EDB Postgres Advanced Server

Version 10

Built at 2021-01-27T01:22:29

1 Requirements Overview

The following sections detail the supported platforms and installation requirements for EDB Postgres Advanced Server 10.

Supported Platforms

The Advanced Server 10 RPM packages are supported on the following platforms:

64 bit Linux:

- Red Hat Enterprise Linux (x86_64) 6.x and 7.x
- CentOS (x86_64) 6.x and 7.x
- PPC-LE 8 running RHEL or CentOS 7.x

The Advanced Server 10 graphical (or interactive) installers are supported on the following platforms:

64 bit Linux:

- Red Hat Enterprise Linux 6.x and 7.x
- CentOS 6.x and 7.x
- Oracle Enterprise Linux 6.x and 7.x
- Ubuntu 14.04 LTS and 16.04 LTS
- Debian 7 and 8
- SELinux Enterprise 12.x

64 bit Windows:

- Windows Server 2016
- Windows Server 2012 R2 Server

Note: The data directory of a production database should not be stored on an NFS file system.

RPM Installation Pre-Requisites

You can use an RPM package to install Advanced Server and its supporting components on a Linux host. Before installing the Advanced Server, you must:

Install the EPEL Release Package

You can use yum to install the epel-release package:

```
yum -y install https://dl.fedoraproject.org/pub/epel/epel-release-latest-7.noarch.rpm
```

Please note that you may need to enable the [extras] repository definition in the CentOS-Base.repo file (located in /etc/yum.repos.d).

If yum cannot access a repository that contains epel-release, you will get an error message:

No package epel available.

Error: Nothing to do

If you receive this error, you can download the EPEL rpm package, and install it manually. To manually install

EPEL, download the rpm package, assume superuser privileges, navigate into the directory that contains the package, and install EPEL with the command:

yum install epel-release

Request Credentials for the EnterpriseDB Repository

Before performing an RPM installation, you must have credentials that allow access to the EnterpriseDB repository. For information about requesting credentials, visit:

https://info.enterprisedb.com/rs/069-ALB-339/images/Repository%20Access%2004-09-2019.pdf

1.1 Using a Package Manager to Install Advanced Server

You can use the yum package manager to install Advanced Server or Advanced Server supporting components. yum will attempt to satisfy package dependencies as it installs a package, but requires access to the Advanced Server repositories. If your system does not have access to a repository via the Internet, you can use RPM to install a package or create a local repository, but you may be required to manually satisfy package dependencies.

The Advanced Server RPM installs Advanced Server and the core components of the database server. For a complete list of the RPM installers available for Advanced Server and its supporting components, see Section 3.1.

Installing the server package creates a database superuser named enterprisedb. The user is assigned a user ID (UID) and a group ID (GID) of 26. The user has no default password; use the passwd command to assign a password for the user. The default shell for the user is bash, and the user's home directory is /var/lib/edb/as10.

By default, Advanced Server logging is configured to write files to the log subdirectory of the data directory, rotating the files each day and retaining one week of log entries. You can customize the logging behavior of the server by modifying the postgresql.conf file; for more information about modifying the postgresql.conf file, please see Section 6.1.

The RPM installers place Advanced Server components in the directories listed in the table below:

Component	Location
Executables	/usr/edb/as10/bin
Libraries	/usr/edb/as10/lib
Cluster configuration files	/etc/edb/as10
Documentation	/usr/edb/as10/share/doc
Contrib	/usr/edb/as10/share/contrib
Data	/var/lib/edb/as10/data
Logs	/var/log/as10
Lock files	/var/lock/as10
Log rotation file	/etc/logrotate.d/as10
Sudo configuration file	/etc/sudoers.d/as10
Binary to access VIP without sudo	/usr/edb/as10/bin/secure
Backup area	/var/lib/edb/as10/backups
Templates	/usr/edb/as10/share
Procedural Languages	/usr/edb/as10/lib or /usr/edb/as10/lib64
Development Headers	/usr/edb/as10/include
Shared data	/usr/edb/as10/share

Component	Location
Regression tests	/usr/edb/as10/lib/pgxs/src/test/regress
SGML Documentation	/usr/edb/as10/share/doc

Installing an RPM Package

Before installing the Advanced Server or a supporting component via an RPM package, you must request access to the EnterpriseDB repository. For more information, visit:

https://info.enterprisedb.com/rs/069-ALB-339/images/Repository%20Access%2004-09-2019.pdf

After receiving your repository credentials you can:

1. Create the repository configuration file.
2. Modify the file, providing your user name and password.
3. Install Advanced Server and its supporting components.

Creating a Repository Configuration File and Installing Advanced Server

To create the repository configuration file, assume superuser privileges and invoke the following command:

```
yum -y install https://yum.enterprisedb.com/edb-repo-rpms/edb-repo-latest.noarch.rpm
```

The repository configuration file is named edb.repo. The file resides in /etc/yum.repos.d.

After creating the edb.repo file, use your choice of editor to ensure that the value of the enabled parameter is 1, and replace the username and password placeholders in the baseurl specification with the name and password of a registered EnterpriseDB user.

[edb]

name=EnterpriseDB RPMs $releasever - $basearch

baseurl=https://\<*username*>:\<*password*>@yum.enterprisedb.com/edb/redhat/rhel-$releasever-$basearch

enabled=1

gpgcheck=1

gpgkey=file:///etc/pki/rpm-gpg/ENTERPRISEDB-GPG-KEY

After saving your changes to the configuration file, you can use yum install command to install Advanced Server. For example, to install the server and its core components, use the command:

```
yum install edb-as10-server
```

When you install an RPM package that is signed by a source that is not recognized by your system, yum may ask for your permission to import the key to your local server. If prompted, and you are satisfied that the packages come from a trustworthy source, enter a y, and press Return to continue.

After installing Advanced Server, you must configure the installation; see Section 3.3, *Configuring a Package Installation*, for details.

During the installation, yum may encounter a dependency that it cannot resolve. If it does, it will provide a list of the required dependencies that you must manually resolve.

Advanced Server RPM Installers

The tables that follow list the packages that are available from EnterpriseDB. Please note that you can also use the yum search command to access a list of the packages that are currently available from your configured repository. To use the yum search command, open a command line, assume root privileges, and enter:

yum search *package*

Where *package* is the search term that specifies the name (or partial name) of a package. The repository search will return a list of available packages that include the specified search term.

