EDB Postgres Extended Server

Version 18

Built at 2025-11-25T16:52:50

1 EDB Postgres Extended Server

EDB Postgres Extended Server is a Postgres database server distribution built on open-source, community PostgreSQL. It's fully compatible with PostgreSQL. If you have applications written and tested to work with PostgreSQL, they will behave the same with EDB Postgres Extended Server. We will support and fix any functionality or behavior differences between community PostgreSQL and EDB Postgres Extended Server.

EDB Postgres Extended Server's primary purpose is to extend PostgreSQL with a limited number of features that can't be implemented as extensions, such as enhanced replication optimization used by EDB Postgres Distributed and Transparent Data Encryption, while maintaining parity in other respects.

Additional value-add enterprise features include:

- Security though Transparent Data Encryption
- WAL pacing delays to avoid flooding transaction logs
- Additional tracing and diagnostics options

To see how EDB Postgres Extended Server compares to other databases, see Choosing your Postgres distribution.

2 Release notes

The EDB Postgres Extended Server documentation describes the latest version of EDB Postgres Extended Server 18, including minor releases and patches. These release notes cover what was new in each release.

Version	Release date
18.1.0	25 Nov 2025

2.1 EDB Postgres Extended Server 18.1.0 release notes

Released: 25 Nov 2025

> **Deprecation**
>
> From EDB Postgres Extended Server 18.1.0 onwards, support for Debian 11 has been deprecated. Users are advised to upgrade to Debian 12 or later versions to ensure continued support and access to the latest features and security updates.

EDB Postgres Extended Server 18.1.0 includes the following enhancements and bug fixes:

Type	Description	Addresses
Upstream merge	Merged with community PostgreSQL 18.1. See the PostgreSQL 18.1 Release Notes for more information.	
Feature	Added `password_profile` extension to enable the profile-based system for enforcing advanced password management rules on user accounts.	
Feature	Added `data_redaction` extension to provide capabilities for redacting sensitive data in query results.	

3 Deployment options

The deployment options include:

- Installing on a virtual machine or physical server using native packages
- Deploying it with EDB Postgres Distributed
- Deploying it on EDB Postgres AI Cloud Service with extreme-high-availability cluster types

4 Installing EDB Postgres Extended Server on Linux

Select a link to access the applicable installation instructions:

Linux x86-64 (amd64)

Red Hat Enterprise Linux (RHEL) and derivatives

- RHEL 9, RHEL 8
- Oracle Linux (OL) 9, Oracle Linux (OL) 8
- Rocky Linux 9, Rocky Linux 8
- AlmaLinux 9, AlmaLinux 8

SUSE Linux Enterprise (SLES)

- SLES 15

Debian and derivatives

- Ubuntu 24.04, Ubuntu 22.04
- Debian 12

Linux IBM Power (ppc64le)

Red Hat Enterprise Linux (RHEL) and derivatives

- RHEL 9, RHEL 8

SUSE Linux Enterprise (SLES)

- SLES 15

Linux AArch64 (ARM64)

Red Hat Enterprise Linux (RHEL) and derivatives

- RHEL 9
- Oracle Linux (OL) 9

Debian and derivatives

- Debian 12

4.1 Installing EDB Postgres Extended Server on Linux x86 (amd64)

Operating system-specific install instructions are described in the corresponding documentation:

Red Hat Enterprise Linux (RHEL) and derivatives

- RHEL 9
- RHEL 8
- Oracle Linux (OL) 9
- Oracle Linux (OL) 8
- Rocky Linux 9
- Rocky Linux 8
- AlmaLinux 9
- AlmaLinux 8

SUSE Linux Enterprise (SLES)

- SLES 15

Debian and derivatives

- Ubuntu 24.04
- Ubuntu 22.04
- Debian 12

4.1.1 Installing EDB Postgres Extended Server on RHEL 9 or OL 9 x86_64

Prerequisites

Before you begin the installation process:

- Set up the EDB repository.

  Setting up the repository is a one-time task. If you have already set up your repository, you don't need to perform this step.

  To determine if your repository exists, enter this command:

  ```
  dnf repolist | grep enterprisedb
  ```

  If no output is generated, the repository isn't installed.

  To set up the EDB repository:

  1. Go to EDB repositories.
  2. Select the button that provides access to the EDB repository.
  3. Select the platform and software that you want to download.
  4. Follow the instructions for setting up the EDB repository.

- Install the EPEL repository:

  ```
  sudo dnf -y install https://dl.fedoraproject.org/pub/epel/epel-release-latest-9.noarch.rpm
  ```

Install the package

```
sudo dnf -y install edb-postgresextended18-server edb-postgresextended18-contrib
```

Initial configuration

Getting started with your cluster involves logging in, ensuring the installation and initial configuration was successful, connecting to your cluster, and creating the user password.

First, you need to initialize and start the database cluster. The `edb-pge-18-setup` script creates a cluster.

```
sudo PGSETUP_INITDB_OPTIONS="-E UTF-8" /usr/edb/pge18/bin/edb-pge-18-setup initdb

sudo systemctl start edb-pge-18
```

To work in your cluster, log in as the postgres user. Connect to the database server using the psql command-line client. Alternatively, you can use a client of your choice with the appropriate connection string.

```
sudo -iu postgres

psql postgres
```

The server runs with the `peer` or `ident` permission by default. You can change the authentication method by modifying the `pg_hba.conf` file.

Before changing the authentication method, assign a password to the database superuser, postgres. For more information on changing the authentication, see Modifying the pg_hba.conf file.

```
ALTER ROLE postgres with PASSWORD
'password';
```

Experiment

Now you're ready to create and connect to a database, create a table, insert data in a table, and view the data from the table.

First, use psql to create a database named `hr` to hold human resource information.

```
# running in psql
CREATE DATABASE
hr;
```

output

```
CREATE DATABASE
```

Connect to the `hr` database inside psql:

```
\c hr
```

output

```
You are now connected to database "hr" as user "postgres".
```

Create columns to hold department numbers, unique department names, and locations:

```
CREATE TABLE public.dept (deptno numeric(2) NOT NULL CONSTRAINT dept_pk
PRIMARY KEY, dname varchar(14) CONSTRAINT dept_dname_uq UNIQUE, loc
varchar(13));
```

output

```
CREATE TABLE
```

Insert values into the `dept` table:

```
INSERT INTO dept VALUES (10,'ACCOUNTING','NEW YORK');
```

output

```
INSERT 0 1
```

```
INSERT into dept VALUES (20,'RESEARCH','DALLAS');
```

output

```
INSERT 0 1
```

View the table data by selecting the values from the table:

```
SELECT * FROM dept;
```

output

```
deptno  |   dname    |   loc
--------+------------+----------
10      | ACCOUNTING | NEW YORK
20      | RESEARCH   | DALLAS
(2 rows)
```

4.1.2 Installing EDB Postgres Extended Server on RHEL 8 or OL 8 x86_64

Prerequisites

Before you begin the installation process:

- Set up the EDB repository.

  Setting up the repository is a one-time task. If you have already set up your repository, you don't need to perform this step.

  To determine if your repository exists, enter this command:

  ```
  dnf repolist | grep enterprisedb
  ```

  If no output is generated, the repository isn't installed.

  To set up the EDB repository:

  1. Go to EDB repositories.
  2. Select the button that provides access to the EDB repository.
  3. Select the platform and software that you want to download.
  4. Follow the instructions for setting up the EDB repository.

- Install the EPEL repository:

  ```
  sudo dnf -y install https://dl.fedoraproject.org/pub/epel/epel-release-latest-8.noarch.rpm
  ```

Install the package

```
sudo dnf -y install edb-postgresextended18-server edb-postgresextended18-contrib
```

Initial configuration

Getting started with your cluster involves logging in, ensuring the installation and initial configuration was successful, connecting to your cluster, and creating the user password.

First, you need to initialize and start the database cluster. The `edb-pge-18-setup` script creates a cluster.

```
sudo PGSETUP_INITDB_OPTIONS="-E UTF-8" /usr/edb/pge18/bin/edb-pge-18-setup initdb

sudo systemctl start edb-pge-18
```

To work in your cluster, log in as the postgres user. Connect to the database server using the psql command-line client. Alternatively, you can use a client of your choice with the appropriate connection string.

```
sudo -iu postgres

psql postgres
```

The server runs with the `peer` or `ident` permission by default. You can change the authentication method by modifying the `pg_hba.conf` file.

Before changing the authentication method, assign a password to the database superuser, postgres. For more information on changing the authentication, see Modifying the pg_hba.conf file.

```
ALTER ROLE postgres with PASSWORD
'password';
```

Experiment

Now you're ready to create and connect to a database, create a table, insert data in a table, and view the data from the table.

First, use psql to create a database named `hr` to hold human resource information.

```
# running in psql
CREATE DATABASE
hr;
```

output

```
CREATE DATABASE
```

Connect to the `hr` database inside psql:

```
\c hr
```

output

```
You are now connected to database "hr" as user "postgres".
```

Create columns to hold department numbers, unique department names, and locations:

```
CREATE TABLE public.dept (deptno numeric(2) NOT NULL CONSTRAINT dept_pk
PRIMARY KEY, dname varchar(14) CONSTRAINT dept_dname_uq UNIQUE, loc
varchar(13));
```

output

```
CREATE TABLE
```

Insert values into the `dept` table:

```
INSERT INTO dept VALUES (10,'ACCOUNTING','NEW YORK');
```

output

```
INSERT 0 1
```

```
INSERT into dept VALUES (20,'RESEARCH','DALLAS');
```

output

```
INSERT 0 1
```

View the table data by selecting the values from the table:

```
SELECT * FROM dept;
```

output

```
deptno  |   dname    |   loc
--------+------------+----------
10      | ACCOUNTING | NEW YORK
20      | RESEARCH   | DALLAS
(2 rows)
```

4.1.3 Installing EDB Postgres Extended Server on AlmaLinux 9 or Rocky Linux 9 x86_64

Prerequisites

Before you begin the installation process:

- Set up the EDB repository.

  Setting up the repository is a one-time task. If you have already set up your repository, you don't need to perform this step.

  To determine if your repository exists, enter this command:

  ```
  dnf repolist | grep enterprisedb
  ```

  If no output is generated, the repository isn't installed.

  To set up the EDB repository:

  1. Go to EDB repositories.
  2. Select the button that provides access to the EDB repository.
  3. Select the platform and software that you want to download.
  4. Follow the instructions for setting up the EDB repository.

- Install the EPEL repository:

  ```
  sudo dnf -y install epel-release
  ```

- Enable additional repositories to resolve dependencies:

  ```
  sudo dnf config-manager --set-enabled crb
  ```

Install the package

```
sudo dnf -y install edb-postgresextended18-server edb-postgresextended18-contrib
```

Initial configuration

Getting started with your cluster involves logging in, ensuring the installation and initial configuration was successful, connecting to your cluster, and creating the user password.

First, you need to initialize and start the database cluster. The `edb-pge-18-setup` script creates a cluster.

```
sudo PGSETUP_INITDB_OPTIONS="-E UTF-8" /usr/edb/pge18/bin/edb-pge-18-setup initdb

sudo systemctl start edb-pge-18
```

To work in your cluster, log in as the postgres user. Connect to the database server using the psql command-line client. Alternatively, you can use a client of your choice with the appropriate connection string.

```
sudo -iu postgres

psql postgres
```

The server runs with the `peer` or `ident` permission by default. You can change the authentication method by modifying the `pg_hba.conf` file.

Before changing the authentication method, assign a password to the database superuser, postgres. For more information on changing the authentication, see Modifying the pg_hba.conf file.

```
ALTER ROLE postgres with PASSWORD
'password';
```

Experiment

Now you're ready to create and connect to a database, create a table, insert data in a table, and view the data from the table.

First, use psql to create a database named `hr` to hold human resource information.

```
# running in psql
CREATE DATABASE
hr;
```

output

```
CREATE DATABASE
```

Connect to the `hr` database inside psql:

```
\c hr
```

output

```
You are now connected to database "hr" as user "postgres".
```

Create columns to hold department numbers, unique department names, and locations:

```
CREATE TABLE public.dept (deptno numeric(2) NOT NULL CONSTRAINT dept_pk
PRIMARY KEY, dname varchar(14) CONSTRAINT dept_dname_uq UNIQUE, loc
varchar(13));
```

output

```
CREATE TABLE
```

Insert values into the `dept` table:

```
INSERT INTO dept VALUES (10,'ACCOUNTING','NEW YORK');
```

output

```
INSERT 0 1
```

```
INSERT into dept VALUES (20,'RESEARCH','DALLAS');
```

output

```
INSERT 0 1
```

View the table data by selecting the values from the table:

```
SELECT * FROM dept;
```

output

```
deptno  |   dname    |   loc
--------+------------+----------
10      | ACCOUNTING | NEW YORK
20      | RESEARCH   | DALLAS
(2 rows)
```

4.1.4 Installing EDB Postgres Extended Server on AlmaLinux 8 or Rocky Linux 8 x86_64

Prerequisites

Before you begin the installation process:

- Set up the EDB repository.

  Setting up the repository is a one-time task. If you have already set up your repository, you don't need to perform this step.

  To determine if your repository exists, enter this command:

  ```
  dnf repolist | grep enterprisedb
  ```

  If no output is generated, the repository isn't installed.

  To set up the EDB repository:

  1. Go to EDB repositories.
  2. Select the button that provides access to the EDB repository.
  3. Select the platform and software that you want to download.
  4. Follow the instructions for setting up the EDB repository.

- Install the EPEL repository:

  ```
  sudo dnf -y install epel-release
  ```

- Enable additional repositories to resolve dependencies:

  ```
  sudo dnf config-manager --set-enabled powertools
  ```

Install the package

```
sudo dnf -y install edb-postgresextended18-server edb-postgresextended18-contrib
```

Initial configuration

Getting started with your cluster involves logging in, ensuring the installation and initial configuration was successful, connecting to your cluster, and creating the user password.

