

# Replication

## Overview of Logical Replication



### Copyright

© Postgres Professional, 2017–2025

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

Translated by: Liudmila Mantrova, Alexander Meleshko, Elena Sharafutdinova

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](mailto: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 Replication

WAL Levels

Publications and Subscriptions

Conflict Detection and Resolution

Replica Usage

## Publisher

- reads its own WAL records
- decodes them into table row changes
- sends to subscribers

## Subscriber

- receives WAL records from the replication stream
- applies the changes to its own tables

## Features

- publication-subscription: data flow is possible in both directions
- requires protocol-level compatibility
- can replicate individual tables

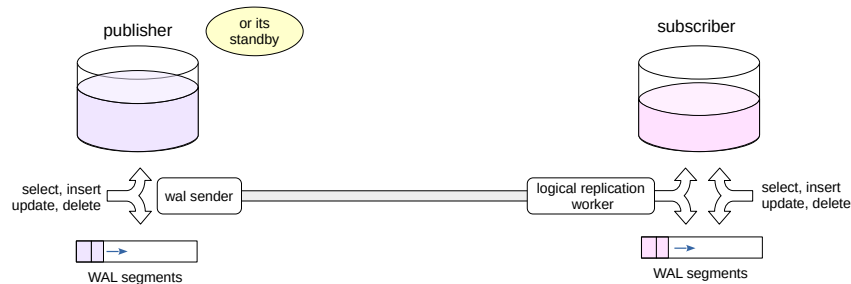
In physical replication only one server (the primary) makes changes and generates WAL records. Other servers (standbys) only read the primary's WAL records and apply them.

In logical replication all servers operate normally, can modify data, and generate their own WAL records. Any server can *publish* its changes, while others can *subscribe* to them. A single server can act as both a publisher and a subscriber, enabling flexible data flows between servers.

The publisher reads its own WAL records, but unlike physical replication, it first decodes them into a logical format that is platform-independent and not tied to specific PostgreSQL version before transmitting to subscribers. Therefore, binary compatibility is not required — only the replication protocol must be understood. It also supports selective replication, allowing only specific tables to be replicated.

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

# Logical Replication



On the publisher the `wal_sender` process generates WAL records reflecting changes to published data. On the subscriber the logical replication worker receives the information from the publisher and applies it.

The subscriber can receive changes either directly from the publisher or from the publisher's standby. In the latter case, the standby's `wal_sender` process sends changes to the subscriber. This architecture helps reduce load on the publisher.

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

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

## Publication

- includes one or more database tables
- can specify columns and row filters
- processes INSERT, UPDATE, DELETE, TRUNCATE commands
- sends row-level changes after transaction commit
- uses a logical replication slot

## Subscription

- receives and applies changes
- can do initial synchronization
- no parsing, rewriting and planning, just direct execution
- possible conflicts with local data

Logical replication uses a publish and subscribe model.

A *publication* is created on one server and can include several tables in a single database. Starting with version 15, PostgreSQL allows publishing partial table data by specifying column subsets and row filter conditions. Other servers can *subscribe* to this publication to receive and apply changes to the tables.

Only table row modifications are replicated (not SQL commands). DDL commands are not transmitted, so target tables on the subscriber must be created manually. Initial synchronization of table contents can be performed when a subscription is created.

After transaction commit, information about modified rows is extracted from WAL records on the publisher through *logical decoding*. The resulting messages are transmitted to subscribers via the replication protocol in a platform- and version-independent format.

Changes are applied without executing SQL commands, avoiding parsing and planning overhead. On the other hand, a single SQL command can result in multiple one-row changes.

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

# WAL Levels



*wal\_level* parameter

minimal	<	replica	<	logical
crash recovery		crash recovery restore from backup, replication		crash recovery restore from backup, replication logical replication

6

To enable the publisher to generate messages about changes at the row level, the WAL must include additional information, such as row identifiers involved in modifications and notifications about changes to table definitions and data types. This allows the subscriber to have up-to-date knowledge of the structure of replicated objects in any moment.

This extended level of logging is called **logical**. Since the default level is *replica*, logical replication requires changing the *wal\_level* parameter on the publisher.