The following table lists the packages that are stored in the Advanced Server repository and the corresponding software installed by those packages:

Package Name	Package Installs
edb-as10-server	This package installs core components of the Advanced Server database server.
edb-as10-server-client	The edb-as10-server-client package contains client programs and utilities that you can use to access and manage Advanced Server.
edb-as10-server-contrib	The edb-as10-contrib package installs contributed tools and utilities that are distributed with Advanced Server. Files for these modules are installed in: Documentation: /usr/edb/as10/share/doc Loadable modules: /usr/edb/as10/lib Binaries: /usr/edb/as10/bin
edb-as10-server-core	The edb-as10-server-core package includes the programs needed to create the core functionality behind the Advanced Server database.
edb-as10-server-devel	The edb-as10-server-devel package contains the header files and libraries needed to compile C or C++ applications that directly interact with an Advanced Server server and the ecpg or ecpgPlus C preprocessor.
edb-as10-server-docs	The edb-as10-server-docs package installs the readme file.
edb-as10-server-indexadvisor	This package installs Advanced Server's Index Advisor feature. The Index Advisor utility helps determine which columns you should index to improve performance in a given workload.
edb-as10-server-libs	The edb-as10-server-libs package provides the essential shared libraries for any Advanced Server client program or interface.
edb-as10-server-pldebugger	This package implements an API for debugging PL/pgSQL functions on Advanced Server.
edb-as10-server-plperl	The edb-as10-server-plperl package installs the PL/Perl procedural language for Advanced Server. Please note that the edb-as10-server-plperl package is dependent on the platform-supplied version of Perl.
edb-as10-server-plpython	The edb-as10-server-plpython package installs the PL/Python procedural language for Advanced Server. Please note that the edb-as10-server-plpython package is dependent on the platform-supplied version of Python.
edb-as10-server-pltcl	The edb-as10-pltcl package installs the PL/Tcl procedural language for Advanced Server. Please note that the edb-as10-pltcl package is dependent on the platform-supplied version of TCL.
edb-as10-server-sqlprofiler	This package installs Advanced Server's SQL Profiler feature. SQL Profiler helps identify and optimize SQL code.
edb-as10-server-sqlprotect	This package installs Advanced Server's SQL Protect feature. SQL Protect provides protection against SQL injection attacks.
edb-as10-server-sslutils	This package installs functionality that provides SSL support.

Package Name	Package Installs
edb-as10-server-cloneschema	This package installs the EDB Clone Schema extension. For more information about EDB Clone Schema, see the EDB Postgres Advanced Server Guide.
edb-as10-server-parallel-clone	This package installs functionality that supports the EDB Clone Schema extension.
edb-as10-edbplus	The edb-edbplus package contains the files required to install the EDB Plus command line client. EDB Plus commands are compatible with Oracle's SQL*Plus.
edb-as10-pgagent	This package installs pgAgent; pgAgent is a job scheduler for Advanced Server. Before installing this package, you must install EPEL; for detailed information about installing EPEL, see Section 2.2.
edb-icache	This package installs InfiniteCache.
edb-icache-devel	This is a supporting package for InfiniteCache.
edb-as10-pgsnmpd	SNMP (Simple Network Management Protocol) is a protocol that allows you to supervise an apparatus connected to the network.
edb-as10-pljava	This package installs PL/Java, providing access to Java stored procedures, triggers and functions via the JDBC interface.
edb-as10-pgpool35-extensions	This package creates pgPool extensions required by the server.
edb-as10-slony-replication	This package contains the meta installer for Slony replication and documentation. Slony facilitates master-standby replication, and is suited for large databases with a limited number of standbys.
edb-as10-slony-replication-core	This package contains the files required to install Slony replication. Slony facilitates master-standby replication, and is suited for large databases with a limited number of standby systems.
edb-as10-slony-replication-docs	This package contains the Slony project documentation (in pdf form).
edb-as10-slony-replication-tools	This package contains the Slony altperl tools and utilities that are useful when deploying Slony replication environments. Before installing this package, you must install EPEL; for detailed information about installing EPEL, see Section 2.2.

The following table lists the packages for Advanced Server 10 supporting components:

Package Name	Package Installs
edb-pgpool35	This package contains the pgPool-II installer. pgPool provides connection pooling for Advanced Server installations.
edb-pgpool35-devel	This package contains the pgPool-II headers and libraries.
edb-jdbc	The edb-jdbc package includes the .jar files needed for Java programs to access an Advanced Server database.
edb-migrationtoolkit	The edb-migrationtoolkit package installs Migration Toolkit, facilitating migration to an Advanced Server database from Oracle, PostgreSQL, MySQL, Sybase and SQL Server.
edb-oci	The edb-oci package installs the EnterpriseDB Open Client library, allowing applications that use the Oracle Call Interface API to connect to an Advanced Server database.
edb-oci-devel	This package installs the OCI include files; install this package if you are developing C/C++ applications that require these files.

Package Name	Package Installs
edb-odbc	This package installs the driver needed for applications to access an Advanced Server system via ODBC.
edb-odbc-devel	This package installs the ODBC include files; install this package if you are developing C/C++ applications that require these files.
edb-pgbouncer17	This package contains PgBouncer (a lightweight connection pooler). This package requires the libevent package.
edb-xdb	This package contains the xDB installer; xDB provides asynchronous cross-database replication. For more information, visit http://www.enterprisedb.com/faq-xdb-multi-master
edb-xdb-console	This package provides support for xDB.
edb-xdb-libs	This package provides support for xDB.
edb-xdb-publisher	This package provides support for xDB.
edb-xdb-subscriber	This package provides support for xDB.

Please Note: Available packages are subject to change.

Updating an RPM Installation

If you have an existing Advanced Server RPM installation, you can use yum to upgrade your repository configuration file and update to a more recent product version. To update the edb.repo file, assume superuser privileges and enter:

```
yum upgrade edb-repo
```

yum will update the edb.repo file to enable access to the current EDB repository, configured to connect with the credentials specified in your edb.repo file. Then, you can use yum to upgrade all packages whose names include the expression edb:

```
yum upgrade *edb**
```

Please note that the yum upgrade command will only perform an update between minor releases; to update between major releases, you must use pg_upgrade.

For more information about using yum commands and options, enter yum --help on your command line, or visit:

https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Deployment_Guide/ch-yum.html

Configuring a Package Installation

The packages that install the database server component create a service configuration file (on version 6.x hosts) or unit file (on version 7.x hosts), and service startup scripts.

