

Replication Logical Replication



Copyright

© Postgres Professional, 2017–2025

Authors: Egor Rogov, Pavel Luzanov, Ilya Bashtanov, Alexey Beresnev

Photo: Oleg Bartunov (Phu Monastery and Bhrikuti Peak, Nepal)

Use of Course Materials

Non-commercial use of course materials (presentations, demonstrations) is allowed without restrictions. Commercial use is possible only with the written permission of Postgres Professional. It is prohibited to make changes to the course materials.

Feedback

Please send your feedback, comments and suggestions to:

edu@postgrespro.ru

Disclaimer

Postgres Professional assumes no responsibility for any damages and losses, including loss of income, caused by direct or indirect, intentional or accidental use of course materials. Postgres Professional company specifically disclaims any warranties on course materials. Course materials are provided “as is,” and Postgres Professional company has no obligations to provide maintenance, support, updates, enhancements, or modifications.

Logical vs Physical Replication
Publications and Subscriptions
Initial Synchronization
Row Identification and Filtration
Logical Decoding
Large Transactions
Conflicts
Replication Sources

Physical

- leader-replica: data flow in one direction only
- stream WAL records or send WAL files
- binary server compatibility is required
- whole cluster replication only

Logical

- publication-subscriptions: servers have no dedicated roles
- row-level changes are replicated
- protocol-level compatibility is required
- can replicate individual tables

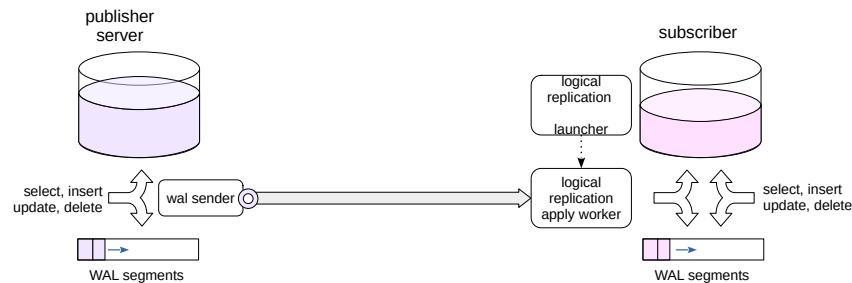
During physical replication, one server is assigned the leader role and others become replicas. The leader transfers WAL records to replicas to be applied to data files there. The replica does not “understand” the changes, just applies them mechanically without any interpretation of what the changes mean, so binary compatibility between servers is necessary (the same platforms and major PostgreSQL versions). Since the WAL is shared across the entire cluster, only the cluster as a whole can be replicated.

In logical replication, one server creates a publication that other servers can subscribe to. Servers don't have fixed roles: the same server can both publish changes and subscribe to publications from other servers (or even its own).

By default, information about table row changes is transmitted to the subscription in a text format that's platform-independent and doesn't require binary compatibility.

<https://postgrespro.com/docs/postgresql/16/logical-replication>

How It Works



max_wal_senders = 10
max_replication_slots = 10

max_logical_replication_workers = 4
max_worker_processes = 8

4

Logical replication uses the publish-subscribe pattern.

When the instance starts, the **logical replication launcher** background process starts with it, periodically checking the `pg_subscription` table in the system catalog for new subscriptions. As soon as a subscription appears, a background **logical replication apply worker** process is launched to handle it. It connects to the publishing server, which, like any replication connection, starts the **wal sender** process. This process reads the WAL, selects the necessary records and converts them to platform-independent protocol messages. A logical replication slot is always used for that.

On the subscription side, the worker receives messages and applies changes. Meanwhile, the subscriber server can receive normal read and write queries.

Note that the publishing server can run multiple wal sender processes, one for each subscription. Both *max_wal_senders* and *max_replication_slots* must be no less than the number of subscriptions.

On the subscriber server, you need to configure *max_logical_replication_workers* (for processes that receive subscription changes) and *max_worker_processes* in general (more by one at least, since there's also the logical replication launcher process, though this pool is actually used for other purposes as well).

<https://postgrespro.com/docs/postgresql/16/logical-replication-architecture>

Database object

sends data changes row by row in the order they are committed in
only for regular and partitioned tables
`wal_level = logical`

Published operations

insert, update, delete, truncate

Privileges

creating a new publication – CREATE privilege for the database
add table – owner or superuser
add all tables for a schema or a database – superuser

A *publication* is created on the server to collect changes from one or more tables (regular or partitioned) in the same database. To create a publication, you need the CREATE privilege on the database. To replicate from multiple databases, multiple publications must be created.

To include a table in a publication, you must be either a superuser or the owner of the table. To publish all tables belonging to a schema or a database, you must be a superuser.

For logical replication to work, the publishing server's WAL must contain additional information (`wal_level = logical`).

Publishing includes row changes that occur in tables as a result of DML commands. The MERGE command executed on the publishing server is applied as INSERT, UPDATE, or DELETE.

By default, changes are not transferred immediately, but only when the transaction is committed.

<https://postgrespro.com/docs/postgresql/16/logical-replication-publication>

<https://postgrespro.com/docs/postgresql/16/sql-createpublication>

Database object

- receives messages from one or more publications
- binary subscription possible
- applies changes row by row
- Tables and columns are matched by full names

Privileges

- create subscription – pg_create_subscription role or superuser

By defining subscriptions to publications in a database, you can receive and apply changes.

Basically, a subscription is a connection to the publishing server that serves for data replication.

Members of the pg_create_subscription role or superusers can create subscriptions.

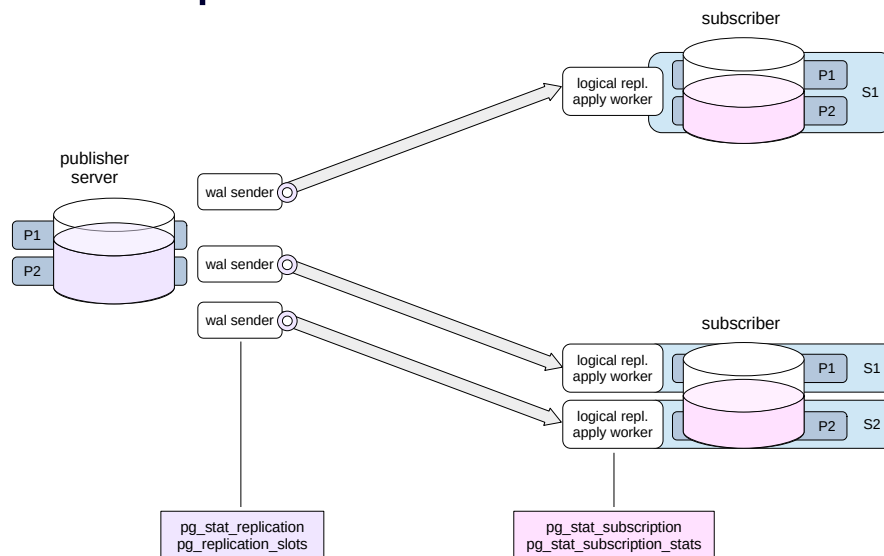
Changes are always applied row by row. Although each individual change doesn't require query parsing and planning, bulk changes will still apply more slowly because of this row-by-row approach.

Tables are identified by their full names (including the schema), and columns are identified by their names as well. This allows the subscription to use a different data schema (for example, having additional columns in the table).

When creating a subscription, you can specify the parameter binary to transfer the data in binary format. The binary format can speed up processing if binary receive and send functions are defined for the required data types. However, when using binary format, it's important that the machine representation of data is compatible across different platforms.

<https://postgrespro.com/docs/postgresql/16/logical-replication-subscription>

Publications and Subscriptions



7

You can distribute objects to publications in any way on the publishing server. You can include all tables in one publication, you can split them between multiple publications – this does not affect performance.

However, each subscription creates a separate connection to the publishing server, creating unnecessary load.

Several wal sender processes will run on the publishing server, each of which will independently read the WAL and perform logical decoding (more on that below) in its local memory.

Therefore, if you need to subscribe to several publications, it is better to do it in one common subscription.

Remember that publications and subscriptions are database objects, so each database will require a separate connection.

Logical replication monitoring on the publishing server is identical to physical replication monitoring. Your most important tools here are the views `pg_stat_replication` and `pg_replication_slots`.

Subscription status information is stored in `pg_subscription`.

<https://postgrespro.com/docs/postgresql/16/catalog-pg-subscription#CATALOG-PG-SUBSCRIPTION>

Information about applying changes is found in the `pg_stat_subscription` view.

