How to Upgrade Major PostgreSQL Versions: A Practical Production Guide

PostgreSQL versions follow a well-defined five-year support lifecycle. Each major release receives bug fixes, security patches, and minor updates for five years from its initial release date. After that point, the version reaches end-of-life (EOL) and no longer receives official updates.

Staying on an EOL version exposes your systems to security risks, potential compatibility issues, and missing performance improvements introduced in later releases. You can always check the current support status of PostgreSQL versions on the official PostgreSQL Versioning Policy page.

Upgrading to the latest version ensures long-term stability, access to new features, and better support. Recently, I worked on upgrading a critical production PostgreSQL environment from version 11 to 15. Version 15 was chosen because the client's application had only been tested up to that release. The system supported large batch workloads and live applications, so we had to be meticulous. While this article draws from that specific project, the steps are broadly applicable to anyone planning a major PostgreSQL upgrade, especially when crossing several versions.

This guide outlines a generalized, production-ready approach for performing major version upgrades using the pg_dump/pg_restore method.

Upgrade Methods

PostgreSQL provides two primary upgrade options, each with distinct advantages.

1. In-place upgrade using pg_upgrade
This method is designed for rapid transitions and minimal downtime. It upgrades the system catalog in place and reuses existing data files, making it highly efficient. However, it requires careful compatibility checks, especially around tablespaces, file system layout, and extensions.

2. Logical upgrade using pg_dump and pg_restore
This method involves exporting the database schema and data from the old cluster and importing them into a new one. While it involves longer downtime and more disk I/O, it avoids binary compatibility issues and is well-suited for multi-version jumps and cross-platform migrations.

If you have a downtime window and are upgrading across multiple versions, the dump/restore method is often the simpler and safer path. In our case, we had a one-day downtime window and also needed to migrate to a new server, so using the pg_dump/pg_restore method was the most practical and reliable approach. It gave us full control over the migration process and allowed us to verify the restored data and performance on the new instance before final cutover.

Pre-Upgrade Preparation

A major PostgreSQL version upgrade can be performed either on the same host or by migrating to a different server. In our case, we opted for a two-server setup:

- **Source**: PostgreSQL 11 (actively serving the application)
- **Target**: PostgreSQL 15 (fresh install on a separate server)
  At the time of migration, the application was actively connected to the PostgreSQL 11 instance. The goal of this upgrade was to migrate the database from version 11 to 15 on a new server. The migration was carried out on Red Hat Enterprise Linux 9, though the overall approach can be adapted to other operating systems depending on your environment and tooling.

Stop Application

Prior to the upgrade, all client connections, batch jobs, and scheduled processes must be stopped. This guarantees a consistent state and prevents any post-dump changes from being lost. Application access to the source database should be disabled entirely for the duration of the backup.

Prepare Target Server

If PostgreSQL is not yet installed on the target server, you'll need to set it up before proceeding with the upgrade. The following instructions demonstrate how to install PostgreSQL 15 on a Red Hat 9 system. You may adjust the version number as needed based on your upgrade target.

First, install the official PostgreSQL repository and disable the system's default PostgreSQL module:

```
sudo dnf install -y https://download.postgresql.org/pub/repos/yum/reporpms/EL-9-x86_64/pgdg-redhat-repo-latest.noarch.rpm

sudo dnf -qy module disable postgresqlCopy to Clipboard
```

Next, install the PostgreSQL server package for your desired version:

```
sudo dnf install -y postgresql15-serverCopy to Clipboard
```

Initialize the database cluster and configure the service to start automatically on boot:

```
sudo /usr/pgsql-15/bin/postgresql-15-setup initdb

sudo systemctl enable postgresql-15

sudo systemctl start postgresql-15Copy to Clipboard
```

Some applications rely on specific PostgreSQL extensions that must be installed on the target server prior to restoration. During the restore process, you may encounter warnings or errors if these extensions were present in the source database but are missing from the target environment.

In our case, the only extension we were using was pg_stat_statements, which did not impact the restore itself and could safely be added afterward. However, if your application or schema depends on certain extensions (for custom functions, data types, or triggers), it's important to ensure those extensions are available before the restore begins to avoid failures or broken dependencies.

