EUROPEAN ORGANIZATION FOR NUCLEAR RESEARCH

DBOD
DATABASE ON DEMAND

Summer Student Report

Zero-Downtime PostgreSQL Upgrades Using Replication and Connection Switching

Author:
Wojciech Rzepliński
IT-DA-DB

Supervisors:
Maurizio De Giorgi
Nikolaos Smyrnioudis

September 5, 2025

Abstract

This report presents a procedure for zero-downtime PostgreSQL version upgrades on the DBOD platform at CERN, fully automated by a Python prototype. Several approaches were identified each with different strengths and weaknesses. One method was chosen for which the prototype was implemented. The chosen method utilizes a PgBouncer proxy to orchestrate the transition, reducing downtime to roughly one second, visible as a brief increase in transaction latency by using a transaction pause. While the approach is constrained by factors like replication throughput, a freeze on DDL operations, and the need for client-side retry logic, it demonstrates that zero-downtime upgrades are achievable and offer a substantial enhancement to database availability.

Contents

1 INTRODUCTION

Upgrading database systems in production environments poses a critical and challenging task. Traditional upgrade procedures often involve downtime, which can disrupt services and significantly impact users. While small outages may be acceptable in some contexts, in large-scale or mission-critical systems, even a few seconds of unavailability can have significant consequences.

The primary goal of this project was to explore and implement strategies for near zero-downtime database upgrades for the DBOD platform running on the CERN infrastructure. Specifically, the work is focused on comparing different approaches both seen in the industry and new ones, with an emphasis on limitations and benchmarking the most promising method.

The solution was implemented as part of the DBOD service running in the CERN datacenter.

2 BACKGROUND AND RELATED WORK

Zero-downtime upgrades are a concept widely researched and applied within the industry. Thanks to logical replication, a feature introduced in PostgreSQL 13, it became easier to replicate databases across different major PostgreSQL versions For replica-based updates, a key challenge is the switchover—the process of seamlessly redirecting all client connections from the old primary PostgreSQL instance to the new one. To perform the switchovers which are invisible to the users a proxy between the client and instance has to exists. The PgBouncer proxy is a tool originally designed for connection pooling that can be configured to pause transactions and switch backends, enabling a seamless transition.

Different approaches were invented and compared [1]. One of the most impressive examples of this work comes from the GitLab team [2] which existing PgBouncer setup with HA to flawlessly switch connections without any connections or transaction broken. Another impressive method was performed by the Instacart [3] who managed to perform the zero-downtime-upgrades using the databases managed by database managment system Amazon Relational Database Service (RDS). Their method although simpler was critized on the forums for potentialy losing transactions. It can be said that with each passing month, more companies are adopting zero-downtime upgrades, with a recent example being a company that published its results just a few days ago [4].

Despite these impressive results, there are also critics who argue that in most cases, a database is not so absolutely critical that it would suffer from a brief downtime. Given the significant effort required, engineering resources might be better allocated elsewhere, with users simply being notified of a planned maintenance window.

3 TIMING OF THE GITLAB (CERN) DATABASE UPGRADE

Database for the Gitlab service at CERN is an example of a big database amounting to the 400 GB of data. To motivate the need for reduced downtime, the upgrade was thoroughly benchmarked and the results are visible in Figure 1.

The main upgrade which consists mainly of rewriting the database data files from the old format into the new format took about 3 minutes.The most time consuming part was vacuuming. This operation doesn't require downtime, but it is a good practice to make