The PostgreSQL initdb command creates a database cluster. If you are using an RPM package to install Advanced Server, you must manually configure the service and invoke initdb to create your cluster.

When invoking initdb, you can:

- Specify environment options on the command line.
- Include the service command on RHEL or CentOS 6.x, and use service configuration file to configure the environment.
- Include the systemd service manager on RHEL or CentOS 7.x use the service configuration file to configure the

environment.

If you are using the interactive graphical installer to install Advanced Server, the installer will invoke initdb to create a cluster for you; for details about specifying cluster preferences when using the interactive installer, see ***Setting Cluster Preferences with the Graphical Installer*** in Chapter 4.

Creating a Database Cluster and Starting the Service

After specifying any options in the service configuration file, you must create the database cluster and start the service; these steps are platform specific.

On RHEL or CentOS 6.x

To create a database cluster in the PGDATA directory that listens on the port specified by the PGPORT specified in the service configuration file described in Section 3.3.2, assume root privileges, and invoke the service script:

```
service edb-as-10 initdb
```

You can also assign a locale to the cluster when invoking initdb. By default, initdb will use the value specified by the $LANG operating system variable, but if you append a preferred locale when invoking the script, the cluster will use the alternate value. For example, to create a database cluster that uses simplified Chinese, invoke the command:

```
service edb-as-10 initdb zh_CH.UTF-8
```

After creating a database cluster, start the database server with the command:

```
service edb-as-10 start
```

On RHEL or CentOS 7.x

To invoke initdb on a RHEL or CentOS 7.x system, with the options specified in the service configuration file, assume the identity of the operating system superuser:

```
su - root
```

Then, invoke initdb:

```
/usr/edb/as10/bin/edb-as-10-setup initdb
```

After creating the cluster, use systemctl to start, stop, or restart the service:

> systemctl { start | stop | restart } edb-as-10

For more information about using the service command, please see Section 5.2.

Using a Service Configuration File on CentOS or Redhat 6.x

On a CentOS or RedHat version 6.x host, the RPM installer creates a service configuration file named edb-as-10.sysconfig in /etc/sysconfig/edb/as10. Please note that options specified in the service configuration file are only enforced if initdb is invoked via the service command; if you manually invoke initdb (at the command line), you must specify the other options (such as the location of the data directory and installation mode) on the command line.

Figure 3.5 -The Advanced Server service configuration file.

The file contains the following environment variables:

- PGENGINE specifies the location of the engine and utility executable files.
- PGPORT specifies the listener port for the database server.
- PGDATA specifies the path to the data directory.
- PGLOG specifies the location of the log file to which the server writes startup information.
- Use INITDBOPTS to specify any initdb option or options that you wish to apply to the new cluster.

You can modify the edb-as-10.sysconfig file before using the service command to invoke the startup script to change the listener port, data directory location, startup log location or installation mode. If you plan to create more than one instance on the same system, you may wish to copy the edb-as-10.sysconfig file (and the associated edb-as-10 startup script) and modify the file contents for each additional instance that resides on the same host.

Specifying INITDBOPTS Options

You can use the INITDBOPTS variable to specify your cluster configuration preferences. By default, the INITDBOPTS variable is commented out in the service configuration file; unless modified, when you run the service startup script, the new cluster will be created in a mode compatible with Oracle databases. Clusters created in this mode will contain a database named edb, and have a database superuser named enterprisedb.

To create a new cluster in PostgreSQL mode, remove the pound sign (#) in front of the INITDBOPTS variable, enabling the "--no-redwood-compat" option. Clusters created in PostgreSQL mode will contain a database named postgres, and have a database superuser named postgres.

You may also specify multiple initdb options. For example, the following statement:

INITDBOPTS="--no-redwood-compat -U alice --locale=en_US.UTF-8"

Creates a database cluster (without compatibility features for Oracle) that contains a database named postgres that is owned by a user named alice; the cluster uses UTF-8 encoding.

In addition to the cluster configuration options documented in the PostgreSQL core documentation, Advanced Server supports the following initdb options:

--no-redwood-compat

Include the --no-redwood-compat keywords to instruct the server to create the cluster in PostgreSQL mode. When the cluster is created in PostgreSQL mode, the name of the database superuser will be postgres, the name of the default database will be postgres, and Advanced Server’s features compatible with Oracle databases will not be available to the cluster.

--redwood-like

Include the --redwood-like keywords to instruct the server to use an escape character (an empty string ('')) following the LIKE (or PostgreSQL-compatible ILIKE) operator in a SQL statement that is compatible with Oracle syntax.

--icu-short-form

Include the --icu-short-form keywords to create a cluster that uses a default ICU (International Components for Unicode) collation for all databases in the cluster. For more information about Unicode collations, please refer to the *EDB Postgres Advanced Server Guide* available at:

http://www.enterprisedb.com/products-services-training/products/documentation

For more information about using initdb, and the available cluster configuration options, see the PostgreSQL Core Documentation available at:

https://www.postgresql.org/docs/10/static/app-initdb.html

You can also view online help for initdb by assuming superuser privileges and entering:

/*path_to_initdb_installation_directory*/initdb --help

Where *path_to_initdb_installation_directory* specifies the location of the initdb binary file.

Modifying the Data Directory Location on CentOS or Redhat 7.x

On a CentOS or RedHat version 7.x host, the unit file is named edb-as-10.service and resides in /usr/lib/systemd/system. The unit file contains references to the location of the Advanced Server data directory. You should avoid making any modifications directly to the unit file because it may be overwritten during package upgrades.

By default, data files reside under /var/lib/edb/as10/data directory. To use a data directory that resides in a non-default location, create a copy of the unit file under the /etc directory:

```
cp /usr/lib/systemd/system/edb-as-10.service /etc/systemd/system/
```

After copying the unit file to the new location, modify the service file (/etc/systemd/system/edb-as-10.service) with your editor of choice, correcting any required paths.

Then, use the following command to reload systemd, updating the modified service scripts:

```
systemctl daemon-reload
```

Then, start the Advanced Server service with the following command:

```
systemctl start edb-as-10
```

Starting Multiple Postmasters with Different Clusters

You can configure Advanced Server to use multiple postmasters, each with its own database cluster. The steps required are version specific to the Linux host.

On RHEL or CentOS 6.x

The edb-as10-server-core RPM contains a script that starts the Advanced Server instance. The script can be copied, allowing you to run multiple services, with unique data directories and that monitor different ports. You must have root access to invoke or modify the script.

