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.

1. In-place upgrade using pg_upgrade

Upgrading PostgreSQL major version using `pg_upgrade`

Upgrading PostgreSQL from version 14 to 15 can be done using `pg_upgrade`, which is a utility that allows for fast upgrades by reusing the existing data files.

Here's a brief overview of the **` — link` and ` — clone`** options and how to use them:

`pg_upgrade` Overview

`pg_upgrade` works by leveraging the existing data files in your old PostgreSQL cluster to speed up the upgrade process. Instead of copying data, it can either link to the existing files or clone them.

Why upgrade is required?

- Security fixes
- Bugfixes
- Performance improvements
- New features

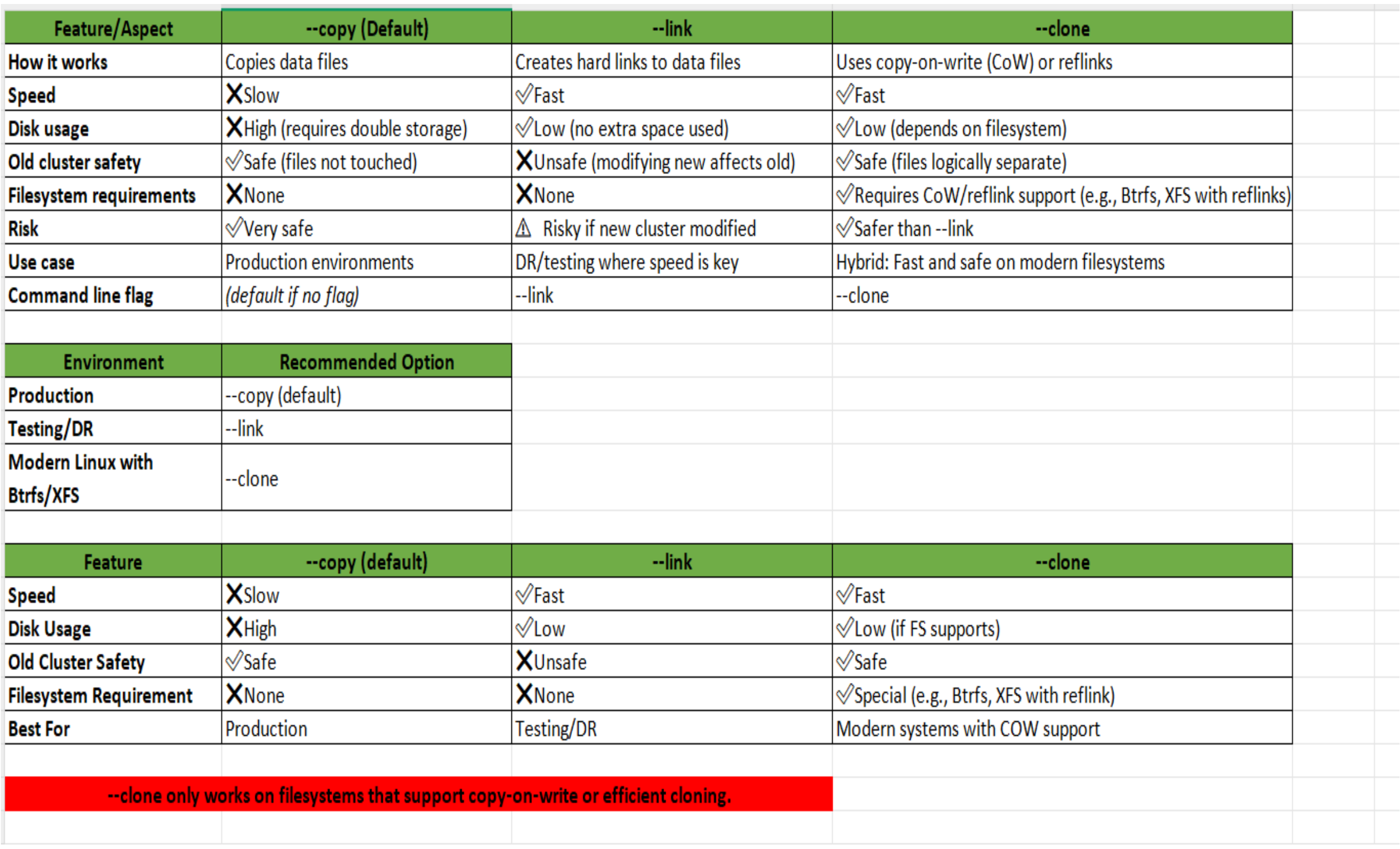
Types of Upgrade:
- Minor upgrade: 13.1 -> 13.5
- Major upgrade: 13.1 -> 15.2