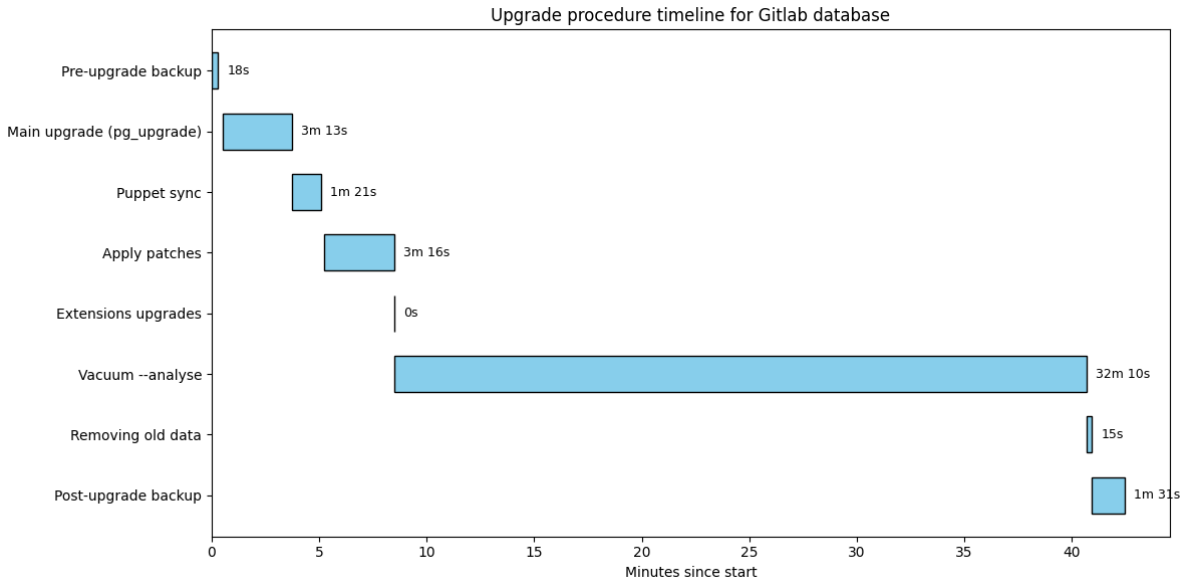


Figure 1: Gitlab upgrade timeline

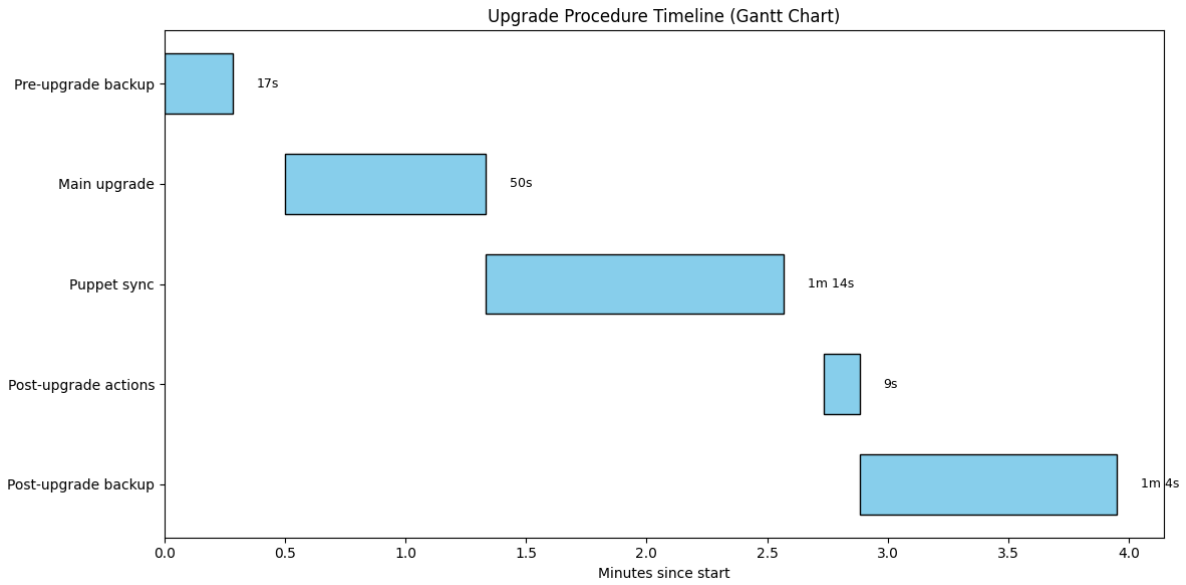


Figure 2: Empty database upgrade timeline

sure bloating is minimized and the new version optimizer can rely on freshly gathered statistics.

Empty database update was performed for the comparison Figure 2. The main upgrade process took 50 seconds, vacuuming was a 6 second operation. This means that the main upgrade process is not that time consuming and even very large databases can upgrade in about ten minutes.

4 DIFFERENT ASYNCHROUNOUS UPGRADES PROCEDURES

To achieve a major version upgrade with minimal downtime, the core strategy involves using an asynchronous replica of the production database. The details of each step are deliberately omitted here, as they are explained in the following sections.

4.1 The industry approach (upgrading the replica)

This procedure, following a common industry pattern, was taken as the starting point. The main idea is to prepare a fully upgraded, ready-to-go replacement for the primary database before making any changes that would be visible to the client. This approach prioritizes safety and preparation. The procedure diagram is shown in Figure 3.

The sequence of steps is as follows:

1. Create a physical replica
2. Create a logical replica from the physical one
3. Stop the logical replication and perform an actual upgrade on the logical replica.
4. Enable back the logical replication and wait for convergence
5. Once ready and replication lag is low, do the switchover.
6. During the switchover downtime, wait until replication lag is zero.
7. After a succesfull switchover, the replica becomes the primary

This approach presents challenges related to the resource usage of the DBOD system. If the replica were to be located on the same volume as the primary, the volume size would need to be increased significantly, and the replica would be constrained on the same host machine as the primary. Furthermore, it would require changing the default data directory of the instance in the volume, which can potentially lead to many parts of the DBOD core library rewritten, as it currently expects the data directory path to be in the specified form.

Alternatively, if the replica were located on a different host, it would require a separate volume. Consequently, the history of backups would be less accessible, as it would require restoring from a snapshot from a different volume.

However, the main problem, as it turned out, was the PostgreSQL `wal_level` configuration parameter. It configures the amount of information to store in the Write-Ahead-Logs. For physical replication, the `wal_level` must be set to 'replica'. However, to serve as a source for logical replication, the `wal_level` must be set to 'logical', which stores