First, you need to initialize and start the database cluster. The `edb-pge-18-setup` script creates a cluster.

```
sudo PGSETUP_INITDB_OPTIONS="-E UTF-8" /usr/edb/pge18/bin/edb-pge-18-setup initdb

sudo systemctl start edb-pge-18
```

To work in your cluster, log in as the postgres user. Connect to the database server using the psql command-line client. Alternatively, you can use a client of your choice with the appropriate connection string.

```
sudo -iu postgres

psql postgres
```

The server runs with the `peer` or `ident` permission by default. You can change the authentication method by modifying the `pg_hba.conf` file.

Before changing the authentication method, assign a password to the database superuser, postgres. For more information on changing the authentication, see Modifying the pg_hba.conf file.

```
ALTER ROLE postgres with PASSWORD
'password';
```

Experiment

Now you're ready to create and connect to a database, create a table, insert data in a table, and view the data from the table.

First, use psql to create a database named `hr` to hold human resource information.

```
# running in psql
CREATE DATABASE
hr;
```

output

```
CREATE DATABASE
```

Connect to the `hr` database inside psql:

```
\c hr
```

output

```
You are now connected to database "hr" as user "postgres".
```

Create columns to hold department numbers, unique department names, and locations:

```
CREATE TABLE public.dept (deptno numeric(2) NOT NULL CONSTRAINT dept_pk
PRIMARY KEY, dname varchar(14) CONSTRAINT dept_dname_uq UNIQUE, loc
varchar(13));
```

output

```
CREATE TABLE
```

Insert values into the `dept` table:

```
INSERT INTO dept VALUES (10,'ACCOUNTING','NEW YORK');
```

output

```
INSERT 0 1
```

```
INSERT into dept VALUES (20,'RESEARCH','DALLAS');
```

output

```
INSERT 0 1
```

View the table data by selecting the values from the table:

```
SELECT * FROM dept;
```

output

```
deptno  |   dname    |   loc
--------+------------+----------
10      | ACCOUNTING | NEW YORK
20      | RESEARCH   | DALLAS
(2 rows)
```

4.1.5 Installing EDB Postgres Extended Server on SLES 15 x86_64

Prerequisites

Before you begin the installation process:

- Set up the EDB repository.

  Setting up the repository is a one-time task. If you have already set up your repository, you don't need to perform this step.

  To determine if your repository exists, enter this command:

  ```
  zypper lr -E | grep enterprisedb
  ```

  If no output is generated, the repository isn't installed.

  To set up the EDB repository:

  1. Go to EDB repositories.
  2. Select the button that provides access to the EDB repository.
  3. Select the platform and software that you want to download.
  4. Follow the instructions for setting up the EDB repository.

- Activate the required SUSE module:

  ```
  sudo SUSEConnect -p PackageHub/15.4/x86_64
  ```

- Refresh the metadata:

  ```
  sudo zypper refresh
  ```

Install the package

```
sudo zypper -n install edb-postgresextended18-server edb-postgresextended18-contrib
```

Initial configuration

Getting started with your cluster involves logging in, ensuring the installation and initial configuration was successful, connecting to your cluster, and creating the user password.

First, you need to initialize and start the database cluster. The `edb-pge-18-setup` script creates a cluster.

```
sudo PGSETUP_INITDB_OPTIONS="-E UTF-8" /usr/edb/pge18/bin/edb-pge-18-setup initdb

sudo systemctl start edb-pge-18
```

To work in your cluster, log in as the postgres user. Connect to the database server using the psql command-line client. Alternatively, you can use a client of your choice with the appropriate connection string.

```
sudo -iu postgres

psql postgres
```

The server runs with the `peer` or `ident` permission by default. You can change the authentication method by modifying the `pg_hba.conf` file.

Before changing the authentication method, assign a password to the database superuser, postgres. For more information on changing the authentication, see Modifying the pg_hba.conf file.

```
ALTER ROLE postgres with PASSWORD
'password';
```

Experiment

Now you're ready to create and connect to a database, create a table, insert data in a table, and view the data from the table.

First, use psql to create a database named `hr` to hold human resource information.

```
# running in psql
CREATE DATABASE
hr;
```

output

```
CREATE DATABASE
```

Connect to the `hr` database inside psql:

```
\c hr
```

output

```
You are now connected to database "hr" as user "postgres".
```

Create columns to hold department numbers, unique department names, and locations:

```
CREATE TABLE public.dept (deptno numeric(2) NOT NULL CONSTRAINT dept_pk
PRIMARY KEY, dname varchar(14) CONSTRAINT dept_dname_uq UNIQUE, loc
varchar(13));
```

output

```
CREATE TABLE
```

Insert values into the `dept` table:

```
INSERT INTO dept VALUES (10,'ACCOUNTING','NEW YORK');
```

output

```
INSERT 0 1
```

```
INSERT into dept VALUES (20,'RESEARCH','DALLAS');
```

output

```
INSERT 0 1
```

View the table data by selecting the values from the table:

```
SELECT * FROM dept;
```

output

```
deptno  |   dname    |   loc
--------+------------+----------
10      | ACCOUNTING | NEW YORK
20      | RESEARCH   | DALLAS
(2 rows)
```

4.1.6 Installing EDB Postgres Extended Server on Ubuntu 24.04 x86_64

Prerequisites

Before you begin the installation process:

- Set up the EDB repository.

  Setting up the repository is a one-time task. If you have already set up your repository, you don't need to perform this step.

  To determine if your repository exists, enter this command:

  ```
  apt-cache search enterprisedb
  ```

  If no output is generated, the repository isn't installed.

  To set up the EDB repository:

  1. Go to EDB repositories.
  2. Select the button that provides access to the EDB repository.
  3. Select the platform and software that you want to download.
  4. Follow the instructions for setting up the EDB repository.

Install the package

```
sudo apt-get -y install edb-postgresextended-18
```

Initial configuration

This section steps you through getting started with your cluster including logging in, ensuring the installation was successful, connecting to your cluster, and creating the user password.

First, you need to initialize and start the database cluster. The `edb-pge-18-setup` script creates a cluster.

```
sudo PGSETUP_INITDB_OPTIONS="-E UTF-8" /usr/lib/edb-pge/18/bin/edb-pge-18-setup initdb

sudo systemctl start edb-pge-18
```

To work in your cluster, log in as the postgres user. Connect to the database server using the psql command-line client. Alternatively, you can use a client of your choice with the appropriate connection string.

```
sudo -iu postgres

psql postgres
```

The server runs with the `peer` or `ident` permission by default. You can change the authentication method by modifying the `pg_hba.conf` file.

Before changing the authentication method, assign a password to the database superuser, postgres. For more information on changing the authentication, see Modifying the pg_hba.conf file.

```
ALTER ROLE postgres with PASSWORD
'password';
```

Experiment

Now you're ready to create and connect to a database, create a table, insert data in a table, and view the data from the table.

First, use psql to create a database named `hr` to hold human resource information.

```
# running in psql
CREATE DATABASE
hr;
```

output

```
CREATE DATABASE
```

Connect to the `hr` database inside psql:

```
\c hr
```

output

```
You are now connected to database "hr" as user "postgres".
```

Create columns to hold department numbers, unique department names, and locations:

```
CREATE TABLE public.dept (deptno numeric(2) NOT NULL CONSTRAINT dept_pk
PRIMARY KEY, dname varchar(14) CONSTRAINT dept_dname_uq UNIQUE, loc
varchar(13));
```

output

```
CREATE TABLE
```

Insert values into the `dept` table:

```
INSERT INTO dept VALUES (10,'ACCOUNTING','NEW YORK');
```

output

```
INSERT 0 1
```

```
INSERT into dept VALUES (20,'RESEARCH','DALLAS');
```

output

```
INSERT 0 1
```

View the table data by selecting the values from the table:

```
SELECT * FROM dept;
```

output

```
deptno  |   dname    |   loc
--------+------------+----------
10      | ACCOUNTING | NEW YORK
20      | RESEARCH   | DALLAS
(2 rows)
```

4.1.7 Installing EDB Postgres Extended Server on Ubuntu 22.04 x86_64

Prerequisites

Before you begin the installation process:

- Set up the EDB repository.

  Setting up the repository is a one-time task. If you have already set up your repository, you don't need to perform this step.

  To determine if your repository exists, enter this command:

  ```
  apt-cache search enterprisedb
  ```

  If no output is generated, the repository isn't installed.

  To set up the EDB repository:

  1. Go to EDB repositories.
  2. Select the button that provides access to the EDB repository.
  3. Select the platform and software that you want to download.
  4. Follow the instructions for setting up the EDB repository.

Install the package

```
sudo apt-get -y install edb-postgresextended-18
```

Initial configuration

This section steps you through getting started with your cluster including logging in, ensuring the installation was successful, connecting to your cluster, and creating the user password.

First, you need to initialize and start the database cluster. The `edb-pge-18-setup` script creates a cluster.

```
sudo PGSETUP_INITDB_OPTIONS="-E UTF-8" /usr/lib/edb-pge/18/bin/edb-pge-18-setup initdb

sudo systemctl start edb-pge-18
```

To work in your cluster, log in as the postgres user. Connect to the database server using the psql command-line client. Alternatively, you can use a client of your choice with the appropriate connection string.

```
sudo -iu postgres

psql postgres
```

The server runs with the `peer` or `ident` permission by default. You can change the authentication method by modifying the `pg_hba.conf` file.

Before changing the authentication method, assign a password to the database superuser, postgres. For more information on changing the authentication, see Modifying the pg_hba.conf file.

```
ALTER ROLE postgres with PASSWORD
'password';
```

Experiment

Now you're ready to create and connect to a database, create a table, insert data in a table, and view the data from the table.

First, use psql to create a database named `hr` to hold human resource information.

```
# running in psql
CREATE DATABASE
hr;
```

output

```
CREATE DATABASE
```

Connect to the `hr` database inside psql:

```
\c hr
```

output

```
You are now connected to database "hr" as user "postgres".
```

Create columns to hold department numbers, unique department names, and locations:

```
CREATE TABLE public.dept (deptno numeric(2) NOT NULL CONSTRAINT dept_pk
PRIMARY KEY, dname varchar(14) CONSTRAINT dept_dname_uq UNIQUE, loc
varchar(13));
```

output

```
CREATE TABLE
```

Insert values into the `dept` table:

```
INSERT INTO dept VALUES (10,'ACCOUNTING','NEW YORK');
```

output

```
INSERT 0 1
```

```
INSERT into dept VALUES (20,'RESEARCH','DALLAS');
```

output

```
INSERT 0 1
```

View the table data by selecting the values from the table:

```
SELECT * FROM dept;
```

output

```
deptno  |   dname    |   loc
--------+------------+----------
10      | ACCOUNTING | NEW YORK
20      | RESEARCH   | DALLAS
(2 rows)
```

4.1.8 Installing EDB Postgres Extended Server on Debian 12 x86_64

Prerequisites

Before you begin the installation process:

- Set up the EDB repository.

  Setting up the repository is a one-time task. If you have already set up your repository, you don't need to perform this step.

  To determine if your repository exists, enter this command:

  ```
  apt-cache search enterprisedb
  ```

  If no output is generated, the repository isn't installed.

  To set up the EDB repository:

  1. Go to EDB repositories.
  2. Select the button that provides access to the EDB repository.
  3. Select the platform and software that you want to download.
  4. Follow the instructions for setting up the EDB repository.

Install the package

```
sudo apt-get -y install edb-postgresextended-18
```

Initial configuration

This section steps you through getting started with your cluster including logging in, ensuring the installation was successful, connecting to your cluster, and creating the user password.

First, you need to initialize and start the database cluster. The `edb-pge-18-setup` script creates a cluster.

```
sudo PGSETUP_INITDB_OPTIONS="-E UTF-8" /usr/lib/edb-pge/18/bin/edb-pge-18-setup initdb

sudo systemctl start edb-pge-18
```

To work in your cluster, log in as the postgres user. Connect to the database server using the psql command-line client. Alternatively, you can use a client of your choice with the appropriate connection string.

```
sudo -iu postgres

psql postgres
```

The server runs with the `peer` or `ident` permission by default. You can change the authentication method by modifying the `pg_hba.conf` file.

Before changing the authentication method, assign a password to the database superuser, postgres. For more information on changing the authentication, see Modifying the pg_hba.conf file.

```
ALTER ROLE postgres with PASSWORD
'password';
```

Experiment

Now you're ready to create and connect to a database, create a table, insert data in a table, and view the data from the table.

First, use psql to create a database named `hr` to hold human resource information.

```
# running in psql
CREATE DATABASE
hr;
```

output

```
CREATE DATABASE
```

Connect to the `hr` database inside psql:

```
\c hr
```

output

```
You are now connected to database "hr" as user "postgres".
```

Create columns to hold department numbers, unique department names, and locations:

```
CREATE TABLE public.dept (deptno numeric(2) NOT NULL CONSTRAINT dept_pk
PRIMARY KEY, dname varchar(14) CONSTRAINT dept_dname_uq UNIQUE, loc
varchar(13));
```

output

```
CREATE TABLE
```

Insert values into the `dept` table:

```
INSERT INTO dept VALUES (10,'ACCOUNTING','NEW YORK');
```

output

```
INSERT 0 1
```

```
INSERT into dept VALUES (20,'RESEARCH','DALLAS');
```

output

```
INSERT 0 1
```

View the table data by selecting the values from the table:

```
SELECT * FROM dept;
```

output

```
deptno  |   dname    |   loc
--------+------------+----------
10      | ACCOUNTING | NEW YORK
20      | RESEARCH   | DALLAS
(2 rows)
```

4.2 Installing EDB Postgres Extended Server on Linux AArch64 (ARM64)

Operating system-specific install instructions are described in the corresponding documentation:

Red Hat Enterprise Linux (RHEL) and derivatives

- RHEL 9
- Oracle Linux (OL) 9

Debian and derivatives

- Debian 12

4.2.1 Installing EDB Postgres Extended Server on RHEL 9 or OL 9 arm64

Prerequisites

Before you begin the installation process:

- Set up the EDB repository.

