EDB Postgres Extended Server

Version 15

Built at 2024-11-22T18:11:22

1 EDB Postgres Extended Server

EDB Postgres Extended Server is a Postgres database server distribution built on open-source, community PostgreSQL. It is 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 cannot 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
- Optional SQL superset to community PostgreSQL
- WAL pacing delays to avoid flooding transaction log
- Additional tracing and diagnostics options

2 Release notes

The EDB Postgres Extended Server documentation describes the latest version of EDB Postgres Extended Server 15, including minor releases and patches. These release notes cover what was new in each release.

Version	Release date
15.10	21 Nov 2024
15.8.1	22 Aug 2024
15.8	08 Aug 2024
15.7	09 May 2024
15.6	08 Feb 2024
15.5	09 Nov 2023
15.4	21 Aug 2023
15.3	11 May 2023
15.2	14 Feb 2023

2.1 EDB Postgres Extended Server 15.10 release notes

Released: 21 Nov 2024

New features, enhancements, bug fixes, and other changes in EDB Postgres Extended Server 15.10 include:

Type	Description	Ticket
Upstream merge	Merged with community PostgreSQL 15.10. This release includes a fix for CVE-2024-10978. See the PostgreSQL 15.10 Release Notes for more information.	CVE-2024-10978

2.2 EDB Postgres Extended Server 15.8.1 release notes

Released: 22 Aug 2024

New features, enhancements, bug fixes, and other changes in EDB Postgres Extended Server 15.8.1 include:

Type	Description	Ticket
Bug fix	A previous release introduced a new `in_create` field to the LogicalDecodingContext structure and changed its memory layout. This means that any code that expects the old layout will no longer be compatible. The release fixed the issue to ensure that the layout of previous database server versions is still compatible with the creation of a logical replication slot.	

2.3 EDB Postgres Extended Server 15.8 release notes

Released: 8 Aug 2024

New features, enhancements, bug fixes, and other changes in EDB Postgres Extended Server 15.8 include:

Type	Description	Ticket
Upstream merge	Merged with community PostgreSQL 15.8. Includes a fix for CVE-2024-7348 (Prevent unauthorized code execution during pg_dump) - See the PostgreSQL 15 Release Notes for more information.	
Bug fix	Added cleanup for incomplete transactions in reorder buffer after a crash.	RT9963 6

2.4 EDB Postgres Extended Server 15.7 release notes

Released: 9 May 2024

Updated: 17 May 2024

New features, enhancements, bug fixes, and other changes in EDB Postgres Extended Server 15.7 include:

Type	Description	Ticket
Upstream merge	Merged with community PostgreSQL 15.7. Includes a fix for CVE-2024-4317. See the PostgreSQL 15 Release Notes for more information.	
Bug fix	Fixed issue in WAL-logging of XID assignments that could crash standby	#99297/35451

2.5 EDB Postgres Extended Server 15.6 release notes

Released: 8 Feb 2024

New features, enhancements, bug fixes, and other changes in EDB Postgres Extended Server 15.6 include:

Type	Description
Upstream merge	Merged with community PostgreSQL 15.6. See the PostgreSQL 15 Release Notes for more information.

2.6 EDB Postgres Extended Server 15.5 release notes

Released: 9 Nov 2023

New features, enhancements, bug fixes, and other changes in EDB Postgres Extended Server 15.5 include:

Type	Description
Upstream merge	Merged with community PostgreSQL 15.5. See the PostgreSQL 15 Release Notes for more information.

2.7 EDB Postgres Extended Server 15.4 release notes

Released: 21 Aug 2023

New features, enhancements, bug fixes, and other changes in EDB Postgres Extended Server 15.4 include:

Type	Description
Upstream merge	Merged with community PostgreSQL 15.4. See the PostgreSQL 15 Release Notes for more information.
Bug fix	Fixed a memory leak experienced when using EDB Postgres Distributed (PGD) with Transparent Data Encryption (TDE). [Support issue: #93936]

2.8 EDB Postgres Extended Server 15.3 release notes

Released: 11 May 2023

New features, enhancements, bug fixes, and other changes in EDB Postgres Extended Server 15.3 include:

Type	Description	Category
Upstream merge	Merged with community PostgreSQL 15.3. See the PostgreSQL 15 Release Notes for more information.	
Enhancement	SQL Profiler and Index Advisor are now extensions and can be downloaded from EDB Repos.	