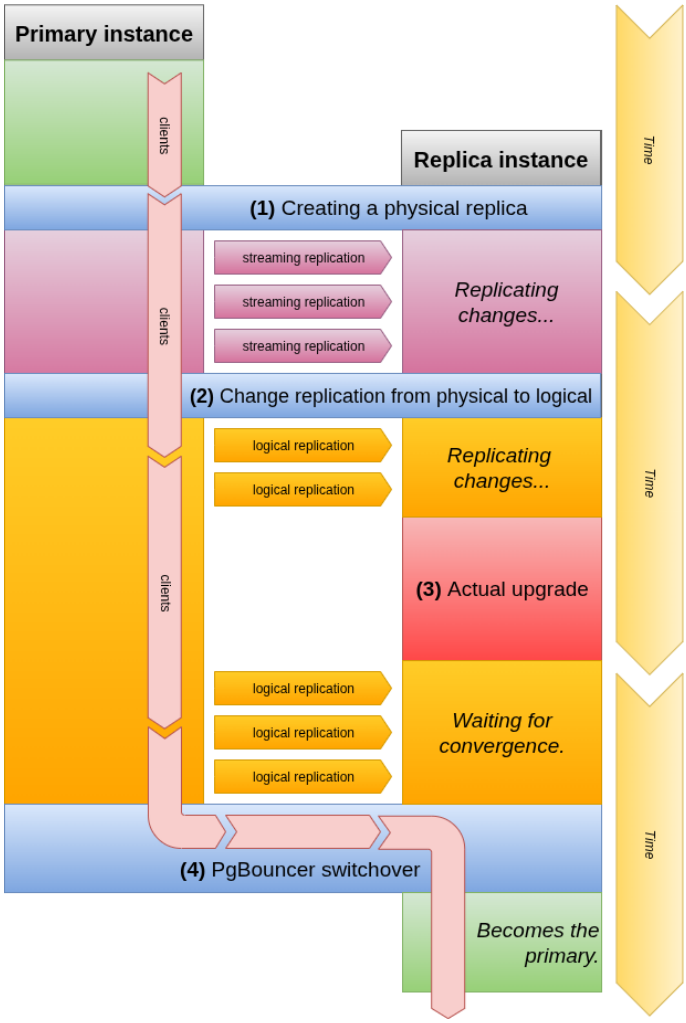


Figure 3: The industry approach

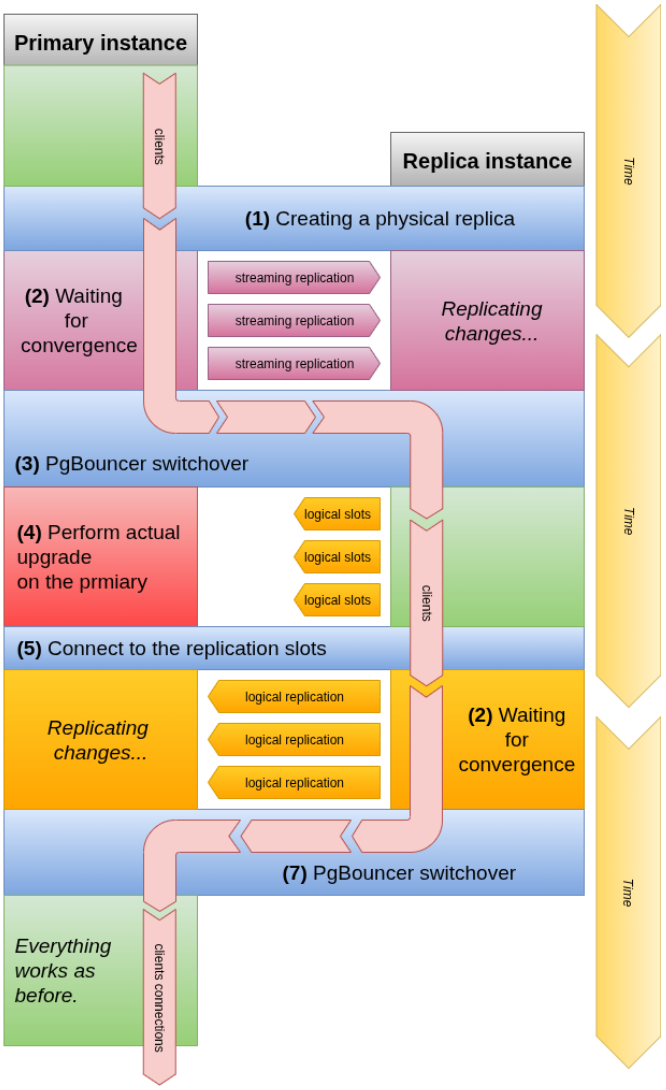


Figure 4: The DBOD approach

a greater amount of information in the WAL files for logical decoding. Changing this parameter requires a database restart, a step that results in a downtime (from a few seconds up to one minute).

4.2 The alternative approach (upgrading the primary)

The alternative approach involves upgrading the primary instance while using a replica to temporarily maintain client connections. It would require two switchovers, the first one to redirection connections to the replica and the second one back to the upgraded primary. The procedure is listed in Figure 4.

The procedure is as follows:

1. Create a physical replica
2. Wait until the replication lag is low
3. Switch the client connections to the replica. During the downtime promote the replica and create the logical replication slots on the replica

4. Perform the actual upgrade on the primary
5. Connect the upgraded primary to the logical replication slots and start replicating changes from the replica (so this is reverse replication)
6. Wait until the replication lag is low
7. Switch the client connections back to the upgraded primary instance. During the switchover, wait for the replication lag to become zero.
8. Everything is as before

It is crucial to note that the direction of the logical replication is reversed compared to the previous approach. Now the replica is the source of logical replication. This means that the replica must have necessary `wal_level` configured which can be done before the replica starts, without interfering with the primary database.

This approach has the key advantage that, from the primary instance's perspective, the zero-downtime upgrade appears identical to a standard upgrade. After the upgrade, the transactions are replayed on the primary, which is very similar to having those transactions applied to the primary directly.

4.3 Comparison of the approaches

Arguments in favour of the industry approach:

- All steps leading up to the final switchover do not make any permanent changes and are transparent to clients. In contrast, with the alternative approach, a failure during the switchover could result in a serious issue.
- In the alternative approach during the switch the replica accumulates WAL files. Assuming the upgrade procedure can last for an hour that means that an hour of WAL files has to be stored on disk to be later replayed.

