT id=1
P0 dbz.analytics.events INSERT id=1
P1 dbz.shop.products INSERT id=2
P0 dbz.shop.orders INSERT id=1
P1 dbz.message MESSAGE prefix=pgstream.ddl
P0 dbz.shop.products INSERT id=3
P1 dbz.analytics.events INSERT id=2
P0 dbz.shop.products UPDATE id=1
P0 dbz.shop.products UPDATE id=1

Debezium's Postgres connector emits no schema change of its own.

The one DDL row it carries is pgstream's event-trigger message, relayed as opaque bytes on dbz.message.

the same twelve changes, one database, two tools reading it at once

pgstream · one topic · key: primary_key

P0 shop DDL CREATE TABLE

P0 shop DDL CREATE TABLE

P2 analytics DDL CREATE TABLE

P0 shop.products:1 INSERT shop.products

P0 analytics.events:1 INSERT analytics.events

P2 shop.products:2 INSERT shop.products

P0 shop.orders:1 INSERT shop.orders

P0 shop DDL ALTER TABLE

P0 shop.products:3 INSERT shop.products

P1 analytics.events:2 INSERT analytics.events

P0 shop.products:1 UPDATE shop.products

P0 shop.products:1 UPDATE shop.products

Debezium · one topic per table

(no event)

(no event)

(no event)

P0 dbz.shop.products INSERT {"id":1}

P0 dbz.analytics.events INSERT {"id":1}

P1 dbz.shop.products INSERT {"id":2}

P0 dbz.shop.orders INSERT {"id":1}

(no event)

P0 dbz.shop.products INSERT {"id":3}

P1 dbz.analytics.events INSERT {"id":2}

P0 dbz.shop.products UPDATE {"id":1}

P0 dbz.shop.products UPDATE {"id":1}

Debezium's Postgres connector emits no schema change of its own.

Both key by row here. Debezium's key needs no table name — the topic already carries it.