2.9 EDB Postgres Extended Server 15.2 release notes

Released: 14 Feb 2023

New features, enhancements, bug fixes, and other changes in EDB Postgres Extended Server 15.2 include:

Type	Description
Upstream merge	Merged with community PostgreSQL 15.2. See the PostgreSQL 15 Release Notes for more information.
Feature	Support for Transparent Data Encryption (TDE). TDE encrypts, transparently to the user, any user data stored in the database system.

3 Deployment options

The deployment options include:

- Installing on a virtual machine or physical server using native packages
- Deploying it with EDB Postgres Distributed using Trusted Postgres Architect
- 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

Debian and derivatives

- Ubuntu 22.04, Ubuntu 20.04
- Debian 11

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

Debian and derivatives

- Ubuntu 22.04
- Ubuntu 20.04
- Debian 11

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-postgresextended15-server edb-postgresextended15-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-15-setup` script creates a cluster.

```
sudo PGSETUP_INITDB_OPTIONS="-E UTF-8" /usr/edb/pge15/bin/edb-pge-15-setup initdb

sudo systemctl start edb-pge-15
```

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-postgresextended15-server edb-postgresextended15-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-15-setup` script creates a cluster.

```
sudo PGSETUP_INITDB_OPTIONS="-E UTF-8" /usr/edb/pge15/bin/edb-pge-15-setup initdb

sudo systemctl start edb-pge-15
```

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-postgresextended15-server edb-postgresextended15-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-15-setup` script creates a cluster.

```
sudo PGSETUP_INITDB_OPTIONS="-E UTF-8" /usr/edb/pge15/bin/edb-pge-15-setup initdb

sudo systemctl start edb-pge-15
```

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-postgresextended15-server edb-postgresextended15-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-15-setup` script creates a cluster.

```
sudo PGSETUP_INITDB_OPTIONS="-E UTF-8" /usr/edb/pge15/bin/edb-pge-15-setup initdb

sudo systemctl start edb-pge-15
```

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 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-15
```

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-15-setup` script creates a cluster.

```
sudo PGSETUP_INITDB_OPTIONS="-E UTF-8" /usr/lib/edb-pge/15/bin/edb-pge-15-setup initdb

sudo systemctl start edb-pge-15
```

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 20.04 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-15
```

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-15-setup` script creates a cluster.

```
sudo PGSETUP_INITDB_OPTIONS="-E UTF-8" /usr/lib/edb-pge/15/bin/edb-pge-15-setup initdb

sudo systemctl start edb-pge-15
```

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 Debian 11 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-15
```

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-15-setup` script creates a cluster.

```
sudo PGSETUP_INITDB_OPTIONS="-E UTF-8" /usr/lib/edb-pge/15/bin/edb-pge-15-setup initdb

sudo systemctl start edb-pge-15
```

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 Ubuntu 18.04 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-15
```

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-15-setup` script creates a cluster.

```
sudo PGSETUP_INITDB_OPTIONS="-E UTF-8" /usr/lib/edb-pge/15/bin/edb-pge-15-setup initdb

sudo systemctl start edb-pge-15
```

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 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/pge15/bin`
Libraries	`/usr/edb/pge15/lib`
Cluster configuration files	`/var/lib/edb-pge/15`
Documentation	`/usr/edb/pge15/share/man`
Contrib	`/usr/edb/pge15/share/contrib`
Data	`/var/lib/edb-pge/15/data`
Logs	`/var/log/edb/pge15`
Lock files	`/var/lock/edb/pge15`
Backup area	`/var/lib/edb-pge/15/backups`
Templates	`/usr/edb/pge15/share`
Procedural Languages	`/usr/edb/pge15/lib`
Development Headers	`/usr/edb/pge15/include`
Shared data	`/usr/edb/pge15/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/15/bin`
Libraries	`/usr/lib/edb-pge/15/lib`
Cluster configuration files	`/var/lib/edb-pge/15/main`
Data	`/var/lib/edb-pge/15/main`
Logs	`/var/log/edb-pge/`
Lock files	`/var/lock/edb/pge15`

5 Administration

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

- Setting Configuration Parameters covers how to configure GUC parameters at runtime, modifying `postgresql.conf` for persistent changes and editing `pg_hba.conf` to change access and authentication settings.

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 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.

8 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 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, 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` .

9 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.

9.1 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.

9.2 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.

10 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`