<https://postgrespro.com/docs/postgresql/16/monitoring-stats#MONITORING-PG-STAT-SUBSCRIPTION>

Replication errors go to `pg_stat_subscription_stats`.

<https://postgrespro.ru/docs/postgresql/16/monitoring-stats#MONITORING-PG-STAT-SUBSCRIPTION-STATS>

Logical Replication

We're going to set up logical replication for the test table from the alpha server to the beta server.

Let's start by creating a database.

```
α=> CREATE DATABASE replica_logical;
```

```
CREATE DATABASE
```

```
α=> \c replica_logical
```

You are now connected to database "replica_logical" as user "student".

Logical replication requires another WAL level.

```
α=> ALTER SYSTEM SET wal_level = logical;
```

```
ALTER SYSTEM
```

```
student$ sudo pg_ctlcluster 16 alpha restart
```

We'll make the second database cluster a copy of the first one, so let's back it up to the second server's data directory.

```
student$ sudo pg_ctlcluster 16 beta stop
```

Cluster is not running.

```
postgres$ rm -rf /var/lib/postgresql/16/beta
```

```
postgres$ pg_basebackup --checkpoint=fast --pgdata=/var/lib/postgresql/16/beta
```

```
student$ sudo pg_ctlcluster 16 beta start
```

```
student$ psql -d replica_logical
```

Now let's create a table on the first server and populate it with data.

```
α=> CREATE TABLE test(  
  id int PRIMARY KEY GENERATED BY DEFAULT AS IDENTITY,  
  descr text,  
  cluster_name text DEFAULT 'alpha'  
);
```

```
CREATE TABLE
```

The test table is not present on the second server. Since DDL commands are not replicated, we need to create this table manually. Its structure may be different, as long as all the published tables are present.

```
student$ psql -p 5433 -d replica_logical
```

```
β=> CREATE TABLE test(  
  id int PRIMARY KEY GENERATED BY DEFAULT AS IDENTITY,  
  descr text,  
  cluster_name text DEFAULT 'beta',  
  additional text  
);
```

```
| CREATE TABLE
```

Create a publication on the first server.

```
α=> CREATE PUBLICATION test_pub;
```

```
CREATE PUBLICATION
```

```
α=> \dRp+
```

	Publication test_pub					
Owner	All tables	Inserts	Updates	Deletes	Truncates	Via root
student	f	t	t	t	t	f

(1 row)

Notice that the publication was created even though it doesn't contain any tables yet. Typically, you create a publication by specifying:

- a list of tables, and in this case you can specify which columns to publish
- all tables in specified schemas (requires superuser privileges)

- all tables in the database (requires superuser privileges)

Let's add the table to the publication.

```
α=> ALTER PUBLICATION test_pub ADD TABLE test;
```

ALTER PUBLICATION

```
α=> \dRp+
```

```

                                Publication test_pub
 Owner | All tables | Inserts | Updates | Deletes | Truncates | Via root
-----+-----+-----+-----+-----+-----+-----
 student | f          | t       | t       | t       | t       | f
Tables:
 "public.test"
```

On the second server, subscribe to the publication:

```
β=> CREATE SUBSCRIPTION test_sub
CONNECTION 'host=localhost port=5432 user=student password=student dbname=replica_logical'
PUBLICATION test_pub;
```

```
NOTICE: created replication slot "test_sub" on publisher
CREATE SUBSCRIPTION
```

```
β=> \dRs
```

```

                        List of subscriptions
   Name   | Owner | Enabled | Publication
-----+-----+-----+-----
 test_sub | student | t      | {test_pub}
(1 row)
```

Check the subscription we created.

```
α=> INSERT INTO test(descr) VALUES ('One');
```

INSERT 0 1

```
β=> SELECT * FROM test;
```

```

 id | descr | cluster_name | additional
-----+-----+-----+-----
  1 | One   | alpha        |
(1 row)
```

The values for id, descr and cluster_name columns are retrieved from the publication.

Starting with PostgreSQL 15, you can publish only specific columns instead of all of them. Let's modify the publication:

```
α=> ALTER PUBLICATION test_pub SET TABLE test (id, descr);
```

ALTER PUBLICATION

```
α=> \dRp+
```

```

                                Publication test_pub
 Owner | All tables | Inserts | Updates | Deletes | Truncates | Via root
-----+-----+-----+-----+-----+-----+-----
 student | f          | t       | t       | t       | t       | f
Tables:
 "public.test" (id, descr)
```

Check:

```
α=> INSERT INTO test(descr) VALUES ('Two');
```

INSERT 0 1

```
β=> SELECT * FROM test;
```

```

 id | descr | cluster_name | additional
-----+-----+-----+-----
  1 | One   | alpha        |
  2 | Two   | beta         |
(2 rows)
```

Notice that the cname column now has the default value (beta) on the subscriber.

The following view shows the state of the subscription:

```
beta=> SELECT * FROM pg_stat_subscription \gx
```

```
-[ RECORD 1 ]-----+-----  
subid          | 16419  
subname        | test_sub  
pid            | 82645  
leader_pid     |  
reliid         |  
received_lsn   | 0/3084B30  
last_msg_send_time | 2026-09-17 00:02:54.298711+03  
last_msg_receipt_time | 2026-09-17 00:02:54.299514+03  
latest_end_lsn | 0/3084B30  
latest_end_time | 2026-09-17 00:02:54.298711+03
```

- received_lsn – the LSN up to which changes have been received
- latest_end_lsn – the LSN confirmed to the WAL sender process

The logical replication worker process starts (you can see its ID in pg_stat_subscription.pid):

```
student$ ps f -C postgres -o pid,command | grep beta
```

```
82278 /usr/lib/postgresql/16/bin/postgres -D /var/lib/postgresql/16/beta -c  
config_file=/etc/postgresql/16/beta/postgresql.conf  
82279 \_ postgres: 16/beta: checkpointer  
82280 \_ postgres: 16/beta: background writer  
82283 \_ postgres: 16/beta: walwriter  
82284 \_ postgres: 16/beta: autovacuum launcher  
82285 \_ postgres: 16/beta: logical replication launcher  
82410 \_ postgres: 16/beta: student replica_logical [local] idle  
82645 \_ postgres: 16/beta: logical replication apply worker for subscription 16419
```

Not replicated

- sequences
- materialized views
- temporary and unlogged tables
- foreign tables
- large objects
- DDL

TRUNCATE restriction

- tables sharing foreign keys must belong to the same subscription

Only changes to the content of regular and partitioned tables caused by DML commands are replicated.

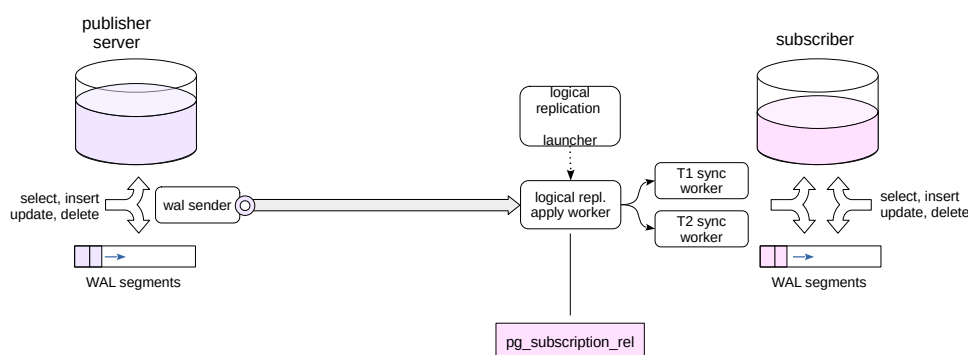
The contents of the rest of the objects, broadly defined as “relations”, are not replicated. This includes sequences, materialized views, temporary and unlogged tables, foreign tables. Large objects are not replicated.

DDL commands are also not replicated, so you must first create the necessary tables on the subscription side.

TRUNCATE replicates successfully only if all subscriber tables whose foreign keys reference the tables being truncated are included in the same subscription.

<https://postgrespro.com/docs/postgresql/16/logical-replication-restrictions>

Initial Synchronization



One synchronization process per table

but no more than $\text{max_sync_workers_per_subscription} = 2$

10

By default, if you do not specify a value for `copy_data = false`, initial table content synchronization occurs when a subscription is created. This happens seamlessly through the use of the snapshot export mechanism.