pg_upgrade workflow:

1. install new major binaries
2. initdb — initialize the new cluster
3. shut down the old cluster
4. run pg_upgrade
5. start the new cluster

Before upgrade:

- Read release notes: incompatibilities must be addressed before pg_upgrade.
- Try pg_upgrade — check: if there are any problems reported — fix them.
- Make a backup database using barman/pgbackrest/pg_dump/pg_dumpall and also restored test.
- Check extentions whether it is compatible.



Feature/Aspect	--copy (Default)	--link	--clone
How it works	Copies data files	Creates hard links to data files	Uses copy-on-write (CoW) or reflinks
Speed	✗Slow	✓Fast	✓Fast
Disk usage	✗High (requires double storage)	✓Low (no extra space used)	✓Low (depends on filesystem)
Old cluster safety	✓Safe (files not touched)	✗Unsafe (modifying new affects old)	✓Safe (files logically separate)
Filesystem requirements	✗None	✗None	✓Requires CoW/reflink support (e.g., Btrfs, XFS with reflinks)
Risk	✓Very safe	⚠ Risky if new cluster modified	✓Safer than --link
Use case	Production environments	DR/testing where speed is key	Hybrid: Fast and safe on modern filesystems
Command line flag	*(default if no flag)*	--link	--clone

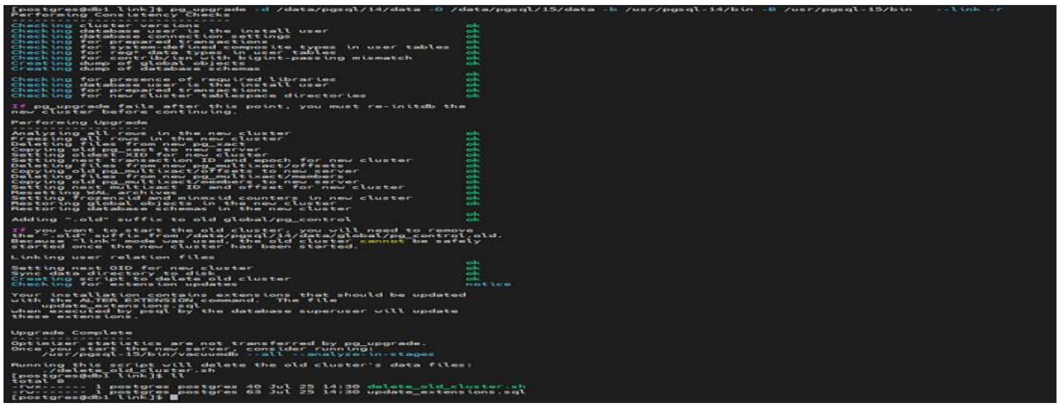
Environment	Recommended Option
Production	--copy (default)
Testing/DR	--link
Modern Linux with Btrfs/XFS	--clone

Feature	--copy (default)	--link	--clone
Speed	✗Slow	✓Fast	✓Fast
Disk Usage	✗High	✓Low	✓Low (if FS supports)
Old Cluster Safety	✓Safe	✗Unsafe	✓Safe
Filesystem Requirement	✗None	✗None	✓Special (e.g., Btrfs, XFS with reflink)
Best For	Production	Testing/DR	Modern systems with COW support

--clone only works on filesystems that support copy-on-write or efficient cloning.