The example that follows creates a second instance on an Advanced Server host; the secondary instance is named secondary:

1. Create a hard link in /etc/rc.d/init.d (or equivalent location) to edb-as-10 named secondary-edb-as-10:

```
ln edb-as-10 secondary-edb-as-10
```

Be sure to pick a name that is not already used in /etc/rc.d/init.d.

2. Create a file in /etc/sysconfig/edb/as10/ named secondary-edb-as-10. This file is where you would typically define PGDATA and PGOPTS. Since $PGDATA/postgresql.conf will override many of these settings (except PGDATA) you might notice unexpected results on startup.

3. Create the target PGDATA directory.

4. Assume the identity of the Advanced Server database superuser (enterprisedb) and invoke initdb on the target PGDATA. For information about using initdb, please see the PostgreSQL Core Documentation available at:

https://www.postgresql.org/docs/10/static/app-initdb.html

5. Edit the postgresql.conf file to specify the port, address, TCP/IP settings, etc. for the secondary instance.

6. Start the postmaster with the following command:

```
service secondary-edb-as-10 start
```

On RHEL or CentOS 7.x

The edb-as10-server-core RPM for version 7.x contains a unit file that starts the Advanced Server instance. The file allows you to start multiple services, with unique data directories and that monitor different ports. You must have root access to invoke or modify the script.

The example that follows creates an Advanced Server installation with two instances; the secondary instance is named secondary:

1. Make a copy of the default file with the new name. As noted at the top of the file, all modifications must reside under /etc. You must pick a name that is not already used in /etc/systemd/system.

```
cp /usr/lib/systemd/system/edb-as-10.service /etc/systemd/system/secondary-edb-as-10.service
```

2. Edit the file, changing PGDATA to point to the new data directory that you will create the cluster against.

3. Create the target PGDATA with user enterprisedb.

4. Run initdb, specifying the setup script:

/usr/edb/as10/bin/edb-as-10-setup initdb secondary-edb-as-10

5. Edit the postgresql.conf file for the new instance, specifying the port, the IP address, TCP/IP settings, etc.

6. Make sure that new cluster runs after a reboot:

```
systemctl enable secondary-edb-as-10
```

7. Start the second cluster with the following command:

```
systemctl start secondary-edb-as-10
```

Installing Advanced Server on an Isolated Network

You can create a local yum repository to act as a host for the Advanced Server RPM packages if the server on which you wish to install Advanced Server (or supporting components) cannot directly access the EnterpriseDB repository. Please note that this is a high-level listing of the steps requires; you will need to modify the process for your individual network.

To create and use a local repository, you must:

1. Use yum to install the yum-utils and createrepo packages:

```
yum install yum-utils yum install createrepo
```

2. Create a directory in which to store the repository:

```
mkdir /srv/repos
```

3. Copy the RPM installation packages to your local network repository. You can download the individual RPM files from:

```
yum.enterprisedb.com
```

4. Sync the RPM packages and create the repository.

```
reposync -r edbas10 -p /srv/repos createrepo /srv/repos
```

5. Install your preferred webserver on the host that will act as your local repository, and ensure that the repository directory is accessible to the other servers on your network. For example, you might install lighttpd:

```
yum install lighttpd
```

6. If you are using lighttpd, you must provide a configuration file that identifies the location of the repository on your local network. For example, the configuration file might contain:

```
$HTTP["host"] == "yum.domain.com"{ server.document-root = "/srv/repos"
server.errorlog="/var/log/lighttpd/yum_error.log" accesslog.filename = "/var/log/lighttpd/yum_access.log"}
```

For detailed information about installing, configuring and using lighttpd, visit the official project site at:

http://redmine.lighttpd.net/projects/1/wiki/Docs

7. On each isolated database server, configure yum to pull updates from the mirrored repository on your local network. For example, you might create a file called /etc/yum.repos.d/edb-repo with connection information that specifies:

```
[edbas10] name=EnterpriseDB Advanced Server 10 baseurl=http://yum.domain.com/edbas10 enabled=1
gpgcheck=0
```

After specifying the location and connection information for your local repository, you can use yum commands to install Advanced Server and its supporting components on the isolated servers. For example:

```
yum install edb-as10
```

For more information about creating a local repository, visit:

http://yum.baseurl.org/

1.2 Installing Advanced Server with the Interactive Installer

The Advanced Server installer is available from the EnterpriseDB website at:

http://www.enterprisedb.com/downloads/postgres-postgresql-downloads

After navigating to the Software Downloads page, use the drop-down listboxes to select the Advanced Server version you wish to install and your platform, and then click the Download Now button. When the download completes, extract files using your system-specific file extractor.

You can use the extracted installer in different installation modes to perform an Advanced Server installation:

- For information about using the extracted files to perform a graphical installation on Windows, See Section 4.3.1.
- For information about performing a graphical installation on Linux, see Section 4.3.2.
- For information about using the installer to perform a command line installation, see Section 4.4.
- For information about performing an unattended installation, see Section 4.4.2.
- For information about performing an installation with limited privileges, see Section 4.4.3.
- For information about the command line options you can use when invoking the installer, see Section 4.4.4.

During the installation process, the Advanced Server installer program copies a number of temporary files to the location specified by the TEMP or TMP environment variable (on Windows), or to the /tmp directory (on Linux). You can optionally specify an alternate location for the installer to place the temporary files by modifying or creating the TEMP environment variable.

If invoking the installer from the command line, you can set the value of the variable on the command line:

On Windows, use the command:

SET TMP=temp_file_location

On Linux, use the command:

```
export TEMP=temp_file_location
```

Where temp_file_location specifies the alternate location for the temporary files.

Please Note: If you are invoking the installer to perform a system upgrade, the installer will preserve the configuration options specified during the previous installation.

Setting Cluster Preferences with the Graphical Installer

During an installation, the graphical installer invokes the PostgreSQL initdb utility to initialize a cluster. If you are using the graphical installer, you can use the INITDBOPTS environment variable to specify your initdb preferences. Before invoking the graphical installer, set the value of INITDBOPTS at the command line, specifying one or more cluster options. For example, on Linux use an export statement to set the value:

```
export INITDBOPTS="-k -E=UTF-8"
```

or on Windows, use a SET statement:

```
SET INITDBOPTS= -k -E=UTF-8
```

On Linux, enclose the options in double-quotes ("); on Windows, double-quotes are not required. If you specify values in INITDBOPTS that are also provided by the installer (such as the –D option, which specifies the installation directory), the value specified in the graphical installer will supersede the value if specified in INITDBOPTS.