The initial synchronization should be completed as soon as possible so as not to delay the vacuuming horizon longer than necessary. Therefore, synchronization is usually done in parallel: the apply worker process looks at the list of replicated tables and starts additional workers that do the actual synchronization. One table is always synchronized by one process.

The maximum number of concurrent synchronization processes is limited by `max_sync_workers_per_subscription` (default is two).

The replication status for each table in the subscription is shown in the `pg_subscription_rel` view.

<https://postgrespro.com/docs/postgresql/16/catalog-pg-subscription-rel#CATALOG-PG-SUBSCRIPTION-REL>

If the subscription is flagged as binary, then not only replication will be performed in binary format, but also initial synchronization.

Initial Synchronization

By default, the subscriber collects table contents first, and replication begins after that. This is seamless, and data retention is guaranteed.

Create three tables with a huge number of rows, and a publication:

```
α=> CREATE TABLE t1 AS
    SELECT id, 'This is row nubmer' || id descr FROM generate_series(1,1_000_000) id;

SELECT 1000000

α=> CREATE TABLE t2 AS
    SELECT id, 'This is row nubmer' || id descr FROM generate_series(1,1_000_000) id;

SELECT 1000000

α=> CREATE TABLE t3 AS
    SELECT id, 'This is row nubmer' || id descr FROM generate_series(1,1_000_000) id;

SELECT 1000000

α=> CREATE PUBLICATION t123_pub FOR TABLE t1, TABLE t2, TABLE t3;

CREATE PUBLICATION
```

On the subscriber, create empty tables with identical structure:

```
| β=> CREATE TABLE t1( id integer, descr text );
|
| CREATE TABLE
|
| β=> CREATE TABLE t2( id integer, descr text );
|
| CREATE TABLE
|
| β=> CREATE TABLE t3( id integer, descr text );
|
| CREATE TABLE
```

Subscribe to the publication, then peek into the system table that tracks the replication state of each table, and into the process list.

```
| β=> CREATE SUBSCRIPTION t123_sub
| CONNECTION 'host=localhost port=5432 user=student password=student dbname=replica_logical'
| PUBLICATION t123_pub;
|
| NOTICE: created replication slot "t123_sub" on publisher
| CREATE SUBSCRIPTION
|
| β=> SELECT *, srrelid::regclass::text tablename FROM pg_subscription_rel;
|
| srsubid | srrelid | srsubstate | srsublsn | tablename
|-----+-----+-----+-----+-----
| 16419 | 16411 | r | 0/3084198 | test
| 16435 | 16430 | i | | t3
| 16435 | 16420 | i | | t1
| 16435 | 16425 | i | | t2
| (4 rows)
```

Here, it says that the table test is being replicated (r), and that the three new tables are still synchronizing.

```
student$ ps f -C postgres -o pid,command | grep beta

82278 /usr/lib/postgresql/16/bin/postgres -D /var/lib/postgresql/16/beta -c
config_file=/etc/postgresql/16/beta/postgresql.conf
82279 \_ postgres: 16/beta: checkpointer
82280 \_ postgres: 16/beta: background writer
82283 \_ postgres: 16/beta: walwriter
82284 \_ postgres: 16/beta: autovacuum launcher
82285 \_ postgres: 16/beta: logical replication launcher
82410 \_ postgres: 16/beta: student replica_logical [local] idle
82645 \_ postgres: 16/beta: logical replication apply worker for subscription 16419
83332 \_ postgres: 16/beta: logical replication apply worker for subscription 16435
83353 \_ postgres: 16/beta: logical replication worker for subscription 16435 sync
16430
83356 \_ postgres: 16/beta: logical replication worker for subscription 16435 sync
16420
```

In the process list, we see a new process apply worker serving the new subscription, as well as two synchronization processes. There are three tables, but the number of concurrent processes is limited by the parameter:

```
| β=> SHOW max_sync_workers_per_subscription;
```

```
max_sync_workers_per_subscription
```

```
-----
```

```
2
```

```
(1 row)
```

Wait for synchronization to complete.

Now all the tables are being replicated.

```
| β=> SELECT * FROM pg_subscription_rel;
```

```
srsubid | srrelid | srsubstate | srsublsn
```

```
-----+-----+-----+-----
```

```
16419 | 16411 | r          | 0/3084198
```

```
16435 | 16430 | r          | 0/12D22528
```

```
16435 | 16420 | r          | 0/12D22528
```

```
16435 | 16425 | r          | 0/12D22598
```

```
(4 rows)
```

```
| β=> SELECT count(*) FROM t1;
```

```
count
```

```
-----
```

```
1000000
```

```
(1 row)
```

We don't need the publication and subscription anymore, delete them:

```
| β=> DROP SUBSCRIPTION t123_sub;
```

```
NOTICE: dropped replication slot "t123_sub" on publisher
DROP SUBSCRIPTION
```

```
| α=> DROP PUBLICATION t123_pub;
```

```
DROP PUBLICATION
```

There must be a suitable row for change and delete operations

Logical row ID

```
ALTER TABLE ... REPLICA IDENTITY ...
```

DEFAULT	primary key columns (default)
USING INDEX	unique index columns with NOT NULL constraint
FULL	all columns
NOTHING	no identification

(default for the system catalog)

12

New rows are inserted unambiguously on the subscription side. Changes and deletions are more complicated. You need to determine which row on the subscriber to apply the operation to. Therefore, a *logical* row ID is defined for each table. By default, rows are identified by primary key columns, but the identifier can be changed with `ALTER TABLE ... REPLICA IDENTITY`.

Rows can be identified by another unique index, in which all columns have a NOT NULL constraint.

Another option is identification by all columns. In PostgreSQL

16 and higher, you can use full B-tree index lookup, where the first field is a column from the publication. PostgreSQL 17 and up can also use hash indexes. If there is no index available, a full scan is performed, which is extremely ineffective for large tables.

You can also completely disable replication support for certain tables (this is how system catalog tables are set up by default).

When changing or deleting, the column values of the tuple to be deleted do not get into the replica WAL – this is one of the reasons why the logical WAL level is required on the publishing server. At logical, it stores the old values of the columns included in the logical identifier. This is why with identification by all columns, the WAL size can be significantly larger compared to the replica level.

<https://postgrespro.com/docs/postgresql/16/sql-altertable#SQL-ALTERTABLE-REPLICA-IDENTITY>

Filter at publication

Filtering possible

insert	for any published columns
delete	for logical ID columns
update	for logical ID columns
	this may transform into insert or delete
truncate	filter ignored

Plain expressions only

By default, the publishing server sends out all published table rows.

Since PostgreSQL 15, you can specify a filter expression. Rows that evaluate as false or NULL will not be published.

Filtering does not affect the truncate operation. For insert operations, any published columns can be included in the filter expression, and for update and delete, only the columns included in the logical row ID.

For update, the filter expression is evaluated for both the old and the new row version.

If the filter expression is false or NULL for both, the row is not replicated. If the old row doesn't match the filter but the new one does, the new row is inserted. If the opposite is true — the old one matches but the new one does not — the row is deleted.

If both match, update goes through.

You can only filter using plain expressions: no user-defined functions, operators, types and collations, no referencing system columns, no calling built-in functions with volatile and stable variability.

<https://postgrespro.com/docs/postgresql/16/logical-replication-row-filter.html>

Row Identification and Filtering

By default, the logical row ID is determined by the primary key columns:

```
α=> SELECT relreplident FROM pg_class WHERE relname = 'test';
```

```
relreplident
-----
d
(1 row)
```

- d – default (primary key, if present)
- i – index
- f – all columns
- n – none

Let's say we need to replicate only rows where the cluster_name field has the value alpha. The cluster_name column is used in the filter expression, but it's not published yet. Publish all the columns, without listing any specific ones.

```
α=> ALTER PUBLICATION test_pub SET TABLE test WHERE (cluster_name = 'alpha');
```

ALTER PUBLICATION

```
α=> \dRp+
```

```

          Publication test_pub
 Owner | All tables | Inserts | Updates | Deletes | Truncates | Via root
-----+-----+-----+-----+-----+-----+-----
student | f          | t       | t       | t       | t       | f
Tables:
"public.test" WHERE (cluster_name = 'alpha'::text)
```

Observe how change replication works.

```
α=> INSERT INTO test(descr) VALUES ('Three');
```

INSERT 0 1

```
α=> INSERT INTO test(descr, cluster_name) VALUES ('Four', 'gamma');
```

INSERT 0 1

```
α=> SELECT * FROM test;
```

```
id | descr | cluster_name
---+-----+-----
1  | One   | alpha
2  | Two   | alpha
3  | Three | alpha
4  | Four  | gamma
(4 rows)
```

```
| β=> SELECT * FROM test;
```

```
| id | descr | cluster_name | additional
| ---+-----+-----+-----
| 1  | One   | alpha       |
| 2  | Two   | beta        |
| 3  | Three | alpha       |
| (3 rows)
```

The row with the value gamma was not replicated.