Arguments in favour of the alternative approach:

- This method avoids changing the WAL level parameter and thus avoids the necessity to restart the database.
- It allows the replica to be hosted on a different machine and volume, while still ensuring the procedure output exactly the same as standard upgrade. All the paths and the names stay the same, and the backup system remain unchanged.

5 WAYS OF CREATING LOGICAL AND PHYSICAL REPLICA

The foundation of any zero-downtime upgrade strategy involving a standby server is the creation of a replica. Depending on the chosen zero-downtime upgrade approach, the type of the replication setup will differ. Replicas can be created using various method, each with different performance, resource usage and complexity.

5.1 Creating a physical replica

A standard PostgreSQL tool for this task is `pg_basebackup`, which streams the entire data directory over the network. However, for large databases, this can be time-consuming operation (for example creating the PostgreSQL replica for GitLab using this tool took 3 hours).

An alternative approach relevant in environments using incremental backups storage provided is to use storage-level snapshots, such as those provided by NetApp, which can create a clone from a volume snapshot almost instantaneously. The drawback is that the snapshot and the original volume must persist as long as the clone volume is used. For Disaster Recovery it would be required to have the replica volume in a different datacenter, meaning this method is not suitable for long running replicas.

However, when the replica is only temporarily needed, this alternative approach to replica creation is suitable. This approach was implemented successfully. The total time to get the replica running is about 4 minutes to create a snapshot and 1 minute to create a volume in NetApp on my benchmark database.

5.2 Creating a logical replica

The logical replication transmits high-level changes (in text format).PostgreSQL uses 'logical decoding' under the hood, which is a routine that translates the physical WAL file changes back to SQL-like language. To configure decoding, a 'publication' must be created on the source database and a "subscription" on the target instance.

The subscription functions in PostgreSQL have the option to automatically transfer all the table data "a process referred to as the 'synchronization' step. For the Gitlab database it would take about 40 hours, which would make it an unfeasible solution for this use case.

A common procedure in the database world is to convert the physical replica into the logical one. The challenge lies in establishing a single point in the Write-Ahead-Log where the physical replication has ended and logical replication has started from. This can be achieved by using a combination of `pg_replication_slot_advance()` function to move the slots forward and `recovery_target_lsn` recovery configuration option to stop physical replication at the defined LSN value. Beginning with PostgreSQL 17 version function `pg_createsubscriber` tool was introduced which creates a new logical replica from a physical standby server, but it was not taken into account.

During the development, the second approach was implemented which proved that it is possible. However, creating logical replica from the physical one ultimately was not a part of the final approach.

6 WAYS OF PERFORMING THE SWITCHOVER

The switchover is a critical and sensitive phase. It is the moment when client connections are redirected from the old primary database to the newly upgraded instance.

6.1 Restart of both primary and replica instance

The simplest solution is to stop both instances, which must to be on the same host machine, change their port settings and start them again. This approach was not pursued

because the restart of the PostgreSQL instance takes a few seconds and is highly dependent on the current instance load (a restart of the GitLab database with no active client connections took 4 seconds, but this is not comparable to a restart under an active load). During the restart, active connections are broken and new connections are not accepted.

6.2 PgBouncer

Downtime can be avoided by using a proxy. The most popular tool for that is PgBouncer. It was also used by other companies performing zero-downtime upgrades. Originally created as a connection pooling program, it supports several levels of pooling granularity.

- Session pooling - when a client connects, a server connection will be assigned to it for the whole duration it stays connected.
- Transaction pooling - a server connection is assigned to a client only during a transaction. When PgBouncer notices that the transaction is over, the server will put the connection back into the pool.

The transaction pooling is what enables zero-downtime upgrades. It can switch the backend connections to a different PostgreSQL instance without the client to reconnect.

6.2.1 Problems with attaching the PgBouncer

One of the assumptions was to keep the client connection parameters (host, port) to stay the same during the entire zero-downtime upgrade. Getting the PgBouncer working on the same host and port is difficult as there already was a PostgreSQL instance working. This problem was solved by using the linux packet filters, to redirect the packets into a different port.

The **nft** tools allows to add redirection on all connections from one port into a different port. Let's say for example that the PostgreSQL instance is working under port 6616 and the PgBouncer under 6620 (as on Figure 5). Client connections arrive on port 6616. PgBouncer was configured to use the 6616 port as PostgreSQL backend instance (either through localhost which is a different type of nft redirection and external interface or through the socket). Then I enable the packet redirection from port 6616 to 6620 on external network interface. The active connections are not redirected and have to be terminated, but the clients can recover instantly. New connections are redirected to the 6620 port without any downtime. The result is presented on Figure 6. In the future steps, when performing a switchover to a different backend (potentially on a different host), the situation may look like on Figure 7.

6.2.2 Problems with the authentication

PgBouncer acts like a proxy, but makes its own connections to the backends. There are two places where authentification takes place. First one is between the clients and the PgBouncer and the second one is between the PgBouncer and the backend. Both use the same authentication methods (md5, scram-sha256) and permission file (hba file).