For more information about using initdb cluster configuration options, see the PostgreSQL Core Documentation available at:

https://www.postgresql.org/docs/10/static/app-initdb.html

In addition to the cluster configuration options documented in the PostgreSQL core documentation, Advanced Server supports the following initdb options:

--no-redwood-compat

Include the --no-redwood-compat keywords to instruct the server to create the cluster in PostgreSQL mode. When the cluster is created in PostgreSQL mode, the name of the database superuser will be postgres, the name of the default database will be postgres, and Advanced Server's features compatible with Oracle databases will not be available to the cluster.

--redwood-like

Include the --redwood-like keywords to instruct the server to use an escape character (an empty string ('')) following the LIKE (or PostgreSQL-compatible ILIKE) operator in a SQL statement that is compatible with Oracle syntax.

--icu-short-form

Include the --icu-short-form keywords to create a cluster that uses a default ICU (International Components for Unicode) collation for all databases in the cluster. For more information about Unicode collations, please refer to the *EDB Postgres Advanced Server Guide* available at:

http://www.enterprisedb.com/products-services-training/products/documentation

Graphical Installation Prerequisites

User Privileges

Before invoking the installer on a Linux system, you must have superuser privileges to perform an Advanced Server installation. To perform an Advanced Server installation on a Windows system, you must have administrator privileges. If you are installing Advanced Server into a Windows system that is configured with User Account Control enabled, you can assume sufficient privileges to invoke the graphical installer by right clicking on the name of the installer and selecting Run as administrator from the context menu.

Linux-specific Software Requirements

You must install xterm, konsole, or gnome-terminal before executing any console-based program installed by the Advanced Server installer. Without a console program, you will not be able to access Advanced Server configuration files through menu selections.

Before invoking the StackBuilder Plus utility on a Linux system, you must install the redhat-lsb package. To install the package, open a terminal window, assume superuser privileges, and enter:

```
# yum install redhat-lsb
```

For more information about using StackBuilder Plus, see Section 4.5.

SELinux Permissions

Before invoking the installer on a system that is running SELinux, you must set SELinux to permissive mode.

The following example works on Redhat Enterprise Linux, Fedora Core or CentOS distributions. Use comparable commands that are compatible with your Linux distribution to set SELinux to permissive mode during installation and return it to enforcing mode when installation is complete.

Before installing Advanced Server, set SELinux to permissive mode with the command:

```
# setenforce Permissive
```

When the installation is complete, return SELinux to enforcing mode with the command:

```
# setenforce Enforcing
```

Windows-specific Software Requirements

You should apply Windows operating system updates before invoking the Advanced Server installer. If (during the installation process) the installer encounters errors, exit the installation, and ensure that your version of Windows is up-to-date before restarting the installer.

Migration Toolkit or EDB*Plus Installation Pre-requisites

Before using an RPM or StackBuilder Plus to install Migration Toolkit or EDB*Plus, you must first install Java (version 1.7 or later). On a Linux system, you can use the yum package manager to install Java. Open a terminal window, assume superuser privileges, and enter:

```
# yum install java
```

Follow the onscreen instructions to complete the installation.

If you are using Windows, Java installers and instructions are available online at:

http://www.java.com/en/download/manual.jsp

Locales Requiring Product Keys

The Advanced Server 10 installer will request a product key before completing an installation into a host system using one of the locales listed in the table below. Product keys are available from your local Advanced Server distributor.

Note: The product key applies only to the Advanced Server installation program. The Advanced Server database program has no built-in limitations or expiration features that require a product key or any other activation technique.

Locale	Locale Identifier
Traditional Chinese with Hong Kong SCS	zh_HK
Traditional Chinese for Taiwan	zh_TW
Simplified Chinese	zh_CN
Japanese	ja_JP
Korean	ko_KR
Argentina – Spanish	es_ar
Beliz – English	en_bz
Brazil - Portuguese	pt_br
Bolivia - Spanish	es_bo
Chile - Spanish	es_cl
Colombia - Spanish	es_co
Costa Rica - Spanish	es_cr
Dominican Republic - Spanish	es_do
Ecuador - Spanish	es_ec
Guatemala - Spanish	es_gt
Guyana - English	en_gy
Honduras - Spanish	es_hn
Mexico - Spanish	es_mx
Nicaragua - Spanish	es_ni
Panama - Spanish	es_pa
Peru - Spanish	es_pe
Puerto Rico - Spanish	es_pr
Paraguay - Spanish	es_py
El Salvador - Spanish	es_sv
Uruguay - Spanish	es_uy
Venezuela - Spanish	es_ve

During an installation in one of the listed locales, the Product Key window (shown in Figure 4.1) will open, prompting you to provide a valid product key. Enter a product key, and press Next to continue with the installation.

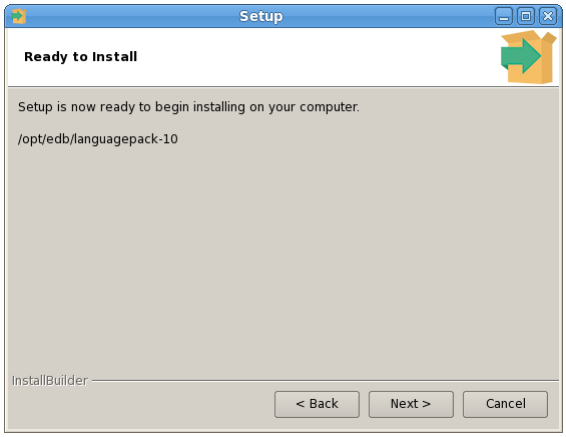

Figure 4.1 -The Advanced Server Product Key Window

Performing a Graphical Installation

A graphical installation wizard provides a quick and easy way to install Advanced Server 10 on a Linux or Windows system. As the Setup wizard's easy-to-follow dialogs lead you through the installation process, specify information about your system, your system usage, and the modules that will best complement your installation of Advanced Server. When you have completed the dialogs, the installer performs an installation based on the selections made during the setup process.

When the Advanced Server installation finishes, you will be offered the option to invoke EDB Postgres StackBuilder Plus. StackBuilder Plus provides an easy-to-use graphical interface that can update installed products, or download and add any omitted modules (and the resulting dependencies) after your Advanced Server setup and installation completes. See Section 4.5 for more information about StackBuilder Plus.

Using the Graphical Installer with Windows

To perform an installation using the graphical wizard on a Windows system, you must have administrator privileges. To start the Setup wizard, assume administrator privileges, and double-click the edb-as10-server-10.x.*x*-*x*-windows-x64 executable file.