Let's try to modify the row.

```
α=> UPDATE test SET cluster_name = 'delta' WHERE id = 2;
```

ERROR: cannot update table "test"

DETAIL: Column used in the publication WHERE expression is not part of the replica identity.

For upgrade and delete, the filter expression accepts only columns contained in the logical row ID, but for insert, all columns are good.

Let's change the filter condition.

```
α=> ALTER PUBLICATION test_pub SET TABLE test WHERE (id > 0);
```

ALTER PUBLICATION

Now modify the row:

```
α=> UPDATE test SET cluster_name = 'delta' WHERE id = 2;
```

UPDATE 1

```
| β=> SELECT * FROM test WHERE id = 2;
|
|  id | descr | cluster_name | additional
|-----+-----+-----+-----
|  2 | Two   | delta        |
| (1 row)
```

The row change has been replicated.

The new filter condition works:

```
α=> INSERT INTO test(id, descr) VALUES (0, 'Zero');
```

INSERT 0 1

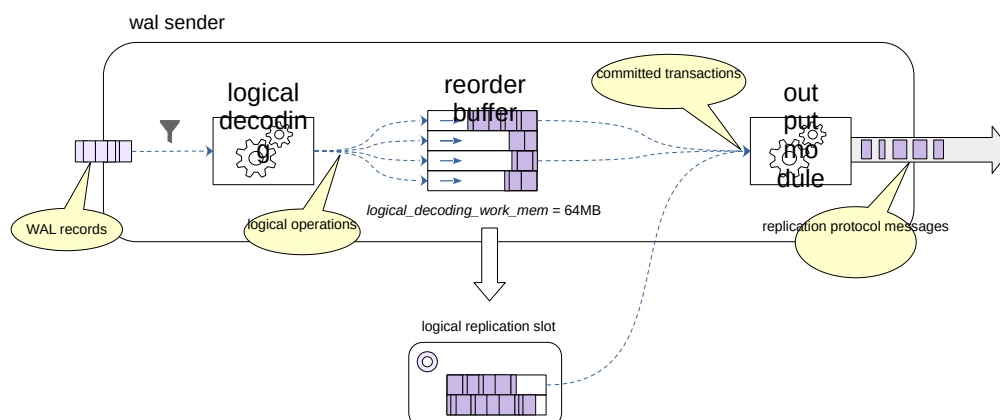
```
α=> SELECT * FROM test ORDER BY id;
```

```
id | descr | cluster_name
---+-----+-----
0 | Zero  | alpha
1 | One   | alpha
2 | Two   | delta
3 | Three | alpha
4 | Four  | gamma
(5 rows)
```

```
| β=> SELECT * FROM test ORDER BY id;
|
|  id | descr | cluster_name | additional
|-----+-----+-----+-----
|  1 | One   | alpha        |
|  2 | Two   | delta        |
|  3 | Three | alpha        |
| (3 rows)
```

The row with id = 0 has not been published, so it wasn't replicated.

Logical Decoding



15

During streaming physical replication, the wal sender process reads WAL records and passes them to the replica unchanged.

During logical replication, records are converted. *The logical decoding* module selects records that correspond to the published operations and tables, taking into account the columns and filter conditions. Selected records are converted to logical operations information, which is sent to the *reorder buffer* and stored there in accordance with the transaction IDs.

At the logical level, WAL additionally records the information required for logical decoding, in particular:

- for update – new values of all columns, not just changed ones,
- for update and delete – the old values of the columns in the logical ID,
- for commit – database OID.

The reorder buffer is in the local memory of the wal sender process. If the amount of data in it approaches the value of the `logical_decoding_work_mem` parameter, the largest transaction is sent to the replication slot (a file in `PGDATA/pg_replslots/name`).

When a transaction is committed, all its operations are sent to the *output module*, which converts them into replication protocol messages.

<https://postgrespro.com/docs/postgresql/16/logicaldecoding>

<https://postgrespro.com/docs/postgresql/16/runtime-config-resource#GUC-LOGICAL-DECODING-WORK-MEM>

Slots and Logical Decoding

Let's clear the table and check out logical decoding.

```
α=> TRUNCATE test;
```

```
TRUNCATE TABLE
```

```
| β=> SELECT * FROM test ORDER BY id;
|
|  id | descr | cluster_name | additional
|-----+-----+-----+-----
| (0 rows)
```

First, manually create a logical replication slot. The pgoutput plugin is used to send changes to the subscriber, but for observation purposes, the test_decoding plugin is more convenient:

```
α=> SELECT pg_create_logical_replication_slot('test_slot', 'test_decoding');
```

```
pg_create_logical_replication_slot
-----
(test_slot,0/12D2C990)
(1 row)
```

Now there are two slots on the alpha server:

```
α=> SELECT slot_name, plugin, slot_type, active FROM pg_replication_slots;
```

```
slot_name | plugin      | slot_type | active
-----+-----+-----+-----
test_sub  | pgoutput    | logical   | t
test_slot | test_decoding | logical   | f
(2 rows)
```

In a separate session, start a transaction and insert a row into the table:

```
student$ psql -d replica_logical
```

```
|| α=> BEGIN;
||
|| BEGIN
||
|| α=> INSERT INTO test(descr) VALUES ('Πέντε');
||
|| INSERT 0 1
```

The output plugin can request changes from the slot:

```
|| α=> SELECT * FROM pg_logical_slot_get_changes('test_slot', NULL, NULL);
||
|| lsn | xid | data
||-----+-----+-----
|| (0 rows)
```

The transaction isn't complete yet, so the output plugin hasn't received anything. Let's make another change and commit the transaction.

```
|| α=> UPDATE test SET descr = 'Five' WHERE id = 5;
||
|| UPDATE 1
||
|| α=> UPDATE test SET id = 1, descr='One' WHERE id = 5;
||
|| UPDATE 1
||
|| α=> COMMIT;
||
|| COMMIT
||
|| α=> SELECT * FROM pg_logical_slot_get_changes('test_slot', NULL, NULL) \gx (xheader_width=90)
```

```

-[ RECORD 1 ]-----
lsn | 0/12D44FA8
xid | 766
data | BEGIN 766
-[ RECORD 2 ]-----
lsn | 0/12D44FA8
xid | 766
data | table public.test: INSERT: id[integer]:5 descr[text]:'Πέντε'
cluster_name[text]:'alpha'
-[ RECORD 3 ]-----
lsn | 0/12D4CC78
xid | 766
data | table public.test: UPDATE: id[integer]:5 descr[text]:'Five'
cluster_name[text]:'alpha'
-[ RECORD 4 ]-----
lsn | 0/12D4EEE8
xid | 766
data | table public.test: UPDATE: old-key: id[integer]:5 new-tuple: id[integer]:1
descr[text]:'One' cluster_name[text]:'alpha'
-[ RECORD 5 ]-----
lsn | 0/12D51810
xid | 766
data | COMMIT 766

```

Now the output plugin has received the changes. The subscriber collects the published tables. When the logical ID changes, its old value is sent too.