Take Dump – Using the target version tools

To ensure compatibility during a major version upgrade, it is strongly recommended to use the pg_dump and pg_dumpall binaries from the target PostgreSQL version (in our case, version 15). This helps avoid potential issues that can arise from using outdated dump formats when restoring to a newer server. If the target binaries are not already available on the source (older) server, you can install just the client tools without the server package using the following command:

```
sudo dnf install -y postgresql15
```

If installing PostgreSQL 15 tools on the source server is not possible due to system constraints or compatibility issues, you can run the dump commands remotely from the target server (or any server that has PostgreSQL 15 binaries installed), using the -h flag to connect to the source database over the network. In our scenario, we encountered compatibility issues while trying to install PostgreSQL 15 tools on the production server. Instead, we executed both dump commands remotely from the target Red Hat 9 server using PostgreSQL 15 binaries. This approach worked reliably after setting a password for the postgres user to allow authenticated remote access.

Export the main database using custom format:

```
/usr/pgsql-15/bin/pg_dump -Fc -h <source-host> -U postgres -d <database> -f
/path/to/backup.dump
```

The custom format is recommended because it allows greater control during restoration such as selective restores and parallelism, etc. Note that backup time will vary depending on database size and hardware. In our case, backing up an 800 GB database took approximately two hours on moderately provisioned infrastructure.

Next, export global objects such as roles, tablespaces, and ownership metadata separately:

```
/usr/pgsql-15/bin/pg_dumpall -g -h <source-host> -U postgres > /path/to/globals.sql
```

Once the backup is complete, copy both files to the target server. Additionally, store a copy of both on a separate host (outside of the source and target environments) to serve as a recovery fallback in case of unexpected failure during the upgrade process.

Upgrade Execution Plan

Once the backup files have been transferred to the target server and validated, proceed with the following steps to complete the database restoration.

Begin by restoring global objects such as roles, tablespaces, and their associated privileges. These were captured using pg_dumpall -g and are essential for preserving access control and ownership:

```
psql -U postgres -f /path/to/globals.sql
```

Next, create a fresh, empty database with the same name as the original source database:

```
createdb -U postgres <database>
```

With the database shell in place, restore the main database dump using pg_restore. For improved performance, enable parallel restore mode using the -j flag, as this can greatly speed up the process. The number of parallel jobs should be adjusted based on available CPU and I/O capacity on the target system:

```
nohup pg_restore -U postgres -d <database> -j 4 -v /path/to/backup.dump > restore.log 2>&1 &
```

Using nohup allows the command to continue running in the background even if the terminal session is closed. The -v flag enables verbose output, and restore.log captures both standard output and error messages for review.

Monitor the restore.log file to track progress and check for any errors during the restoration process. Depending on the database size and server resources, this step can take significant time. In our case, the restore of an 800 GB dump completed in approximately 2.5 hours.

After the restoration is complete, run ANALYZE on the database to refresh PostgreSQL's planner statistics. This ensures the query planner can make informed decisions based on the current data distribution:

```
psql -U postgres -d <database> -c "ANALYZE;"
```

Install Required Extensions

For extensions provided by PostgreSQL's contrib modules like pg_stat_statements you must first install the appropriate package.

```
sudo dnf install -y postgresql15-contrib
```

Next, configure PostgreSQL to preload the extension by modifying the postgresql.conf file:

```
shared_preload_libraries = 'pg_stat_statements'Copy to Clipboard
```

After updating the configuration, restart the PostgreSQL service for changes to take effect:

```
sudo systemctl restart postgresql-15Copy to Clipboard
```

Finally, enable the extension within the database:

```
psql -U postgres -d <database> -c "CREATE EXTENSION pg_stat_statements;"Copy to Clipboard
```

To verify that the extension is successfully installed and active, connect to the database and run:

```
\dxCopy to Clipboard
```

This command lists all extensions installed in the current database. You should see pg_stat_statements or any others you've enabled in the output.