<https://postgrespro.com/docs/postgresql/16/protocol-logical-replication#PROTOCOL-LOGICAL-MESSAGES-FLOW>

## Logical Replication

Assume the first server contains a table:

```
=> CREATE DATABASE replica_overview_logical_dba;
```

```
CREATE DATABASE
```

```
=> \c replica_overview_logical_dba
```

You are now connected to database "replica\_overview\_logical\_dba" as user "student".

```
=> CREATE TABLE test(id integer PRIMARY KEY, descr text);
```

```
CREATE TABLE
```

Let's clone the cluster by creating a standalone backup (as covered in the Physical Replication Overview lesson), but omit the -R option from pg\_basebackup call since we need an independent server, not a standby.

```
student$ rm -rf /home/student/tmp/backup
```

```
student$ pg_basebackup --pgdata=/home/student/tmp/backup --checkpoint=fast
```

If the second server is running, we stop it.

```
student$ sudo pg_ctlcluster 16 replica stop
```

Cluster is not running.

Move the backup to the data directory of the second server, changing the owner of the files:

```
student$ sudo rm -rf /var/lib/postgresql/16/replica
```

```
student$ sudo mv /home/student/tmp/backup /var/lib/postgresql/16/replica
```

```
student$ sudo chown -R postgres: /var/lib/postgresql/16/replica
```

Start the second server:

```
student$ sudo pg_ctlcluster 16 replica start
```

We got two independent servers, each of them has an empty test table. Let's add a couple of lines to the table on the first server:

```
=> INSERT INTO test VALUES (1, 'One'), (2, 'Two');
```

```
INSERT 0 2
```

To establish logical replication between servers, additional information in the WAL of the publishing server is required:

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

```
ALTER SYSTEM
```

```
student$ sudo pg_ctlcluster 16 main restart
```

Create a publication on the first server:

```
student$ psql -d replica_overview_logical_dba
```

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

```
CREATE PUBLICATION
```

```
=> \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:

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

```
=> CREATE SUBSCRIPTION test_sub
CONNECTION 'port=5432 user=student dbname=replica_overview_logical_dba'
PUBLICATION test_pub;
```

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

```
=> \dRs
```

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

Verify the replication:

```
=> INSERT INTO test VALUES (3, 'Three');
```

```
INSERT 0 1
```

```
=> SELECT * FROM test;
```

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

The following view shows the state of the subscription:

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

```
-[ RECORD 1 ]-----+-----
subid          | 24578
subname        | test_sub
pid            | 26471
leader_pid     |
relid         |
received_lsn   | 0/3004298
last_msg_send_time | 2025-09-24 17:04:37.336051+03
last_msg_receipt_time | 2025-09-24 17:04:37.337585+03
latest_end_lsn  | 0/3004298
latest_end_time | 2025-09-24 17:04:37.336051+03
```

The logical replication apply worker process has been started (you can see its ID in pg\_stat\_subscription.pid):

```
student$ ps -o pid,command --ppid 26164
```

```
  PID COMMAND
26165 postgres: 16/replica: checkpointer
26166 postgres: 16/replica: background writer
26178 postgres: 16/replica: walwriter
26179 postgres: 16/replica: autovacuum launcher
26180 postgres: 16/replica: logical replication launcher
26438 postgres: 16/replica: student replica_overview_logical_dba [local] idle
26471 postgres: 16/replica: logical replication apply worker for subscription 24578
```



## Identification modes for modifying and deleting rows

- primary key columns (default)
- columns of a specific unique index with the NOT NULL constraint
- all columns
- no identification (default for the system catalog)

## Conflicts: violation of integrity constraints

- replication is suspended until the conflict is resolved manually

Inserting new rows is straightforward. Changes and deletions are more complicated. These operations need to somehow identify the old version of the row. By default, primary key columns are used for this, but you can specify other ways (replica identity) when defining a table, i.e. use a unique index or all the table columns. Or you can disable replication for some tables altogether (system catalog tables have it disabled by default).

Since the table on the publisher and the table on the subscriber can be changed independently of each other, conflicts in the form of integrity constraint violations are possible when inserting new row versions. Whenever this happens, the process of applying records is suspended until the conflict is resolved manually.

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

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

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

## Conflicts

Local modifications on the subscriber are not prohibited. Let's insert a row into the table on the second server:

```
=> INSERT INTO test VALUES (4, 'Four (local)');  
INSERT 0 1
```

If we add a row with the same primary key value on the publisher, the subscription will have a conflict when trying to apply the change.

```
=> INSERT INTO test VALUES (4, 'Four');  
INSERT 0 1  
=> INSERT INTO test VALUES (5, 'Five');  
INSERT 0 1
```

The subscription is unable to apply the change, replication stops.

```
=> SELECT * FROM pg_stat_subscription \gx  
  
-[ RECORD 1 ]-----+-----  
subid          | 24578  
subname        | test_sub  
pid            |  
leader_pid     |  
relid          |  
received_lsn   |  
last_msg_send_time |  
last_msg_receipt_time |  
latest_end_lsn |  
latest_end_time |  
  
=> SELECT * FROM test;  
  
 id | descr  
----+-----  
  1 | One  
  2 | Two  
  3 | Three  
  4 | Four (local)  
(4 rows)
```

To resolve the conflict, remove the row on the second server and wait a moment...