Option	Description
--old-datadir=DIR	Path to the **old PostgreSQL data directory** (e.g., /var/lib/pgsql/13/data).
--new-datadir=DIR	Path to the **new PostgreSQL data directory** (e.g., /var/lib/pgsql/15/data).
--old-bindir=DIR	Path to the **old PostgreSQL binaries** (e.g., /usr/pgsql-13/bin).
--new-bindir=DIR	Path to the **new PostgreSQL binaries** (e.g., /usr/pgsql-15/bin).
--check	Run a **dry-run check** to see if upgrade is possible. No actual changes are made.
--link	Use **hard links** instead of copying files, which makes the upgrade faster and saves disk space. **Risk**: corrupts old cluster if new one is modified.
--clone	Like --link, but uses copy_file_range() syscall or reflinks (safer on supported systems).
--jobs=N	Use **parallel jobs** to speed up upgrade. Good for large databases (e.g., --jobs=4).
--verbose	Outputs detailed logs for debugging or audit purposes.
--username=NAME	Database superuser to use for connecting (default is postgres).
--socketdir=DIR	Location of the Unix socket file (if not the default).
--old-options='OPTIONS'	Additional options passed to the **old cluster's postgres process**.
--new-options='OPTIONS'	Additional options passed to the **new cluster's postgres process**.
--retain	Retain temporary files (e.g., pg_upgrade_dump.*) generated during upgrade. Useful for debugging.
--debug	Enable debugging mode — forces pg_upgrade to pause at key steps so you can manually inspect.
--disable-link	Explicitly disables hard linking even if the system supports it.
--no-sync	Skips syncing data to disk after upgrade. **Faster** but **risky** if system crashes.
--check	Repeated here to emphasize: it's the **safe pre-upgrade validation** step.

Using`pg_upgrade --link`



```
[postgres@db1 link]$ pg_upgrade -d /data/pgsql/14/data -D /data/pgsql/15/data -b /usr/pgsql-14/bin -B /usr/pgsql-15/bin   --link -r
Performing Consistency Checks
-----------------------------
Checking cluster versions                                   ok
Checking database user is the install user                  ok
Checking database connection settings                       ok
Checking for prepared transactions                          ok
Checking for system-defined composite types in user tables  ok
Checking for reg* data types in user tables                 ok
Checking for contrib/isn with bigint-passing mismatch       ok
Creating dump of global objects                             ok
Creating dump of database schemas
                                                            ok
Checking for presence of required libraries                 ok
Checking database user is the install user                  ok
Checking for prepared transactions                          ok
Checking for new cluster tablespace directories             ok

If pg_upgrade fails after this point, you must re-initdb the
new cluster before continuing.

Performing Upgrade
------------------
Analyzing all rows in the new cluster                       ok
Freezing all rows in the new cluster                        ok
Deleting files from new pg_xact                             ok
Copying old pg_xact to new server                           ok
Setting oldest XID for new cluster                          ok
Setting next transaction ID and epoch for new cluster       ok
Deleting files from new pg_multixact/offsets                ok
Copying old pg_multixact/offsets to new server              ok
Deleting files from new pg_multixact/members                ok
Copying old pg_multixact/members to new server              ok
Setting next multixact ID and offset for new cluster        ok
Resetting WAL archives                                      ok
Setting frozenxid and minmxid counters in new cluster       ok
Restoring global objects in the new cluster                 ok
Restoring database schemas in the new cluster
                                                            ok
Adding ".old" suffix to old global/pg_control               ok

If you want to start the old cluster, you will need to remove
the ".old" suffix from /data/pgsql/14/data/global/pg_control.old.
Because "link" mode was used, the old cluster cannot be safely
started once the new cluster has been started.

Linking user relation files
                                                            ok
Setting next OID for new cluster                            ok
Sync data directory to disk                                 ok
Creating script to delete old cluster                       ok
Checking for extension updates                              notice

Your installation contains extensions that should be updated
with the ALTER EXTENSION command.  The file
    update_extensions.sql
when executed by psql by the database superuser will update
these extensions.


Upgrade Complete
----------------
Optimizer statistics are not transferred by pg_upgrade.
Once you start the new server, consider running:
    /usr/pgsql-15/bin/vacuumdb --all --analyze-in-stages

Running this script will delete the old cluster's data files:
    ./delete_old_cluster.sh
[postgres@db1 link]$ ll
total 8
-rwx------ 1 postgres postgres 40 Jul 25 14:30 delete_old_cluster.sh
-rw------- 1 postgres postgres 63 Jul 25 14:30 update_extensions.sql
[postgres@db1 link]$ 
```