Let's remove the test slot, otherwise it will prevent WAL file segments from being deleted.

```

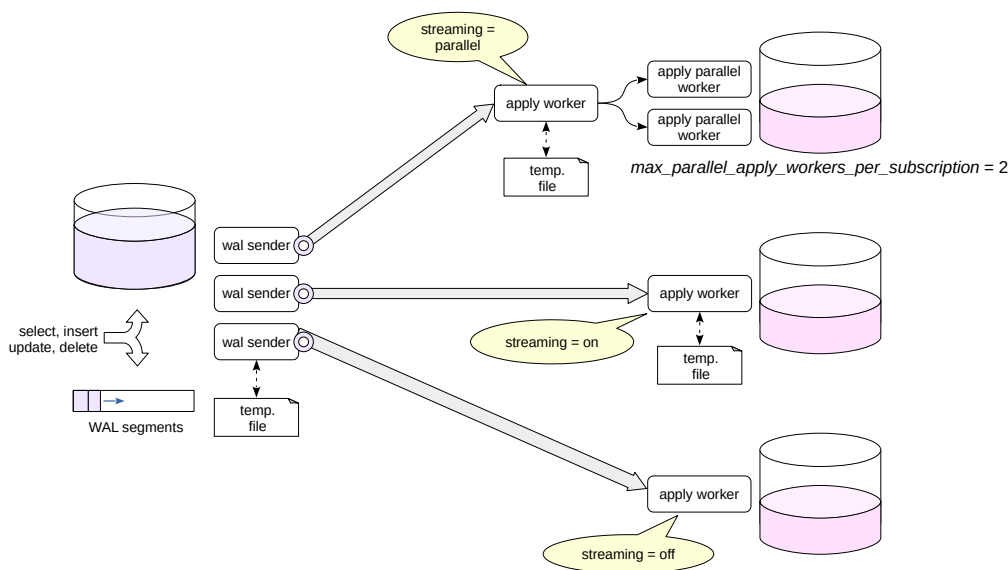
α=> SELECT pg_drop_replication_slot('test_slot');

pg_drop_replication_slot
-----
(1 row)

α=> \q

```

Large Transactions



17

By default (subscription streaming = off), changes are accumulated in the reorder buffer and sent to the subscriber after the transaction is committed.

If a transaction changes a lot of data, problems can occur:

- wal sender's reorder buffer memory costs,
- writing files when the reorder buffer exceeds (`logical_decoding_work_mem`) memory,
- replica lag.

If you set streaming = on, the publishing server will pass data changes to the subscriber while the transaction is still running.

When the total amount of changes exceeds the value of the `logical_decoding_work_mem` parameter, the accumulated changes of the largest transaction in terms of volume are not dumped into a file, but are immediately sent to the output module.

The subscriber saves these changes in its temporary files and applies them when committing the transaction.

Starting with PostgreSQL 16, if streaming = parallel and there are workers available, changes are not saved at the subscriber, but are immediately applied by the free workers. The number of such processes per subscription is limited by the `max_parallel_apply_workers_per_subscription` parameter.

<https://postgrespro.com/docs/postgresql/16/sql-createsubscription#SQL-CREATE-SUBSCRIPTION-WITH-STREAMING>

<https://postgrespro.com/docs/postgresql/16/runtime-config-replication#GUC-MAX-PARALLEL-APPLY-WORKERS-PER-SUBSCRIPTION>

Lead to errors

- constraint violations,
- insufficient rights to target table or row level security
- replication error,
- subscription with `disable_on_error` goes down
- you can manually correct subscription-side data
- or skip the conflicting transaction

Ignored

- row being updated or deleted does not exist at the subscriber
- replication continues, but consistency may be compromised

Because the tables on the publishing server and at the subscriber can change independently of each other, a subscription operation may *conflict* with existing data.

Often a conflict is a situation where an operation cannot be performed, usually due to a violation of an integrity constraint, or the lack of permissions on the target table, or row level security restrictions.

In this case, the operation fails. The subscriber periodically repeats attempts, but they will fail until the conflict is resolved. Conflicts can only be resolved manually, automatic resolution has not yet been implemented. There are two options: fix the data at the subscriber or skip the conflicting transaction. In the second case, you need to disable replication, call the `pg_replication_origin_advance` function or execute the `ALTER SUBSCRIPTION ... SKIP...`, and enable replication again. You can also pre-configure automatic disabling of replication by specifying the `disable_on_error` subscription option.

If the row to be modified or deleted does not exist at the subscriber, the operation will fail. In this case, no error occurs, but such ignored conflicts indicate that the data consistency between the publishing server and the subscriber is compromised.

<https://postgrespro.com/docs/postgresql/16/logical-replication-conflicts>

Conflicts

Note that sequences are not replicated. The second server created its own sequence:

```
β=> INSERT INTO test(descr) VALUES ('Six - local');  
ERROR:  duplicate key value violates unique constraint "test_pkey"  
DETAIL:  Key (id)=(1) already exists.
```

And this will work:

```
β=> INSERT INTO test VALUES (6, 'Six - local');  
INSERT 0 1
```

What will happen if a value with the same key (6) appears on the publishing server?

```
α=> INSERT INTO test(descr) VALUES ('Six');  
INSERT 0 1  
α=> INSERT INTO test(descr) VALUES ('Seven');  
INSERT 0 1
```

A conflict will occur during replication, and the replication will be paused.

```
β=> SELECT * FROM test ORDER BY id;  
  
 id | descr      | cluster_name | additional  
----+-----+-----+-----  
  1 | One        | alpha        |  
  6 | Six - local | beta         |  
(2 rows)
```

Technically, the logical replication apply worker process will continuously restart and check if the conflict is resolved. This resets the `pg_stat_subscription` information:

```
β=> SELECT * FROM pg_stat_subscription \gx  
  
-[ RECORD 1 ]-----+-----  
subid          | 16419  
subname         | test_sub  
pid            |  
leader_pid     |  
reliid         |  
received_lsn   |  
last_msg_send_time |  
last_msg_receipt_time |  
latest_end_lsn |  
latest_end_time |
```

The log will contain records about constraint violations:

```
student$ sudo tail -n 3 /var/log/postgresql/postgresql-16-beta.log
```

```
2026-09-17 00:03:37.210 MSK [85757] DETAIL:  Key (id)=(6) already exists.  
2026-09-17 00:03:37.210 MSK [85757] CONTEXT:  processing remote data for replication  
origin "pg_16419" during message type "INSERT" for replication target relation  
"public.test" in transaction 767, finished at 0/12D6E280  
2026-09-17 00:03:37.224 MSK [82278] LOG:  background worker "logical replication worker"  
(PID 85757) exited with exit code 1
```

And the error counter in `pg_stat_subscription_stats` will increment with each restart:

```
β=> SELECT * FROM pg_stat_subscription_stats;  
  
 subid | subname | apply_error_count | sync_error_count | stats_reset  
-----+-----+-----+-----+-----  
 16419 | test_sub |                2 |                0 |  
(1 row)
```

(PostgreSQL 18 has introduced separate counters for each conflict type.)

To resolve this conflict, delete the conflicting row from the table and wait a moment:

```
β=> DELETE FROM test WHERE id = 6;
```

```
| DELETE 1
```

Check:

```
| β=> SELECT * FROM test ORDER BY id;

 id | descr | cluster_name | additional
-----+-----+-----+-----
  1 | One   | alpha        |
  6 | Six   | alpha        |
  7 | Seven | alpha        |
(3 rows)
```

The data has appeared, replication has been restored.

Skipping a Conflicting Transaction

Another option is to skip the conflicting transaction entirely.

The conflicting transaction:

```
| β=> INSERT INTO test(id,descr) VALUES (8,'Eight - local');
| INSERT 0 1

α=> INSERT INTO test(descr) VALUES ('Eight');
INSERT 0 1
```

The subscription has to be stopped before skipping the transaction:

```
| β=> ALTER SUBSCRIPTION test_sub DISABLE;
| ALTER SUBSCRIPTION
```

Instead of stopping manually, you can set the `disable_on_error` subscription parameter.

Subscription stopped:

```
| β=> \dRs

          List of subscriptions
   Name   | Owner | Enabled | Publication
-----+-----+-----+-----
 test_sub | student | f       | {test_pub}
(1 row)
```

Check the log:

```
| β=> \! tail -4 /var/log/postgresql/postgresql-16-beta.log

2026-09-17 00:03:43.408 MSK [86237] ERROR:  duplicate key value violates unique
constraint "test_pkey"
2026-09-17 00:03:43.408 MSK [86237] DETAIL:  Key (id)=(8) already exists.
2026-09-17 00:03:43.408 MSK [86237] CONTEXT:  processing remote data for replication
origin "pg_16419" during message type "INSERT" for replication target relation
"public.test" in transaction 771, finished at 0/12E27298
2026-09-17 00:03:43.421 MSK [82278] LOG:  background worker "logical replication worker"
(PID 86237) exited with exit code 1
```

The message contains the LSN of the conflicting transaction commit record.

Skip the transaction:

```
| β=> ALTER SUBSCRIPTION test_sub SKIP (lsn = '0/12E27298');
| ALTER SUBSCRIPTION
```

A superuser could use the function `pg_replication_origin_advance` instead.

