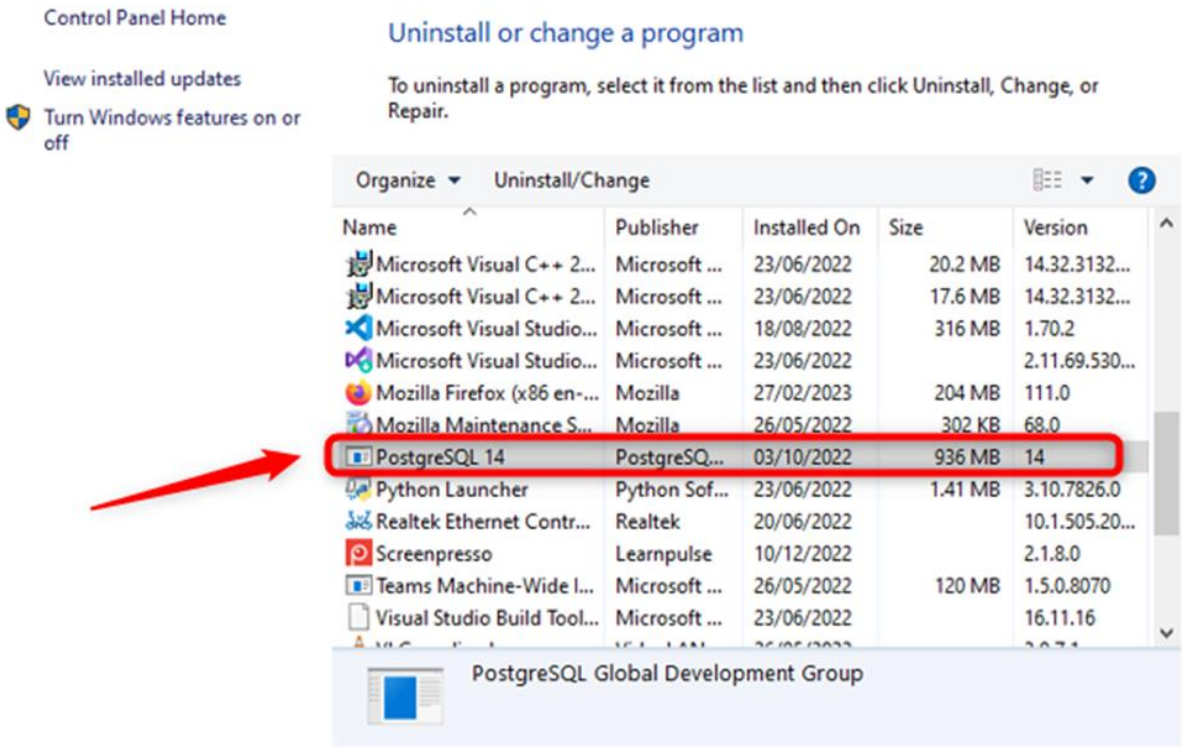
Mukesh Chaurasia – SQL DBA

COURSE: POSTGRESQL DATABASE ADMINISTRATOR

DAY – 9

TOPIC: PostgreSQL Upgrade & Uninstallation Process



Detailed guide to upgrading PostgreSQL, covering why upgrades matter, how to prepare, and step-by-step instructions for both minor and major version upgrades.

🚀 Why Upgrade PostgreSQL?

Upgrading your PostgreSQL database is essential for:

- **Security**: New versions patch vulnerabilities.
- **Performance**: Improved indexing, query planning, and memory usage.
- **Features**: New data types (e.g., jsonb), SQL enhancements, and better concurrency.
- **Support**: Older versions reach End of Life (EOL), losing updates and fixes.

🧭 Types of PostgreSQL Upgrades

Upgrade Type	Description	Method
Minor	Patch-level updates (e.g., 16.4 → 16.5)	Replace binaries, restart server
Major	Version jump (e.g., 14 → 16)	Use pg_upgrade, pg_dump, or replication

Minor upgrades are simple and backward-compatible. Major upgrades may involve changes to internal storage formats and system catalogs.

🧰 Pre-Upgrade Checklist

1. **Read Release Notes** Review all changes and migration notes for the target version.
2. **Check Extension Compatibility** Ensure all extensions (e.g., PostGIS, pg_stat_statements) support the new version.
3. **Test in a Staging Environment** Validate application behavior and performance before production rollout.
4. **Backup Your Data** Use:

```
pg_dumpall > alldb.bak
pg_dump dbname > dbname.bak
```

5. **Plan for Downtime** Schedule during off-peak hours or use replication for near-zero downtime.

🛠️ Step-by-Step: Major Upgrade Using pg_upgrade

1. Install New PostgreSQL Version

Download and install the new binaries from postgresql.org.

2. Initialize New Cluster

```
/usr/local/pgsql/bin/initdb -D /usr/local/pgsql/data_new
```

3. Run pg_upgrade

```
/usr/local/pgsql/bin/pg_upgrade \
 -b /usr/lib/postgresql/14/bin \
 -B /usr/lib/postgresql/16/bin \
```

```
-d /var/lib/postgresql/14/main \
-D /var/lib/postgresql/16/main \
-o "-c config_file=/etc/postgresql/14/main/postgresql.conf" \
-O "-c config_file=/etc/postgresql/16/main/postgresql.conf"
```

4. Analyze and Reindex

```
./analyze_new_cluster.sh
./delete_old_cluster.sh
```

🧪 **Alternative: Dump and Restore**

Use when pg_upgrade isn't viable (e.g., cross-platform migration):

```
pg_dump -Fc dbname > dbname.dump
pg_restore -d newdb dbname.dump
```

🔄 **Near-Zero Downtime Option**

Use logical replication:

1. Set up a replica on the new version.
2. Stream changes using pglogical or native replication.
3. Switch traffic once synced.

Learn more

🧩 **Post-Upgrade Tasks**

- Verify application functionality.
- Monitor logs for errors or performance issues.
- Keep the old version temporarily for rollback if needed.

⚠️ **Common Challenges in PostgreSQL Upgrades**

1. Deprecated Features

- **Issue**: Older PostgreSQL versions may rely on features that are removed or changed in newer releases.
- **Example**: Implicit casts removed in PostgreSQL 8.3 broke many legacy queries.

- **Solution**: Review release notes and test queries for compatibility before upgrading.

2. Downtime Risk

- **Issue**: Upgrades can cause service interruptions.
- **Solution**: Use pg_upgrade or logical replication for minimal downtime. Always schedule upgrades during off-peak hours.

```
pg_upgrade -b /usr/local/pgsql.old/bin -B /usr/local/pgsql.new/bin \
-d /usr/local/pgsql.old/data -D /usr/local/pgsql.new/data \
-p 5432 -P 5433
```

3. Compatibility Issues

- **Issue**: SQL behavior, data types, or extensions may change.
- **Example**: JSON vs. JSONB—older versions stored JSON as plain text, while newer versions validate structure and support indexing.
- **Solution**: Validate application logic and extension compatibility in a staging environment.

4. Data Loss

- **Issue**: Improper upgrade procedures can corrupt or delete data.
- **Solution**: Always back up using pg_dump, pg_dumpall, or filesystem snapshots. Test restore procedures.

5. Performance Regressions

- **Issue**: New versions may introduce changes that degrade performance for certain workloads.
- **Solution**: Benchmark queries before and after upgrade. Tune configuration parameters for the new version.

✅ **Benefits of Upgrading PostgreSQL**

Upgrading unlocks powerful features and performance improvements. For example, PostgreSQL 13 introduced:

- **Improved Query Planning**: Better statistics for complex filters.
- **Parallel Vacuuming**: Faster table maintenance.

```
VACUUM (PARALLEL 4) customers;
```

- **Incremental Sorting**: Efficient multi-key sorting.

```
SELECT name FROM employees ORDER BY salary DESC, name ASC;
```

🧭 Step-by-Step Upgrade Guide

Here's a high-level upgrade workflow:

🔷 Preparation

1. Read release notes
2. Audit extensions and custom functions
3. Back up your data:

```
pg_dumpall > alldb.bak
```

🔷 Installation

4. Install new PostgreSQL binaries
5. Initialize new cluster:

```
initdb -D /new/data
```

🔷 Migration

6. Use pg_upgrade:

```
pg_upgrade -b /old/bin -B /new/bin -d /old/data -D /new/data
```

7. Run post-upgrade scripts:

```
./analyze_new_cluster.sh
./delete_old_cluster.sh
```

🔷 Validation

8. Test application behavior
9. Monitor logs and performance
10. Keep old version temporarily for rollback

⚠️ Common Challenges in PostgreSQL Upgrades

1. Deprecated Features

- **Issue**: Older PostgreSQL versions may rely on features that are removed or changed in newer releases.
- **Example**: Implicit casts removed in PostgreSQL 8.3 broke many legacy queries.
- **Solution**: Review release notes and test queries for compatibility before upgrading.

2. Downtime Risk

- **Issue**: Upgrades can cause service interruptions.
- **Solution**: Use pg_upgrade or logical replication for minimal downtime. Always schedule upgrades during off-peak hours.

```
pg_upgrade -b /usr/local/pgsql.old/bin -B /usr/local/pgsql.new/bin \
-d /usr/local/pgsql.old/data -D /usr/local/pgsql.new/data \
-p 5432 -P 5433
```

3. Compatibility Issues

- **Issue**: SQL behavior, data types, or extensions may change.
- **Example**: JSON vs. JSONB—older versions stored JSON as plain text, while newer versions validate structure and support indexing.
- **Solution**: Validate application logic and extension compatibility in a staging environment.

4. Data Loss

- **Issue**: Improper upgrade procedures can corrupt or delete data.
- **Solution**: Always back up using pg_dump, pg_dumpall, or filesystem snapshots. Test restore procedures.

5. Performance Regressions

- **Issue**: New versions may introduce changes that degrade performance for certain workloads.
- **Solution**: Benchmark queries before and after upgrade. Tune configuration parameters for the new version.