To install Advanced Server on some versions of Windows, you may be required to right click on the file icon and select Run as Administrator from the context menu to invoke the installer with Administrator privileges.

The wizard opens a Language Selection popup; select an installation language from the drop-down listbox and click OK to continue to the Setup window (shown in Figure 4.2):

Figure 4.2 -The Advanced Server installer Welcome window

Click Next to continue.

The EnterpriseDB License Agreement (see Figure 4.3) opens.

Figure 4.3 -The EnterpriseDB License Agreement

Carefully review the license agreement before highlighting the appropriate radio button; click Next to continue.

The User Authentication window opens, as shown in Figure 4.4.

Figure 4.4 -The User Authentication window.

Before continuing, you must provide the email address and password associated with your EnterpriseDB user account. Registration is free; if you do not have an EnterpriseDB user account, click the link provided to open a web browser, and supply your user information.

Enter the email address of a registered account in the Email field, and the corresponding password in the Password field, and click Next to continue.

The Installation Directory window opens, as shown in Figure 4.5.

Figure 4.5 -The Installation Directory window.

By default, the Advanced Server installation directory is:

C:\Program Files\edb\as10

You can accept the default installation location, and click Next to continue, or optionally click the File Browser icon to open the Browse For Folder dialog to choose an alternate installation directory.

Figure 4.6 -The Select Components window

The Select Components window (shown in Figure 4.6) contains a list of optional components that you can install with the Advanced Server Setup wizard. You can omit a module from the Advanced Server installation by deselecting the box next to the components name.

The Setup wizard can install the following components while installing Advanced Server 10:

EDB Postgres Advanced Server

Select the EDB Postgres Advanced Server option to install Advanced Server 10.

pgAdmin 4

Select the EDB Postgres pgAdmin 4 option to install the pgAdmin 4 client. pgAdmin 4 provides a powerful graphical interface for database management and monitoring.

StackBuilder Plus

> The StackBuilder Plus utility is a graphical tool that can update installed products, or download and add supporting modules (and the resulting dependencies) after your Advanced Server setup and installation completes. See Section 4.5 for more information about StackBuilder Plus.

Command Line Tools

> The Command Line Tools option installs command line tools and supporting client libraries including:
>
> - libpq
> - psql
> - EDB*Loader
> - ecpgPlus
> - pg_basebackup, pg_dump, and pg_restore
> - pg_bench
> - and more.

> Please note: the Command Line Tools are required if you are installing Advanced Server or pgAdmin 4.

After selecting the components you wish to install, click Next to open the Additional Directories window (shown in Figure 4.7).

Figure 4.7 -The Additional Directories window.

By default, the Advanced Server data files are saved to:

C:\Program Files\edb\as10\data

The default location of the Advanced Server Write-Ahead Log (WAL) Directory is:

C:\Program Files\edb\as10\data\pg_wal

Advanced Server uses write-ahead logs to promote transaction safety and speed transaction processing; when you make a change to a table, the change is stored in shared memory and a record of the change is written to the write-ahead log. When you perform a COMMIT, Advance Server writes contents of the write-ahead log to disk.

Accept the default file locations, or use the File Browser icon to select an alternate location; click Next to continue to the Advanced Server Dialect window (shown in Figure 4.8).

Figure 4.8 -The Advanced Server Dialect window.

Use the drop-down listbox on the Advanced Server Dialect window to choose a server dialect. The server dialect specifies the compatibility features supported by Advanced Server.

By default, Advance Server installs in Compatible with Oracle mode; you can choose between Compatible with Oracle and Compatible with PostgreSQL installation modes.

Compatible with Oracle

If you select Compatible with Oracle on the Configuration Mode dialog, the installation will include the following features:

- Data dictionary views compatible with Oracle databases.
- Oracle data type conversions.
- Date values displayed in a format compatible with Oracle syntax.
- Support for Oracle-styled concatenation rules (if you concatenate a string value with a NULL value, the returned value is the value of the string).
- Schemas (dbo and sys) compatible with Oracle databases added to the SEARCH_PATH.
- Support for the following Oracle built-in packages:

Package	Functionality Compatible with Oracle Databases
dbms_alert	Provides the ability to register for, send and receive alerts.
dbms_aq	Provides queueing functionality for Advanced Server.
dbms_aqadm	Provides supporting functionality for dbms_aq.
dbms_crypto	Provides a way to encrypt or decrypt RAW, BLOB or CLOB data.
dbms_job	Implements job-scheduling functionality.
dbms_lob	Provides the ability to manage large objects.
dbms_lock	Provides support for the DBMS_LOCK.SLEEP procedure.
dbms_mview	Provides a way to manage and refresh materialized views.
dbms_output	Provides the ability to display a message on the client.
dbms_pipe	Provides the ability to send a message from one session and read it in another session.
dbms_profiler	Collects and stores performance data about PL/pgSQL and SPL statements.
dbms_random	Provides a way to generate random numbers.
dbms_rls	Implements row level security.
dbms_scheduler	Provides a way to create and manage Oracle-style jobs.
dbms_session	A partial implementation that provides support for DBMS_SESSION.SET_ROLE.
dbms_sql	Implements use of Dynamic SQL
dbms_utility	Provides a collection of misc functions and procedures.
utl_encode	Provides a way to encode or decode data.
utl_file	Provides a way for a function, procedure or anonymous block to interact with files stored in the server’s file system.
utl_http	Provides a way to use HTTP or HTTPS to retrieve information found at a URL.
utl_mail	Provides a simplified interface for sending email and attachments.
utl_raw	Provides a way to manipulate or retrieve the length of raw data types.
utl_smtp	Implements smtp email functions.

Package	Functionality Compatible with Oracle Databases
utl_url	Provides a way to escape illegal and reserved characters in a URL.

This is not a comprehensive list of the compatibility features for Oracle included when Advanced Server is installed in Compatible with Oracle mode; more information about see the *Database Compatibility for Oracle Developer's Guide* available from the EnterpriseDB website at:

http://www.enterprisedb.com/products-services-training/products/documentation

If you choose to install in Compatible with Oracle mode, the Advanced Server superuser name is enterprisedb.

Compatible with PostgreSQL

If you select Compatible with PostgreSQL, Advanced Server will exhibit compatibility with PostgreSQL version 10. If you choose to install in Compatible with PostgreSQL mode, the default Advanced Server superuser name is postgres.