  Setting up the repository is a one-time task. If you have already set up your repository, you don't need to perform this step.

  To determine if your repository exists, enter this command:

  ```
  dnf repolist | grep enterprisedb
  ```

  If no output is generated, the repository isn't installed.

  To set up the EDB repository:

  1. Go to EDB repositories.
  2. Select the button that provides access to the EDB repository.
  3. Select the platform and software that you want to download.
  4. Follow the instructions for setting up the EDB repository.

- Install the EPEL repository:

  ```
  sudo dnf -y install https://dl.fedoraproject.org/pub/epel/epel-release-latest-9.noarch.rpm
  ```

Install the package

```
sudo dnf -y install edb-postgresextended18-server edb-postgresextended18-contrib
```

Initial configuration

Getting started with your cluster involves logging in, ensuring the installation and initial configuration was successful, connecting to your cluster, and creating the user password.

First, you need to initialize and start the database cluster. The `edb-pge-18-setup` script creates a cluster.

```
sudo PGSETUP_INITDB_OPTIONS="-E UTF-8" /usr/edb/pge18/bin/edb-pge-18-setup initdb

sudo systemctl start edb-pge-18
```

To work in your cluster, log in as the postgres user. Connect to the database server using the psql command-line client. Alternatively, you can use a client of your choice with the appropriate connection string.

```
sudo -iu postgres

psql postgres
```

The server runs with the `peer` or `ident` permission by default. You can change the authentication method by modifying the `pg_hba.conf` file.

Before changing the authentication method, assign a password to the database superuser, postgres. For more information on changing the authentication, see Modifying the pg_hba.conf file.

```
ALTER ROLE postgres with PASSWORD
'password';
```

Experiment

Now you're ready to create and connect to a database, create a table, insert data in a table, and view the data from the table.

First, use psql to create a database named `hr` to hold human resource information.

```
# running in psql
CREATE DATABASE
hr;
```

output

```
CREATE DATABASE
```

Connect to the `hr` database inside psql:

```
\c hr
```

output

```
You are now connected to database "hr" as user "postgres".
```

Create columns to hold department numbers, unique department names, and locations:

```
CREATE TABLE public.dept (deptno numeric(2) NOT NULL CONSTRAINT dept_pk
PRIMARY KEY, dname varchar(14) CONSTRAINT dept_dname_uq UNIQUE, loc
varchar(13));
```

output

```
CREATE TABLE
```

Insert values into the `dept` table:

```
INSERT INTO dept VALUES (10,'ACCOUNTING','NEW YORK');
```

output

```
INSERT 0 1
```

```
INSERT into dept VALUES (20,'RESEARCH','DALLAS');
```

output

```
INSERT 0 1
```

View the table data by selecting the values from the table:

```
SELECT * FROM dept;
```

output

```
deptno  |   dname    |   loc
--------+------------+----------
10      | ACCOUNTING | NEW YORK
20      | RESEARCH   | DALLAS
(2 rows)
```

4.2.2 Installing EDB Postgres Extended Server on Debian 12 arm64

Prerequisites

Before you begin the installation process:

- Set up the EDB repository.

  Setting up the repository is a one-time task. If you have already set up your repository, you don't need to perform this step.

  To determine if your repository exists, enter this command:

  ```
  apt-cache search enterprisedb
  ```

  If no output is generated, the repository isn't installed.

  To set up the EDB repository:

  1. Go to EDB repositories.
  2. Select the button that provides access to the EDB repository.
  3. Select the platform and software that you want to download.
  4. Follow the instructions for setting up the EDB repository.

Install the package

```
sudo apt-get -y install edb-postgresextended-18
```

Initial configuration

This section steps you through getting started with your cluster including logging in, ensuring the installation was successful, connecting to your cluster, and creating the user password.

First, you need to initialize and start the database cluster. The `edb-pge-18-setup` script creates a cluster.

```
sudo PGSETUP_INITDB_OPTIONS="-E UTF-8" /usr/lib/edb-pge/18/bin/edb-pge-18-setup initdb

sudo systemctl start edb-pge-18
```

To work in your cluster, log in as the postgres user. Connect to the database server using the psql command-line client. Alternatively, you can use a client of your choice with the appropriate connection string.

```
sudo -iu postgres

psql postgres
```

The server runs with the `peer` or `ident` permission by default. You can change the authentication method by modifying the `pg_hba.conf` file.

Before changing the authentication method, assign a password to the database superuser, postgres. For more information on changing the authentication, see Modifying the pg_hba.conf file.

```
ALTER ROLE postgres with PASSWORD
'password';
```

Experiment

Now you're ready to create and connect to a database, create a table, insert data in a table, and view the data from the table.

First, use psql to create a database named `hr` to hold human resource information.

```
# running in psql
CREATE DATABASE
hr;
```

output

```
CREATE DATABASE
```

Connect to the `hr` database inside psql:

```
\c hr
```

output

```
You are now connected to database "hr" as user "postgres".
```

Create columns to hold department numbers, unique department names, and locations:

```
CREATE TABLE public.dept (deptno numeric(2) NOT NULL CONSTRAINT dept_pk
PRIMARY KEY, dname varchar(14) CONSTRAINT dept_dname_uq UNIQUE, loc
varchar(13));
```

output

```
CREATE TABLE
```

Insert values into the `dept` table:

```
INSERT INTO dept VALUES (10,'ACCOUNTING','NEW YORK');
```

output

```
INSERT 0 1
```

```
INSERT into dept VALUES (20,'RESEARCH','DALLAS');
```

output

```
INSERT 0 1
```

View the table data by selecting the values from the table:

```
SELECT * FROM dept;
```

output

```
deptno  |   dname    |   loc
--------+------------+----------
10      | ACCOUNTING | NEW YORK
20      | RESEARCH   | DALLAS
(2 rows)
```

4.3 Installing EDB Postgres Extended Server on Linux IBM Power (ppc64le)

Operating system-specific install instructions are described in the corresponding documentation:

Red Hat Enterprise Linux (RHEL)

- RHEL 9
- RHEL 8

SUSE Linux Enterprise (SLES)

- SLES 15

4.3.1 Installing EDB Postgres Extended Server on RHEL 9 ppc64le

Prerequisites

Before you begin the installation process:

- Set up the EDB repository.

  Setting up the repository is a one-time task. If you have already set up your repository, you don't need to perform this step.

  To determine if your repository exists, enter this command:

  ```
  dnf repolist | grep enterprisedb
  ```

  If no output is generated, the repository isn't installed.

  To set up the EDB repository:

  1. Go to EDB repositories.
  2. Select the button that provides access to the EDB repository.
  3. Select the platform and software that you want to download.
  4. Follow the instructions for setting up the EDB repository.

- Install the EPEL repository:

  ```
  sudo dnf -y install https://dl.fedoraproject.org/pub/epel/epel-release-latest-9.noarch.rpm
  ```

- Refresh the cache:

  ```
  sudo dnf makecache
  ```

Install the package

```
sudo dnf -y install edb-postgresextended18-server edb-postgresextended18-contrib
```

Initial configuration

Getting started with your cluster involves logging in, ensuring the installation and initial configuration was successful, connecting to your cluster, and creating the user password.

First, you need to initialize and start the database cluster. The `edb-pge-18-setup` script creates a cluster.

```
sudo PGSETUP_INITDB_OPTIONS="-E UTF-8" /usr/edb/pge18/bin/edb-pge-18-setup initdb

sudo systemctl start edb-pge-18
```

To work in your cluster, log in as the postgres user. Connect to the database server using the psql command-line client. Alternatively, you can use a client of your choice with the appropriate connection string.

```
sudo -iu postgres

psql postgres
```

The server runs with the `peer` or `ident` permission by default. You can change the authentication method by modifying the `pg_hba.conf` file.

Before changing the authentication method, assign a password to the database superuser, postgres. For more information on changing the authentication, see Modifying the pg_hba.conf file.

```
ALTER ROLE postgres with PASSWORD
'password';
```

Experiment

Now you're ready to create and connect to a database, create a table, insert data in a table, and view the data from the table.

First, use psql to create a database named `hr` to hold human resource information.

```
# running in psql
CREATE DATABASE
hr;
```

output

```
CREATE DATABASE
```

Connect to the `hr` database inside psql:

```
\c hr
```

output

```
You are now connected to database "hr" as user "postgres".
```

Create columns to hold department numbers, unique department names, and locations:

```
CREATE TABLE public.dept (deptno numeric(2) NOT NULL CONSTRAINT dept_pk
PRIMARY KEY, dname varchar(14) CONSTRAINT dept_dname_uq UNIQUE, loc
varchar(13));
```

output

```
CREATE TABLE
```

Insert values into the `dept` table:

```
INSERT INTO dept VALUES (10,'ACCOUNTING','NEW YORK');
```

output

```
INSERT 0 1
```

```
INSERT into dept VALUES (20,'RESEARCH','DALLAS');
```

output

```
INSERT 0 1
```

View the table data by selecting the values from the table:

```
SELECT * FROM dept;
```

output

```
deptno  |   dname    |   loc
--------+------------+----------
10      | ACCOUNTING | NEW YORK
20      | RESEARCH   | DALLAS
(2 rows)
```

4.3.2 Installing EDB Postgres Extended Server on RHEL 8 ppc64le

Prerequisites

Before you begin the installation process:

- Set up the EDB repository.

  Setting up the repository is a one-time task. If you have already set up your repository, you don't need to perform this step.

  To determine if your repository exists, enter this command:

  ```
  dnf repolist | grep enterprisedb
  ```

  If no output is generated, the repository isn't installed.

  To set up the EDB repository:

  1. Go to EDB repositories.
  2. Select the button that provides access to the EDB repository.
  3. Select the platform and software that you want to download.
  4. Follow the instructions for setting up the EDB repository.

- Install the EPEL repository:

  ```
  sudo dnf -y install https://dl.fedoraproject.org/pub/epel/epel-release-latest-8.noarch.rpm
  ```

- Refresh the cache:

  ```
  sudo dnf makecache
  ```

Install the package

```
sudo dnf -y install edb-postgresextended18-server edb-postgresextended18-contrib
```

Initial configuration

Getting started with your cluster involves logging in, ensuring the installation and initial configuration was successful, connecting to your cluster, and creating the user password.

First, you need to initialize and start the database cluster. The `edb-pge-18-setup` script creates a cluster.

```
sudo PGSETUP_INITDB_OPTIONS="-E UTF-8" /usr/edb/pge18/bin/edb-pge-18-setup initdb

sudo systemctl start edb-pge-18
```

To work in your cluster, log in as the postgres user. Connect to the database server using the psql command-line client. Alternatively, you can use a client of your choice with the appropriate connection string.

```
sudo -iu postgres

psql postgres
```

The server runs with the `peer` or `ident` permission by default. You can change the authentication method by modifying the `pg_hba.conf` file.

Before changing the authentication method, assign a password to the database superuser, postgres. For more information on changing the authentication, see Modifying the pg_hba.conf file.

```
ALTER ROLE postgres with PASSWORD
'password';
```

Experiment

Now you're ready to create and connect to a database, create a table, insert data in a table, and view the data from the table.

First, use psql to create a database named `hr` to hold human resource information.

```
# running in psql
CREATE DATABASE
hr;
```

output

```
CREATE DATABASE
```

Connect to the `hr` database inside psql:

```
\c hr
```

output

```
You are now connected to database "hr" as user "postgres".
```

Create columns to hold department numbers, unique department names, and locations:

```
CREATE TABLE public.dept (deptno numeric(2) NOT NULL CONSTRAINT dept_pk
PRIMARY KEY, dname varchar(14) CONSTRAINT dept_dname_uq UNIQUE, loc
varchar(13));
```

output

```
CREATE TABLE
```

Insert values into the `dept` table:

```
INSERT INTO dept VALUES (10,'ACCOUNTING','NEW YORK');
```

output

```
INSERT 0 1
```

```
INSERT into dept VALUES (20,'RESEARCH','DALLAS');
```

output

```
INSERT 0 1
```

View the table data by selecting the values from the table:

```
SELECT * FROM dept;
```

output

```
deptno  |   dname    |   loc
--------+------------+----------
10      | ACCOUNTING | NEW YORK
20      | RESEARCH   | DALLAS
(2 rows)
```

4.3.3 Installing EDB Postgres Extended Server on SLES 15 ppc64le

Prerequisites

Before you begin the installation process:

- Set up the EDB repository.

  Setting up the repository is a one-time task. If you have already set up your repository, you don't need to perform this step.

  To determine if your repository exists, enter this command:

  ```
  zypper lr -E | grep enterprisedb
  ```

  If no output is generated, the repository isn't installed.

  To set up the EDB repository:

  1. Go to EDB repositories.
  2. Select the button that provides access to the EDB repository.
  3. Select the platform and software that you want to download.
  4. Follow the instructions for setting up the EDB repository.

- Activate the required SUSE module:

  ```
  sudo SUSEConnect -p PackageHub/15.4/ppc64le
  ```

- Refresh the metadata:

  ```
  sudo zypper refresh
  ```

Install the package

```
sudo zypper -n install edb-postgresextended18-server edb-postgresextended18-contrib
```

Initial configuration

Getting started with your cluster involves logging in, ensuring the installation and initial configuration was successful, connecting to your cluster, and creating the user password.

First, you need to initialize and start the database cluster. The `edb-pge-18-setup` script creates a cluster.

```
sudo PGSETUP_INITDB_OPTIONS="-E UTF-8" /usr/edb/pge18/bin/edb-pge-18-setup initdb

sudo systemctl start edb-pge-18
```

To work in your cluster, log in as the postgres user. Connect to the database server using the psql command-line client. Alternatively, you can use a client of your choice with the appropriate connection string.

```
sudo -iu postgres

psql postgres
```

The server runs with the `peer` or `ident` permission by default. You can change the authentication method by modifying the `pg_hba.conf` file.