✅ **Benefits of Upgrading PostgreSQL**

Upgrading unlocks powerful features and performance improvements. For example, PostgreSQL 13 introduced:

- **Improved Query Planning**: Better statistics for complex filters.
- **Parallel Vacuuming**: Faster table maintenance.

```
VACUUM (PARALLEL 4) customers;
```

- **Incremental Sorting**: Efficient multi-key sorting.

```
SELECT name FROM employees ORDER BY salary DESC, name ASC;
```

🧭 Step-by-Step Upgrade Guide

Here's a high-level upgrade workflow:

🔷 Preparation

1. Read release notes
2. Audit extensions and custom functions
3. Back up your data:

```
pg_dumpall > alldb.bak
```

🔷 Installation

4. Install new PostgreSQL binaries
5. Initialize new cluster:

```
initdb -D /new/data
```

🔷 Migration

6. Use pg_upgrade:

```
pg_upgrade -b /old/bin -B /new/bin -d /old/data -D /new/data
```

7. Run post-upgrade scripts:

```
./analyze_new_cluster.sh
./delete_old_cluster.sh
```

🔷 Validation

8. Test application behavior
9. Monitor logs and performance
10. Keep old version temporarily for rollback

📚 Recommended Guides

- Percona: Common PostgreSQL Upgrade Errors and How to Avoid Them
- Reintech: Best Practices and Pitfalls
- SQL-Easy: Comprehensive Upgrade Guide

I used ClusterControl to deploy a PostgreSQL 11 database cluster for a single node. The following have been tested on Centos 7 and Ubuntu Focal (20.04.1):

$ /usr/pgsql-11/bin/postgres --version

postgres (PostgreSQL) 11.13

postgres=# dx

List of installed extensions

```
      Name       | Version |   Schema   |                     Description
-----------------------+---------+------------+------------------------------------------------------------
 fuzzystrmatch        | 1.1     | public     | determine similarities and distance between strings
 pg_stat_statements   | 1.6     | public     | track execution statistics of all SQL statements executed
 plpgsql              | 1.0     | pg_catalog | PL/pgSQL procedural language
 postgis              | 3.1.4   | public     | PostGIS geometry and geography spatial types and functions
 postgis_raster       | 3.1.4   | public     | PostGIS raster types and functions
 postgis_sfcgal       | 3.1.4   | public     | PostGIS SFCGAL functions
 postgis_tiger_geocoder | 3.1.4 | tiger      | PostGIS tiger geocoder and reverse geocoder
 postgis_topology     | 3.1.4   | topology   | PostGIS topology spatial types and functions
 timescaledb          | 2.3.1   | public     | Enables scalable inserts and complex queries for time-series data
(9 rows)
```

So I got the following,

PostgreSQL server version: 11.13

TimescaleDB version: 2.3.1

PostGIS version: 3.1.4

If you want to test this with ClusterControl, there are two ways to have TimescaleDB.

Setup the required repositories

First, let's add the PostgreSQL repository.

For CentOS/RHEL/Oracle Linux

You have to make sure that you have the right repository. For Enterprise Linux (EL) 7, you can do the following:

sudo yum -y install https://download.postgresql.org/pub/repos/yum/reporpms/EL-7-x86_64/pgdg-redhat-repo-latest.noarch.rpm

Let's add the TimescaleDB repository as well.

vi /etc/yum.repos.d/timescale_timescaledb.repo

Then add the following contents for this file,

[timescale_timescaledb]

name=timescale_timescaledb

baseurl=https://packagecloud.io/timescale/timescaledb/el/7/$basearch

repo_gpgcheck=1

gpgcheck=0

enabled=1

gpgkey=https://packagecloud.io/timescale/timescaledb/gpgkey

sslverify=1

sslcacert=/etc/pki/tls/certs/ca-bundle.crt

metadata_expire=300

Just replace the baseurl accordingly if you are using a version other than EL 7.

For Ubuntu/Debian

Add the PG repository for Ubuntu Focal:

deb http://apt.postgresql.org/pub/repos/apt/ focal-pgdg main

For other Ubuntu/Debian distributions, just replace the focal accordingly, which you can find here http://apt.postgresql.org/pub/repos/apt/dists/. For example, replace focal-pgdg with buster-pgdg.

Now, let's add the repository for TimescaleDB,

sudo sh -c "echo 'deb [signed-by=/usr/share/keyrings/timescale.keyring] https://packagecloud.io/timescale/timescaledb/ubuntu/ $(lsb_release -c -s) main' > /etc/apt/sources.list.d/timescaledb.list"

Import the keyring,

wget --quiet -O - https://packagecloud.io/timescale/timescaledb/gpgkey | sudo gpg --dearmor -o /usr/share/keyrings/timescale.keyring

and update the package lists for upgrades for packages that need upgrading, as well as new packages that have just come to the repositories.

sudo apt-get update

You can replace the sub-directory in the URL if you are using Debian from ubuntu.

Now that we have the repository ready, we're good to go.

Install PostgreSQL version 13 with TimescaleDB and PostGIS

Installing PostgreSQL 13 can be done on the same host. First, you must make sure things such as the database port are unique. In other words, it has to be different from the current PostgreSQL 11 installed on the same host.

For CentOS/RHEL/Oracle Linux

Run the command below to install PostgreSQL 13 and its dependent packages:

yum install postgresql13.x86_64 postgresql13-server.x86_64 postgresql13-contrib.x86_64 postgresql13-libs.x86_64

Then initialize the database cluster and its required collection of databases by running the command below:

$ /usr/pgsql-13/bin/postgresql-13-setup initdb

At this point, there should be two data directories both for PG 11 and PG 13:

[root@pupnode28 ~]# ls -alth /var/lib/pgsql/* -d

drwx------. 4 postgres postgres 51 Sep 22 14:19 /var/lib/pgsql/13

drwx------. 4 postgres postgres 33 Sep 21 18:53 /var/lib/pgsql/11

Now that we're good with PostgreSQL 13 let's install TimescaleDB. We need to make sure that the plugin to be installed is the same version on PostreSQL 11.

Take note that, in order to make sure that pg_upgrade will work smoothly, the plugins of your source and major version destination should be the same version. This is because pg_upgrade will look for its designated libraries linked to the plugins or extensions that have been loaded or used by your old or source database version of your PostgreSQL. You can verify this in your Enterprise Linux by running showduplicates or by verifying with info just like below either with dnf or yum:

$ yum --showduplicates list timescaledb_13.x86_64 timescaledb-2-postgresql-13.x86_64 timescaledb-2-oss-postgresql-13.x86_64 timescaledb-2-loader-postgresql-13.x86_64|grep '2.3.1'

Repository pgdg-common is listed more than once in the configuration

timescaledb-2-loader-postgresql-13.x86_64 2.3.1-0.el7 timescale_timescaledb

timescaledb-2-oss-postgresql-13.x86_64 2.3.1-0.el7 timescale_timescaledb

timescaledb-2-postgresql-13.x86_64 2.3.1-0.el7 timescale_timescaledb

Or verify it with the info option:

$ yum info timescaledb-2-loader-postgresql-13-2.3.1-0.el7.x86_64 timescaledb-2-postgresql-13-2.3.1-0.el7.x86_64

Now, we're ready to install the TimescaleDB package for the PG 13 version.

$ yum install timescaledb-2-loader-postgresql-13-2.3.1-0.el7.x86_64 timescaledb-2-postgresql-13-2.3.1-0.el7.x86_64

After you've installed it, you can try to run timescaledb-tune tool to tune up your postgresql.conf configuration file. Just run the command below:

$ timescaledb-tune --pg-config=/usr/pgsql-13/bin/pg_config

Now, let's install the PostGIS package for the PG 13 version as well.

$ yum install -y postgis31_13.x86_64

For Ubuntu/Debian

Simply run:

$ apt install postgresql-client-13 postgresql-13

The great thing with Ubuntu/Debian distributions is that there are tools for PostgreSQL that are very handy for managing your PostgreSQL clusters, such as pg_lsclusters, pg_ctlcluster, etc.

You can verify your available clusters installed.

$ pg_lsclusters

Ver Cluster Port Status Owner Data directory Log file

11 main 5432 online postgres /var/lib/postgresql/11/main log/postgresql-%Y-%m-%d_%H%M%S.log

13 main 5433 online postgres /var/lib/postgresql/13/main /var/log/postgresql/postgresql-13-main.log

In Ubuntu/Debian, there's no need to change the port since it will be handled during the installation phase and it detects and sets it uniquely, accordingly.

Now, let's install TimescaleDB.

$ apt install timescaledb-2-loader-postgresql-13 timescaledb-2-postgresql-13

Optionally, you can run the timescaledb-tune tool to tune up your postgresql.conf configuration file by simply just invoking the tool as the following:

$ timescaledb-tune

Now, we're ready to install the PostGIS package for PG 13.

$ apt install postgresql-13-postgis-3-scripts postgresql-13-postgis-3

Review your postgresql.conf

It's always better to review your postgresql.conf configuration file. In Enterprise Linux versions, you can locate your postgresql.conf either in your data_directory or PGDATA path. Whereas for Ubuntu/Debian, you can locate it in /etc/postgresql///postgresql.conf. Make sure that in your postgresql.conf, the following lines are correctly configured:

shared_preload_libraries = 'pg_stat_statements,timescaledb' # pg_stat_statements is not required but if you are using ClusterControl, make sure this is appended.

port = 5532 # make sure that the port number is unique than the old version of your PostgreSQL

listen_address = * # depends on your setup but if you need to specify the available network interfaces to its IP addresses (IPv4 or IPv6) set it accordingly.

It's best practice to compare your old and new versions of your PostgreSQL configuration files to ensure that your postgresql.conf is identical to what is needed and set.

Before proceeding with the next step, we also need to check that your PostgreSQL version 13 is loaded accordingly. Make sure that you have the latest version acquired or installed in your host. Start the database and make sure that it starts and runs correctly.

To start in EL distributions, run the command below:

$ sudo -iu postgres /usr/pgsql-13/bin/pg_ctl -D /var/lib/pgsql/13/data/ -o "-c config_file=/var/lib/pgsql/13/data/postgresql.conf" start

or for Ubuntu/Debian, run the command below:

$ sudo -iu postgres /usr/lib/postgresql/13/bin/pg_ctl -D /var/lib/postgresql/13/main/ -o "-c config_file=/etc/postgresql/13/main/postgresql.conf" start

or use the pg_ctlcluster tool to start, restart, or stop your PG Cluster.

When ready, run pg_upgrade...

Before moving on, first make sure that you always have your backup from your old server ready and available. Always take a logical backup and physical backup as good practice before proceeding with a major upgrade.

Now that you're ready, you are good to run pg_upgrade. In practice, you have to initially run pg_upgrade with a check to determine the incompatibility and issues before proceeding to the main procedure of pg_upgrade. Before running the pg_upgrade, make sure that both PG 11 and PG 13 are down while doing this process. That just means downtime is needed for this process.

To do that, run the following command with –check option:

$ sudo -iu postgres /usr/lib/postgresql/13/bin/pg_upgrade -o "-c config_file=/etc/postgresql/11/main/postgresql.conf" --old-datadir=/var/lib/postgresql/11/main/ -O "-c config_file=/etc/postgresql/13/main/postgresql.conf" --new-datadir=/var/lib/postgresql/13/main/ --old-bindir=/usr/lib/postgresql/11/bin --new-bindir=/usr/lib/postgresql/13/bin --check

Replace the –old-datadir value or config_file accordingly if it's different from your setup.

This shall run compatibility checks just like the result below:

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 tables WITH OIDS ok

Checking for invalid "sql_identifier" user columns 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

If all checks signal "ok," that means a successful check, and the bottom message shows that clusters are compatible, then you should be good to go.