For detailed information about PostgreSQL functionality, visit the official PostgreSQL website at:

http://www.postgresql.org

After specifying a configuration mode, click Next to continue to the Password window (shown in Figure 4.9).

![image](./images/image12.png)

Figure 4.9 -The Password window.

Advanced Server uses the password specified on the Password window for the database superuser. The specified password must conform to any security policies existing on the Advanced Server host.

After you enter a password in the Password field, confirm the password in the Retype Password field, and click Next to continue.

The Additional Configuration window opens (shown in Figure 4.10).

![image](./images/image13.png)

Figure 4.10 -The Additional Configuration window.

Use the fields on the Additional Configuration window to specify installation details:

- Use the Port field to specify the port number that Advanced Server should listen to for connection requests from client applications. The default is 5444.
- If the Locale field is set to [Default locale], Advanced Server uses the system locale as the working locale. Use the drop-down listbox next to Locale to specify an alternate locale for Advanced Server.
- By default, the Setup wizard installs corresponding sample data for the server dialect specified by the compatibility mode (Oracle or PostgreSQL). Clear the checkbox next to Install sample tables and procedures if you do not wish to have sample data installed.

After verifying the information on the Additional Configuration window, click Next to open the Dynatune Dynamic Tuning: Server Utilization window (shown in Figure 4.11).

The graphical Setup wizard facilitates performance tuning via the Dynatune Dynamic Tuning feature. Dynatune functionality allows Advanced Server to make optimal usage of the system resources available on the host machine on which it is installed.

![image](./images/image14.png)

Figure 4.11 -The Dynatune Dynamic Tuning: Server Utilization window.

The edb_dynatune configuration parameter determines how Advanced Server allocates system resources. Use

the radio buttons on the Server Utilization window to set the initial value of the edb_dynatune configuration parameter:

- Select Development to set the value of edb_dynatune to 33. A low value dedicates the least amount of the host machine's resources to the database server. This is a good choice for a development machine.
- Select General Purpose to set the value of edb_dynatune to 66. A mid-range value dedicates a moderate amount of system resources to the database server. This would be a good setting for an application server with a fixed number of applications running on the same host as Advanced Server.
- Select Dedicated to set the value of edb_dynatune to 100. A high value dedicates most of the system resources to the database server. This is a good choice for a dedicated server host.

After the installation is complete, you can adjust the value of edb_dynatune by editing the postgresql.conf file, located in the data directory of your Advanced Server installation. After editing the postgresql.conf file, you must restart the server for your changes to take effect.

Select the appropriate setting for your system, and click Next to continue to the Dynatune Dynamic Tuning: Workload Profile window (shown in Figure 4.12).

Figure 4.12 -The Dynatune Dynamic Tuning: Workload Profile window.

Use the radio buttons on the Workload Profile window to specify the initial value of the edb_dynatune_profile configuration parameter. The edb_dynatune_profile parameter controls performance-tuning aspects based on the type of work that the server performs.

- Select Transaction Processing (OLTP systems) to specify an edb_dynatune_profile value of oltp. Recommended when Advanced Server is supporting heavy online transaction processing.
- Select General Purpose (OLTP and reporting workloads) to specify an edb_dynatune_profile value of mixed. Recommended for servers that provide a mix of transaction processing and data reporting.
- Select Reporting (Complex queries or OLAP workloads) to specify an edb_dynatune_profile value of reporting. Recommended for database servers used for heavy data reporting.

After the installation is complete, you can adjust the value of edb_dynatune_profile by editing the postgresql.conf file, located in the data directory of your Advanced Server installation. After editing the postgresql.conf file, you must restart the server for your changes to take effect.

For more information about edb_dynatune and other performance-related topics, see the *EDB Postgres Advanced Server Guide* available from the EnterpriseDB website at:

http://www.enterprisedb.com/products-services-training/products/documentation

Click Next to continue. The Update Notification Service window (shown in Figure 4.13) opens.

Figure 4.13 -The Update Notification Service window.

When enabled, the update notification service notifies you of any new updates and security patches available for your installation of Advanced Server.

By default, Advanced Server is configured to start the service when the system boots; clear the Install Update Notification Service checkbox, or accept the default, and click Next to continue.

The Pre Installation Summary opens as shown in Figure 4.14.

Figure 4.14 -The Pre Installation Summary.

The Pre Installation Summary provides an overview of the options specified during the Setup process. Review the options before clicking Next; use the Back button to navigate back through the dialogs and update any options.

The Ready to Install window (see Figure 4.15) confirms that the installer has the information it needs about your configuration preferences to install Advanced Server. Click Next to continue.

Figure 4.15 -The Ready to Install window.

Figure 4.16 -Popup dialogs confirm the installation of supporting modules.

As each supporting module is unpacked and installed, the module's installation is confirmed with a progress bar (see Figure 4.16).

Before the Setup wizard completes the Advanced Server installation, it offers to Launch StackBuilder Plus at exit? (see Figure 4.17).

Figure 4.17 -The Setup wizard offers to Launch StackBuilder Plus at exit.

You can clear the StackBuilder Plus checkbox and click Finish to complete the Advanced Server installation, or accept the default and proceed to StackBuilder Plus.

EDB Postgres StackBuilder Plus is included with the installation of Advanced Server and its core supporting components. StackBuilder Plus is a graphical tool that can update installed products, or download and add supporting modules (and the resulting dependencies) after your Advanced Server setup and installation completes. See Section 4.5 for more information about StackBuilder Plus.

Using the Graphical Installer on a Linux System

To use the graphical installation wizard on a Linux system, you must have superuser privileges. To invoke the Setup wizard, open a Terminal window, navigate to the directory that contains the Advanced Server installer, and enter the command:

./edb-as10-server-10.*x.x.x*-linux-x64.run

The wizard opens a Language Selection popup; select an installation language from the drop-down listbox and click OK to continue to the Welcome window (shown in Figure 4.18).

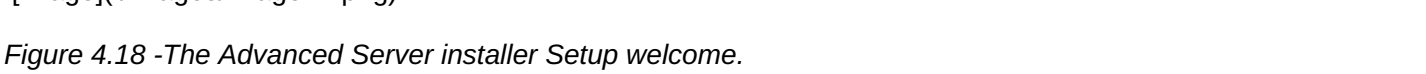

Figure 4.18 -The Advanced Server installer Setup welcome.

Click Next to continue.

The License Agreement window (shown in Figure 4.19) opens.

Figure 4.19 -The EnterpriseDB License Agreement.