What it does:

- **- With the ` — link` option**, `pg_upgrade` creates hard links between the old and new data directories. This means that both the old and new clusters share the same data files, avoiding the need to duplicate data on disk.

Advantages:

- - Speed: Linking is faster because it avoids copying data files
- .
  - Disk Space: It conserves disk space since the data files are not duplicated.

Disadvantages:

- - Limited Use Case: This option is only available if the data directories are on the same filesystem. If the new cluster is on a different filesystem, this option cannot be used.
- - Risk: If the new cluster or the old cluster's data files are corrupted or deleted, it could affect both clusters due to the shared data.
- --link tells pg_upgrade not to copy the data files, but instead to create hard links from the old cluster's data directory to the new cluster's data directory.
- A hard link means both directories point to the same physical file on disk. There's no duplicate — just two directory entries for the same underlying data.
- If you delete the old cluster's data directory after upgrading with --link, the files themselves will be deleted from disk (because the last hard link will be removed).
- That will cause corruption or complete data loss in the new cluster, because the files it depends on will be gone.

How to use it:

1. Stop the PostgreSQL server for version 14.

2. Initialize the new PostgreSQL 15 cluster.

3. Run `pg_upgrade` with the ` — link` option also with version 15, it means pg_upgrade verson should be higher:

- mkdir -p /data/pgsql/15/data   # Create a data directory for postgresql 15
- sudo /usr/pgsql-15/bin/postgresql-15-setup initdb   # Should have a initialized data files in
- /data/pgsql/15/data, and this path should be configured in service file
- pg_upgrade --link -r -d /data/pgsql/14/data -D /data/pgsql/15/data -b /usr/pgsql-14/bin -B /usr/pgsql-15/bin   --check   # Used for dry run
- pg_upgrade --link   -r   -d /data/pgsql/14/data -D /data/pgsql/15/data -b /usr/pgsql-14/bin -B /usr/pgsql-15/bin   # Used for final execution

--link
Tells pg_upgrade to use hard links instead of copying data files.
? Much faster, but dangerous if you later remove old cluster, since both clusters share same files.

-r
Runs in rollback mode. If the upgrade fails, this flag ensures the old cluster can still be used safely. (It doesn’t perform the upgrade; it just prepares rollback safety.)

-d /data/pgsql/14/data
Location of the old cluster’s data directory (PostgreSQL 14 in your case).