Finally, run the command again without the –check option:

$ sudo -iu postgres /usr/lib/postgresql/13/bin/pg_upgrade -o "-c config_file=/etc/postgresql/11/main/postgresql.conf" --old-datadir=/var/lib/postgresql/11/main/ -O "-c config_file=/etc/postgresql/13/main/postgresql.conf" --new-datadir=/var/lib/postgresql/13/main/ --old-bindir=/usr/lib/postgresql/11/bin --new-bindir=/usr/lib/postgresql/13/bin

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 tables WITH OIDS ok

Checking for invalid "sql_identifier" user columns 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 cluste ok

Copying user relation files ok

Setting next OID for new cluster ok

Sync data directory to disk ok

Creating script to analyze new cluster 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 so,

once you start the new server, consider running:

./analyze_new_cluster.sh

Running this script will delete the old cluster's data files:

./delete_old_cluster.sh

Depending on the size of your dataset, it might take a bit of time. In the example command above, it does copy the transaction logs, which are of physical files. However, if your disk size is tight, then you can use the -k or –link option, which uses hard links. If all goes well, it should provide you with messages such as the above notifying you with complete upgrade and notices to update the extensions you might have. In this setup, it goes smoothly. The update_extensions.sql contains only the following:

$ cat update_extensions.sql

connect postgres

ALTER EXTENSION "pg_stat_statements" UPDATE;

As you noticed, pg_upgrade does a series of actions. It analyzes all rows from the source cluster, copying transaction metadata logs and its multi-transaction status data and the rest.

Once you figured out that everything looks good, then run the analyze_new_cluster.sh script, which shall run

vacuumdb --all --analyze-only.

$ sudo -iu postgres PGPORT=5433 ./analyze_new_cluster.sh

Take note that you should specify the port of your PostgreSQL 13 so that vacuumdb will run correctly to the right target server.

In this case, the upgrade outcome with TimescaleDB and PostGIS extensions enabled goes well. Once all appears in order, make sure to start the server and do a series of tests and checks.

Test the upgrade process

Always test and review the upgrade process. Here are a few tables and a user-defined database, which contains hypertables and PostGIS tables that rely on using geometry and geography spatial types and functions.

$ sudo -iu postgres psql -p5433

psql (13.4 (Ubuntu 13.4-1.pgdg20.04+1))

Type "help" for help.

postgres=# l

List of databases

```
   Name    |  Owner   | Encoding | Collate |  Ctype  |   Access privileges
-----------+----------+----------+---------+---------+-----------------------
 mydb      | postgres | UTF8     | C.UTF-8 | C.UTF-8 |
 postgres  | postgres | UTF8     | C.UTF-8 | C.UTF-8 |
 template0 | postgres | UTF8     | C.UTF-8 | C.UTF-8 | =c/postgres          +
           |          |          |         |         | postgres=CTc/postgres
 template1 | postgres | UTF8     | C.UTF-8 | C.UTF-8 | postgres=CTc/postgres+
           |          |          |         |         | =c/postgres
(4 rows)
```

postgres=# c mydb

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

mydb=# set search_path="$user",public;

SET

mydb=# dt+

List of relations

Schema	Name	Type	Owner	Persistence	Size	Description
public	conditions	table	postgres	permanent	8192 bytes	
public	global_points	table	postgres	permanent	16 kB	
public	roads	table	postgres	permanent	16 kB	
public	sample_table	table	postgres	permanent	8192 bytes	
public	spatial_ref_sys	table	postgres	permanent	6968 kB	

(5 rows)

Checking some of my PostGIS and hypertables:

mydb=# d roads

Table "public.roads"

Column	Type	Collation	Nullable	Default
road_id	integer			
road_name	character varying			
roads_geom	geometry(LineString)			

mydb=# d sample_table

Table "public.sample_table"

Column	Type	Collation	Nullable	Default
id	integer		not null	nextval('sample_table_id_seq'::regclass)

time | timestamp with time zone | | not null |

name | character varying | | not null |

Indexes:

"sample_table_pkey" PRIMARY KEY, btree (id, "time")

"sample_table_time_idx" btree ("time" DESC)

Triggers:

ts_insert_blocker BEFORE INSERT ON sample_table FOR EACH ROW EXECUTE FUNCTION _timescaledb_internal.insert_blocker()

Number of child tables: 371 (Use d+ to list them.)

mydb=# d conditions

Table "public.conditions"

Column	Type	Collation	Nullable	Default
time	timestamp with time zone		not null	
location	text		not null	
location2	character(10)		not null	
temperature	double precision			
humidity	double precision			

Indexes:

"conditions_time_idx" btree ("time" DESC)

Triggers:

ts_insert_blocker BEFORE INSERT ON conditions FOR EACH ROW EXECUTE FUNCTION _timescaledb_internal.insert_blocker()

Number of child tables: 366 (Use d+ to list them.)

mydb=# select count(*) from sample_table;

```
 count
-------
  2588
(1 row)

mydb=# SELECT name FROM global_points WHERE ST_DWithin(location,
'SRID=4326;POINT(-110 29)'::geography, 1000000);
  name
--------
 Town
 Forest
(2 rows)

mydb=# SELECT n, ST_AsEWKT(ST_GeometryN(the_geom, n)) As geomewkt
mydb-# FROM (
mydb(# VALUES (ST_GeomFromEWKT('MULTIPOINT(1 2 7, 3 4 7, 5 6 7, 8 9 10)') ),
mydb(# ( ST_GeomFromEWKT('MULTICURVE(CIRCULARSTRING(2.5 2.5,4.5 2.5, 3.5
3.5), (10 11, 12 11))') )
mydb(# )As foo(the_geom)
mydb-# CROSS JOIN generate_series(1,100) n
mydb-# WHERE n <= ST_NumGeometries(the_geom);
 n |            geomewkt
---+---------------------------------------
 1 | POINT(1 2 7)
 1 | CIRCULARSTRING(2.5 2.5,4.5 2.5,3.5 3.5)
 2 | POINT(3 4 7)
 2 | LINESTRING(10 11,12 11)
 3 | POINT(5 6 7)
 4 | POINT(8 9 10)
(6 rows)
```

Now everything looks ready to serve as my new cluster.

Once you get things going, then you can run:

$ sudo -iu postgres ./delete_old_cluster.sh

Make sure you only run the script since this is a very dangerous script, as it will delete all files in your old PostgreSQL cluster.

Cluster overview

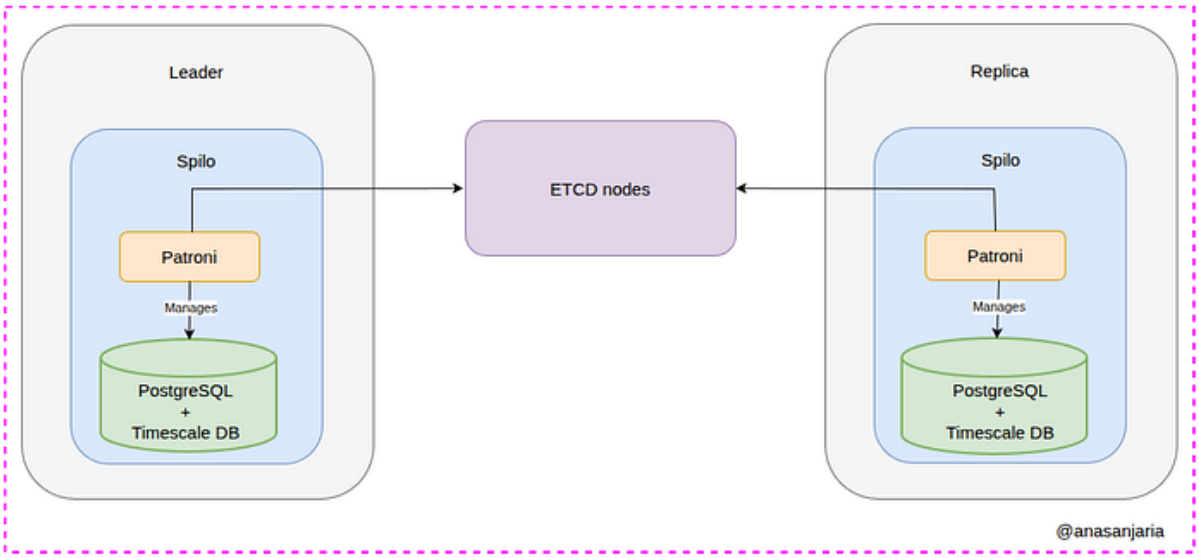
The setup involves a **2-node PostgreSQL cluster** using Zalando's Spilo, which bundles:

- **PostgreSQL + TimescaleDB** — For time-series data storage.
- **Patroni** — Manages the cluster state, handles failover, and ensures high availability.

Each node is either a **Leader** or a **Replica**, and both communicate with a set of **ETCD nodes** that serve as a distributed key-value store for cluster coordination.

Here's a high-level architecture diagram to visualize the setup:

Press enter or click to view image in full size



High-level overview of PostgreSQL cluster using Spilo

Note: While this example uses only 2 nodes for simplicity, the setup is extendable to larger clusters.

With that in place, let's explore the upgrade process step by step.

Checklist — If You're Doing This in a Production Setup

If you're planning a **major version upgrade** in a live environment, several precautions are critical — such as validating functionality, choosing the right upgrade window, and informing stakeholders.

Rather than repeating the full **checklist** here, I recommend referring to the Minor Version Upgrade Guide, which outlines a comprehensive pre-upgrade checklist suitable for both minor and major upgrades.

The only major difference: expect **longer downtime and more complexity** with a major version upgrade — especially for replica nodes.

High-level overview

📝 This section provides a high-level overview of the process without going into technical details. Detailed technical explanations are provided later in the post.

Major version upgrades involve distinct steps for leader and replica nodes, including:

1. Pre-upgrade processing.
2. Upgrade a leader node.
3. Upgrade a replica node.

4. Post-upgrade processing.

Pre-upgrade processing

Before upgrading, ensure that replica nodes can communicate with the leader node via SSH, and have an additional node for complete rollback or failure recovery.

Upgrade a leader node

Utilize PostgreSQL's pg_upgrade tool for the leader node upgrade. The process involves the following steps:

1. Disable all write activity on the leader node.
2. Temporarily step down patroni from managing the cluster.
3. Shutdown database (or Postgres service).
4. Initialize DB (initdb).
5. Copy configurations such as pg_hba.conf, postgres.conf etc.
6. Perform upgrade (pg_upgrade).
7. Please check the output of the upgrade for any warnings, it might guide you on things you need to take care of after the upgrade.
8. Adjust patroni configuration.
9. Wipe complete cluster state from DCS (Dynamic Configuration Settings).
10. Start Postgres service with target version.
11. Start patroni.