PgBouncer can be configured to use the same `pg_hba.txt` file to authentificate the clients and use the (hashed) passwords from the text file, which should be generated by querying the original instance. An alternative way is to configure PgBouncer to query

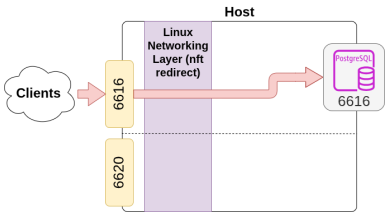


Figure 5: Before PgBouncer is attached

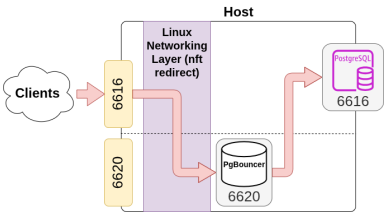


Figure 6: After PgBouncer is attached

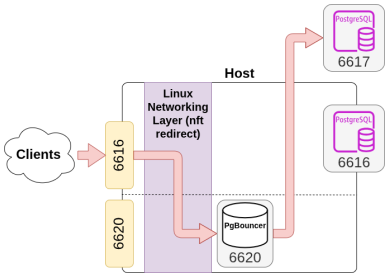


Figure 7: After PgBouncer redirects to different instance

the original instance directly once the new client connection is pending. Both approaches were tested and work, but the text file method should be considerably faster.

There is a problem with authentication between PgBouncer and the PostgreSQL instance, since the connection's source IP address is different than the original client connection. The connection parameters can also be different (like no SSL security). The database may require the connection to be secured through SSL, but PgBouncer will not be able to make such a connection and will make the usual one instead. Similarly, there exist option that the only TCP connection accepted are the ones without SSL encryption.

To mitigate this, an automated script was created that detects if the entry in the `pg_hba.txt` file defines the source IP address or connection encryption, and loosens those requirements while PgBouncer is attached. It creates a new record for PgBouncer. This new record can be defined with the PgBouncer's source IP address.

6.2.3 Problems with pooling settings

The PgBouncer is a connection pooler and this pooling feature is not needed in our use case. However, as it is always enabled, it can interfere with client connections. There are configuration parameters that limit the user or database connections, which should be taken into account before and configured based on typical usage.

6.2.4 Problems with throughput

The PgBouncer is a single thread process. When the transaction pooling is enabled there are a lot of transactions which has to be routed to a connection from the pool it can create a bottleneck. It is visible on the stress tests on Figure 11.

This problem is widely known. There are benchmarks that show that PgBouncer transaction mode creates major overhead when there is a large number of small transactions (https://www.percona.com/blog/scaling-PostgreSQL-with-pgbouncer-you-may-need-a-connection-pooler-sooner-than-you-expect/). The advantages of using a pooler are visible from 100+ client connections.

The solution may be to run PgBouncer on separate CPU or create many PgBouncer processes.

6.2.5 PgBouncer backend switch steps

PgBouncer supports `PAUSE` command, which stops the new transactions from executing. From the client's perspective, this is only visible in an increased latency. For example,

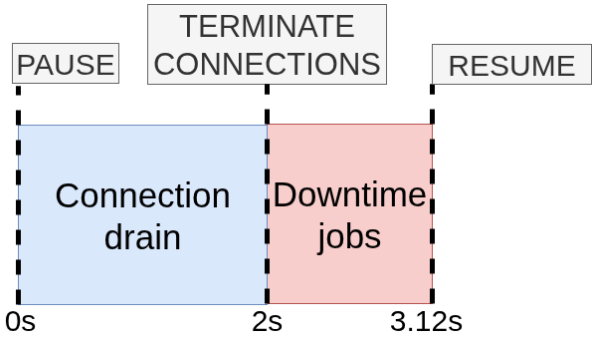


Figure 8: Switch steps

when transactions are paused for 2 seconds and a transaction took 100 ms, then the total latency would be 2.1 s.

The procedure implemented performs a connection drain, as follows:

1. Pause the transactions.
2. Wait for a configured period (e.g., 2 seconds) for current transactions to end.
3. If there are still active transactions terminate them
4. Perform downtime jobs.
5. Resume the transactions.

If any transaction takes more than 2 seconds then it will be terminated. As a consequence, all new transactions may have up to 2 seconds + downtime increased latency.

6.3 Nft linux networking

During the development phase, it was determined that the Linux packet filtering with `nft` can also be employed for the switchover. The redirection can also point outside of the host. Jobs requiring downtime must be executed when the instance is not accepting connections. resulting in a period of approximately 1 second of complete downtime.

7 LOGICAL REPLICATION LIMITATIONS

Logical replication has some well known limilimiatationstations that are particularly important for a zero-downtime upgrade procedure.

7.1 DDL operations are not replicated

During the upgrade procedure no DDL operations can be performed, since they would be lost and not replicated (but it is changed in PostgreSQL version 18). It is a responsibility of the application to ensure this contraint is met.

7.2 Table identity

For the logical replication to work each table must have the replication identity. It is a key that allows each row to be uniquely identified.

The best practice is to have a primary key or unique value present on all tables. A query was written to detect whether every table has a replication identity. When the primary key is not present the table identity can be set to "full", which means that the entire row serves as the identity."

7.3 Sequences

Sequence generators are not replicated in logical replication. It means that when the primary database generates new numbers and increases the counter of the generator, the replica also inserts those rows but it does not increase the generator counter. After the replica becomes the primary the counter will generate numbers that have already been used.

This issue is easily resolved. The moment before the switchover set all the sequence counters on the replica to coresponding values fetched from the primary at the same moment. A safety offset, such as a large number, should also be applied.

7.4 Configuration and workers

The replication workers are required to decode the WAL files on the primary and to replay the changes on the replica. They are configured by a set of parameters, which define the maximum number of workers for each task. Every database involved in logical replication must have its own worker. It means that that if the number of databases is large there won't be enough replication workers. The default number in PostgreSQL is 4, when there are more databases the number worker has to be configured accordingly.