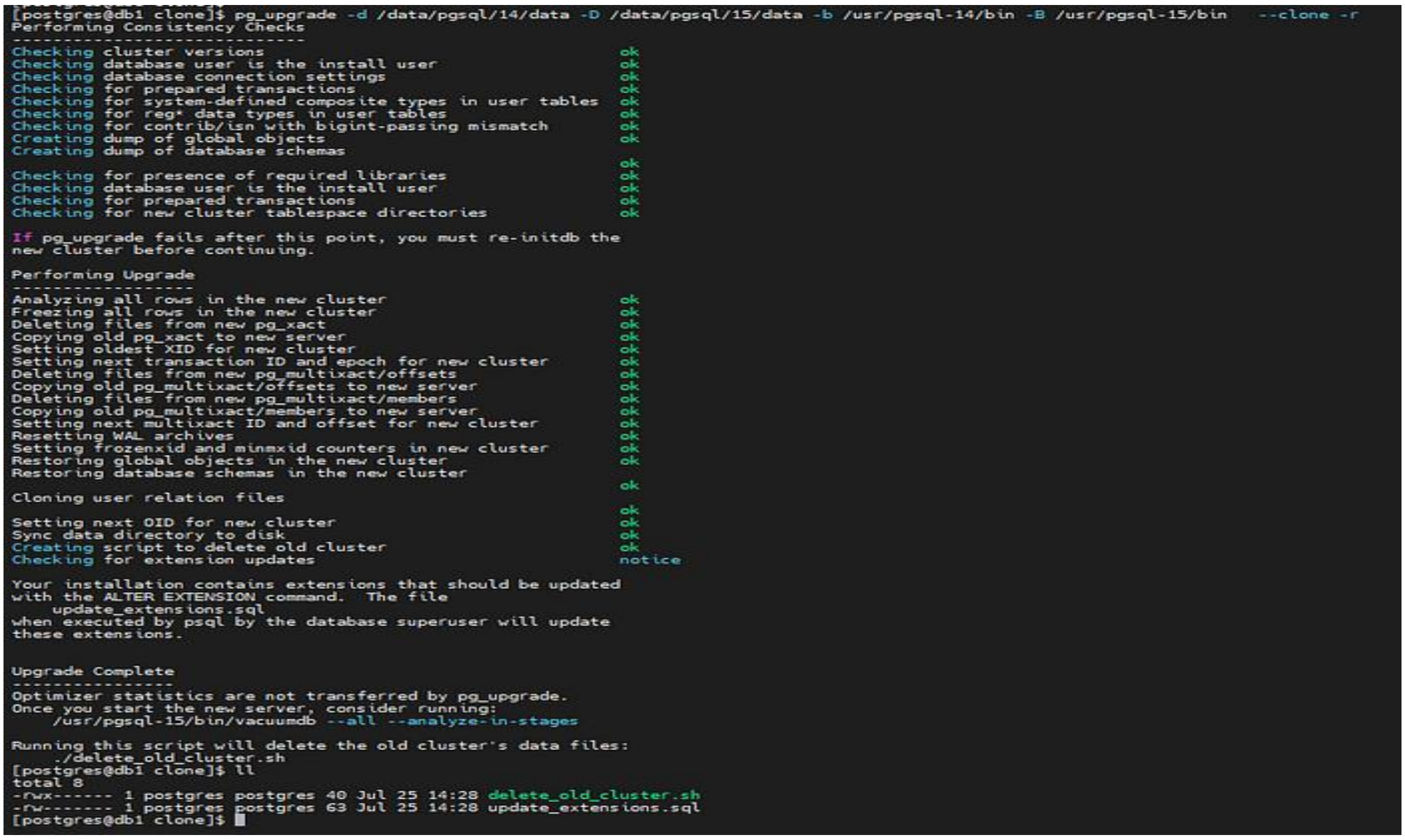
-D /data/pgsql/15/data
Location of the new cluster’s data directory (PostgreSQL 15, empty before running upgrade).

-b /usr/pgsql-14/bin
Path to old version’s binaries (Postgres 14 executables).

-B /usr/pgsql-15/bin
Path to new version’s binaries (Postgres 15 executables).

--check
Runs in check-only mode. It will not actually perform the upgrade, but only validate compatibility between old and new clusters.