Congrats 🎉, you have successfully upgraded the leader node.

Upgrade a replica node

Unlike the leader node, replicas cannot use pg_upgrade. The basic idea for upgrading an existing replica node is to synchronize folder structure (data & Write Ahead Log (WAL) directories).

1. Shut down database (or Postgres service).
2. Synchronize data and Write Ahead Log (WAL) directories.
3. Adjust patroni configuration
4. Start up database (or Postgres service).
5. Configure replication.

Congrats 🎉, you have successfully upgraded the replica node.

Post-upgrade processing

Enable write activity on the leader node and rebuild statistics for tables.

Technical details

Close to zero downtime using in-place upgrade

Major version upgrades require downtime, but in-place upgrades minimize this downtime. Using pg_upgrade with the --link option reduces downtime to near zero by copying only metadata.

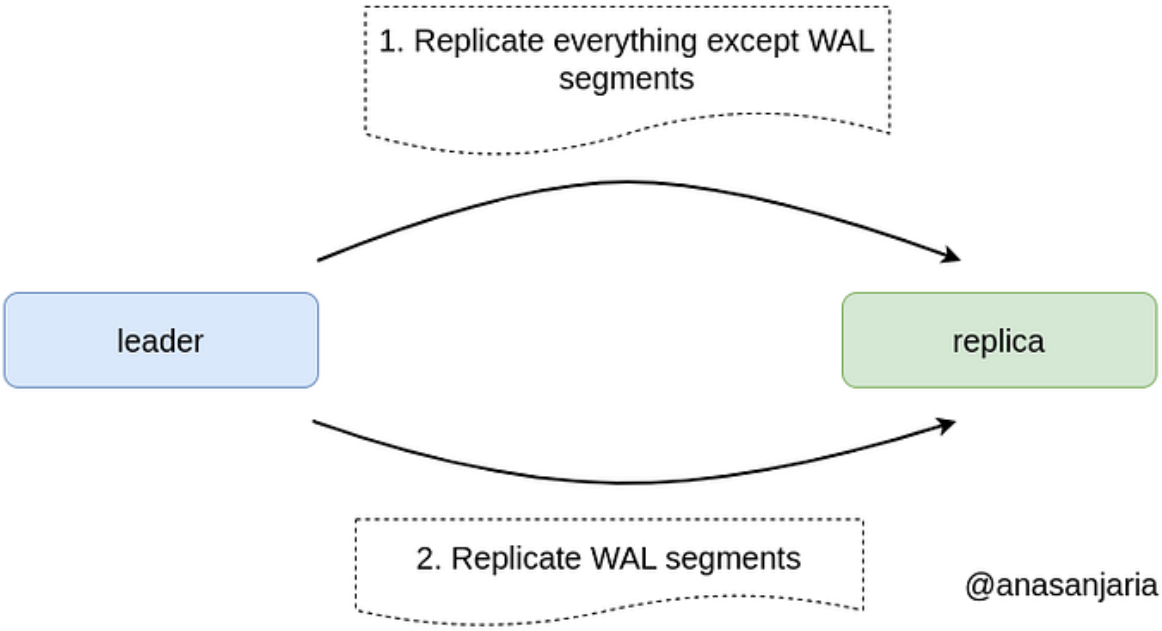
Tools	**Reloading data**	**Downtime needed**
`pg_upgrade`	Metadata and files are copied	Downtime is needed
`pg_upgrade --link`	Only metadata are copied	Close to zero

Source: Upgrading and updating PostgreSQL

Replica's Upgrade Process — Synchronization of Folder Structure

You can understand this concept if you are familiar with replication.

Press enter or click to view image in full size



High-level overview of a replication

When incorporating a new replica node into our cluster, the leader node performs the following steps:

1. **Initial Data Replication:** The leader node replicates everything except for WAL segments to the new replica.
2. **WAL Segment Replication:** Subsequently, it replicates WAL segments to the replica.

Synchronizing folder structures manually performs replication, effectively upgrading existing replica nodes.

Ensure Replica-Node Communication with Leader via SSH

Use rsync for folder synchronization, requiring SSH communication between replica and leader nodes.

You can use the following script to ensure successful communication.

```
#!/bin/bash

# Configuration
LEADER_IP="..."
LEADER_USER="..."

REPLICA_IP="..."
REPLICA_USER="..."

# generate key
ssh -i ~/.ssh/your.pem "$REPLICA_USER@$REPLICA_IP" \
 'sudo ssh-keygen -t rsa -b 2048 -f /root/.ssh/id_rsa -N ""'

# download generated key on your machine
rsync -avz --progress --rsync-path="sudo rsync" \
 -e "ssh -i ~/.ssh/your.pem" \
 "$REPLICA_USER@$REPLICA_IP:/root/.ssh/id_rsa.pub" "$PWD/replica-ssh-key.pub"

# copying it to a leader node
cat replica-ssh-key.pub | ssh -i ~/.ssh/your.pem "$LEADER_USER@$LEADER_IP" \
 'cat >> /home/ec2-user/.ssh/authorized_keys && echo "Key copied"'
```

⚠ You need to run this script from your machine.

Importance of Temporarily Stepping Down Patroni

Patroni manages cluster. It keeps track of a leader's health and act accordingly. It will perform a failover when a leader is unhealthy to ensure high availability.

As upgrading process makes leader unhealthy, we let Patroni to step down temporarily and don't do anything, and we're aware of the situation.

Upgrade in Action

This section provides practical insights and applies the aforementioned workflow to enhance understanding.

Source code: https://github.com/anasanjaria/blogs/tree/main/postgres-upgrade

For demonstration, assume a cluster of two nodes: one leader and one replica, running PostgreSQL version 13.x and upgrading to version 15.x.

📝 For the sake of similicity, I am running all the containers locally and persisting data on different directories.

Data for **node-1** & **node-2** are stored in **data-1** & **data-2** respectively.

Upgrading leader node

Execute the provided script to upgrade the leader node. The script guides through the process, requiring user input at various steps.

docker exec -it node-1 /opt/pgupgrade.sh

Upgrading replica node

Similarly, use the provided script to upgrade an existing replica node. Execute the script and follow the prompts for a successful upgrade.

./pgupgrade-replica.sh

Post upgrade process

Enabling write activity — Let's deactivate write lock now.

```
curl --request PATCH \
  --url http://localhost:8008/config \
  --header 'Content-Type: application/json' \
  --data '{
 "postgresql": {
  "parameters": {
   "default_transaction_read_only": "off"
  }
```

```
  }
 }'
```

Rebuilding statistics — Rebuilding statistics is extremely crucial to have better performance of your production system.

🔥 *If you skip this step, you will end up with extremely slow queries.*

Typically, analyze command rebuilds statistics, but it's extremely slow. Hence, it's not practical for big databases.

vacuumdb is another tool for the same purpose and also supports parallelism. Hence, the preferred way for production system.

```
/path/to/vacuumdb -d <DATABASE> \
 -t <TABLE-1> -t <TABLE-2> ... -t <TABLE-n> \
 --analyze-in-stages \
 -j <CONCURRENT_CONNECTIONS>
```

💡**TIP:** Use the same number of connections as the number of tables because a single connection is used per table.

Conclusion

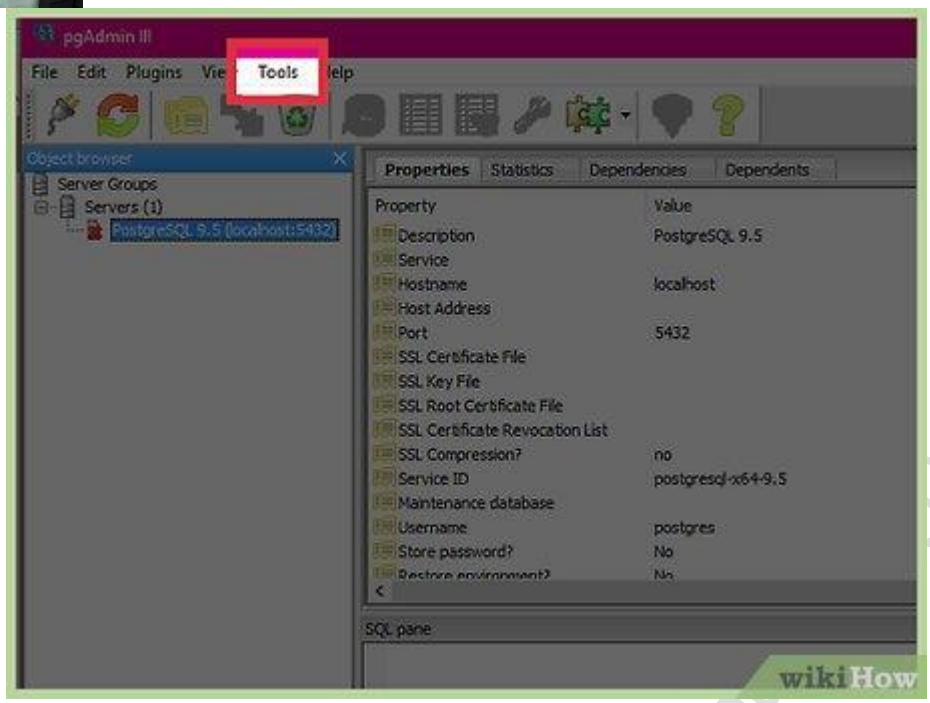
The successful upgrade of our production cluster marks a significant milestone in our journey towards maintaining a robust and reliable infrastructure.

By extending the upgrade process to include both leader and replica nodes, we've not only mitigated potential risks but also enhanced the stability and resilience of our critical services.