Also an important insight is that the replica applies the changes from the source using only one worker and one thread. The primary can have multiple client connections, where each connections has its own thread so it benefits from the parallelism. Although not observed during these specific tests, this single-threaded replay can create a bottleneck. There are countermeasures for this. PostgreSQL offers a parallel replication, which splits the replay jobs into multiple workers.

8 IMPLEMENTED PROCEDURE

The implemented procedure is based on the alternative approach of upgrading the primary instance. It utilizes the PgBouncer for the connection switchover, NetApp "clone from snapshot" function to create a physical replica and performs automatic configuration of the tooling.

The detailed steps are as follows:

1. Perform a check for the replication identity of every table.
2. Create a backup that is saved in a snapshot. The core library is used directly for this step.
3. Restore a backup from a snapshot onto a different volume using the "clone volume from snapshot" feature. This new volume will serve as the replica.

4. Modify the PostgreSQL configuration file of the replica volume so it can be the source of logical replication.
5. Start physical streaming replication from the backup snapshot.
   (a) Create a replication user.
   (b) Create a "primary_conninfo" connection string and a restore command.
   (c) Disable archive mode, enable hot standby, create a `standby.signal` file.
   (d) Wait for the instance to start and the replication to become active
6. Attach PgBouncer to the primary instance
   (a) Query the password file
   (b) Modify instance `pg_hba.txt` file to add entries for the PgBouncer connections
   (c) Add nft port redirection from the primary's port to PgBouncer's port
   (d) Terminate connections on the primary instance
7. Wait for convergence and perform a switchover of connections to the replica.
   (a) Modify the replica instance pg_hba.txt file to add entries for the PgBouncer connections.
   (b) Wait until the replication lag becomes low.
   (c) Perform the connection drain switchover as described in the subsubsection 6.2.5. During the downtime perform:
      i. Promote the replica
      ii. Create one "all tables" publication per database for every database on the replica
      iii. Create replication slots for every database.
8. Perform the upgrade of the primary. This can be done by starting an upgrade job within the DBOD service. The instance remains operational and healthy, but since traffic is being routed to the replica, no new transactions are being processed on the primary.
9. After the upgrade, start reverse logical replication by subscribing from the primary to the replication slots on the replica.
10. Wait until the replication lag is low
11. Switch the connections back to the primary. It consists of the following steps:
    (a) Increase the sequences on the primary to match the values on the replica, plus a safety margin of approximately one million.
    (b) Check if the replication lag is higher than expected, and fail the procedure if it is.
    (c) Perform the switchover. During the switchover downtime, do:
       i. Wait for the replication lag to become zero.

ii. Disable the subscriptions

(d) Drop the subscriptions.

12. Detach the PgBouncer

(a) Remove the nft port redirection introduced while attaching the PgBouncer

(b) Stop the PgBouncer with the "terminate connections" shutdown option.

(c) Restore the original `pg_hba.txt` files on both instances.

13. Cleanup jobs (optional). Stop the replica, delete the replica volume.

8.1 Performed tests

To test the procedure, the following tests and benchmarks were created:

- Missing transactions test - every 0.1 second new unique entry was added to the table and also saved locally in a Python dictionary. At the end, a query was run to check if all values from the dictionary were present in the database.
- OLTP-like transaction test - python script that resembles the transactions of the popular OLTP system.
- WAL stress testing - inserting large batches of artificially generated data, with LSN increase per second ranging from 2MiB to 30 MiB, to observe replica converge.
- Sysbench write_read Lua test, executed on a more powerful machine via Rundeck.
- Long running transactions test (to see if the upgrade will not fail).

9 RESULTS

9.1 Downtime and latency

To measure the downtime two approaches were implemented. First, the precise clock times were measured in the zero-downtime upgrade script of the key events: "PAUSE" command in PgBouncer and "RESUME" command. Also the measurements were taken by the benchmark tool.

The downtime measured locally for the first switch (replica promotion and creating replication slots) was about **1 second**. This consists of 0.5 second for replia promotion and 0.5 for creating replication slots for each database. It is a significant overhead to connect to every database to replicate, and there may be an optimization here to create the connections and replication slots to each database concurrently. The second switch downtime took about **0.01 s** repeatedly. This consists of only waiting for the lag to become 0.

The main result is the increase in the latency. The custom test created many OLTP transaction like queries and measured their downtimes. No transactions were interrupted.