Before changing the authentication method, assign a password to the database superuser, postgres. For more information on changing the authentication, see Modifying the pg_hba.conf file.

```
ALTER ROLE postgres with PASSWORD
'password';
```

Experiment

Now you're ready to create and connect to a database, create a table, insert data in a table, and view the data from the table.

First, use psql to create a database named `hr` to hold human resource information.

```
# running in psql
CREATE DATABASE
hr;
```

output

```
CREATE DATABASE
```

Connect to the `hr` database inside psql:

```
\c hr
```

output

```
You are now connected to database "hr" as user "postgres".
```

Create columns to hold department numbers, unique department names, and locations:

```
CREATE TABLE public.dept (deptno numeric(2) NOT NULL CONSTRAINT dept_pk
PRIMARY KEY, dname varchar(14) CONSTRAINT dept_dname_uq UNIQUE, loc
varchar(13));
```

output

```
CREATE TABLE
```

Insert values into the `dept` table:

```
INSERT INTO dept VALUES (10,'ACCOUNTING','NEW YORK');
```

output

```
INSERT 0 1
```

```
INSERT into dept VALUES (20,'RESEARCH','DALLAS');
```

output

```
INSERT 0 1
```

View the table data by selecting the values from the table:

```
SELECT * FROM dept;
```

output

```
deptno  |   dname    |   loc
--------+------------+----------
10      | ACCOUNTING | NEW YORK
20      | RESEARCH   | DALLAS
(2 rows)
```

4.4 Default component locations

The package managers for the various Linux variations install EDB Postgres Extended Server components in different locations. If you need to access the components after installation, see:

- RHEL/OL/Rocky Linux/AlmaLinux/CentOS/SLES locations
- Debian/Ubuntu locations

RHEL/OL/Rocky Linux/AlmaLinux/CentOS/SLES Locations

The RPM installers place EDB Postgres Extended Server components in the directories listed in the table.

Component	Location
Executables	`/usr/edb/pge17/bin`
Libraries	`/usr/edb/pge17/lib`
Cluster configuration files	`/var/lib/edb-pge/17`
Documentation	`/usr/edb/pge17/share/man`
Contrib	`/usr/edb/pge17/share/contrib`
Data	`/var/lib/edb-pge/17/data`
Logs	`/var/log/edb/pge17`
Lock files	`/var/lock/edb/pge17`
Backup area	`/var/lib/edb-pge/17/backups`
Templates	`/usr/edb/pge17/share`
Procedural Languages	`/usr/edb/pge17/lib`
Development Headers	`/usr/edb/pge17/include`
Shared data	`/usr/edb/pge17/share`

Debian/Ubuntu Locations

The Debian package manager places EDB Postgres Extended Server components in the directories listed in the table.

Component	Location
Executables	`/usr/lib/edb-pge/17/bin`
Libraries	`/usr/lib/edb-pge/17/lib`
Cluster configuration files	`/var/lib/edb-pge/17/main`
Data	`/var/lib/edb-pge/17/main`
Logs	`/var/log/edb-pge/`
Lock files	`/var/lock/edb/pge17`

5 Database configuration

EDB Postgres Extended Server includes features to help you to maintain, secure, and operate EDB Postgres Extended Server databases.

You can configure grand unified configuration (GUC) parameters at runtime by modifying the `postgresql.conf` and `pg_hba.conf` files.

- The `postgresql.conf` file allows you to make persistent changes to your database configuration.
- The `pg_hba.conf` file allows you to change access and authentication settings.

See Setting configuration parameters for more information.

5.1 Setting configuration parameters

Set each configuration parameter using a name/value pair. Parameter names aren't case sensitive. The parameter name is typically separated from its value by an optional equals sign (`=`).

This example shows some configuration parameter settings in the `postgresql.conf` file:

```
# This is a
comment
log_connections = yes
log_destination =
'syslog'
search_path = '"$user", public'
shared_buffers = 128MB
```

Types of parameter values

Parameter values are specified as one of five types:

- **Boolean** — Acceptable values are `on` , `off` , `true` , `false` , `yes` , `no` , `1` , `0` , or any unambiguous prefix of these.
- **Integer** — Number without a fractional part.
- **Floating point** — Number with an optional fractional part separated by a decimal point.
- **String** — Text value enclosed in single quotes if the value isn't a simple identifier or number, that is, the value contains special characters such as spaces or other punctuation marks.
- **Enum** — Specific set of string values. The allowed values can be found in the system view `pg_settings.enumvals` . Enum values are not case sensitive.

Some settings specify a memory or time value. Each of these has an implicit unit, which is kilobytes, blocks (typically 8 kilobytes), milliseconds, seconds, or minutes. You can find default units by referencing the system view `pg_settings.unit` . You can specify a different unit explicitly.

Valid memory units are:

- `kB` (kilobytes)
- `MB` (megabytes)
- `GB` (gigabytes).

Valid time units are:

- `ms` (milliseconds)
- `s` (seconds)
- `min` (minutes)
- `h` (hours)
- `d` (days).

The multiplier for memory units is 1024.

Specifying configuration parameter settings

A number of parameter settings are set when the EDB Postgres Extended Server database product is built. These are read-only parameters, and you can't change their values. A couple of parameters are also permanently set for each database when the database is created. These parameters are read-only and you can't later change them for the database. However, there are a number of ways to specify the configuration parameter settings:

- The initial settings for almost all configurable parameters across the entire database cluster are listed in the `postgresql.conf` configuration file. These settings are put into effect upon database server start or restart. You can override some of these initial parameter settings. All configuration parameters have built-in default settings that are in effect unless you explicitly override them.

- Configuration parameters in the `postgresql.conf` file are overridden when the same parameters are included in the `postgresql.auto.conf` file. Use the `ALTER SYSTEM` command to manage the configuration parameters in the `postgresql.auto.conf` file.

- You can modify parameter settings in the configuration file while the database server is running. If the configuration file is then reloaded (meaning a SIGHUP signal is issued), for certain parameter types, the changed parameters settings immediately take effect. For some of these parameter types, the new settings are available in a currently running session immediately after the reload. For others, you must start a new session to use the new settings. And for some others, modified settings don't take effect until the database server is stopped and restarted. See the PostgreSQL core documentation for information on how to reload the configuration file.

- You can use the SQL commands `ALTER DATABASE`, `ALTER ROLE`, or `ALTER ROLE IN DATABASE` to modify certain parameter settings. The modified parameter settings take effect for new sessions after you execute the command. `ALTER DATABASE` affects new sessions connecting to the specified database. `ALTER ROLE` affects new sessions started by the specified role. `ALTER ROLE IN DATABASE` affects new sessions started by the specified role connecting to the specified database. Parameter settings established by these SQL commands remain in effect indefinitely, across database server restarts, overriding settings established by the other methods. You can change parameter settings established using the `ALTER DATABASE`, `ALTER ROLE`, or `ALTER ROLE IN DATABASE` commands by either:

    - Reissuing these commands with a different parameter value.

    - Issuing these commands using the `SET parameter TO DEFAULT` clause or the `RESET parameter` clause. These clauses change the parameter back to using the setting set by the other methods. See the PostgreSQL core documentation for the syntax of these SQL commands.

- You can make changes for certain parameter settings for the duration of individual sessions using the `PGOPTIONS` environment variable or by using the `SET` command in the EDB-PSQL or PSQL command-line programs. Parameter settings made this way override settings established using any of the methods discussed earlier, but only during that session.

Modifying the postgresql.conf file

The configuration parameters in the `postgresql.conf` file specify server behavior with regard to auditing, authentication, encryption, and other behaviors. On Linux and Windows hosts, the `postgresql.conf` file resides in the `data` directory under your EDB Postgres Extended Server installation.

Parameters that are preceded by a pound sign (#) are set to their default value. To change a parameter value, remove the pound sign and enter a new value. After setting or changing a parameter, you must either `reload` or `restart` the server for the new parameter value to take effect.

In the `postgresql.conf` file, some parameters contain comments that indicate `change requires restart`. To view a list of the parameters that require a server restart, use the following query at the psql command line:

```
SELECT name FROM pg_settings WHERE context =
'postmaster';
```

Modifying the pg_hba.conf file

Appropriate authentication methods provide protection and security. Entries in the `pg_hba.conf` file specify the authentication methods that the server uses with connecting clients. Before connecting to the server, you might need to modify the authentication properties specified in the `pg_hba.conf` file.

When you invoke the initdb utility to create a cluster, the utility creates a `pg_hba.conf` file for that cluster that specifies the type of authentication required from connecting clients. You can modify this file. After modifying the authentication settings in the `pg_hba.conf` file, restart the server and apply the changes. For more information about authentication and modifying the `pg_hba.conf` file, see the PostgreSQL core documentation.

When the server receives a connection request, it verifies the credentials provided against the authentication settings in the `pg_hba.conf` file before allowing a connection to a database. To log the `pg_hba.conf` file entry to authenticate a connection to the server, set the `log_connections` parameter to `ON` in the `postgresql.conf` file.

A record specifies a connection type, database name, user name, client IP address, and the authentication method to authorize a connection upon matching these parameters in the `pg_hba.conf` file. Once the connection to a server is authorized, you can see the matched line number and the authentication record from the `pg_hba.conf` file.

This example shows a log detail for a valid `pg_hba.conf` entry after successful authentication:

```
2020-05-08 10:42:17 IST LOG:  connection received: host=[local]
2020-05-08 10:42:17 IST LOG:  connection authorized: user=u1 database=edb
application_name=psql
2020-05-08 10:42:17 IST DETAIL:  Connection matched pg_hba.conf line 84:
"local   all             all               md5"
```

6 Transparent data encryption

Transparent data encryption (TDE) encrypts any user data stored in the database system. This encryption is transparent to the user. User data includes the actual data stored in tables and other objects as well as system catalog data such as the names of objects.

See Transparent data encryption for more information.

7 Managing password profiles

The `password_profile` extension introduces a profile-based system for enforcing advanced password management rules on user accounts. It gives privileged users the ability to apply crucial security restrictions, such as:

- Controlling detailed password policies (e.g., minimum length, complexity).
- Automatically locking accounts after repeated failed logins.

Essentially, this tool provides administrators with greater control and security over user passwords than standard PostgreSQL offers, which is vital for meeting modern compliance requirements.

7.1 Profile management key concepts

The EDB Postgres Extended Server provides a password profile extension to define and enforce password management rules for user accounts through profiles. This extension creates a new schema `password_profile` that contains functions, views and required function's to create and manage password profiles.

You implement password profiles by defining a profile using the `PG_PASSWORD_PROFILE` function. The profiles are then assigned to user accounts using the `PG_ATTACH_ROLE_PROFILE` function.

When a user attempts to change their password, the system checks the password against the rules defined in the assigned profile. If the password does not meet the criteria, the system rejects the change and provides feedback on which rules were violated.

7.2 Creating a password profile

The `password_profile` extension automatically creates a system-generated `DEFAULT` profile that sets the default limits for all the parameters. You can change the parameter values for the `DEFAULT` profile, however it can't be dropped or renamed. You can create new profiles and set any parameter to -1 to inherit the corresponding value from the `DEFAULT` profile.

The `PG_CREATE_PROFILE` function defines a new profile for password management.

Synopsis

```
PASSWORD_PROFILE.PG_CREATE_PROFILE('profilename',
                                   failed_login_attempts,
                                   password_lock_time,
                                   password_life_time,
                                   password_grace_time,
                                   password_verify_function)
```

Description

The `PG_CREATE_PROFILE` function creates a new profile for password management. The profile defines the profile name, limits for failed login attempts, account lock time, password expriation and grace period, and the password verification function for complexity checks.

Parameters

- `profilename` — Name of the password profile to create.
- `failed_login_attempts` — Number of consecutive failed login attempts allowed before the account is locked for the length of time specified by `PASSWORD_LOCK_TIME`. Supported values are:
    - An `integer` value greater than `0`.
    - `Default` — The value of `failed_login_attempts` specified in the `DEFAULT` profile.
    - `unlimited` — The connecting user can make an unlimited number of failed login attempts.
- `password_lock_time` — Duration (in days) for which the account remains locked after exceeding the allowed failed login attempts. Supported values are:
    - A `numeric` value greater than or equal to `0`. To specify a fractional portion of a day, specify a decimal value. For example, use the value `4.5` to specify 4 days, 12 hours.
    - `Default` — The value of `password_lock_time` specified in the `DEFAULT` profile.
    - `unlimited` — The account is locked until a database superuser manually unlocks it.

- `password_life_time` — Duration (in days) after which the password expires and must be changed. Include the `password_grace_time` clause when using the `password_life_time` clause to specify the number of days that pass after the password expires before connections by the role are rejected. If you don't specify `password_grace_time`, the password expires on the day specified by the default value of `password_grace_time`, and the user can't execute any command until they provide a new password. Supported values are:

    - A `numeric` value greater than or equal to 0. To specify a fractional portion of a day, specify a decimal value. For example, use the value `4.5` to specify 4 days, 12 hours.
    - `Default` — The value of `password_life_time` specified in the `DEFAULT` profile.
    - `unlimited` — The password doesn't have an expiration date.

- `password_grace_time` — Duration (in days) after password expiration during which the user can still log in and change the password. When the grace period expires, a user can connect but can't execute any command until they update their expired password. Supported values are:

    - A `numeric` value greater than or equal to 0. To specify a fractional portion of a day, specify a decimal value. For example, use the value `4.5` to specify 4 days, 12 hours.
    - `Default` — The value of `password_grace_time` specified in the `DEFAULT` profile.
    - `unlimited` — The grace period is infinite.

- `password_verify_function` — Name of the password verification function that checks the complexity of the new password. This function should return true if the password meets the complexity requirements, and false otherwise. Supported values are:

- The name of a PL/SQL function.

- `Default` — The value of `password_verify_function` specified in the `default` profile.

- `NULL`

Caveats

- Role accounts are not automatically attached to the DEFAULT profile. Users must explicitly assign this profile if desired.

- The DEFAULT profile is essential for system operation and must not be removed. Users may, however, reset its values to the original settings.