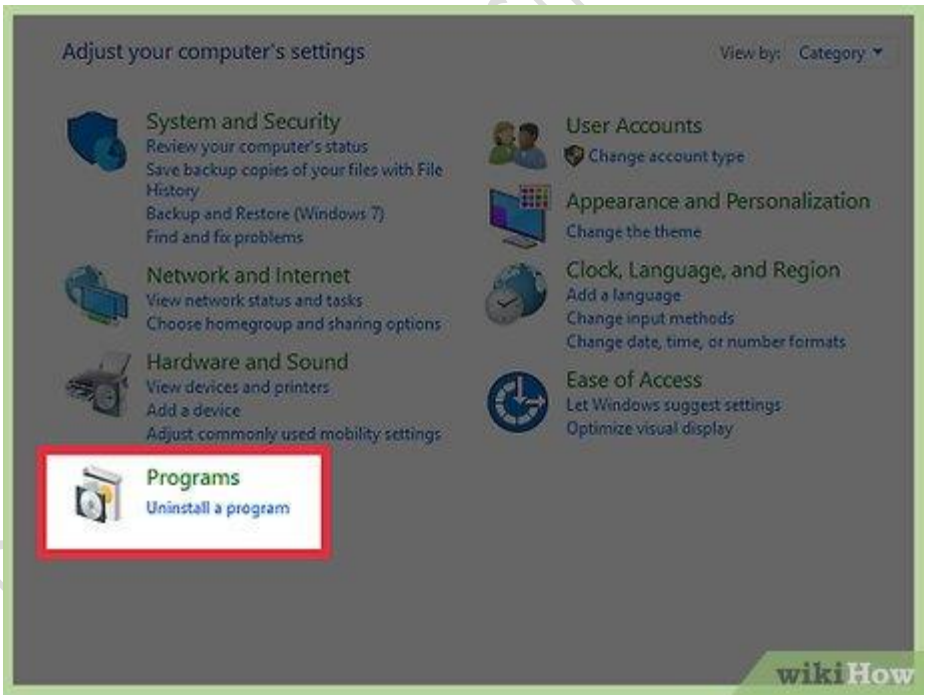
I hope you've found this journey as enlightening and empowering as I have. Here's to smooth upgrades, robust clusters, and endless possibilities! 🚀

How to Uninstall PostgreSQL

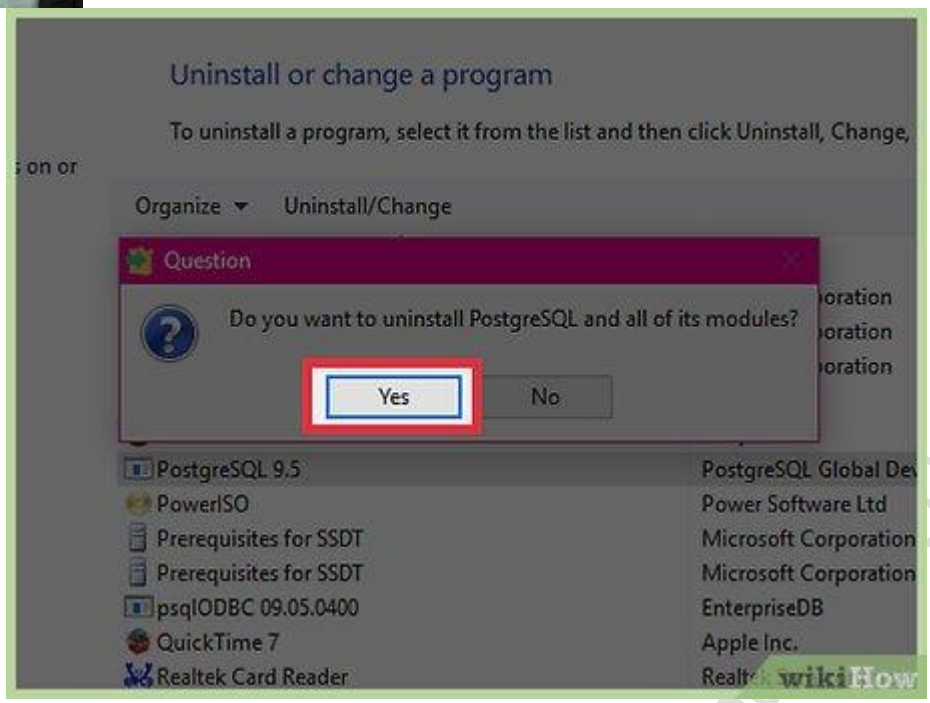
Steps

1. 

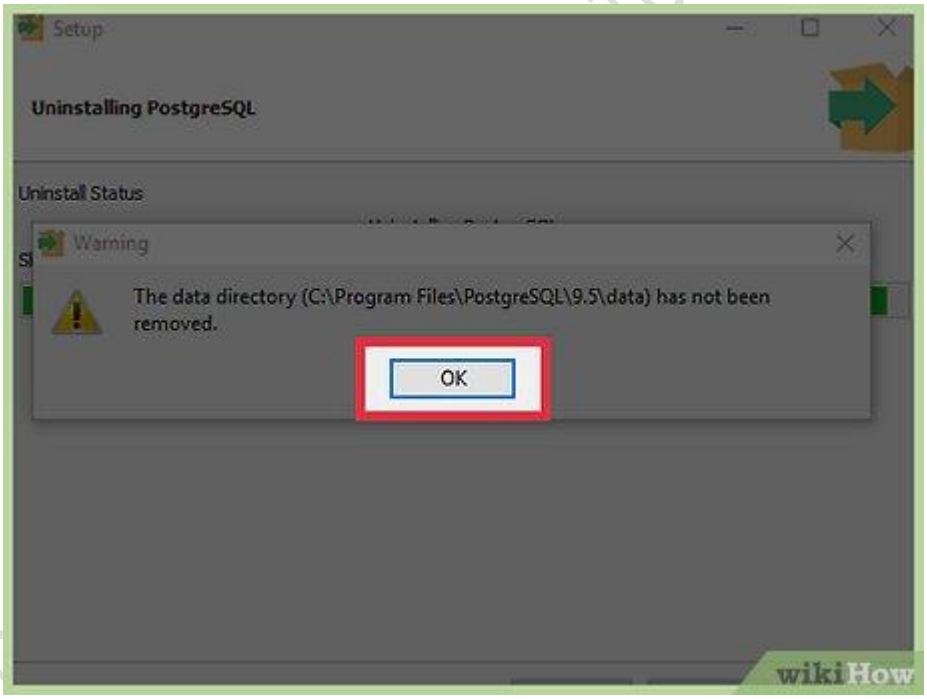
Prepare. Before you start to uninstall "P - SQL", you may need to save your personal settings and data files: open up pgAdmin III - Object browser, visit the Tools menu; scroll down and run the option "Backup...".

2. 

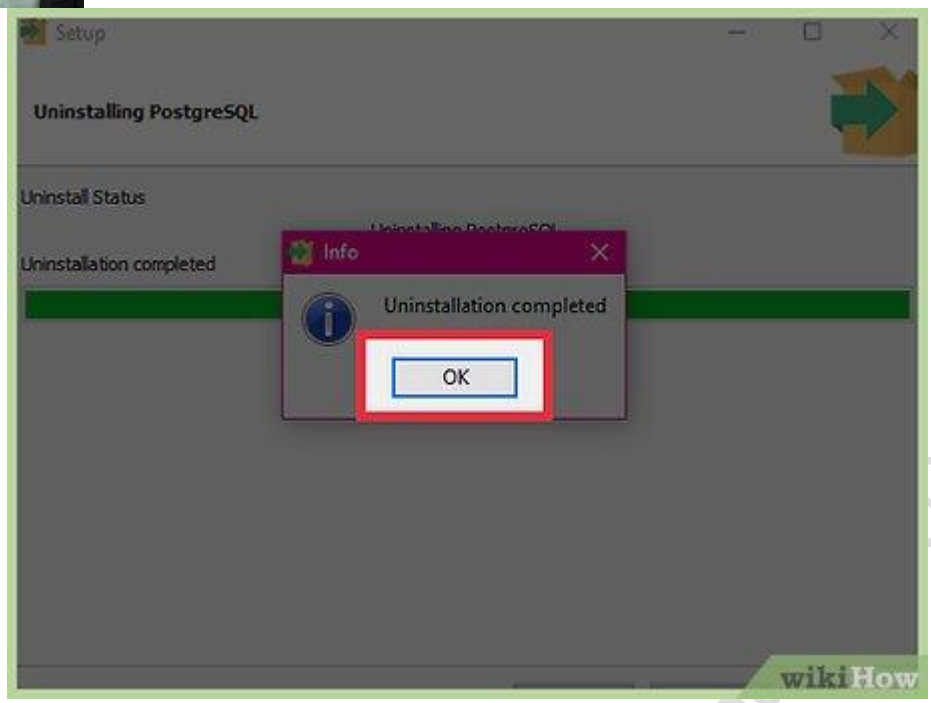
Launch the option "Chang/Remove" for PostgreSQL program, version 9.1 (size, 172.00MB).

3. 

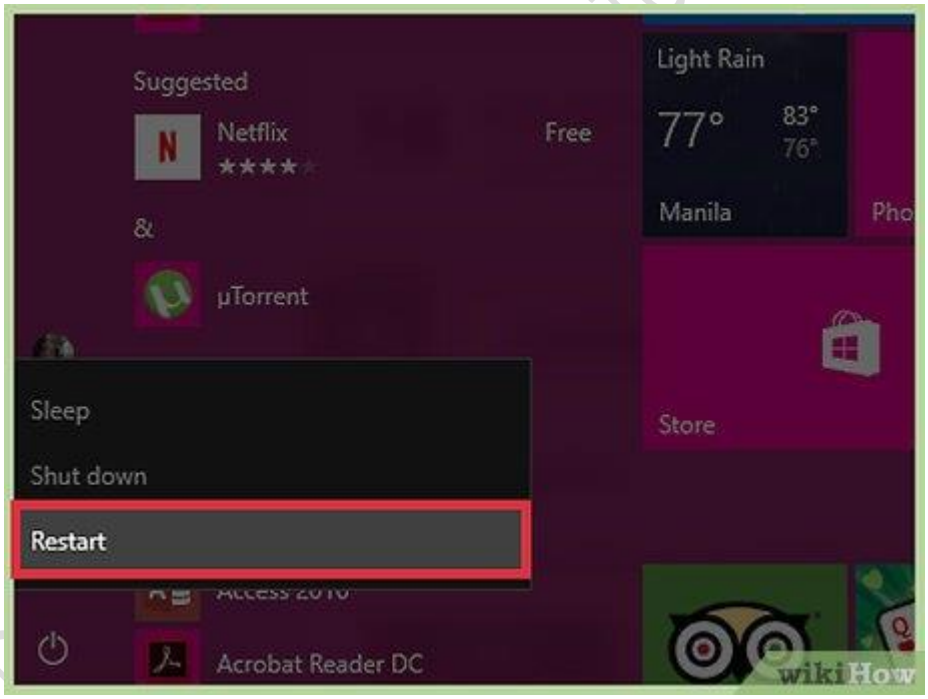
Click on the "Yes" button on the "Question" window and then wait.

4. 