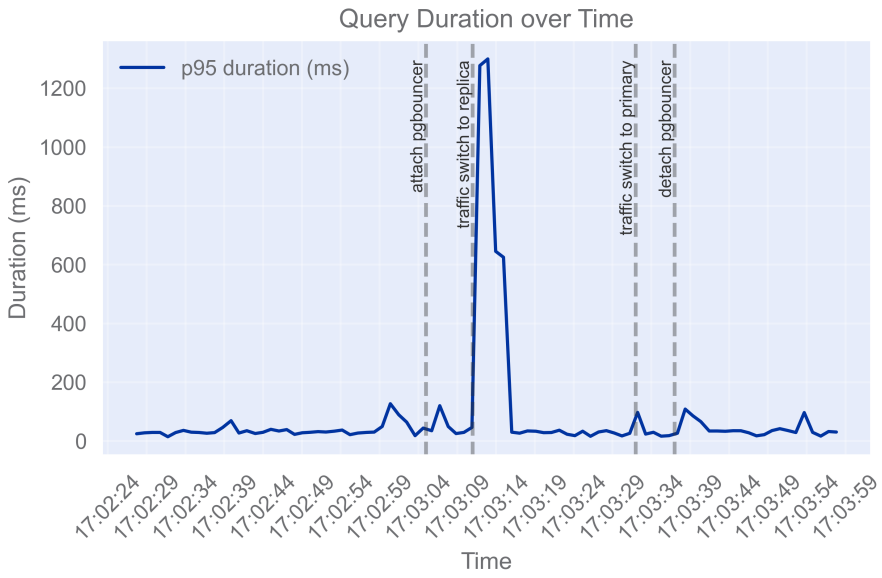


Figure 9: OLTP custom python benchmark latency

9.1.1 Sysbench

The first results failed during the Sysbench test, due to the lack of retry logic for broken connections for the PostgreSQL backend which are a side effect of attaching and detaching PgBouncer. The custom patches were applied and then the measurments were taken which are visible on the figures Figure 11, Figure 12, Figure 10. Sysbench performs queries in a loop and the more latency in queries the less transaction throughput is.

In the results there is an increased latency during the time the first switchover is happening and there are two places where the reconnection happens, during attachment and detachment of the PgBouncer.

What was surprising, there is a visible increase in the latency when the PgBouncer is attached. This may be due to the PgBouncer being a proxy and forwarding the queries and the results. Also the sysbench uses prepared transactions which may be an additional overhead for the PgBouncer. There is a possibility that the PgBouncer running on the same host as the instance does not have sufficient CPU resources, making it a potential bottleneck. The backend_host parameter in the PgBouncer configuration plays an important role here. When a localhost connection was used, latency increased by about 8 ms, whereas using a Unix domain socket resulted in an increase of about 3 ms. The compute of the benchmark client and the compute of the instance host can also change the results significantly, as well as connection latency between them. In summary, PgBouncer introduces a non-negligible amount of additional latency.

9.2 Convergence under load

To measure the properties of logical replication convergence, replication lag values were measured under a workload that generated a significant volume of LSN changes.

The script for testing heavy WAL generation was a Python program in which each thread performed a batched operation in a loop.The parameters included the number of threads, batch size, and the row data size in kilobytes. Each thread in a loop inserts or updates a batch of rows. In each row, the content value is a string of a predefined size,

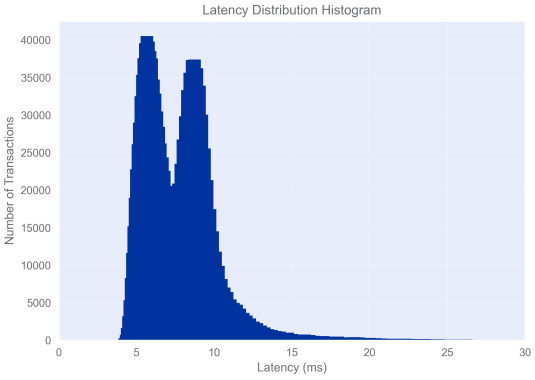


Figure 10: Sysbench latency distribution

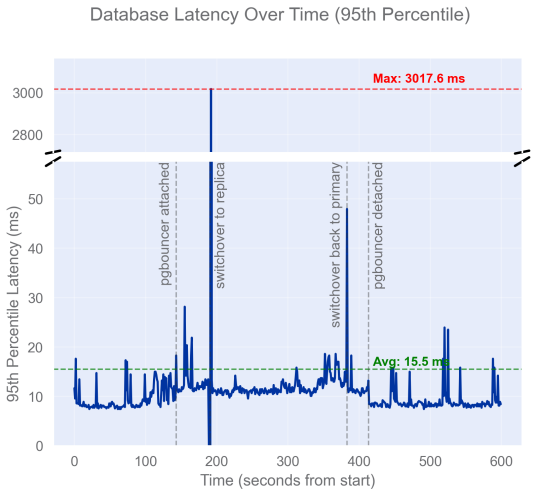


Figure 11: Sysbench latency over time

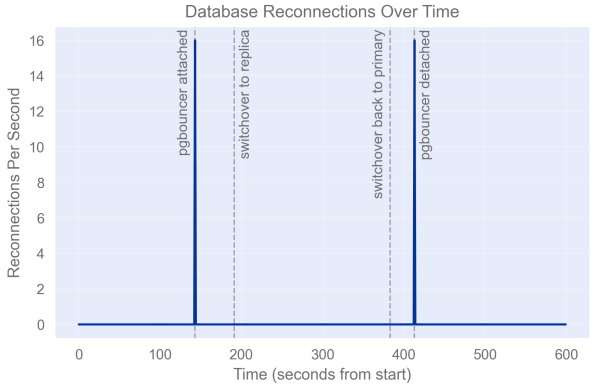


Figure 12: Sysbench reconnections over time

followed by a sleep of 0.1 seconds.

The results are presented in Figure 13, Figure 14, Figure 15, Figure 16. Replication slots were created at the beginning of the test, but the replica was not subscribed to them. After 2 minutes the replica subscribes to the slots and should start replicating changes and decrease the replication lag.

The 'lag increase value' was calculated as the LSN lag increase from the start of the benchmark until the maximum lag was reached. The "lag decrease value" was measured from the start of the replication until the lag fell below 1 MB.