Let's enable replication.

```
| β=> ALTER SUBSCRIPTION test_sub ENABLE;
| ALTER SUBSCRIPTION

β=> SELECT * FROM pg_stat_subscription \gx
```

```

-[ RECORD 1 ]-----+-----
subid          | 16419
subname        | test_sub
pid            | 86738
leader_pid     |
relid          |
received_lsn   | 0/12E27010
last_msg_send_time | 2026-09-17 00:03:47.222544+03
last_msg_receipt_time | 2026-09-17 00:03:47.224331+03
latest_end_lsn | 0/12E27010
latest_end_time | 2026-09-17 00:03:47.222544+03

```

Replication is working, but since we skipped the transaction, the data is now inconsistent:

```
α=> SELECT * FROM test ORDER BY id;
```

```

id | descr | cluster_name
-----+-----
1  | One   | alpha
6  | Six   | alpha
7  | Seven | alpha
8  | Eight | alpha
(4 rows)

```

```
β=> SELECT * FROM test ORDER BY id;
```

```

id | descr      | cluster_name | additional
-----+-----+-----
1  | One        | alpha        |
6  | Six        | alpha        |
7  | Seven      | alpha        |
8  | Eight - local | beta         |
(4 rows)

```

Clear the table:

```
α=> TRUNCATE test;
```

TRUNCATE TABLE

Triggers on the Subscriber

Triggers can execute on the subscriber, but by default a trigger will fire on local changes only and never on replication. This is useful when both servers have identical tables with the same set of triggers – in that case, the trigger has already fired on the publishing server and doesn't need to run again on the subscriber.

Let's try it.

```

β=> CREATE OR REPLACE FUNCTION change_descr() RETURNS trigger AS $$
BEGIN
    NEW.additional := 'from the publication';
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

```

```
CREATE FUNCTION
```

```

β=> CREATE TRIGGER test_before_row
BEFORE INSERT OR UPDATE ON test
FOR EACH ROW
EXECUTE FUNCTION change_descr();

```

```
CREATE TRIGGER
```

```
α=> INSERT INTO test(descr) VALUES ('Nine');
```

INSERT 0 1

```
β=> SELECT * FROM test ORDER BY id;
```

```

id | descr | cluster_name | additional
-----+-----+-----
9  | Nine  | alpha        |
(1 row)

```

The trigger didn't fire.

You can alter the table so the trigger fires only during replication:

```
| β=> ALTER TABLE test ENABLE REPLICA TRIGGER test_before_row;
```

```
| ALTER TABLE
```

Or in both cases: during replication and during local changes.

```
| β=> ALTER TABLE test ENABLE ALWAYS TRIGGER test_before_row;
```

```
| ALTER TABLE
```

You can distinguish between these situations using the session_replication_role parameter:

```
| β=> CREATE OR REPLACE FUNCTION change_descr() RETURNS trigger AS $$  
BEGIN  
  NEW.additional := current_setting('session_replication_role');  
  RETURN NEW;  
END;  
$$ LANGUAGE plpgsql;
```

```
| CREATE FUNCTION
```

Add one row on each server:

```
α=> INSERT INTO test(descr) VALUES ('Ten');
```

```
INSERT 0 1
```

```
| β=> INSERT INTO test(id, descr) VALUES (11, 'Eleven');
```

```
| INSERT 0 1
```

```
| β=> SELECT * FROM test ORDER BY id;
```

```
|  id | descr | cluster_name | additional  
|----+-----+-----+-----  
|  9 | Nine  | alpha        |  
| 10 | Ten   | alpha        | replica  
| 11 | Eleven| beta         | origin  
| (3 rows)
```

The trigger fired in both cases.



<https://postgrespro.com/docs/postgresql/16/view-pg-replication-origin-status#VIEW-PG-REPLICATION-ORIGIN-STATUS>

Replication origin

Delete rows from the table:

```
α=> TRUNCATE test;
```

```
TRUNCATE TABLE
```

Publish several columns from table test on server beta.

```
| β=> CREATE PUBLICATION test_pub FOR TABLE test(id, descr, cluster_name);  
| CREATE PUBLICATION
```

Now the publications exist on both servers. If you create a subscription of alpha, it will receive all changes from beta, including those sent by alpha's publication. But alpha cannot apply these changes because of integrity constraints, so replication will stop.

For beta to receive just the local changes from alpha, set the origin subscription parameter:

```
| β=> ALTER SUBSCRIPTION test_sub SET (origin = none);  
| ALTER SUBSCRIPTION
```

For the changes from replication, the origin will be non-empty, and the publication will not send them to the subscriber.

Now let's make alpha a subscriber to beta. Disable initial synchronization, as the table is empty. Use the subscription parameter origin to receive only the changes made on beta:

```
α=> CREATE SUBSCRIPTION test_sub  
CONNECTION 'host=localhost port=5433 user=student password=student dbname=replica_logical'  
PUBLICATION test_pub WITH (copy_data = false, origin = none);
```

```
NOTICE: created replication slot "test_sub" on publisher  
CREATE SUBSCRIPTION
```

Now there are subscriptions on both servers.

```
α=> \dRs
```

```
          List of subscriptions  
   Name | Owner | Enabled | Publication  
-----+-----+-----+-----  
test_sub | student | t      | {test_pub}  
(1 row)
```

```
| β=> \dRs
```

```
|          List of subscriptions  
|   Name | Owner | Enabled | Publication  
|-----+-----+-----+-----  
|test_sub | student | t      | {test_pub}  
|(1 row)
```

Insert a row on alpha.

```
α=> INSERT INTO test VALUES (1, 'One potato');
```

```
INSERT 0 1
```

Check beta.

```
| β=> SELECT * FROM test;  
|  
| id | descr | cluster_name | additional  
|----+-----+-----+-----  
| 1 | One potato | alpha | replica  
|(1 row)
```

Insert a row on beta.

```
| β=> INSERT INTO test VALUES (2, 'Two potato');
```

```
| INSERT 0 1
```

Check alpha.

```
α=> SELECT * FROM test;
```

id	descr	cluster_name
1	One potato	alpha
2	Two potato	beta

(2 rows)

Verify the changes.

$\alpha \Rightarrow$ **UPDATE** test **SET** descr = 'Three potato, four';

UPDATE 2

$\alpha \Rightarrow$ **SELECT** * **FROM** test;

id	descr	cluster_name
1	Three potato, four	alpha
2	Three potato, four	beta

(2 rows)

$\beta \Rightarrow$ **SELECT** * **FROM** test;

id	descr	cluster_name	additional
1	Three potato, four	alpha	replica
2	Three potato, four	beta	replica

(2 rows)

If replication is no longer needed, the subscription must be dropped – otherwise, the replication slots will remain open on the publishing server.

$\alpha \Rightarrow$ **DROP SUBSCRIPTION** test_sub;

NOTICE: dropped replication slot "test_sub" on publisher
DROP SUBSCRIPTION

$\beta \Rightarrow$ **DROP SUBSCRIPTION** test_sub;

NOTICE: dropped replication slot "test_sub" on publisher
DROP SUBSCRIPTION

Logical replication uses the publish-subscribe pattern.

Row-level changes are replicated

Rows are matched using logical identifiers

Data mismatch on different servers may cause conflicts

Origin information forwarding helps track the progress of cascade replication and eliminate loops

1. Create two databases on a single server.

In the first database, create a table with a primary key and insert a few rows into it. Set up logical replication for the table from the first database to the second, verify that replication is working, and then remove the subscription.

2. Configure logical replication of the table from alpha to beta. Execute a transaction that modifies a large amount of data, and before committing it, verify that the data has been written to disk. Repeat the same steps, but this time in streaming mode. Did the data get to disk?

23

1. Use the `pg_dump` tool with the `--schema-only` option.

If you try the regular way, the subscription creation command will hang because it needs to wait for all active transactions on the publishing server to complete, including itself.

In this case, you must create a logical replication slot in advance, as described in the documentation:

<https://postgrespro.com/docs/postgresql/16/sql-createsubscription>

2. To prevent the publication from getting into the logical copy, use the `--no-publications` option.

Check the value of the `logical_decoding_work_mem` configuration parameter, which sets the maximum size of the reorder buffer. If this value is exceeded, data will be pushed to disk.