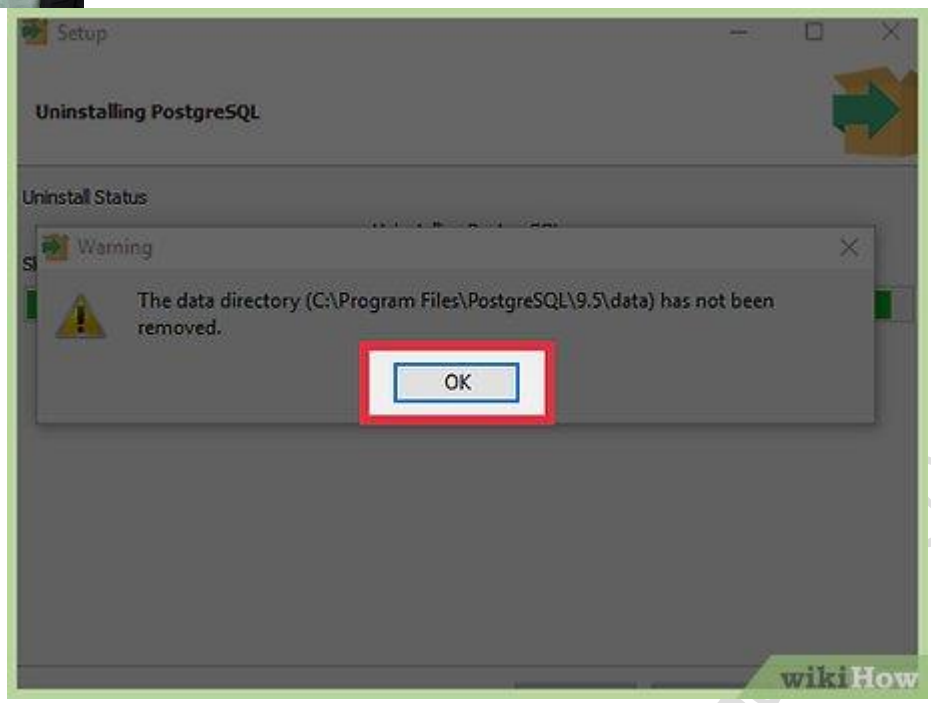
Click OK on Warning. You may then manually delete leftovers related to PostgreSQL, in its installation directory.

5. 

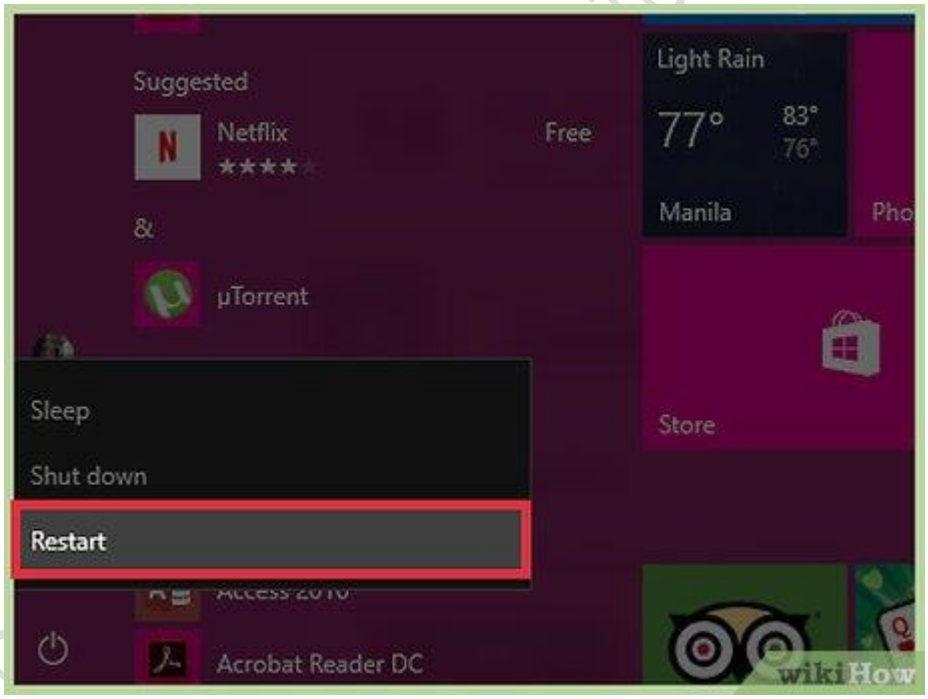
You then click "OK" on "Info".

6. 

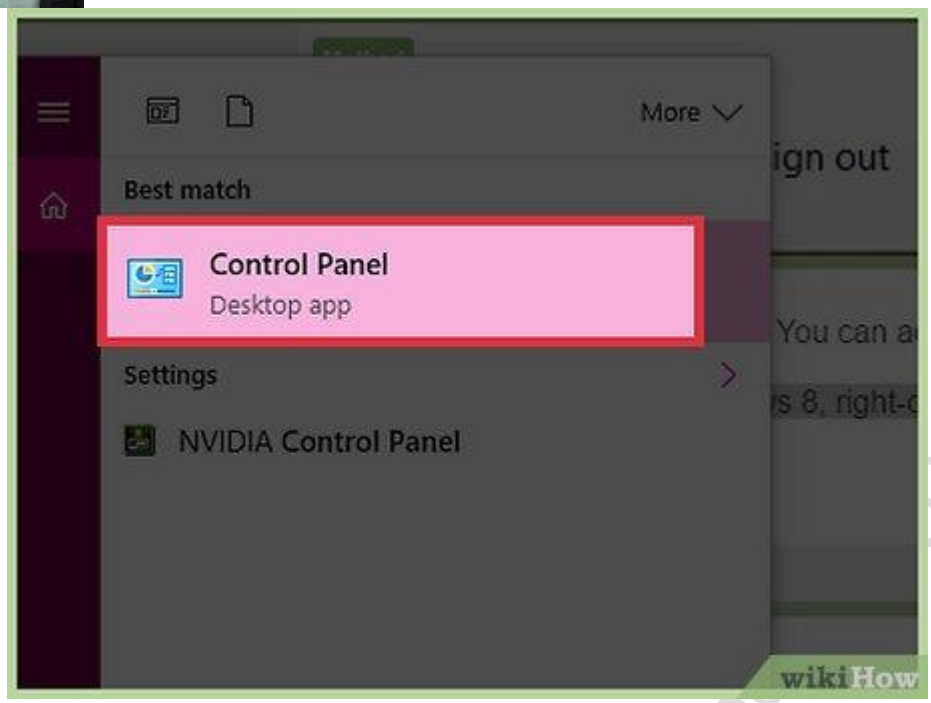
Make sure that you have saved all of your files and programs. You then click Yes in Question information box to restart your device.

7. 

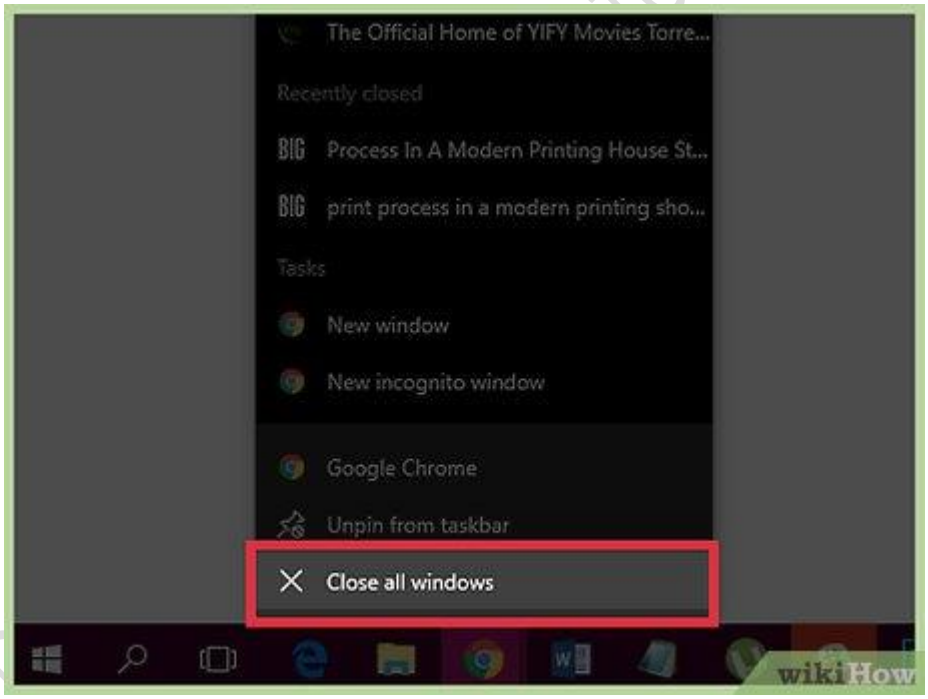
Remove the pgAgent 3.0.1 (size, 9.48MB).

8. 

Again, you may clean the file leftovers by hand.

9. 

Continue to remove MS Visual C++ 2008, a component for PostgreSQL software.

10. 

Remove C++ 2005 project listed.

11. 

Save and reboot your computer. You manually search and clean those traces related the programs installed above.

How to Uninstall PostgreSQL From Windows

What is the Purpose to Uninstall Postgres on Windows?

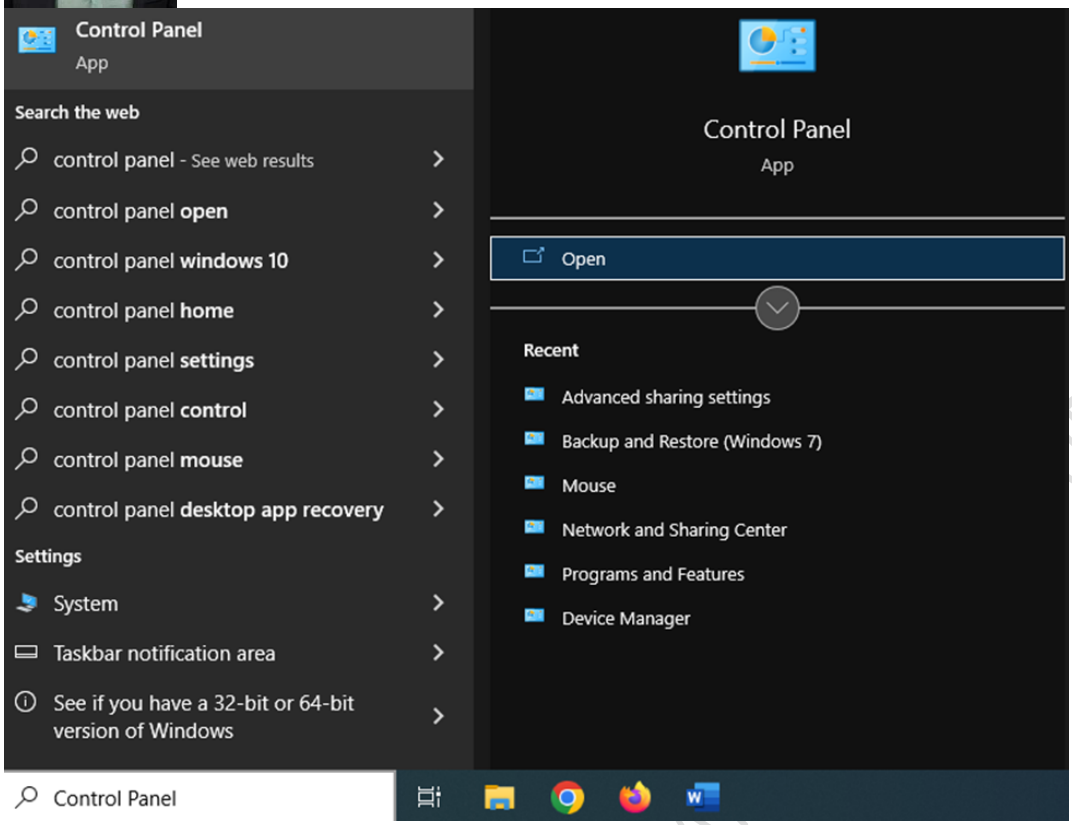
Uninstalling Postgres ensures that all of its files, configurations, and services are removed from the computer, freeing up disk space and resources. Additionally, it is useful if users are experiencing issues with Postgres and need to perform a clean installation to troubleshoot the problem.