- Upon dropping the `password_profile` extension, users must manually clean up data remaining in the PG_PROFILE and PG_AUTH_PROFILE tables.

- If a role account's password is already stored as a hash, direct comparison is not possible. This will result in an incorrect setting for the BOOLEAN argument of the PASSWORD_VERIFY_FUNCTION.

See also

`EXAMPLE`, `MODIFYING A PROFILE`, `REMOVING A PROFILE`

7.3 Modifying a profile

Altering a password profile

The `PG_ALTER_PROFILE` function modifies an existing password profile for password management.

Synopsis

```
PASSWORD_PROFILE.PG_ALTER_PROFILE('profilename',
                                  failed_login_attempts,
                                  password_lock_time,
                                  password_life_time,
                                  password_grace_time,
                                  password_verify_function)
```

Description

The `PG_ALTER_PROFILE` function modifies an existing profile for password management. The profile defines the profile name, limits for failed login attempts, account lock time, password expiration and grace period, and the password verification function for complexity checks.

Parameters

- `profilename` — Name of the password profile to modify.
- `failed_login_attempts` — Number of consecutive failed login attempts allowed before the account is lockedfor the length of time specified by `PASSWORD_LOCK_TIME`. Supported values are:
  - An `integer` value greater than `0`.
  - `Default` — The value of `failed_login_attempts` specified in the `DEFAULT` profile.
  - `unlimited` — The connecting user can make an unlimited number of failed login attempts.
- `password_lock_time` — Duration (in days) for which the account remains locked after exceeding the allowed failed login attempts. Supported values are:
  - A `numeric` value greater than or equal to `0`. To specify a fractional portion of a day, specify a decimal value. For example, use the value `4.5` to specify 4 days, 12 hours.
  - `Default` — The value of `password_lock_time` specified in the `DEFAULT` profile.
  - `unlimited` — The account is locked until a database superuser manually unlocks it.
- `password_life_time` — Duration (in days) after which the password expires and must be changed. Include the `password_grace_time` clause when using the `password_life_time` clause to specify the number of days that pass after the password expires before connections by the role are rejected. If you don't specify `password_grace_time`, the password expires on the day specified by the default value of `password_grace_time`, and the user can't execute any command until they provide a new password. Supported values are:
  - A `numeric` value greater than or equal to 0. To specify a fractional portion of a day, specify a decimal value. For example, use the value `4.5` to specify 4 days, 12 hours.
  - `Default` — The value of `password_life_time` specified in the `DEFAULT` profile.
  - `unlimited` — The password doesn't have an expiration date.

- `password_grace_time` — Duration (in days) after password expiration during which the user can still log in and change the password. When the grace period expires, a user can connect but can't execute any command until they update their expired password. Supported values are:
    - A `numeric` value greater than or equal to 0. To specify a fractional portion of a day, specify a decimal value. For example, use the value `4.5` to specify 4 days, 12 hours.
    - `Default` — The value of `password_grace_time` specified in the `DEFAULT` profile.
    - `unlimited` — The grace period is infinite.
- `password_verify_function` — Name of the password verification function that checks the complexity of the new password. This function should return true if the password meets the complexity requirements, and false otherwise. Supported values are:
- The name of a PL/SQL function.
- `Default` — The value of `password_verify_function` specified in the `default` profile.
- `NULL`

Renaming a password profile

The `PG_RENAME_PROFILE` function renames an existing password profile for password management.

Synopsis

```
PASSWORD_PROFILE.PG_RENAME_PROFILE('old_profilename',
                                   'new_profilename')
```

Description

The `PG_RENAME_PROFILE` function renames an existing profile for password management.

Parameters

- `old_profilename` — Current name of the password profile to rename.
- `new_profilename` — New name for the password profile.

See also

`EXAMPLE`, `CREATING A PROFILE`, `REMOVING A PROFILE`

7.4 Dropping a password profile

The `PG_DROP_PROFILE` function drops a password profile.

Synopsis

```
PASSWORD_PROFILE.PG_DROP_PROFILE('profilename',

missing_ok)
```

Description

`PG_DROP_PROFILE` drops the specified password profile for password management.

To use `PG_DROP_PROFILE`, you must not have any users assigned to the profile.

Parameters

- `profilename` — Name of the password profile to drop.
- `missing_ok` — If set to true, the function does not raise an error if the specified profile does not exist. If set to false, an error is raised when the specified profile does not exist.

See also

`CREATE PROFILE`, `MODIFYING A PROFILE`

7.5 Roles and profiles

The `password_profile` extension allows you to manage user roles and their associated profiles effectively. Each role can be assigned a specific profile that dictates the password management rules applicable to that role.

Attach role and profile

When a role is created, it can be associated with a password profile using the `PG_ATTACH_ROLE_PROFILE` function. This association ensures that the password management rules defined in the profile are enforced for the role.

```
PASSWORD_PROFILE.PG_ATTACH_ROLE_PROFILE('role_name', 'profile_name');
```

This function accepts following parameters:

- `role_name` — Name of the role to which the profile is being attached.
- `profile_name` — Name of the password profile to attach to the role.

Detach role and profile

When a role is no longer needed or requires a different profile, it can be unassociated from its current profile using the `PG_DETACH_ROLE_PROFILE` function.

```
PASSWORD_PROFILE.PG_DETACH_ROLE_PROFILE('role_name');
```

This function accepts following parameters:

- `role_name` — Name of the role from which the profile is being detached.

Role account status

You can check the status of a role's account by using the `PG_GET_ROLE_STATUS` function that is attached to the profile.

```
PASSWORD_PROFILE.PG_GET_ROLE_STATUS('role_name');

or

PASSWORD_PROFILE.PG_GET_ROLE_STATUS('roleid');
```

This function accepts either parameter:

- `role_name` — Name of the role whose account status is being queried.
- `roleid` — ID of the role whose account status is being queried.

Lock/unlock a role account

You can lock or unlock a role's account using the `PG_ROLE_ACCOUNT_LOCK` function. This action prevents the role from logging in until it is unlocked.

```
PASSWORD_PROFILE.PG_ROLE_ACCOUNT_LOCK('role_name', lock);
```

This function accepts following parameters:

- `role_name` — Name of the role whose account is to be locked or unlocked.
- `lock` — If true, lock the role's account; if false, unlock the role's account.

Set a role password to expire

You can set a role's password to expire using the `PG_ROLE_PASSWORD_EXPIRE` function. This action forces the role to change its password upon the next login.

```
PASSWORD_PROFILE.PG_ROLE_PASSWORD_EXPIRE('role_name');
```

This function accepts following parameters:

- `role_name` — Name of the role whose password is to be expired.

See also

`CREATING A PROFILE`, `MODIFYING A PROFILE`, `REMOVING A PROFILE`

7.6 Password profiles system catalogs

The `password_profile` extension creates the system catalogs that stores profile information.

pg_catalog.pg_profile

The `pg_catalog.pg_profile` system catalog stores information about the profiles and the password management rules added to that profile.

Column	Description
`oid`	The unique number to identify a profile. `oid` is assigned to a profile by a sequence.
`prfname`	Name of the profile.
`prffailedloginattempts`	Number of failed login attempts allowed by the profile.
`prfpasswordlocktime`	Locking duration in seconds after a role is locked.
`prfpasswordlifetime`	Validity of current role password in seconds.
`prfpasswordgracetime`	Duration in seconds after password expiration during which the role can still log in and change the password.
`prfpasswordverifyfuncdb`	The database OID in which the password verify function is created.
`prfpasswordverifyfunc`	The password verify function name created by a user to impose password rules.

pg_catalog.pg_auth_profile

Column	Description
`roleid`	The unique number to identify a role.
`profileid`	The object identifier for the role's profile.
`roleaccountstatus`	Account status of the role.
`rolefailedlogins`	Number of login failures of the role.
`rolelockdate`	The timestamp when the role was last locked.
`rolepasswordsetat`	The timestamp when the role's password was last set or modified.
`rolepasswordexpire`	The timestamp when the role's password expires.

pg_catalog.pg_profile_info

The `pg_catalog.pg_profile_info` system catalog view stores information about the profiles and the corresponding parameter values.

Column	Description
`profile_name`	Name of the profile.
`failed_login_attempts`	Number of failed login attempts allowed by the profile.
`password_lock_time`	Locking duration in days after a role is locked.
`password_life_time`	Validity of current role password in days.

Column	Description
`password_grace_time`	Duration in days after password expiration during which the role can still log in and change the password.
`password_verify_function`	The password verify function name created by a user, in `<database name>.<schema name>.<function name>` format, to impose password rules.

pg_catalog.pg_role_profile_info

The `pg_catalog.pg_role_profile_info` system catalog view stores information about the roles and their associated profiles.

Column	Description
`role_name`	Name of the role.
`profile_name`	Name of the profile associated with the role.
`role_status`	Status of the role.
`role_lock_date`	The timestamp when the role was last locked.
`role_expiry_date`	The timestamp when the role's password expires.

7.7 Working example of a password profile

This example walks you through the process of creating a password profile and associating it with a role. The CREATE EXTENSION command is used to add the password profile functionality to your database.

- You need to load the `password_profile` libraries via `shared_preload_libraries` in postgresql.conf file.

  Add `password_profile` to the `shared_preload_libraries` line:

  ```
  shared_preload_libraries = 'password_profile'
  ```

  Restart the database server.

- Create the extension in your database:

  ```
  CREATE EXTENSION password_profile;
  ```

- Create a user and `GRANT EXECUTE` privileges to the user for all functions in the `password_profile` schema:

  ```
  CREATE USER user1 WITH PASSWORD 'password';
  GRANT EXECUTE ON SCHEMA password_profile TO
  user1;
  ```

  ```
  GRANT EXECUTE ON FUNCTION password_profile.pg_create_profile(text, numeric, numeric, numeric,
  numeric, text) TO user1;
  GRANT EXECUTE ON FUNCTION password_profile.pg_alter_profile(text, numeric, numeric, numeric, numeric,
  text) TO user1;
  GRANT EXECUTE ON FUNCTION password_profile.pg_rename_profile(text, text) TO
  user1;
  GRANT EXECUTE ON FUNCTION password_profile.pg_attach_role_profile(text, text) TO
  user1;
  GRANT EXECUTE ON FUNCTION password_profile.pg_role_account_lock(text, boolean) TO
  user1;
  GRANT EXECUTE ON FUNCTION password_profile.pg_get_role_status(oid) TO
  user1;
  GRANT EXECUTE ON FUNCTION password_profile.pg_get_role_status(text) TO
  user1;
  GRANT EXECUTE ON FUNCTION password_profile.pg_role_password_expire(text) TO
  user1;
  GRANT EXECUTE ON FUNCTION password_profile.pg_detach_role_profile(text) TO
  user1;
  GRANT EXECUTE ON FUNCTION password_profile.pg_drop_profile(text, boolean) TO
  user1;
  ```

- Create a new profile as a superuser or as user1

  ```
  SELECT PASSWORD_PROFILE.PG_CREATE_PROFILE('my_profile');
  ```

  This command creates a new profile and adds an entry to `PG_PROFILE` catalog.

- Alter the profile to set `failed_login_attempts` to 2, `password_lock_time` to 1 day, `password_life_time` to 3 days and `password_grace_time` to 1 day.

```
SELECT PASSWORD_PROFILE.PG_ALTER_PROFILE('my_profile', 2, 1, 3,
1);

SELECT * FROM
pg_profile;
```

```
output
  oid  |  prfname   | prffailedloginattempts | prfpasswordlocktime | prfpasswordlifetime |
prfpasswordgracetime | prfpasswordverifyfuncdb | prfpasswordverifyfunc
-------+-----------+------------------------+---------------------+---------------------+-------------
------+------------------------+----------------------
  6108 | default   |                     -2 |                  -2 |                  -2 |
-2 |                        |
 16397 | myprofile |                      2 |               86400 |              259200 |
86400 |                      0 |                      0
 (2 rows)
```

The `prfpasswordverifyfunc` and `prfpasswordverifyfuncdb` column values are null for `DEFAULT` profile. For `myprofile` they are set to 0, as no values were provided while creating this profile, so it referes to `DEFAULT` profile values.

- Attach the profile to a role

```
SELECT PASSWORD_PROFILE.PG_ATTACH_ROLE_PROFILE('user1', 'myprofile');

SELECT * FROM
pg_auth_profile;
```

```
output
 roleid | profileid | roleaccountstatus | rolefailedlogins | rolelockdate |       rolepasswordsetat
| rolepasswordexpire
--------+-----------+-------------------+------------------+--------------+--------------------------
-+--------------------
  16384 |     16397 |                 0 |                0 |              | 2025-08-23
05:02:28.70456+00 |
(1 row)
```

Here the `rolepasswordsetat` column shows the timestamp when the role is attached to the profile. This column value gets updated whenever the role password is updated.

- Now try one failed login attempt and then query:

```
SELECT roleid, password_profile.PG_GET_ROLE_STATUS('myuser'), rolefailedlogins, rolelockdate FROM
pg_auth_profile;
```

```
output
roleid | pg_get_role_status | rolefailedlogins | rolelockdate
-------+--------------------+------------------+--------------
 16384 | OPEN               |                1 |
(1 row)
```

You can see the `rolefailedlogins` column has been incremented by 1 and the role status is `open`.

- Now try more failed log in attempts and then query again:

```
\c -
myuser
Password for user myuser:
FATAL:  role "myuser" is locked
Previous connection kept

SELECT roleid, password_profile.PG_GET_ROLE_STATUS('myuser'), rolefailedlogins, rolelockdate FROM
pg_auth_profile;
```

output

```
roleid | pg_get_role_status | rolefailedlogins |          rolelockdate
-------+--------------------+------------------+-------------------------------
 16384 | LOCKED(TIMED)      |                0 | 2025-08-24 07:53:58.524348+00
(1 row)
```

Due to multiple failed login attempts, the user account has been locked. The role status shows as `LOCKED(TIMED)`, and the failed login attempt counter has been reset to `0`.