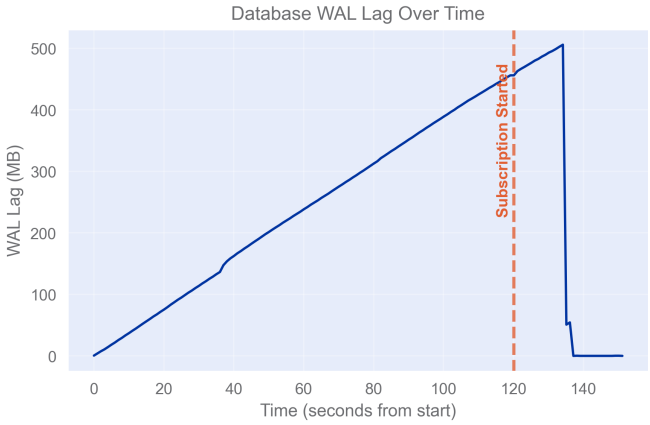
Replication lag was measured as the difference between the replication slot's restart LSN and the current LSN.

Replication was routed through the CERN network to simulate the configuration with two different hosts, but in this case the host connected to itself (with the dbod hostname, not localhost).

9.2.1 Observations

There are a few points to note from the charts.

- Logical replication takes a few seconds to show a decrease in LSN lag; it is not instantaneous. decrease in replication lag under a heavier load occurs in chunks.



Experiment 1	
Configuration	batch_size=100, workers=6, data_kb=0.2
Maximum lag value	505.6 MB
Lag increase period	134.1 s
Lag increase rate	3.77 MB/s
Lag decrease rate	31.58 MB/s

Figure 13: Results of Experiment 1 – WAL lag over time and benchmark metrics

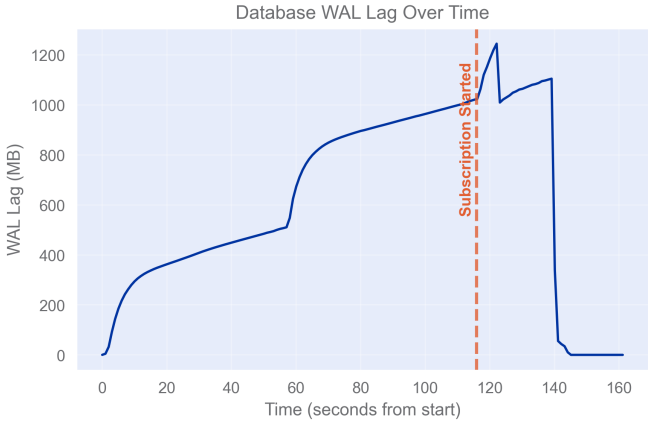
Experiment 2	
Configuration	batch_size=1200, workers=6, data_kb=0.2
Maximum lag value	3733.5 MB
Lag increase period	150.0 s
Lag increase rate	25.03 MB/s
Lag decrease rate	9.80 MB/s

Figure 14: Results of Experiment 2 – WAL lag over time and benchmark metrics

- The effective replication LSN speed is the sum of the 'lag increase rate' and the 'lag decrease rate,' as the lag continues to increase during the replication phase. This results in an effective replication speed of about **35 MB/s**. Consequently, the maximum LSN generation speed the replica can handle is about **30 MB/s**.

Experiment 3	
Configuration	batch_size=1200, workers=9, data_kb=0.2
Maximum lag value	4409.5 MB
Lag increase period	152.2 s
Lag increase rate	28.98 MB/s
Lag decrease rate	N/A

Figure 15: Results of Experiment 3 – WAL lag over time and benchmark metrics



Experiment 4	
Configuration	sysbench oltp read write, table_size=100,000, tables=12, threads=16
Maximum lag value	1244.8 MB
Lag increase period	122.1 s
Lag increase rate	10.19 MB/s
Lag decrease rate	30.08 MB/s

Figure 16: Results of Experiment 4 – WAL lag over time and benchmark metrics

Another interesting phenomenon observed during testing is that creating replication slots while the instance is under heavy load can take between 20-30 seconds.

10 CONCLUSION

The prototype successfully demonstrated a viable procedure for near zero-downtime PostgreSQL upgrades, reducing downtime to approximately one second. The chosen "upgrade the primary" approach, combined with the use of PgBouncer for transaction drainage and NetApp snapshots for temporary replicas, proved to be a practical and effective solution tailored to the DBOD environment.

While the procedure offers a substantial improvement in database availability for mission-critical systems at CERN, it's important to acknowledge its inherent complexity requirements. The implementation and maintenance of this system demand a deep understanding of PostgreSQL internals, logical replication, PgBouncer and network configuration (e.g., nft rules). Furthermore, it strongly benefits from a automated framework to orchestrate the numerous, precise steps involved, like for example handling sequence counters. This complexity introduces many fail points that can be very easily overlooked. The successful outcome of this project demonstrates that with sufficient engineering effort and automation, near zero-downtime upgrades are achievable.

References

[1] pganalyze. (2021). PostgreSQL Zero-Downtime Upgrades with Logical Replication. Available at: https://pganalyze.com/blog/5mins-postgres-zero-downtime-upgrades-logical-replication

[2] GitLab. (2023). How we execute PostgreSQL major upgrades at GitLab with zero downtime. Available at: https://www.postgresql.eu/events/pgconfeu2023/schedule/session/4791-how-we-execute-postgresql-major-upgrades-at-gitlab-with-zero-downtime/

[3] Instacart. (2021). Zero-downtime PostgreSQL cutovers at Instacart. Available at: https://www.instacart.com/company/how-its-made/zero-downtime-PostgreSQL-cutovers/

[4] Gadget. (2023). Zero-downtime PostgreSQL upgrades using logical replication. Available at: https://gadget.dev/blog/zero-downtime-postgres-upgrades-using-logical-replication

[5] PostgreSQL documentation. Available at: https://www.postgresql.org/docs/