Method: Using Control Panel to Uninstall Postgres

Uninstalling PostgreSQL on Windows can be done through the Control Panel or by using the PostgreSQL installer. Here are the steps to follow:

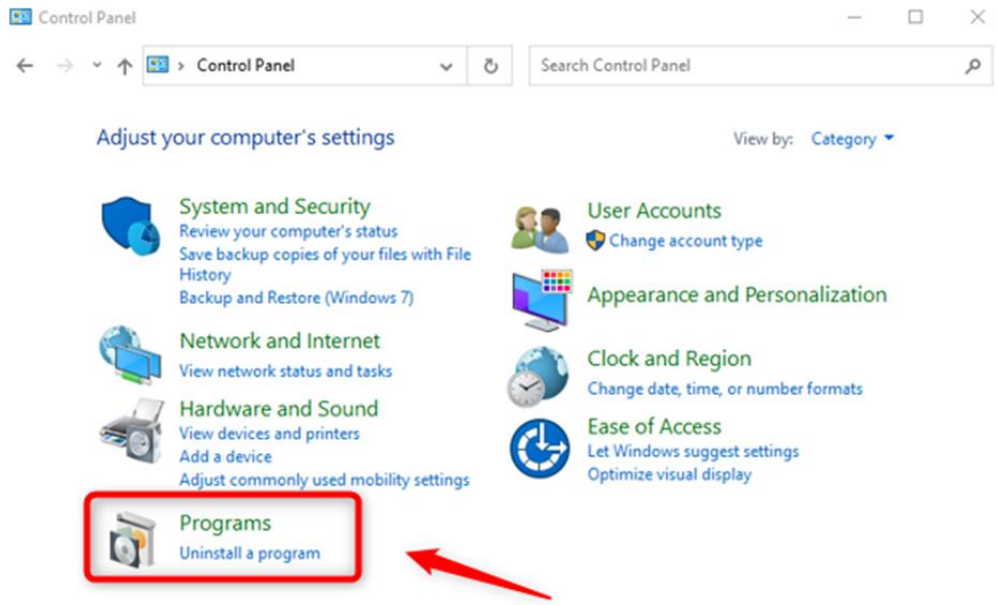
Step 1: Open the Control Panel

Go to the Windows Start menu and search for Control Panel, then click on it to open it:



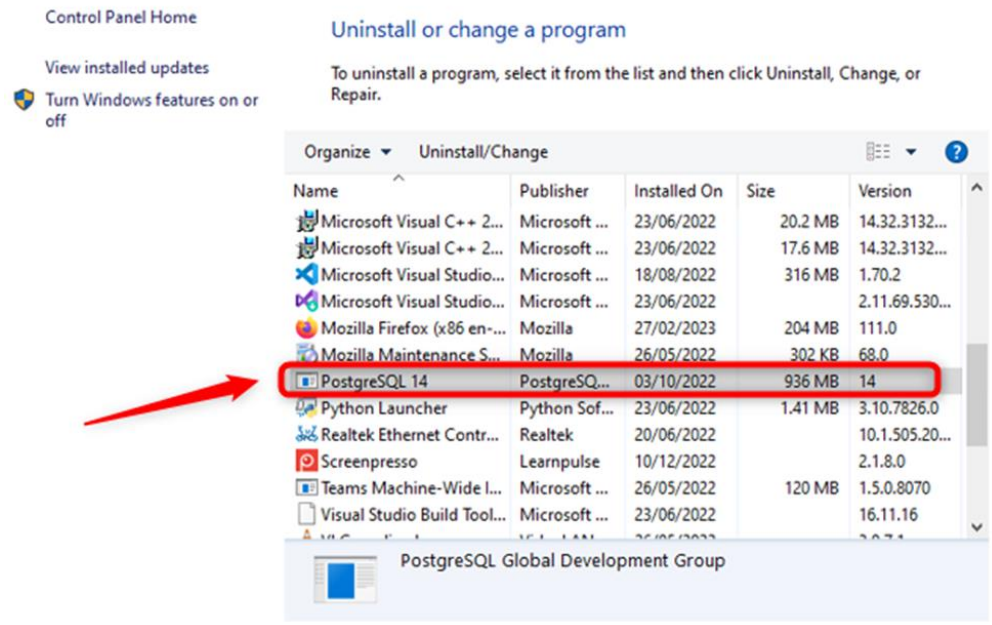
Step 2: Go to "Programs and Features"

Once you reach the Control Panel, click on "**Programs and Features**" (or "Uninstall a program" depending on the version of Windows):



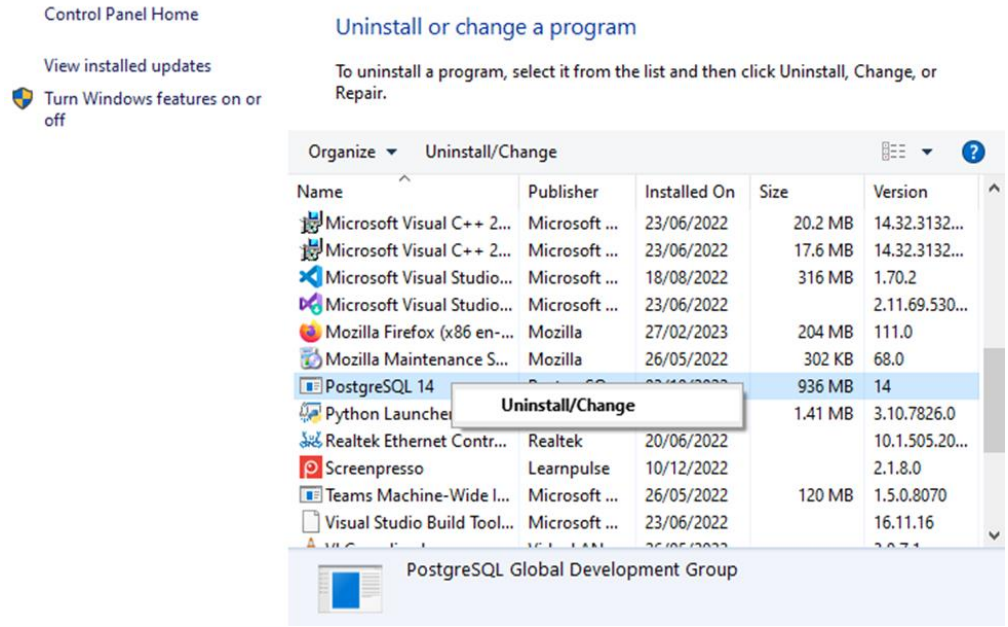
Step 3: Locate PostgreSQL

Scroll down the list of programs until you find PostgreSQL. Click on it to select it:

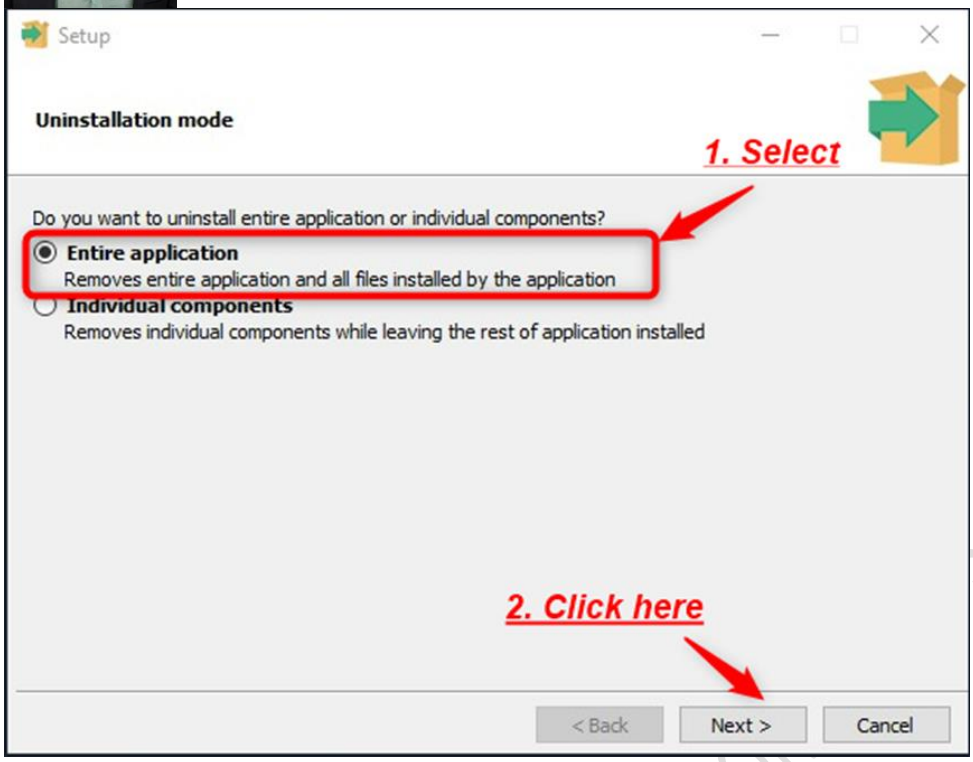


Step 4: Uninstall PostgreSQL

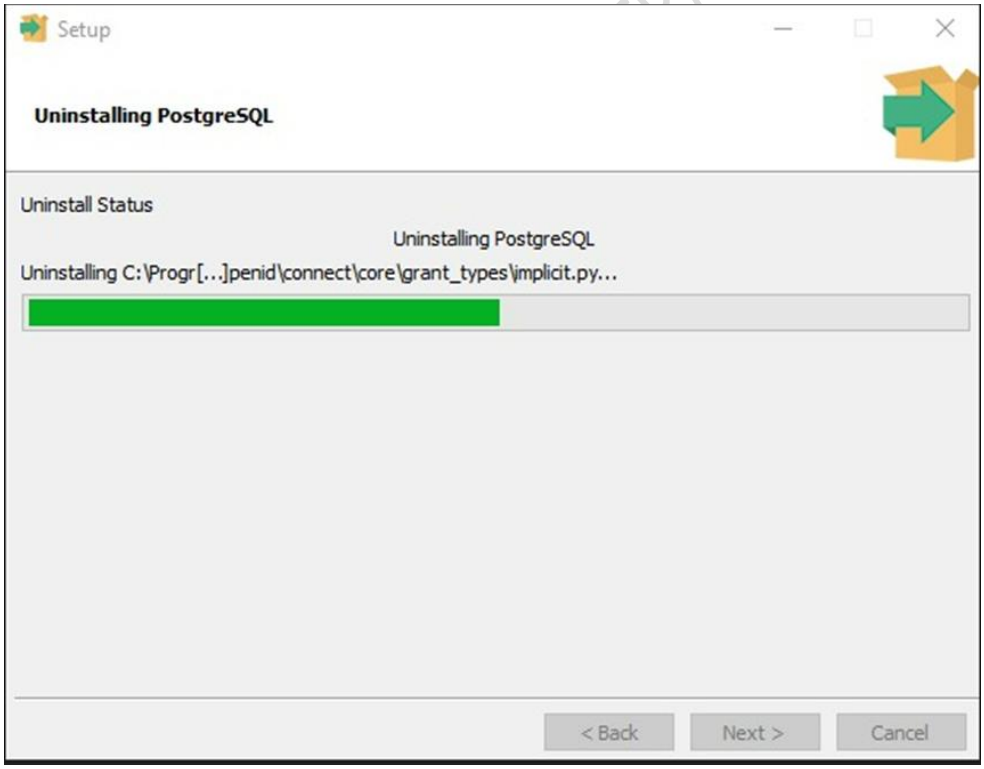
Click on the "**Uninstall**" button by pressing the right button of the mouse on the "PostgreSQL" program:



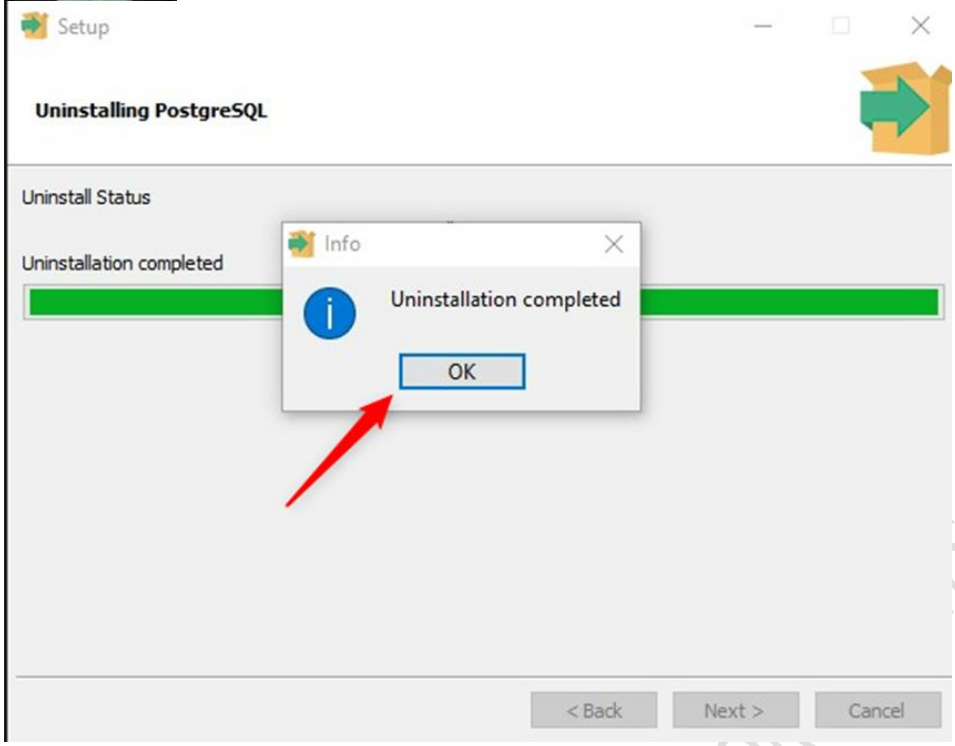
Follow the prompts to complete the uninstallation process:



Hit the “Next” button to begin the uninstallation of Postgres:

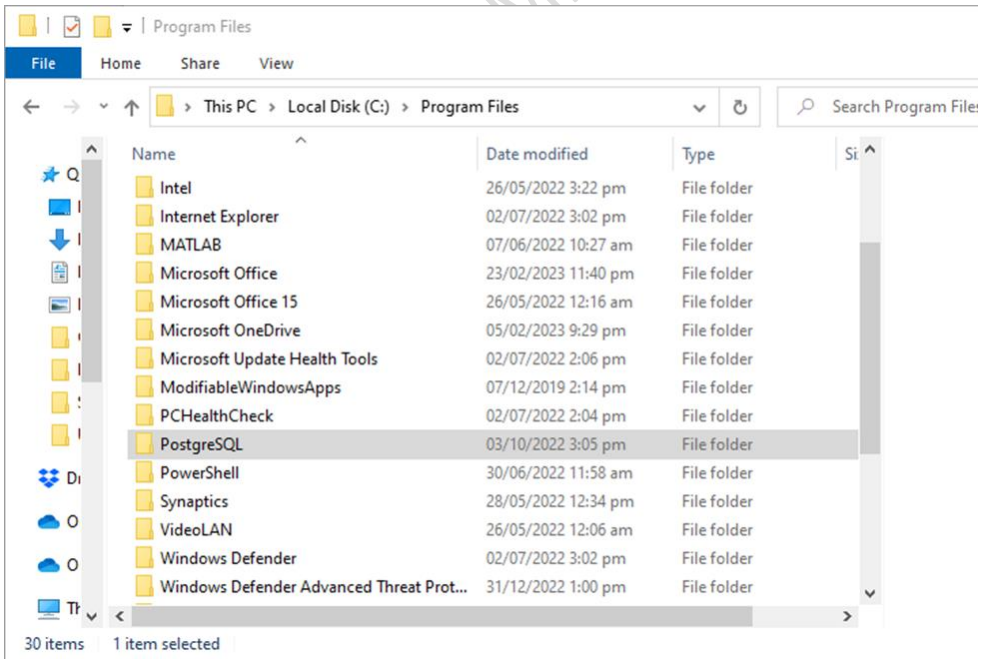


After completing the uninstallation, a new window will appear:



Hit the “Ok” button to finish.

Note: Once the uninstallation is done, the user can manually delete the PostgreSQL files and folders that were left behind. These can typically be found in the “**Program Files**” or “**Program Files (x86)**” folder in the Windows installation:



Remove the “PostgreSQL” directory to completely uninstall Postgres from your Windows system.

How To Remove PostgreSQL in UNIX

Step 1: List the PostgreSQL Packages

Use the dpkg tool to list packages pertaining to the PostgreSQL setup.

dpkg -l | grep postgres

Output:

```
root@newclient:~# dpkg -l | grep postgres
ii  pgdg-keyring                      2018.2                  all          keyring for apt.postgresql.org
ii  postgresql-10                     10.10-1.pgdg18.04+1     amd64        object-relational SQL database, versio
n 10 server
ii  postgresql-client-10              10.10-1.pgdg18.04+1     amd64        front-end programs for PostgreSQL 10
ii  postgresql-client-common          204.pgdg18.04+1         all          manager for multiple PostgreSQL client
 versions
ii  postgresql-common                 204.pgdg18.04+1         all          PostgreSQL database-cluster manager
root@newclient:~#
```

Step 2: Delete the PostgreSQL Packages

In Step 1, all of the software packages related to the PostgreSQL install are shown. To remove and delete them all with one command, Start with this command.

apt-get --purge remove command

This is followed by each package name indicated separated with a space. For my particular version of PostgreSQL, the software installed was:

- pgdg-keyring
- postgresql-10
- postgresql-client-10
- postgresql-client-common
- postgresql-common

so, my purge remove command will look like this:

sudo apt-get --purge remove pgdg-keyring postgresql-10 postgresql-client-10 postgresql-client-common postgresql-common

Step 3: Verifying the Deletion of PostgreSQL

Once you remove these packages you should no longer be able to enter into the PostgreSQL environment. You can verify that by running the grep command again and searching for postgres:

root@newclient:~# dpkg -l | grep postgres

root@newclient:~#

As you can see, there is no output from that command which means, PostgreSQL has been successfully uninstalled!

Removing and Reinstalling PostgreSQL

Step-by-Step Guide to Remove PostgreSQL

Stop PostgreSQL Services

Before uninstalling PostgreSQL, stop all running PostgreSQL processes. To check for active PostgreSQL processes, use this command:

```
ps aux | grep postgres
```

If any PostgreSQL processes are running, stop them with:

```
sudo systemctl stop postgresql
```

Tip: Verify PostgreSQL Service Status

After stopping the PostgreSQL service, you can verify its status using:

```
sudo systemctl status postgresql
```

This command will show you if the service is inactive or if there are any remaining processes.

Uninstall PostgreSQL Packages

To remove PostgreSQL packages, use this command:

```
sudo apt-get remove --purge postgresql*
```

This command removes all PostgreSQL packages and their config files.

Clean Up Remaining Files and Directories

After uninstalling the packages, you may need to remove some leftover directories:

```
sudo rm -rf /etc/postgresql/
```

```
sudo rm -rf /var/lib/postgresql/
```

```
sudo rm -rf /var/log/postgresql/
```

These commands delete the PostgreSQL config, data, and log directories.

Remove PostgreSQL User and Group

Remove the PostgreSQL system user and group:

sudo userdel postgres

sudo groupdel postgres

These commands delete the PostgreSQL user and group from the system.

Reinstalling PostgreSQL on Ubuntu

Preparing for a Fresh Installation

Before reinstalling PostgreSQL, update your package lists:

sudo apt update

Check that your system meets the requirements for PostgreSQL installation. Ubuntu usually includes all needed dependencies.

Tip: Check PostgreSQL Version

To check the available PostgreSQL versions in your repositories, use:

apt-cache search postgresql | grep postgresql

This command lists all available PostgreSQL packages and their versions.

Installing PostgreSQL

To install PostgreSQL, run:

sudo apt install postgresql

This installs the latest version of PostgreSQL from the Ubuntu repositories.

To install a specific PostgreSQL version, use:

sudo apt install postgresql-<version>

Replace <version> with the PostgreSQL version number you want (e.g., 12, 13, 14).

Configuring the New PostgreSQL Installation

After installation, PostgreSQL creates a default database and user. To set up more databases, use:

sudo -u postgres createdb <database_name>

To create a new user role:

sudo -u postgres createuser --interactive

This starts a prompt to set up a new user role.

To set user permissions, access the PostgreSQL prompt:

```
sudo -u postgres psql
```

Then, use SQL commands to give privileges:

```
GRANT ALL PRIVILEGES ON DATABASE <database_name> TO <username>;
```

Replace <database_name> and <username> with your database and user names.