The TIMED status indicates that the account will automatically unlock after one day, as defined by the `PASSWORD_LOCK_TIME` parameter. A timestamp is also recorded when the account becomes locked.

Once the lock period expires, the user will be able to log in again, and the status will return to OPEN.

- If the user successfully logs in after a few failed attempts but before reaching the maximum limit (set by `FAILED_LOGIN_ATTEMPTS`), the failed login counter is also reset to 0.

```
\c -
myuser
Password for user myuser:
You are now connected to database "postgres" as user
"myuser".

SELECT roleid, password_profile.PG_GET_ROLE_STATUS('myuser'), rolefailedlogins, rolelockdate FROM
pg_auth_profile;
```

output

```
roleid | pg_get_role_status | rolefailedlogins |          rolelockdate
-------+--------------------+------------------+-------------------------------
 16384 | OPEN               |                0 | 2025-08-24 07:53:58.524348+00
(1 row)
```

It preserves the last lock date of the role.

8 Redacting data

EDB Postgres Extended Server includes features to help you to maintain, secure, and operate EDB Postgres Extended Server databases. The EDB Postgres Extended Server *Data redaction* feature limits sensitive data exposure by dynamically changing data as it's displayed for certain users.

8.1 Data redaction key concepts

The EDB Postgres Extended Server *Data redaction* extension provides a mechanism to limit sensitive data exposure by dynamically modified data presented to non-privileged users. This extension creates a new schema `data_redaction` that contains functions, views and required function's to create and manage data redaction policies.

To use the data redaction feature, first add the `data_redaction` libraries to `shared_preload_libraries` parameter in the postgresql.conf file and restart the database server.

You implement data redaction by defining a function for each field to which to apply redaction. The function returns the value to display to the users subject to the data redaction.

These functions are then incorporated into a redaction policy by using the `CREATE_REDACTION_POLICY` function. In addition to other options, this command specifies:

- The table on which the policy applies
- The table columns affected by the specified redaction functions
- Expressions to determine the affect session users

When a user queries a table with an active redaction policy, the system evaluates the policy expression. If the expression evaluates to true for that user session, the system applies the redaction functions to the specified columns in the query result.

For example,

- A social security number (SSN) is stored as `021-23-9567`. Privileged users can see the full SSN, while other users see only the last four digits: `xxx-xx-9567`.
- For the SSN field, the redaction function returns `xxx-xx-9567` for an input SSN of `021-23-9567`.
- For a salary field, a redaction function always returns `$0.00`, regardless of the input salary value.

8.2 Creating a data redaction policy

The `CREATE_REDACTION_POLICY` function defines a new data redaction policy for a table.

Synopsis

```
DATA_REDACTION.CREATE_REDACTION_POLICY('policy_name',
                                       'table_name',
                                       'schema_name',
                                       'expression',
                                       'column_name',
                                       'func',
                                       'scope_value',
                                       'exception_value')
```

Description

The `CREATE_REDACTION_POLICY` function creates a new data redaction policy for a table. The data redaction policy specifies how to redact sensitive data for certain users by applying redaction functions to the specified columns of the table.

You must be the owner of a table to create or change data redaction policies for it.

The superuser and the table owner are exempt from the data redaction policy.

Parameters

- `policy_name` — Name of the data redaction policy to create.
- `table_name` — Name of the table the data redaction policy applies to.
- `schema_name` — Schema name in which the table resides.
- `expression` — The data redaction policy expression. No redaction is applied if this expression evaluates to false.
- `column_name` — Name of the existing column of the table on which the data redaction policy is being created.
- `func` — The data redaction function that decides how to compute the redacted column value. Return type of the redaction function must be the same as the column type on which the data redaction policy is being added.
- `scope_value` — The scope identifies the query part to apply redaction for the column. Scope value can be `query`, `top_tlist`, or `top_tlist_or_error`.
    - If the scope is `query`, then the redaction is applied on the column regardless of where it appears in the query.
    - If the scope is `top_tlist`, then the redaction is applied on the column only when it appears in the query's top target list.
    - If the scope is `top_tlist_or_error`, the behavior is the same as the `top_tlist` but throws an errors when the column appears anywhere else in the query.

- `exception_value` — The exception identifies the query part where redaction is exempted. Exception value can be `none`, `equal`, or `leakproof`.

    - If exception is `none`, then there's no exemption.
    - If exception is `equal`, then the column isn't redacted when used in an equality test.
    - If exception is `leakproof`, the column isn't redacted when a leakproof function is applied to it.

Caveats

- The data redaction policies created on inheritance hierarchies aren't cascaded. For example, if the data redaction policy is created for a parent, it isn't applied to the child table that inherits it, and vice versa. A user with access to these child tables can see the non-redacted data. For information about inheritance hierarchies, see the PostgreSQL core documentation.

- Users must manually adjust redaction policies when renaming or dropping any relevant objects.

- If non-super user takes a dump, it has redacted values then user will get redacted values when restores that dump.

See also

`EXAMPLE`, `ENABLE/DISABLE REDACTION POLICY`, `DROP REDACTION POLICY`

8.3 Enable/Disable data redaction policy

The `ENABLE_REDACTION_POLICY` function enables the data redaction policy and `DISABLE_REDACTION_POLICY` function disables the data redaction policy for a table.

Synopsis

```
# Enable a data redaction
policy:
DATA_REDACTION.ENABLE_REDACTION_POLICY('policy_name',
                                       'table_name',
                                       'schema_name')

# Disable a data redaction
policy:
DATA_REDACTION.DISABLE_REDACTION_POLICY('policy_name',
                                        'table_name',
                                        'schema_name')
```

Description

By default, when a data redaction policy is created, it is enabled.

`ENABLE_REDACTION_POLICY` function enables the specified data redaction policy for the table, while `DISABLE_REDACTION_POLICY` function disables it.

Parameters

- `policy_name` — Name of the data redaction policy to enable or disable.
- `table_name` — Name of the table the data redaction policy applies to.
- `schema_name` — Schema name in which the table resides.

If data redaction policy is enabled then,

- Superusers and the table owner bypass data redaction and see the original data.
- All other users have the redaction policy applied and see the reformatted data.

If data redaction policy is disabled then,

- Superusers and the table owner bypass data redaction and see the original data.
- All other users get un-redacted data.

See also

`CREATE REDACTION POLICY`, `DROP REDACTION POLICY`

8.4 Removing a data redaction policy

The `DROP_REDACTION_POLICY` function removes a data redaction policy from a table.

Synopsis

```
DATA_REDACTION.DROP_REDACTION_POLICY('policy_name',
                                     'table_name',
                                     'schema_name')
```

Description

`DROP_REDACTION_POLICY` removes the specified data redaction policy from the table.

To use `DROP_REDACTION_POLICY`, you must own the table that the redaction policy applies to.

Parameters

- `policy_name` — Name of the data redaction policy to remove.
- `table_name` — Name of the table the data redaction policy applies to.
- `schema_name` — Schema name in which the table resides.

See also

`CREATE REDACTION POLICY`, `ENABLE/DISABLE REDACTION POLICY`

8.5 Data redaction system catalogs

System catalogs store the redaction policy information.

pg_redaction_policy

The catalog `data_redaction.pg_redaction_policy` stores information about the redaction policies for tables.

Column	Description
`rppolicyid`	The unique number to identify a data retention policy. The value in this field is added by a sequence.
`rpname`	The name of the data redaction policy.
`rprelname`	The relation on which redaction policy is created.
`rprelschema`	Schema name in which the relation resides.
`rpenable`	Status of the redaction policy. The default value is `true` . If the value is `true` means the policy is enabled and active. If the value is `false` means the policy is disabled and inactive.
`rpexpr`	The data redaction policy expression. If the expression is evaluated true then the policy is applied.
`rpsearchpath`	The value of search path set while creating the redaction policy.

> **Note**
>
> The data redaction policy applies for the relation if it's enabled and the expression ever evaluated true.

pg_redaction_column

The `data_redactionp.pg_redaction_column` system catalog stores information about the data redaction policy attached to the columns of a table.

Column	Description
`rcpolicyid`	The unique number to identify a data redaction policy.
`rcrelname`	The relation on which redaction policy is created.
`rcrelschema`	Schema name in which the relation resides.
`rcattname`	Relation's column name having the redaction policy.
`rcscope`	The scope defined for the redaction policy.
`rcexception`	The exception defined for the redaction policy.
`rcfuncexpr`	The redaction function name that defines on how to redact the data.

> **Note**
>
> The described column is redacted if the redaction policy `data_redaction.pg_redaction_column.rcpolicyid` on the table is enabled and the redaction policy expression `data_redaction.pg_redaction_policy.rpexpr` evaluates to `true`.

redaction_policy_status

The `redaction_policy_status` view shows the health of all current data redaction policies, reporting a status of either `Valid` or `Invalid`.

A `Valid` status means the policy is correctly configured and all required underlying objects (like functions or tables) are present.

An `Invalid` status indicates that the policy cannot be properly applied because one or more of its necessary objects are missing or no longer exist.

8.6 Working example of a data redaction policy

This example walks you through how to create and implement a data redaction policy on a table.

Create the components for a data redaction policy on the `employees` table:

```
CREATE TABLE employees(
 id          integer GENERATED BY DEFAULT AS IDENTITY PRIMARY
KEY,
 name        varchar(40) NOT NULL,
 ssn         varchar(11) NOT
NULL,
 phone       varchar(10),
 birthday
date,
 salary
money,
 email       varchar(100)
);
```

Insert some data to the employees table:

```
INSERT INTO employees (name, ssn, phone, birthday,
salary,
email)
VALUES
( 'Sally Sample', '020-78-9345', '5081234567', '1961-02-02', 51234.34,
'sally.sample@enterprisedb.com'),
( 'Jane Doe', '123-33-9345', '6171234567', '1963-02-14', 62500.00,
'jane.doe@gmail.com');
```

Create a user hr who can see all the data in the employees table:

```
CREATE USER
hr;
```

Create a normal user and grant all privileges to hr and alice:

```
CREATE USER
alice;
GRANT ALL on employees to hr,
alice;
```

Create redaction function in which actual redaction logic resides:

```
CREATE OR REPLACE FUNCTION redact_ssn(ssn name) RETURNS varchar AS
$$
BEGIN
    return overlay (ssn placing 'xxx-xx' from 1);
END;
$$ LANGUAGE plpgsql;
CREATE OR REPLACE FUNCTION redact_salary(salary money) RETURNS money AS
$$
BEGIN
    return 0::money;
END;
$$ LANGUAGE plpgsql;
```

These functions are then incorporated into a redaction policy by using the `CREATE_REDACTION_POLICY` function.

To use the data redaction feature, first add the `data_redaction` libraries by adding the following lines to the `postgresql.conf` file and restarting the database server:

```
# add data redaction libraries in the postgresql.conf file:
shared_preload_libraries = 'data_redaction'
```

Restart the database server and create the extension:

Create an extension and then create a data redaction policy on `employees` table to redact `ssn` and `salary` columns with the default scope and exception. Column `ssn` must be accessible in equality condition. The redaction policy is exempted for the `hr` user.

```
CREATE EXTENSION data_redaction;
SELECT data_redaction.CREATE_REDACTION_POLICY('redact_personal_info_ssn', 'employees',
'public',
                                              'session_user != ''hr''', 'ssn', 'redact_ssn(ssn)',
                                              'query', 'equal');
SELECT data_redaction.CREATE_REDACTION_POLICY('redact_personal_info_salary', 'employees',
'public',
                                              'session_user != ''hr''', 'salary', 'redact_salary(salary)');
```

The `hr` user can view all columns data:

```
\c hr
SELECT * FROM employees;
```

output

```
 id |     name     |     ssn     |   phone    |  birthday  |   salary   |            email
----+--------------+-------------+------------+------------+------------+-------------------------------
  1 | Sally Sample | 020-78-9345 | 5081234567 | 1961-02-02 | $51,234.34 | sally.sample@enterprisedb.com
  2 | Jane Doe     | 123-33-9345 | 6171234567 | 1963-02-14 | $62,500.00 | jane.doe@gmail.com
(2 rows)
```

The `alice` user can't see the actual data in `ssn` and `salary` columns:

```
\c edb
alice
SELECT * FROM employees;
```

output

```
 id |     name     |     ssn     |   phone    |  birthday  | salary |            email
----+--------------+-------------+------------+------------+--------+-------------------------------
  1 | Sally Sample | xxx-xx-9345 | 5081234567 | 1961-02-02 |  $0.00 | sally.sample@enterprisedb.com
  2 | Jane Doe     | xxx-xx-9345 | 6171234567 | 1963-02-14 |  $0.00 | jane.doe@gmail.com
(2 rows)
```

9 Replication

EDB Postgres Extended Server provides the core functionality to support the following replication and high availability features inEDB Postgres Distributed:

- Commit At Most Once (CAMO)
- Group commit
- Eager replication
- Decoding worker
- Assessment tooling
- Lag tracker
- Lag control
- Timestamp snapshots
- Transaction streaming
- Missing partition conflict
- No need for UPDATE trigger on tables with TOAST
- Automatically hold back FREEZE

Asynchronous processing

EDB Postgres Extended Server includes a `synchronous_replication_availability` parameter. A value of `async` for this parameter enables asynchronous processing when not enough standby servers are available (when compared with the values as per `synchronous_standby_names`). The behavior reverts to synchronous replication when the required number of synchronous standby servers reappear.

10 Postgres extensions supported in EDB Postgres Extended Server

EDB provides support for several Postgres extensions on EDB Postgres Extended Server:

- Open-source extensions
- EDB supported open-source extensions
- EDB-developed extensions

See Postgres extensions available by deployment for an overview of all supported extensions and links to their documentation sites.

11 Upgrading EDB Postgres Extended Server

You can upgrade EDB Postgres Extended Server installations to a more recent version.

- See Upgrading a major version of EDB Postgres Extended Server for a major upgrade example.
- See Upgrading a minor version of EDB Postgres Extended Server for minor upgrade examples according to your package format.