Using `pg_upgrade — clone`



```
[postgres@db1 clone]$ pg_upgrade -d /data/pgsql/14/data -D /data/pgsql/15/data -b /usr/pgsql-14/bin -B /usr/pgsql-15/bin   --clone -r
Performing Consistency Checks
-----------------------------
Checking cluster versions                                   ok
Checking database user is the install user                  ok
Checking database connection settings                       ok
Checking for prepared transactions                          ok
Checking for system-defined composite types in user tables  ok
Checking for reg* data types in user tables                 ok
Checking for contrib/isn with bigint-passing mismatch       ok
Creating dump of global objects                             ok
Creating dump of database schemas
                                                            ok
Checking for presence of required libraries                 ok
Checking database user is the install user                  ok
Checking for prepared transactions                          ok
Checking for new cluster tablespace directories             ok

If pg_upgrade fails after this point, you must re-initdb the
new cluster before continuing.

Performing Upgrade
------------------
Analyzing all rows in the new cluster                       ok
Freezing all rows in the new cluster                        ok
Deleting files from new pg_xact                             ok
Copying old pg_xact to new server                           ok
Setting oldest XID for new cluster                          ok
Setting next transaction ID and epoch for new cluster       ok
Deleting files from new pg_multixact/offsets                ok
Copying old pg_multixact/offsets to new server              ok
Deleting files from new pg_multixact/members                ok
Copying old pg_multixact/members to new server              ok
Setting next multixact ID and offset for new cluster        ok
Resetting WAL archives                                      ok
Setting frozenxid and minmxid counters in new cluster       ok
Restoring global objects in the new cluster                 ok
Restoring database schemas in the new cluster
                                                            ok
Cloning user relation files
                                                            ok
Setting next OID for new cluster                            ok
Sync data directory to disk                                 ok
Creating script to delete old cluster                       ok
Checking for extension updates                              notice

Your installation contains extensions that should be updated
with the ALTER EXTENSION command.  The file
    update_extensions.sql
when executed by psql by the database superuser will update
these extensions.


Upgrade Complete
----------------
Optimizer statistics are not transferred by pg_upgrade.
Once you start the new server, consider running:
    /usr/pgsql-15/bin/vacuumdb --all --analyze-in-stages

Running this script will delete the old cluster's data files:
    ./delete_old_cluster.sh
[postgres@db1 clone]$ ll
total 8
-rwx------ 1 postgres postgres 40 Jul 25 14:28 delete_old_cluster.sh
-rw------- 1 postgres postgres 63 Jul 25 14:28 update_extensions.sql
[postgres@db1 clone]$
```

What it does:

- With the ` — clone` option, `pg_upgrade` creates a copy of the old data directory for the new cluster. This means the new cluster will have its own separate set of data files.

Advantages:

- Independence: The new cluster's data files are independent of the old cluster's files, which reduces the risk of data corruption affecting both clusters.

- Filesystem Flexibility: You can use this option even if the old and new clusters are on different filesystems.

Disadvantages:
- Speed: Cloning takes longer than linking because it involves copying data files.

- Disk Space: It requires more disk space as data files are duplicated.

How to use it:

1. Stop the PostgreSQL server for version 14.
2. Initialize the new PostgreSQL 15 cluster.
3. Run `pg_upgrade` with the ` — clone` option also with version 15, it means pg_upgrade verson should be higher:

- mkdir -p /data/pgsql/15/data   # Create a data directory for postgresql 15
- sudo /usr/pgsql-15/bin/postgresql-15-setup initdb   # Should have a initialized data files in
- /data/pgsql/15/data, and this path should be configured in service file
- pg_upgrade --clone -r -d /data/pgsql/14/data -D /data/pgsql/15/data -b /usr/pgsql-14/bin -B /usr/pgsql-15/bin --check   # Used for dry run
- pg_upgrade --clone -r -d /data/pgsql/14/data -D /data/pgsql/15/data -b /usr/pgsql-14/bin -B /usr/pgsql-15/bin   # Used for final execution

--clone

Performs the upgrade by cloning the data from the old cluster to the new one using hard links, instead of copying all files.

Faster and uses less disk space, but the old cluster must remain untouched (read-only) during the upgrade.

-r or --link

This is actually redundant here, because --clone already implies using hard links.

Avoids physically copying files; only links them.

-d /data/pgsql/14/data

The data directory of the old PostgreSQL cluster (version 14 in this case).

-D /data/pgsql/15/data

The data directory for the new PostgreSQL cluster (version 15).

-b /usr/pgsql-14/bin

Binary path for the old PostgreSQL version.

-B /usr/pgsql-15/bin

Binary path for the new PostgreSQL version.

--check

Runs pre-upgrade checks only.

Verifies that the upgrade can succeed without actually performing it.