```
=> DELETE FROM test WHERE id=4;  
DELETE 1  
=> SELECT * FROM test;  
  
 id | descr  
----+-----  
  1 | One  
  2 | Two  
  3 | Three  
  4 | Four  
  5 | Five  
(5 rows)
```

Replication continues.

---

## Dropping a Subscription

If replication is no longer needed, the subscription should be deleted, otherwise the publisher will keep the open replication slot.

```
=> DROP SUBSCRIPTION test_sub;  
  
NOTICE:  dropped replication slot "test_sub" on publisher  
DROP SUBSCRIPTION
```



## Not replicated

- DDL commands
- sequence values
- large objects
- changes to views, materialized views, and foreign tables

## Not supported

- automatic conflict resolution

Logical replication has some fundamental limitations.

DDL commands are not replicated — all schema changes must be copied manually.

Only regular base tables and partitioned tables can be replicated. Other relation types such as views, materialized views, and foreign tables cannot be replicated.

Sequence values are not replicated. This means that if the subscriber inserts rows into a replicated table with a surrogate primary key, conflicts may occur. These conflicts can be avoided by allocating different sequence ranges to each server or by using UUIDs instead of sequences.

There is no built-in conflict resolution mechanism.

These restrictions reduce the applicability of logical replication.

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

## Referential integrity

for TRUNCATE operations, the publication must include all tables referenced by foreign keys

## Persistent connection is mandatory

inactive replication slots prevent WAL segment removal and hold back the vacuum horizon

## Potential issues

bulk data changes

changes made by long-running transactions

To properly replicate TRUNCATE commands, the publication should include all tables that have foreign key references to the truncated table.

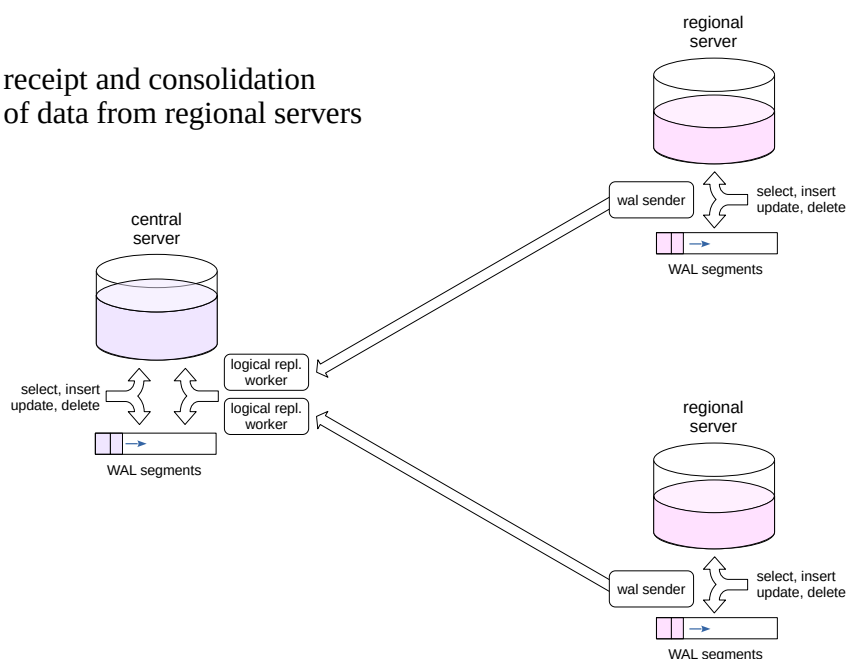
The connection between the publisher and the subscriber must remain stable. If interrupted, the replication slot becomes inactive, forcing the server to retain WAL segments until reconnection. Inactive slots also hold back the vacuum horizon.

Changes are replicated row-by-row, so SQL commands affecting many rows on the publisher create significant subscriber load.

The default configuration handles long-running transactions poorly — they increase publisher load since changes are only sent after transaction commit. The subscription's streaming parameter mitigates this by getting row change without delay, either buffering changes in temporary files or applying changes immediately when background process is available.

# 1. Consolidation

receipt and consolidation  
of data from regional servers



12

Let's discuss some logical replication use cases.

Suppose there are several regional branches, each of which runs on its own PostgreSQL server. The goal is to consolidate some data on a central server.

First, publications of the necessary data are created on regional servers. The central server subscribes to these publications. The received data can be processed using triggers on the central server (for example, unifying the data format).

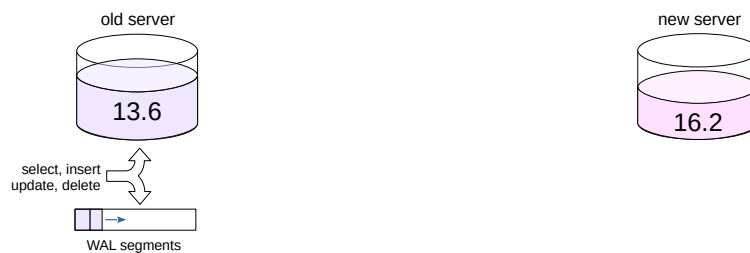
Inverted, the setup allows, for example, to transfer reference information from the central server to regional ones.

The business logic may apply additional constraints on the system. In some use cases, scheduled batch data transfers may be preferable.