1. Logical Replication on a Single Server

Set the WAL level to logical first.

```
α=> ALTER SYSTEM SET wal_level = logical;
```

```
ALTER SYSTEM
```

```
student$ sudo pg_ctlcluster 16 alpha restart
```

Databases:

```
student$ psql
```

```
α=> CREATE DATABASE replica_logical;
```

```
CREATE DATABASE
```

```
α=> CREATE DATABASE replica_logical_bis;
```

```
CREATE DATABASE
```

Table in the first database:

```
α=> \c replica_logical
```

You are now connected to database "replica_logical" as user "student".

```
α=> CREATE TABLE test(  
    id integer PRIMARY KEY GENERATED BY DEFAULT AS IDENTITY,  
    descr text  
);
```

```
CREATE TABLE
```

```
α=> INSERT INTO test(descr) VALUES ('One'), ('Two'), ('Three');
```

```
INSERT 0 3
```

Using logical backup to transfer a schema is especially convenient when there are many tables.

```
student$ pg_dump --no-acl --schema-only replica_logical | psql -d replica_logical_bis
```

```
SET  
SET  
SET  
SET  
SET  
set_config  
-----
```

```
(1 row)
```

```
SET  
SET  
SET  
SET  
SET  
SET  
CREATE TABLE  
ALTER TABLE  
ALTER TABLE  
ALTER TABLE
```

Transferring data with `pg_dump` makes no sense here, since the tables may change during the transfer.

```
student$ psql -d replica_logical_bis
```

```
| α=> \d test
```

```
|      Table "public.test"  
|  Column | Type   | Collation | Nullable | Default  
|-----+-----+-----+-----+-----  
| id      | integer |           | not null | generated by default as identity  
| descr   | text    |           |          |  
| Indexes:  
|    "test_pkey" PRIMARY KEY, btree (id)
```

Publication:

```
α=> CREATE PUBLICATION test_pub FOR TABLE test;
```

```
CREATE PUBLICATION
```

Since replication will be configured on the same server, a logical replication slot must be created first.

```
α=> SELECT pg_create_logical_replication_slot('testslot','pgoutput');
```

```
pg_create_logical_replication_slot
-----
(testslot,0/23EEC58)
(1 row)
```

Then create the subscription:

```
α=> CREATE SUBSCRIPTION test_sub
CONNECTION 'user=student password=student dbname=replica_logical host=localhost'
PUBLICATION test_pub
WITH (create_slot = false, slot_name = testslot);
```

```
CREATE SUBSCRIPTION
```

Verify that replication is working:

```
α=> INSERT INTO test(descr) VALUES ('Four');
```

```
INSERT 0 1
```

```
α=> SELECT * FROM test;
```

```
 id | descr
----+-----
  1 | One
  2 | Two
  3 | Three
  4 | Four
(4 rows)
```

Drop the subscription and the slot.

```
α=> DROP SUBSCRIPTION test_sub;
```

```
NOTICE:  dropped replication slot "testslot" on publisher
DROP SUBSCRIPTION
```

```
α=> \q
```

2. Logical Replication from Alpha to Beta

Start server beta and copy the DDL commands for the test table. Publication creation commands are not required on the subscriber.

```
student$ sudo pg_ctlcluster 16 beta start
```

```
student$ pg_dump \
--create \
--schema-only \
--no-publications \
--no-acl \
--dbname=replica_logical \
| psql -p 5433
```

```
SET
SET
SET
SET
SET
set_config
-----
```

```
(1 row)
```

```
SET
SET
SET
SET
CREATE DATABASE
ALTER DATABASE
You are now connected to database "replica_logical" as user "student".
SET
SET
```

```
SET
SET
SET
set_config
-----
```

(1 row)

```
SET
SET
SET
SET
SET
SET
SET
CREATE TABLE
ALTER TABLE
ALTER TABLE
ALTER TABLE
```

Create a subscription on beta – the slot on alpha is created automatically.

```
student$ psql -p 5433 -d replica_logical
```

```
β=> CREATE SUBSCRIPTION test_sub
CONNECTION 'user=student password=student dbname=replica_logical host=localhost'
PUBLICATION test_pub;
```

```
NOTICE: created replication slot "test_sub" on publisher
CREATE SUBSCRIPTION
```

```
β=> SELECT count(*) FROM test;
```

```
count
-----
      4
(1 row)
```

Replication is working.

Truncate the test table on the publication server (alpha):

```
α=> TRUNCATE test;
```

```
TRUNCATE TABLE
```

```
β=> SELECT count(*) FROM test;
```

```
count
-----
      0
(1 row)
```

The subscriber received the changes.

Start a transaction that inserts a large number of rows into the table.

```
α=> BEGIN;
```

```
BEGIN
```

```
α=> INSERT INTO test(descr)
SELECT now()::text FROM generate_series(1,1_000_000) AS g(id);
```

```
INSERT 0 1000000
```

The logical_decoding_work_mem parameter defines the maximum size of the reordering buffer.

```
α=> SHOW logical_decoding_work_mem;
```

```
logical_decoding_work_mem
-----
64MB
(1 row)
```

When this limit is exceeded, data is spilled to disk in the PGDATA/pg_replslot/slot-name directory. How much disk space does the directory take up?

```
postgres$ du -sh /var/lib/postgresql/16/alpha/pg_replslot/test_sub
```

```
135M    /var/lib/postgresql/16/alpha/pg_replslot/test_sub
```

By default, no data is sent to the subscriber until the transaction is committed.

```
α=> SELECT count(*) FROM test;

 count
-----
      0
(1 row)
```

Commit the transaction...

```
α=> COMMIT;
```

COMMIT

...and wait a moment:

The subscriber received all the rows:

```
β=> SELECT count(*) FROM test;

 count
-----
10000000
(1 row)
```

And the data spilled to disk has been removed:

```
postgres$ du -sh /var/lib/postgresql/16/alpha/pg_replslot/test_sub
```

```
8.0K    /var/lib/postgresql/16/alpha/pg_replslot/test_sub
```

Check pg_stat_replication_slots:

```
α=> SELECT * FROM pg_stat_replication_slots \gx
```

```
-[ RECORD 1 ]+-----
slot_name    | test_sub
spill_txns   | 1
spill_count  | 3
spill_bytes  | 162000000
stream_txns  | 0
stream_count | 0
stream_bytes | 0
total_txns   | 2
total_bytes  | 162001984
stats_reset  |
```

- spill_txns – the number of transactions spilled to disk
- spill_count – the number of times data was spilled to disk due to exceeding logical_decoding_work_mem
- spill_bytes – the number of bytes spilled to disk

Repeat the experiment in streaming mode.

Reset the statistics and truncate the table again.

```
α=> SELECT pg_stat_reset_replication_slot('test_sub');
```

```
pg_stat_reset_replication_slot
-----
(1 row)
```

```
α=> TRUNCATE test;
```

TRUNCATE TABLE

Now enable streaming mode for the subscription:

```
β=> ALTER SUBSCRIPTION test_sub SET (streaming = on);

ALTER SUBSCRIPTION
```

Run the large transaction again.

```
α=> BEGIN;
```

BEGIN

```
α=> INSERT INTO test(descr)
SELECT now()::text FROM generate_series(1,1_000_000) AS g(id);
```

```
INSERT 0 1000000
```

Now there's much less data in the slot's directory – some of it has already been sent to the subscriber:

```
postgres$ du -sh /var/lib/postgresql/16/alpha/pg_replslot/test_sub
```

```
8.0K    /var/lib/postgresql/16/alpha/pg_replslot/test_sub
```

Commit the transaction...

```
α=> COMMIT;
```

```
COMMIT
```

...and wait a moment:

```
| β=> SELECT count(*) FROM test;
```

```
|      count  
|      -----  
|     1000000  
|     (1 row)
```

Check pg_stat_replication_slots:

```
α=> SELECT * FROM pg_stat_replication_slots \gx
```

```
-[ RECORD 1 ]+-----  
slot_name    | test_sub  
spill_txns   | 0  
spill_count  | 0  
spill_bytes  | 0  
stream_txns  | 1  
stream_count | 3  
stream_bytes | 161000000  
total_txns   | 2  
total_bytes  | 161001984  
stats_reset  | 2026-09-17 00:17:52.013907+03
```

This time, when the allocated memory was exhausted, data was not spilled to disk but sent directly to the subscriber.