Validate Schema and Structural Integrity

After restoring the database, it is important to validate that the schema and object structure match the original environment. Start by verifying that the number and types of database objects (tables, indexes, views, etc.) match the expected counts. To do this effectively, ensure that you have captured and stored the corresponding object counts from the original production database (source version) prior to the upgrade.

You can run queries like the following to review object distributions by type:

```
 SELECT
    n.nspname AS schema_name,
    CASE
        WHEN c.relkind = 'r' THEN 'TABLE'
        WHEN c.relkind = 'i' THEN 'INDEX'
        WHEN c.relkind = 'S' THEN 'SEQUENCE'
        WHEN c.relkind = 't' THEN 'TOAST TABLE'
        WHEN c.relkind = 'v' THEN 'VIEW'
        WHEN c.relkind = 'm' THEN 'MATERIALIZED VIEW'
        WHEN c.relkind = 'c' THEN 'COMPOSITE TYPE'
        WHEN c.relkind = 'f' THEN 'FOREIGN TABLE'
        WHEN c.relkind = 'p' THEN 'PARTITIONED TABLE'
        WHEN c.relkind = 'I' THEN 'PARTITIONED INDEX'
        ELSE 'OTHER'
    END AS object_type,
    COUNT(*) AS count

FROM
    pg_class c
JOIN
    pg_namespace n ON c.relnamespace = n.oid
WHERE
    n.nspname IN ('public')
GROUP BY
    n.nspname, object_type
ORDER BY
    n.nspname, object_type;

Copy to Clipboard
```

This query aggregates object counts grouped by schema and object type using pg_class and pg_namespace. By default, the WHERE clause filters for the public schema. You can either replace 'public' with a specific schema name you want to inspect or remove the WHERE clause entirely to include all schemas.

You may also run count checks on critical tables and compare key constraint definitions. Be aware that some catalog-level differences between PostgreSQL versions may lead to minor, expected variations in metadata.

Application Cutover

Once the database has been restored, validated, and tested, the final step is to point the application to the new PostgreSQL server.

Update the application’s connection strings or service configurations to reference the new database host, port, and credentials. On the PostgreSQL side, update the pg_hba.conf file to allow connections from the application hosts, ensuring that appropriate authentication methods are used. Also verify the listen_addresses and port settings in postgresql.conf to confirm that the database is accessible from external systems

Rollback Strategy

Having a rollback plan is essential for any major database upgrade. The rollback approach will differ depending on whether the issue occurs during the upgrade window or after the application has gone live on the new system.

If Issues Occur During the Upgrade Window

Plan A: Redirect the application back to the original PostgreSQL production server. Since no writes would have taken place on the new server at this stage, this provides a clean and immediate fallback with minimal risk.

Plan B: If the original production server is inaccessible or compromised, restore the most recent logical backup (ideally stored on a separate, secure host) to a recovery server. This ensures that a known, consistent version of the database remains accessible, even in the event of infrastructure failure.

If Issues Occur After Go-Live (Writes Have Occurred)

Plan A: Resolve the issue directly on the new PostgreSQL instance. This is the preferred approach, as it preserves any new data written since the cutover and avoids complex recovery operations.

Plan B: Revert to the old PostgreSQL server. This is a last-resort option and involves identifying and manually transferring any data that was created or modified on the new server back to the old environment. This process is time-consuming and introduces risk, and should only be considered when all other remediation efforts have failed.

Mitigation Strategy

To reduce the risk of data inconsistency and simplify rollback procedures, it is advisable to initially run the new PostgreSQL instance in read-only mode after the upgrade. This allows for application-level validation in a production-like environment without making irreversible changes. Once the application has been fully tested and confirmed stable, read/write access can be enabled, completing the transition.

Summary

Upgrading between major PostgreSQL versions requires careful planning. Always test your upgrade process in a staging environment before performing it on production. In our upgrade from PostgreSQL 11 to 15, we prioritized safety and transparency. This approach allowed us to validate the schema, minimize risk, and transition to a supported version with confidence.

Choose the upgrade method that best aligns with your environment, downtime tolerance, and operational requirements.