The image shows two WAL receiving processes running on the central server, one for each subscription.

## 2. Server Update

updating the major version  
without interruption of service



Case: update the major version on the server without interrupting the service.

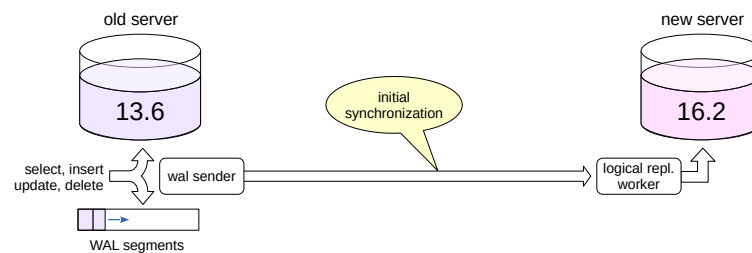
Two major versions are not binary compatible, so physical replication will not work. However, logical replication can solve the problem.

As usual, external tools are required to switch users between servers.

First, a new server is created with the desired PostgreSQL version.

## 2. Server Update

updating the major version  
without interruption of service

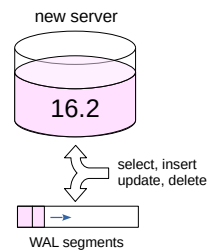


Then, logical replication of all required databases is set up between the servers, and the data is synchronized. This is possible because logical replication does not require binary compatibility between servers.



## 2. Server Update

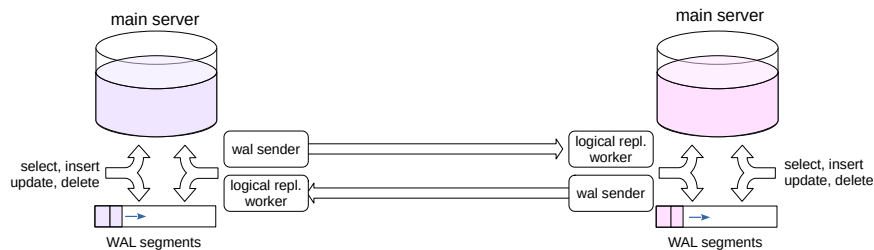
updating the major version  
without interruption of service



Clients are then switched to the new server while the old one is terminated. In practice, the process of updating major server version using logical replication is much more complicated and difficult. It is discussed in more detail in the Server Update lesson of the DBA2 course.

### 3. Primary-Primary

a cluster where  
multiple servers can modify data



16

Case: provide reliable data storage on multiple servers with the ability to write the data on any server (particularly useful for geo-distributed systems).

This can be achieved through bidirectional logical replication, synchronizing changes for the same tables between servers in both directions.

PostgreSQL 16 introduced bidirectional replication, where tables can be both published and subscribed to on different servers simultaneously.

This requires that the applications working with the cluster are built with certain considerations in mind in order to avoid conflicts when modifying data in the same table. For example, to use globally unique identifiers or to ensure that different servers work with different ranges of keys.

Keep in mind that the primary-primary setup with logical replication will not support global distributed transactions and thus cannot guarantee full data consistency between servers. In addition, PostgreSQL does not offer any tools for automatic failure processing, adding nodes to the cluster or removing nodes from it, etc. These tasks must be solved by external means.

Logical replication streams individual row changes

- multidirectional

- requires protocol-level compatibility

Publish and subscribe model

All current restrictions must be considered

1. Set up logical replication of an arbitrary table to another table on the same server.
2. Set up bidirectional logical replication of the same table between two different servers.

1. If you try to perform standard operations, the CREATE SUBSCRIPTION command will hang. Read carefully the documentation:

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

2. Clone the second server from a backup, as shown in demonstration earlier.

When creating subscriptions on both servers, specify the following parameters: `copy_data = false`, `origin = none`.

## 1. Replication on a Single Server

Test table:

```
=> CREATE DATABASE replica_overview_logical_dba;
```

CREATE DATABASE

```
=> \c replica_overview_logical_dba
```

You are now connected to database "replica\_overview\_logical\_dba" as user "student".