Cleanup

Drop the subscription, slot, and databases, and reset configuration.

```
| β=> DROP SUBSCRIPTION test_sub;
```

```
| NOTICE: dropped replication slot "test_sub" on publisher  
| DROP SUBSCRIPTION
```

```
| β=> \c postgres
```

```
| You are now connected to database "postgres" as user "student".
```

```
| β=> DROP DATABASE replica_logical;
```

```
| DROP DATABASE
```

```
student$ sudo pg_ctlcluster 16 beta stop
```

```
α=> \c postgres
```

You are now connected to database "postgres" as user "student".

```
α=> DROP DATABASE replica_logical;
```

```
DROP DATABASE
```

```
α=> DROP DATABASE replica_logical_bis;
```

```
DROP DATABASE
```

```
α=> ALTER SYSTEM RESET wal_level;
```

```
ALTER SYSTEM
```

```
student$ sudo pg_ctlcluster 16 alpha restart
```

1. Create and publish a table with a bytea column on alpha.
Insert several million rows into the table. Compare the initial subscription synchronization time in normal and binary modes using the beta server.
2. Enable concurrent synchronization for the subscription. By adding rows to the table, examine the subscription workers.

1. You can estimate the initial synchronization time by monitoring the `pg_subscription_rel` table. You can set it up on the client side using the `\watch` metacommand or on the server side using a function written in, for example, PL/pgSQL.

<https://postgrespro.com/docs/postgresql/16/catalog-pg-subscription-rel#CATALOG-PG-SUBSCRIPTION-REL>

<https://postgrespro.ru/docs/postgresql/16/plpgsql>

2. Set an appropriate value for the streaming subscription option. Use the `pg_stat_subscription` view to monitor subscription processes.

1. Initial Data Synchronization for Logical Replication

Create a database.

```
α=> CREATE DATABASE replica_logical;
```

```
CREATE DATABASE
```

```
α=> \c replica_logical
```

You are now connected to database "replica_logical" as user "student".

Change the WAL level and restart the server.

```
α=> ALTER SYSTEM SET wal_level = logical;
```

```
ALTER SYSTEM
```

```
student$ sudo pg_ctlcluster 16 alpha restart
```

```
student$ psql -d replica_logical
```

Create a table with a large number of rows.

```
α=> CREATE TABLE t AS  
SELECT n::text::bytea  
FROM generate_series(1, 5_000_000) AS g(n);
```

```
SELECT 5000000
```

Start the beta server and copy the database to it: with the table structure but without any data.

```
student$ sudo pg_ctlcluster 16 beta start
```

```
student$ pg_dump -d replica_logical --schema-only --create --no-acl | psql -p 5433
```

```
SET  
SET  
SET  
SET  
SET  
set_config  
-----
```

```
(1 row)
```

```
SET  
SET  
SET  
SET  
CREATE DATABASE  
ALTER DATABASE
```

You are now connected to database "replica_logical" as user "student".

```
SET  
SET  
SET  
SET  
SET  
set_config  
-----
```

```
(1 row)
```

```
SET  
SET  
SET  
SET  
SET  
SET  
CREATE TABLE  
ALTER TABLE
```

```
student$ psql -p 5433 -d replica_logical
```

Create a publication on alpha.

```
α=> CREATE PUBLICATION pub FOR TABLE t;
```

```
CREATE PUBLICATION
```

The subscription on beta will initially be inactive.

```
β=> CREATE SUBSCRIPTION sub
CONNECTION 'dbname=replica_logical host=localhost port=5432 user=postgres password=postgres'
PUBLICATION pub
WITH (enabled = false);

NOTICE: created replication slot "sub" on publisher
CREATE SUBSCRIPTION
```

The table's status in the subscription is displayed in the following view:

```
β=> SELECT srsubid, srrelid::regclass, srsubstate, srsublsn
FROM pg_subscription_rel;

 srsubid | srrelid | srsubstate | srsublsn
-----+-----+-----+-----
    16397 | t      | i          |
(1 row)
```

Status codes in the srsubstate field:

- i – initialization
- d – data copy
- f – table copy complete
- s – synchronization complete
- r – normal replication

We'll measure the synchronization time using a PL/pgSQL function:

```
β=> CREATE OR REPLACE FUNCTION time4sync() RETURNS interval AS
$$
DECLARE
    st timestamp := clock_timestamp();
BEGIN
    WHILE EXISTS(SELECT 1 FROM pg_subscription_rel WHERE srsubstate <> 'r') LOOP
        PERFORM pg_sleep(0.01); -- достаточная точность
    END LOOP;
    RETURN clock_timestamp() - st;
END;
$$
LANGUAGE plpgsql;

CREATE FUNCTION
```

Activate the subscription and measure the initial synchronization time.

```
β=> ALTER SUBSCRIPTION sub ENABLE; SELECT time4sync();

ALTER SUBSCRIPTION
    time4sync
-----
00:00:09.000456
(1 row)
```

Now let's configure synchronization in binary mode. Drop the subscription, truncate the table, and create a new subscription with the binary parameter.

```
β=> DROP SUBSCRIPTION sub;

NOTICE: dropped replication slot "sub" on publisher
DROP SUBSCRIPTION

β=> TRUNCATE TABLE t;

TRUNCATE TABLE

β=> CREATE SUBSCRIPTION sub
CONNECTION 'dbname=replica_logical host=localhost port=5432 user=postgres password=postgres'
PUBLICATION pub
WITH (enabled = false, binary = true);

NOTICE: created replication slot "sub" on publisher
CREATE SUBSCRIPTION
```

Enable the subscription and measure the synchronization time.

```
β=> ALTER SUBSCRIPTION sub ENABLE; SELECT time4sync();
```

```
ALTER SUBSCRIPTION
    time4sync
-----
00:00:07.282319
(1 row)
```

The initial synchronization is noticeably faster.

The binary parameter also applies to subsequent synchronization. However, binary mode is not supported for all data types; portability across different architectures and major PostgreSQL versions must also be considered. Binary mode was introduced in PostgreSQL 14, and in versions prior to 16, initial synchronization always runs in text mode.

2. Parallel Application of Changes

Enable parallel mode for the subscription.

```
β=> ALTER SUBSCRIPTION sub SET (streaming = parallel);
```

```
ALTER SUBSCRIPTION
```

Processes serving the subscription:

```
β=> SELECT
    a.pid, a.leader_pid, a.backend_type,
    r.srsubstate, s.received_lsn
FROM pg_stat_subscription s
    LEFT JOIN pg_subscription_rel r ON r.srsubid=s.subid AND r.srrelid='t'::regclass
    LEFT JOIN pg_stat_activity a ON s.pid=a.pid
WHERE s.subname = 'sub';
```

pid	leader_pid	backend_type	srsubstate	received_lsn
114667		logical replication worker	r	

(1 row)

Truncate the table, insert a large number of rows, and re-run the query against the system catalog views:

```
α=> TRUNCATE t;
```

```
TRUNCATE TABLE
```

```
α=> INSERT INTO t SELECT g.i::text::bytea FROM generate_series(1, 1_000_000) AS g(i);
```

```
β=> SELECT
    a.pid, a.leader_pid, a.backend_type,
    r.srsubstate, s.received_lsn
FROM pg_stat_subscription s
    LEFT JOIN pg_subscription_rel r ON r.srsubid=s.subid AND r.srrelid='t'::regclass
    LEFT JOIN pg_stat_activity a ON s.pid=a.pid
WHERE s.subname = 'sub';
```

pid	leader_pid	backend_type	srsubstate	received_lsn
114667		logical replication worker	r	0/16FFFFE8
114896	114667	logical replication parallel worker	r	

(2 rows)

An additional process has appeared that applies changes without waiting for the transaction to complete on the publisher.

```
INSERT 0 1000000
```

Row insertion is complete. Wait a little longer...

```
β=> select count(*) from t;
```

```
count
-----
1000000
(1 row)
```

The subscriber has now received all the rows.

Cleanup

Delete the subscription.

```
β=> DROP SUBSCRIPTION sub;
```

```
NOTICE: dropped replication slot "sub" on publisher
DROP SUBSCRIPTION
```

Drop the databases.

```
β=> \c postgres postgres
```

```
You are now connected to database "postgres" as user "postgres".
```

```
β=> DROP DATABASE replica_logical;
```

```
DROP DATABASE
```

Reset the configuration.

```
α=> \c postgres postgres
```

```
You are now connected to database "postgres" as user "postgres".
```

```
α=> ALTER SYSTEM RESET synchronous_standby_names;
```

```
ALTER SYSTEM
```

```
α=> ALTER SYSTEM RESET wal_level;
```

```
ALTER SYSTEM
```

Restart the alpha server.

```
student$ sudo pg_ctlcluster 16 alpha restart
```

```
student$ psql
```

```
α=> DROP DATABASE replica_logical;
```

```
DROP DATABASE
```

Stop the beta server.

```
student$ sudo pg_ctlcluster 16 beta stop
```