11.1 Major version upgrade of EDB Postgres Extended Server

To perform a major version upgrade, install the new version of EDB Postgres Extended Server, initialize an empty cluster and use pg_upgrade to migrate all data.

If a problem occurs during the upgrade process, you can revert to the previous version.

Overview

1. Prepare your upgrade by performing a backup of the existing instance.
2. Install the EDB Postgres Extended Server version you're upgrading toward.
3. Create a new database server:
    1. Create an empty directory for the new server and ensure postgres owns it.
    2. Initialize a server on a different port from the source server.
    3. Start the database server.
    4. Connect to the database server and ensure it's functioning.
4. Upgrade to the target server:
    1. Stop both the source and the new server.
    2. Use pg_upgrade by specifying the source and target bin and data directories.
    3. Start the new database server.
    4. Connect to the encrypted database server and ensure the data was transferred.
5. Clean up and delete the source server:
    1. Clean up the database and its statistics.
    2. Remove the source EDB Postgres Extended Server cluster with the script provided by pg_upgrade.

Worked example

This worked example upgrades an EDB Postgres Extended Server 16 database to EDB Postgres Extended Server 17.

> **Note**
>
> You can perform major upgrades of EDB Postgres Extended Server instances in the same way you upgrade an EDB Postgres Advanced Server installation. If you need more information about the pg_upgrade utility, command line options, troubleshooting, and more, see Upgrading an installation with pg_upgrade.

Preparing your upgrade

Use pg_dumpall, pgBackRest, or Barman to create a backup of your source server.

Installing the target EDB Postgres Extended Server version

Install EDB Postgres Extended Server version 17. Only install the packages. Don't perform any other configurations.

Creating a target server

If you don't want to create a new target instance but want to reuse an existing server with the target EDB Postgres Extended Server version, skip these steps and ensure the target server is empty.

1. As postgres, create an empty directory for the new server:

```
mkdir /var/lib/edb-pge/17/upgrade_target
```

2. As root, ensure the postgres user owns the directory:

```
sudo chown postgres /var/lib/edb-pge/17/upgrade_target
sudo chgrp postgres /var/lib/edb-pge/17/upgrade_target
```

3. As postgres, initialize the new server:

```
/usr/lib/edb-pge/17/bin/initdb -D /var/lib/edb-pge/17/upgrade_target
```

   This command initializes a CONFIG directory with all configuration files for the encrypted server.

4. Before you start the cluster, ensure the new database runs on a different port from the source server. To alter the port, edit `postgresql.conf` by uncommenting the line with `#port` and changing the port number, for example, to 5432.

5. Start the target server:

```
/usr/lib/edb-pge/17/bin/pg_ctl -D /var/lib/edb-pge/17/upgrade_target start
```

   > **Note**
   >
   > You can also start the server with the `logfile` option enabled to print errors into a logfile: `/usr/lib/edb-pge/17/bin/pg_ctl -D /var/lib/edb-pge/17/upgrade_target -l logfile start`
   >
   > In this case, ensure the postgres user has rights to write to the log directory.

6. Connect to the server:

```
/usr/lib/edb-pge/17/bin/psql -p 5432
```

   > **Note**
   >
   > If you're using two different Postgres versions, use the psql utility of the target server. Otherwise, the system will attempt to use psql from the previous instance.

Upgrading to the target server

1. If you have any extensions or component services running in the source cluster, stop them before starting the upgrade. See Stop all component services and servers for more information

2. Stop both the source and target servers:

```
/usr/lib/edb-pge/16/bin/pg_ctl -D /var/lib/edb-pge/16/upgrade-source stop
/usr/lib/edb-pge/17/bin/pg_ctl -D /var/lib/edb-pge/17/upgrade-target stop
```

3. To test for incompatibilities, run the `pg_upgrade` command in check mode.

   With `-b` and `-B`, specify the source and target BIN directories. With `-d` and `-D`, specify the source and target CONFIG directories:

   ```
   /usr/lib/edb-pge/17/bin/pg_upgrade -b /usr/lib/edb-pge/16/bin -B /usr/lib/edb-pge/17/bin -d
   /var/lib/edb-pge/16/upgrade-source -D /var/lib/edb-pge/17/upgrade-target --check
   ```

   > **Note**
   >
   > The `--check` mode performs preliminary checks without executing the command.

4. To copy data from the source server to the target server, run the `pg_upgrade` command in normal mode:

   ```
   /usr/lib/edb-pge/17/bin/pg_upgrade -b /usr/lib/edb-pge/16/bin -B /usr/lib/edb-pge/17/bin -d
   /var/lib/edb-pge/16/upgrade-source -D /var/lib/edb-pge/17/upgrade-target
   ```

5. Start the target server:

   ```
   /usr/lib/edb-pge/17/bin/pg_ctl -D /var/lib/edb-pge/17/upgrade-target start
   ```

6. Connect to the target database server:

   ```
   /usr/lib/edb-as/17/bin/psql -p 5432
   ```

7. Perform a spot check to ensure the databases, tables, schemas, and resources you had in the unencrypted server are available in the new server. For example, list all databases, explore the database objects, views, and so on.

8. Restart the extensions or component services you disabled in the source cluster but in the target cluster.

Cleaning up after upgrade

After you verify that pg_upgrade migrated the data successfully, and the services are running as expected, perform a cleanup.

1. Clean up the database and its statistics:

   ```
   /usr/lib/edb-pge/17/bin/vacuumdb --all --analyze-in-stages
   ```

2. Remove all data files of the unencrypted server with the script generated by `pg_upgrade`:

   ```
   ./delete_old_cluster.sh
   ```

More information

Review Upgrading an installation with pg_upgrade for more information on pg_upgrade options, troubleshooting, and other considerations.

11.2 Minor EDB Postgres Extended Server upgrade

To perform a minor upgrade of your EDB Postgres Extended Server you only need to update your packages and restart the server.

To update your packages, ensure you use the correct package manager for your operating system.

- If you installed an RPM package of EDB Postgres Extended Server (on RHEL, AlmaLinux, Rocky Linux) with `dnf`, see Minor EDB Postgres Extended Server upgrade of RPM packages.
- If you installed a Debian package of EDB Postgres Extended Server (on Ubuntu, Debian) with `apt-get`, see Minor EDB Postgres Extended Server upgrade of Debian packages .

When you upgrade the packages, the packager manager installs the latest available minor version. For example, if you're running an xy.1 minor version, and the latest available minor version is xy.5, the package manager will install xy.5, skipping xy.2 to xy.4.

11.2.1 Minor EDB Postgres Extended Server upgrade of Debian packages

If you used `apt-get` to install a Debian package of EDB Postgres Extended Server (on Ubuntu, Debian), use `apt-get` to perform a minor version upgrade of the packages.

Overview

1. Upgrade the EDB Postgres Extended Server packages with `apt-get install`.
2. Restart the server with `pg_ctl`.
3. Verify the server version with psql.

Worked example

1. To upgrade the existing packages, open a command line, assume root privileges, and enter the command:

   ```
   sudo apt-get install <package_name>
   ```

   For example, if you want to upgrade to the latest minor version of EDB Postgres Extended Server 17, run:

   ```
   sudo apt-get install edb-postgresextended-17
   ```

   > **Note**
   >
   > You can perform a search of the packages to ensure you update the right package beforehand. For example, to browse through all EDB Packages, you can run `sudo apt-cache edb`. For more information about using `apt-get` commands and options, enter `apt-get --help` at the command line.

2. Confirm with `Y`.

   The output displays an overview of all performed processes, where you can see the packages that were upgraded.

3. To finalize the upgrade, restart the server. Replace `<path_to_data_directory>` with the path to the data directory of the server or servers you're upgrading:

   ```
   /usr/lib/edb-pge/17/bin/pg_ctl -D <path_to_data_directory> restart
   ```

   For example:

   ```
   /usr/lib/edb-pge/17/bin/pg_ctl -D /var/lib/edb-pge/17/upgrade restart
   ```

4. Verify the expected database version is running by connecting to psql:

```
/usr/lib/edb-pge/17/bin/psql
```

Check the server version:

```
SHOW server_version;
```

```
                                          output
                          server_version
-----------------------------------------------------------------------
 17.2 (EDB Postgres Extended Server 17.2.0) (Ubuntu 0:17.2-1.jammy)
(1 row)
```

11.2.2 Minor EDB Postgres Extended Server upgrade of RPM packages

If you used `dnf` to install an RPM package of EDB Postgres Extended Server (on RHEL, AlmaLinux, Rocky Linux), use `dnf` to perform a minor version upgrade of the packages.

Overview

1. Check for available updates with `dnf check-update`.
2. Upgrade the EDB Postgres Extended Server packages with `dnf update`.
3. Restart the server with `pg_ctl`.
4. Verify the server version with psql.

Worked example

1. To list the package updates available for your system, open a command line, assume root privileges, and enter the command:

   ```
   sudo dnf check-update <package_name>
   ```

   For example, if you want to upgrade to the latest minor version of EDB Postgres Extended Server 17, run:

   ```
   sudo dnf check-update edb-postgresextended17
   ```

   > **Note**
   >
   > You can include wildcard values in the search term. For example, if you're looking for EDB Packages, you can run `sudo dnf check-update edb-*`. For more information about using `dnf` commands and options, enter `dnf --help` at the command line.

2. Once you've figured the name and version of the package you want to install, use `dnf update` to install the package:

   ```
   sudo dnf update edb-postgresextended17
   ```

   output

   ```
   ================================================================================
   Package                       Arch   Version    Repository              Size
   ================================================================================
   Upgrading:
    edb-postgresextended17         x86_64 17.2-1.el9 enterprisedb-enterprise 1.7 M
    edb-postgresextended17-contrib x86_64 17.2-1.el9 enterprisedb-enterprise 724 k
    edb-postgresextended17-libs    x86_64 17.2-1.el9 enterprisedb-enterprise 330 k
    edb-postgresextended17-server  x86_64 17.2-1.el9 enterprisedb-enterprise 6.8 M

   Transaction Summary
   ================================================================================
   Upgrade  4 Packages

   Total download size: 9.5 M
   Is this ok [y/N]
   ```

3. Confirm with `y`. The output displays an overview of all performed processes, where you can see the packages that were upgraded:

```
y
```

```
                                                output
edb-postgresextended17-17.2-1.el9.x86_64
edb-postgresextended17-contrib-17.2-1.el9.x86_64
edb-postgresextended17-libs-17.2-1.el9.x86_64
edb-postgresextended17-server-17.2-1.el9.x86_64
```

4. To finalize the upgrade, restart the server. Replace `<path_to_data_directory>` with the path to the data directory of the server or servers you're upgrading:

```
/usr/edb/pge17/bin/pg_ctl -D <path_to_data_directory> restart
```

For example:

```
/usr/edb/pge17/bin/pg_ctl -D /var/lib/edb-pge/17/upgrade restart
```

5. Verify the expected database version is running by connecting to psql:

```
/usr/edb/pge17/bin/psql
```

Check the server version:

```
SHOW server_version;
```

```
                                                output
              server_version
-----------------------------------------
 17.2 (EDB Postgres Extended Server 17.2.0)
```

12 Configuration parameters (GUCs)

These Grand Unified Configuration (GUC) configuration parameters are available with EDB Postgres Extended Server.

Backend parameters

Backend parameters introduce a test probe point infrastructure for injecting sleeps or errors into PostgreSQL and extensions.

Any `PROBE_POINT` defined throughout the Postgres code code marks important code paths. These probe points might be activated to signal the current backend or to elog(...) a `LOG` / `ERROR` / `FATAL` / `PANIC`. They might also, or instead, add a delay at that point in the code.

Unless explicitly activated, probe points have no effect and add only a single optimizer-hinted branch, so they're safe on hot paths.

When an active probe point is hit and the counter is satisfied, after any specified sleep interval, a log message is always emitted at `DEBUG1` or higher.

`pg2q.probe_point`

The name of a `PROBE_POINT` in the code of 2ndQPostgres or in an extension that defines a `PROBE_POINT`. This parameter isn't validated. If a nonexistent probe point is named, it's never hit.

Only one probe point can be active. This isn't a list, and attempting to supply a list means nothing matches.

Probe points generally have a unique name, given as the argument to the `PROBE_POINT` macro in the code where it's defined. It's also possible to use the same `PROBE_POINT` name where multiple code paths trigger the same action of interest. A probe fires when either path is taken.

`pg2q.probe_counter`

You might need to act on a probe only after a loop is run for the number of times specified with this parameter. In such cases, set this GUC to the number of iterations at which point the probe point fires, and reset the counter.

The default value is `1`, meaning the probe points always fire when the name matches.

`pg2q.probe_sleep`

Sleep for `pg2q.probe_sleep` milliseconds after hitting the probe point. Then fire the action in `pg2q.probe_action`.

`pg2q.probe_action`

Action to take when the named `pg2q.probe_point` is hit. Available actions are:

- `sleep` — Emit a `DEBUG` message with the probe name.
- `log` — Emit a `LOG` message with the probe name.
- `error` — `elog(ERROR, ...)` to raise an `ERROR` condition.
- `fatal` — `elog(FATAL, ...)`.
- `panic` — `elog(PANIC, ...)`, which generally then calls `abort()` and delivers a `SIGABRT` (signal 6) to cause the backend to core dump. The probe point tries to set the core file limit to enable core dumps if the hard ulimit permits.
- `sigint`, `sigterm`, `sigquit`, `sigkill` — Deliver the named signal to the backend that hit the probe point.

`pg2q.probe_backend_pid`

If nonzero, the probe sleep and action are skipped for backends other than the backend with this ID.

`server_2q_version_num` and `server_2q_version`

The `server_2q_version_num` and `server_2q_version` configuration parameters allow the 2ndQuadrant-specific version number and version substring, respectively, to be accessible to external modules.