```
=> CREATE TABLE test (  
    id int  
);
```

CREATE TABLE

A copy of the database with the table:

```
=> \c student
```

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

```
=> CREATE DATABASE replica_overview_logical_dba2 TEMPLATE replica_overview_logical_dba;
```

CREATE DATABASE

Set the required WAL level:

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

ALTER SYSTEM

```
=> \q
```

```
student$ sudo pg_ctlcluster 16 main restart
```

Connect to the databases:

```
student$ psql replica_overview_logical_dba
```

```
student$ psql replica_overview_logical_dba2
```

Create a publication in the first database:

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

CREATE PUBLICATION

By default, the CREATE SUBSCRIPTION command creates a replication slot, which waits for all active transactions (at the time of slot creation) to complete before proceeding. One such transaction is the CREATE SUBSCRIPTION command itself, causing it to block indefinitely.

This issue can be resolved by manually creating a replication slot and specifying its name when creating the subscription:

```
=> SELECT pg_create_logical_replication_slot('test_slot', 'pgoutput');
```

```
pg_create_logical_replication_slot  
-----  
(test_slot,0/21810F8)  
(1 row)
```

```
| => CREATE SUBSCRIPTION test  
| CONNECTION 'user=student dbname=replica_overview_logical_dba'  
| PUBLICATION test WITH (slot_name = test_slot, create_slot = false);
```

```
| CREATE SUBSCRIPTION
```

Verification:

```
=> INSERT INTO test SELECT * FROM generate_series(1,100);
```

INSERT 0 100

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

```
count
-----
100
(1 row)
```

Replication is working.

Delete the publication, subscription, and the second database.

```
=> DROP PUBLICATION test;
```

DROP PUBLICATION

```
=> DROP SUBSCRIPTION test;
```

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

```
=> DROP DATABASE replica_overview_logical_db2 (FORCE);
```

DROP DATABASE

## 2. Bidirectional Replication

We clone the server using a backup:

```
student$ pg_basebackup --pgdata=/home/student/tmp/backup --checkpoint=fast
```

Make sure the second server is stopped and push the backup:

```
student$ sudo pg_ctlcluster 16 replica stop
```

Cluster is not running.

```
student$ sudo rm -rf /var/lib/postgresql/16/replica
```

```
student$ sudo mv /home/student/tmp/backup /var/lib/postgresql/16/replica
```

```
student$ sudo chown -R postgres: /var/lib/postgresql/16/replica
```

Start the second server:

```
student$ sudo pg_ctlcluster 16 replica start
```

The database with the table and the log level setting were also cloned:

```
student$ psql -p 5433 replica_overview_logical_db2
```

```
=> SHOW wal_level;
```

```
wal_level
-----
logical
(1 row)
```

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

```
count
-----
100
(1 row)
```

We publish the table on both servers:

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

CREATE PUBLICATION

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

CREATE PUBLICATION

Subscribe:

```
=> CREATE SUBSCRIPTION test
CONNECTION 'port=5433 user=student dbname=replica_overview_logical_db2'
PUBLICATION test WITH (copy_data = false, origin = none);
```

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

```
=> CREATE SUBSCRIPTION test
CONNECTION 'port=5432 user=student dbname=replica_overview_logical_dba'
PUBLICATION test WITH (copy_data = false, origin = none);
```

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

Modify table rows:

```
=> UPDATE test SET id = id + 100;
```

ERROR: cannot update table "test" because it does not have a replica identity and publishes updates

HINT: To enable updating the table, set REPLICA IDENTITY using ALTER TABLE.

For changes and deletions to be replicated properly, row identification must be configured. By default, rows are identified by their primary key.

```
=> ALTER TABLE test ADD PRIMARY KEY (id);
```

ALTER TABLE

```
=> ALTER TABLE test ADD PRIMARY KEY (id);
```

ALTER TABLE

Retry the operation:

```
=> UPDATE test SET id = id + 100;
```

UPDATE 100

```
=> UPDATE test SET id = id + 100;
```

UPDATE 100

Check the Result:

```
=> SELECT min(id), max(id) FROM test;
```

```
min | max
-----+-----
201 | 300
(1 row)
```

```
=> SELECT min(id), max(id) FROM test;
```

```
min | max
-----+-----
201 | 300
(1 row)
```