Summary:

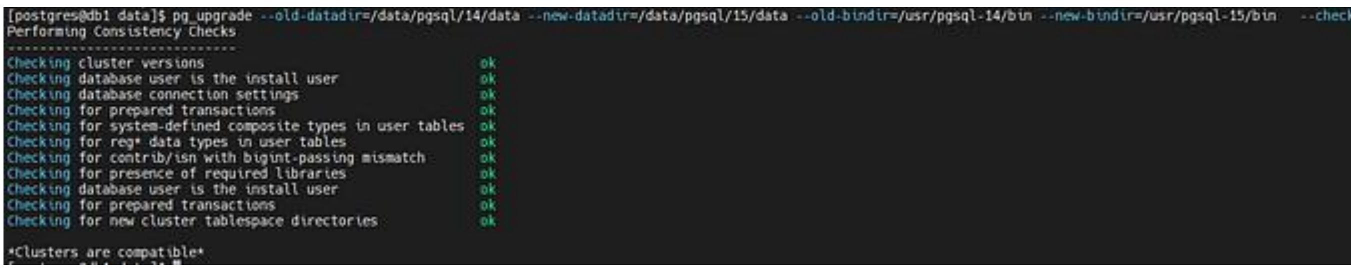
Use ` — link` if you want a faster upgrade and are okay with the old and new clusters sharing the same filesystem.

- Use ` — clone` if you need the new cluster to have its own set of data files or if the old and new data directories are on different filesystems.

Note:

Always make sure to perform a full backup of your data before starting the upgrade process to protect against any unexpected issues.

2. For dry run pelase try with — check option before execuition above command.



```
[postgres@db1 data]$ pg_upgrade --old-datadir=/data/pgsql/14/data --new-datadir=/data/pgsql/15/data --old-bindir=/usr/pgsql-14/bin --new-bindir=/usr/pgsql-15/bin   --check
Performing Consistency Checks
-----------------------------
Checking cluster versions                                   ok
Checking database user is the install user                  ok
Checking database connection settings                       ok
Checking for prepared transactions                          ok
Checking for system-defined composite types in user tables  ok
Checking for reg* data types in user tables                 ok
Checking for contrib/isn with bigint-passing mismatch       ok
Checking for presence of required libraries                 ok
Checking database user is the install user                  ok
Checking for prepared transactions                          ok
Checking for new cluster tablespace directories             ok

*Clusters are compatible*
```

Option	Description
--old-datadir=DIR	Path to the **old PostgreSQL data directory** (e.g., /var/lib/pgsql/13/data).
--new-datadir=DIR	Path to the **new PostgreSQL data directory** (e.g., /var/lib/pgsql/15/data).
--old-bindir=DIR	Path to the **old PostgreSQL binaries** (e.g., /usr/pgsql-13/bin).
--new-bindir=DIR	Path to the **new PostgreSQL binaries** (e.g., /usr/pgsql-15/bin).
--check	Run a **dry-run check** to see if upgrade is possible. No actual changes are made.
--link	Use **hard links** instead of copying files, which makes the upgrade faster and saves disk space. **Risk**: corrupts old cluster if new one is modified.
--clone	Like --link, but uses copy_file_range() syscall or reflinks (safer on supported systems).
--jobs=N	Use **parallel jobs** to speed up upgrade. Good for large databases (e.g., --jobs=4).
--verbose	Outputs detailed logs for debugging or audit purposes.
--username=NAME	Database superuser to use for connecting (default is postgres).
--socketdir=DIR	Location of the Unix socket file (if not the default).
--old-options='OPTIONS'	Additional options passed to the **old cluster's postgres process**.
--new-options='OPTIONS'	Additional options passed to the **new cluster's postgres process**.
--retain	Retain temporary files (e.g., pg_upgrade_dump.*) generated during upgrade. Useful for debugging.
--debug	Enable debugging mode — forces pg_upgrade to pause at key steps so you can manually inspect.
--disable-link	Explicitly disables hard linking even if the system supports it.

--no-sync	Skips syncing data to disk after upgrade. **Faster** but **risky** if system crashes.
--check	Repeated here to emphasize: it's the **safe pre-upgrade validation** step.

2. Logical upgrade using pg_dump and pg_restore

- **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;"
```

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;
```

- 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.