Table-level compression control option

You can set the table-level option `compress_tuple_target` to decide when to trigger compression on a tuple. Previously, you used the `toast_tuple_target` (or the compile time default) to decide whether to compress a tuple. However, this was detrimental when a tuple is large enough and has a good compression ratio but not large enough to cross the toast threshold.

`pg2q.max_tuple_field_size`

Restricts the maximum uncompressed size of the internal representation of any one field that can be written to a table, in bytes.

The default `pg2q.max_tuple_field_size` is 1073740799 bytes, which is 1024 bytes less than 1 GiB. This value is slightly less than the 1 GiB maximum field size usually imposed by PostgreSQL. This margin helps prevent cases where tuples are committed to disk but can't then be processed by logical decoding output plugins and sent to downstream servers.

Set `pg2q.max_tuple_field_size` to `1GB` or `11073741823` to disable the feature.

If your application doesn't rely on inserting large fields, consider setting `pg2q.max_tuple_field_size` to a much smaller value, such as 100MB or even less. Among other issues, large fields can:

- Cause surprising application behavior
- Increase memory consumption for the database engine during queries and replication
- Slow down logical replication

While this parameter is enabled, oversized fields cause queries that `INSERT` or `UPDATE` an oversized field to fail with an `ERROR` such as:

```
ERROR: field big_binary_field_name in row is larger than pg2q.max_tuple_field_size
DETAIL: New or updated row in relation some_table has field big_binary_field_name
    (attno=2) with size 8161 bytes which exceeds limit 1073740799B configured
    in pg2q.max_tuple_field_size
SQLSTATE: 53400 configuration_limit_exceeded
```

Only the superuser can set `pg2q.max_tuple_field_size`. You can use a `SECURITY DEFINER` function wrapper if you want to allow a non-superuser to set it.

If you change `pg2q.max_tuple_field_size`, fields larger than the current `pg2q.max_tuple_field_size` that are already on disk don't change. You can `SELECT` them as usual. Any `UPDATE` that affects tuples with oversized fields fails, even if the oversized field isn't modified, unless the new tuple created by the update operation satisfies the currently active size limits.

A `DELETE` operation doesn't check the field-size limit.

The limit isn't enforced on the text-representation size for I/O of fields because doing so also prevents PostgreSQL from creating and processing temporary in-memory json objects larger than the limit.

The limit isn't enforced for temporary tuples in tuplestores, such as set-returning functions, CTEs, and views. Size checks are deliberately not enforced for any `MATERIALIZED VIEW` either.

> **WARNING**
>
> `pg2q.max_tuple_field_size` is enforced for `pg_restore`. If a database contains oversized tuples, it does a `pg_dump` as usual. However, a subsequent `pg_restore` fails with the error shown previously. To work around this issue, restore the dump with `pg2q.max_tuple_field_size` overridden in connection options using `PGOPTIONS` or the `options` connection-parameter string. For example:
>
> ```
> PGOPTIONS='-c pg2q.max_tuple_field_size=11073741823' pg_restore ...
> ```

Data type specifics:

- For a `bytea` field, the size used is the decoded binary size. It isn't the text-representation size in hex or octal escape form, that is, the `octet_length()` of the field.

  Assuming `bytea_output = 'hex'`, the maximum size of the I/O representation is `2 * pg2q.max_tuple_field_size + 2` bytes.

- For a `text`, `json`, or `xml` field, the measured size is the number of bytes of text in the current database encoding (the `octet_length()` of the field), not the number of characters. In UTF-8 encodings, one character usually consumes one byte but might consume six or more bytes for some languages and scripts.

- For a `jsonb` field, the measured size is that of the PostgreSQL internal jsonb-encoded datatype representation, the text representation of the json document. In some cases the `jsonb` representation for larger json documents is smaller than the text representation. This means that it's possible to insert json documents with text representations larger than any given `pg2q.max_tuple_field_size`, although it's uncommon.

- Extension-defined data type behavior depends on the implementation of the data type.

  The field size used for this limit is the size reported by them `pg_column_size()` function, minus the 4 bytes of header PostgreSQL adds to variable-length data types, when used on a literal of the target data type. For example:

  ```
  demo=> SELECT pg_column_size(BYTEA '\x0001020304050607080 9') - 4;
  14
  ```

  For example, to see the computed size of the jsonb field, use:

  ```
  SELECT pg_column_size(JSONB '{"my_json_document": "yes"}') - 4;
  ```

  Due to TOAST compression, `pg_column_size()` often reports smaller values when called on existing on-disk fields. Also, the header for shorter values on disk might be 1 byte instead of 4.

`pg2q.max_tuple_size`

Restricts the maximum size of a single tuple that can be written to a table. This value is the total row width, including the uncompressed width of all potentially compressible or external-storage-capable field values. Field headers count against the size, but fixed row headers don't.

Many PostgreSQL operations, such as logical replication, work on whole rows, as do many applications. You can use this setting to impose a limit on the maximum row size you consider reasonable for your application to prevent inadvertent creation of oversized rows that might pose operational issues.

When applied to an `UPDATE` of existing tuples, `pg2q.max_tuple_size` isn't enforced as strictly as `pg2q.max_tuple_field_size`. It doesn't count the full size of unmodified values in columns with storage other than `PLAIN`.

> **WARNING**
>
> `pg2q.max_tuple_size` is enforced for `pg_restore`. See the caveat for `pg2q.max_tuple_field_size`.

13 SQL enhancements

EDB Postgres Extended Server includes a number of SQL enhancements.

Rollback options

In PostgreSQL, any error in a transaction rolls back all actions by that transaction. This behavior is different from other DBMS, such as Oracle and SQL Server, where an error causes rollback of only the last statement. This difference in transaction handling semantics doesn't cause a problem in all cases, but it does make implementing business logic in PostgreSQL difficult for Oracle Database and Microsoft SQL Server developers.

One workaround is to manually introduce a savepoint, internally known as subtransactions, into the application code. This is time consuming and difficult to test. A savepoint is an additional statement and therefore increases transaction latency. Given the overhead of additional development work and slower performance, this approach isn't viable in most cases.

EDB Postgres Extended Server allows you to roll back just the current statement. The statement-level rollback feature provides an optional mode to choose whether to allow rollback of the whole transaction or just the current statement. No manual recoding is required. There's some added overhead, but it's lower than for a savepoint.

See `transaction_rollback_scope` for information on setting the transaction rollback scope inside the database and JDBC properties for rollback scope for information on continuing past an error on a JDBC batch job.

Cursors with prepared statements

EDB Postgres Extended Server allows declaring a cursor over a previously created prepared statement.

For example:

```
PREPARE foo AS ...; DECLARE c1 CURSOR FOR foo;
```

PL/pgSQL compatibility

EDB Postgres Extended Server integrates with other migration tools with a number of PL/pgSQL compatibility features.

For general simplicity, EDB Postgres Extended Server allows calling functions using plpqsl without the PERFORM keyword.

For example,

```
BEGIN somefunc(); END
```

Where `somefunc` is not a keyword.

13.1 transaction_rollback_scope parameter

To set the transaction rollback scope inside the database, use the `transaction_rollback_scope` parameter. The `transaction_rollback_scope` parameter has two possible values:

- `transaction` — Standard Postgres behavior, where each error aborts the whole transaction.
- `statement` — An error while executing one statement affects only that statement and not the status of the transaction as a whole.

Setting the parameter

You can set the parameter as a user-level property, a connection option, or the mode for specific functions or procedures.

Set the parameter as a user-level property

```
ALTER USER somebody SET transaction_rollback_scope TO
statement;
```

Set the parameter as a connection option

```
PGOPTIONS="-c transaction_rollback_scope=statement" psql <other options>
```

Set the mode for specific functions or procedures

If using PL/pgSQL, you can set the mode for specific functions or procedures:

```
ALTER FUNCTION myfunc SET transaction_rollback_scope TO
statement;
```

How subtransactions are handled

If you select the `statement` value, then a subtransaction is opened just before each SQL command. If the command is successful, the subtransaction is *committed*. If the command causes an error, the subtransaction is rolled back, and the parent transaction can continue normally. The effect is that an error during execution of one statement affects only that statement and not the status of the transaction as a whole.

Committing a subtransaction assigns the resources it holds only to its parent transaction, which might be the top-level transaction. Or it might be some other subtransaction if there are user-defined savepoints involved. So this is not an "autonomous transaction." Rolling back a subtransaction releases all the resources it holds, such as any locks it acquired.

13.2 JDBC properties for setting rollback scope

If you're using a JDBC connector to connect to a client application, you use the `autosave` and `transaction_rollback_scope` properties together to specify the transaction rollback scope.

You can specify these properties in either the connection URL or as an additional properties object parameter to `DriverManager.getConnection`.

autosave

The `autosave` parameter is a string that specifies what the driver does if a query containing multiple statements fails. The possible values are: `server`, `always`, `never`, and `conservative`.

- In `autosave=server` mode, JDBC relies on the server-side parameter `transaction_rollback_scope` to save each statement by way of internal server savepoints before executing the next. The server rolls back to the previous statement if any statement in the query fails. If this parameter isn't supported on the server side, JDBC rejects the connection.
- In `autosave=always` mode, the JDBC driver first tries to use the server-side `transaction_rollback_scope` property. If it isn't supported, then JDBC driver sets a savepoint before each query statement and rolls back to that savepoint in case of failure.
- In `autosave=never` mode (default), no savepoint activity is ever carried out. In `autosave=conservative` mode, savepoint is set for each query. However, the rollback is done only for rare cases like 'cached statement cannot change return type' or 'statement XXX is not valid', so JDBC driver rolls back and retries.

The default value for this property is `never`.

This `autosave=server` property is useful only with the PostgreSQL server providing `transaction_rollback_scope` functionality.

transaction_rollback_scope

The `autosave` parameter is a string that determines the range of operations that roll back when an SQL statement fails.

The default value is `TRANSACTION`, which causes the entire transaction or current subtransaction to roll back. This is the only mode that you can select with the `SET TRANSACTION` command.

You can specify the other possible mode, `STATEMENT`, only during connection establishment, `ALTER USER`, or `ALTER DATABASE`. In that mode, only the failed `SQL` statement is rolled back, and the transaction is put back in normal mode.

`autosave` test cases

Test cases for trying out values of the `autosave` property are available in the `BatchAutoSaveTest.java` file. The following SQL code shows the behavior that's expected when the server provides `transaction_rollback_scope` functionality and `autosave=server` is used on the JDBC side.

With `autosave=server`, the following query inserts values `(1)`, `(3)`, and `(4)` and disregards the `duplicate key violation` error:

```
CREATE TABLE test (id INT PRIMARY
KEY);
INSERT INTO test VALUES
(2);
BEGIN;
INSERT INTO test VALUES
(1);
INSERT INTO test VALUES
(2);
INSERT INTO test VALUES
(3);
INSERT INTO test VALUES
(4);
COMMIT;
```

The `artifacts` directory contains the `pgjdbc` jar file `postgresql-REL2Q.42.2.3.180601.jar`. This file needs to be added to the CLASSPATH as usual. It also contains the `postgresql-REL2Q.42.2.3.180601-tests.jar` jar that can be used to test the latest `autosave` functionality.

You can test the `BatchAutoSaveTest.java` file provided in the `artifacts` as follows:

1. Export CLASSPATH to build and run the test case:

```
cd artifacts
export CLASSPATH=$PWD:$PWD/postgresql-REL2Q.42.2.3.180601-tests.jar:$PWD/postgresql-
REL2Q.42.2.3.180601.jar:$PWD/junit-4.12.jar:$PWD/hamcrest-core-1.3.jar
```

2. Compile the supplied test file:

```
javac -d .
BatchAutoSaveTest.java
```

3. Run the test (assuming user as `test` and running on localhost):

```
java -Dusername=test -Dport=5432 -Dhost=localhost -Ddatabase=postgres org.junit.runner.JUnitCore
org.postgresql.test.jdbc2.BatchAutoSaveTest
```

output

```
JUnit version 4.12
.Configuration file /Users/altaf/pg/artifacts/../build.properties does not exist. Consider adding it
to specify test db host and login
Configuration file /Users/altaf/pg/artifacts/../build.local.properties does not exist. Consider
adding it to specify test db host and login
Configuration file /Users/altaf/pg/artifacts/../build.properties does not exist. Consider adding it
to specify test db host and login
Configuration file /Users/altaf/pg/artifacts/../build.local.properties does not exist. Consider
adding it to specify test db host and login
.........
Time: 0.556

OK (10 tests)
```

To modify the test cases, you can modify the `BatchAutoSaveTest.java` file in the `artifacts` directory. Then compile and run the test cases.

14 Operations

EDB Postgres Extended Server has a number of features that relate to operations.

Avoid flooding transaction logs

EDB Postgres Extended Server provides WAL pacing delays to avoid flooding transaction logs. The WAL pacing configuration parameters are:

- `wal_insert_delay_enabled`
- `wal_insert_delay`
- `wal_insert_delay_size`

When `wal_insert_delay_enabled` is enabled, a session sleeps based on the value of `wal_insert_delay` after WAL data of at least the value of `wal_insert_delay_size` is generated. The default is off.

Additional tracing and diagnostics options

EDB Postgres Extended Server allows you to enable timeouts based on logging trace messages in specific code paths. Use the `tracelog_timeout` configuration parameter to allow logging of trace messages after a timeout of the specified time occurs.

Selective physical base backup and subsequent selective recovery/restore

By default, backups are always taken of the entire database cluster. You can also back up individual databases or database objects by specifying the `-L` option with the `pg_basebackup` utility multiple times for multiple databases.

Template databases are backed up by default. WAL data for excluded databases is still part of the WAL archives.

The backup activity stores the list of database objects specified using this option in the backup label file. The presence of these objects in the backup label file causes selective recovery of these databases. Recovery of template databases and of global metadata related to users, languages, and so on is also carried out as usual. WAL data belonging to excluded databases is ignored during the recovery process. Attempts to connect to excluded databases cause errors after regular operations start following the recovery.

Additional operations feature

- Reduced locking of `ALTER TABLE ... REPLICA IDENTITY`