Review the EnterpriseDB License Agreement carefully before selecting the radio button next to I accept the agreement. Click Next to continue to the User Authentication window.

The User Authentication window opens, as shown in Figure 4.20.

Figure 4.20 -The User Authentication window.

Before continuing, you must provide the email address and password associated with your EnterpriseDB user account. Registration is free; if you do not have an EnterpriseDB user account, click the link provided to open a web browser, and enter your user information.

Enter the email address of a registered account in the Email field, and the corresponding password in the Password field, and click Next to continue to the Installation Directory window (see Figure 4.21).

Figure 4.21 -The Installation Directory window.

By default, the Advanced Server installation directory is:

/opt/edb/as10

You can accept the default installation location, and click Next to continue, or click the File Browser icon to open a dialog and choose an alternate installation directory.

Figure 4.22 -The Select Components window.

The Select Components window (shown in Figure 4.22) contains a list of the tools and utilities that you can install with the Advanced Server Setup wizard. To omit a component from your installation, deselect the check to the left of the component name.

The Setup wizard can install the following components while installing Advanced Server 10:

EDB Postgres Advanced Server

Select the EDB Postgres Advanced Server option to install Advanced Server.

pgAdmin 4

Select the pgAdmin 4 option to install the pgAdmin 4 client. pgAdmin 4 provides a powerful graphical interface for database management and monitoring.

StackBuilder Plus

> The StackBuilder Plus utility is a graphical tool that can update installed products, or download and add supporting modules (and the resulting dependencies) after your Advanced Server setup and installation completes. See Section 4.5 for more information about StackBuilder Plus.

Command Line Tools

> The Command Line Tools option installs command line tools and supporting client libraries including:
>
> - libpq
> - psql
> - EDB*Loader
> - ecpgPlus
> - pg_basebackup, pg_dump, and pg_restore
> - pg_bench
> - and more.

> This option is required if you are installing Advanced Server or pgAdmin 4.

After selecting the components you wish to install, click Next to open the Additional Directories window (shown in Figure 4.23).

Figure 4.23 -The Additional Directories window.

Use the fields in the Additional Directories window to specify locations for the Advanced Server Data Directory and Write-Ahead Log (WAL) Directory.

The default Data Directory is /opt/edb/as10/data. You can use the file selector icon to specify an alternate location.

The default location of the Advanced Server Write-Ahead Log (WAL) Directory is opt/edb/as10/data/pg_wal. Accept the default location, or specify an alternate location with the file selector icon.

Advanced Server uses write-ahead logs to promote transaction safety and speed transaction processing; when you make a change to a table, the change is stored in shared memory and a record of the change is written to the write-ahead log. When you perform a COMMIT, Advance Server writes contents of the write-ahead log to disk.

Click Next to continue to the Advanced Server Dialect window (shown in Figure 4.24).

![image](./images/image27.png)

Figure 4.24 -The Configuration Mode window.

Use the drop-down listbox on the Advanced Server Dialect window to choose a server dialect. The server dialect specifies the compatibility features supported by Advanced Server.

By default, Advance Server installs with database compatibility with Oracle; you can choose between Compatible with Oracle and Compatible with PostgreSQL installation modes.

Compatible with Oracle Mode

If you select Compatible with Oracle, the installation will include the following features:

- Dictionary views compatible with Oracle databases.
- Oracle data type conversions.
- Date values displayed in a format compatible with Oracle syntax.
- Oracle-styled concatenation rules (if you concatenate a string value with a NULL value, the returned value is the value of the string).
- Schemas (dbo and sys) compatible with Oracle databases added to the SEARCH_PATH.
- Support for the following Oracle built-in packages:

Package	Functionality Compatible with Oracle Databases
dbms_alert	Provides the ability to register for, send and receive alerts.
dbms_aq	Provides queueing functionality for Advanced Server.
dbms_aqadm	Provides supporting functionality for dbms_aq.
dbms_crypto	Provides a way to encrypt or decrypt RAW, BLOB or CLOB data.
dbms_job	Implements job-scheduling functionality.
dbms_lob	Provides the ability to manage large objects.
dbms_lock	Provides support for the DBMS_LOCK.SLEEP procedure.
dbms_mview	Provides a way to manage and refresh materialized views.
dbms_output	Provides the ability to display a message on the client.
dbms_pipe	Provides the ability to send a message from one session and read it in another session.
dbms_profiler	Collects and stores performance data about PL/pgSQL and SPL statements.
dbms_random	Provides a way to generate random numbers.
dbms_rls	Implements row level security.
dbms_scheduler	Provides a way to create and manage Oracle-style jobs.
dbms_session	A partial implementation that provides support for DBMS_SESSION.SET_ROLE.
dbms_sql	Implements use of Dynamic SQL
dbms_utility	Provides a collection of misc functions and procedures.
utl_encode	Provides a way to encode or decode data.

Package	Functionality Compatible with Oracle Databases
utl_file	Provides a way for a function, procedure or anonymous block to interact with files stored in the server's file system.
utl_http	Provides a way to use HTTP or HTTPS to retrieve information found at a URL.
utl_mail	Provides a simplified interface for sending email and attachments.
utl_raw	Provides a way to manipulate or retrieve the length of raw data types.
utl_smtp	Implements smtp email functions.
utl_url	Provides a way to escape illegal and reserved characters in a URL.

This is not a comprehensive list of the compatibility features for Oracle included when Advanced Server is installed in Compatible with Oracle mode. For more information, refer to the *Database Compatibility for Oracle Developer's Guide* available at:

http://www.enterprisedb.com/products-services-training/products/documentation

If you choose to install in Compatible with Oracle mode, the Advanced Server superuser name is enterprisedb.

Compatible with PostgreSQL Mode

When installed in Compatible with PostgreSQL mode, Advanced Server exhibits complete compatibility with Postgres version 10.

For more information about PostgreSQL functionality, visit the official PostgreSQL website at:

http://www.postgresql.org

If you choose to install in Compatible with PostgreSQL mode, the Advanced Server superuser name is postgres.

After specifying a configuration mode, click Next to continue to the Password window (shown in Figure 4.25).

![image](./images/image28.png)

Figure 4.25 -The Password window.

Advanced Server uses the password specified in the Password window for the database superuser. The specified password must conform to any security policies existing on the Advanced Server host.

After you enter a password in the Password field, confirm the password in the Retype Password field, and click Next to continue.

![image](./images/image29.png)

Figure 4.26 -The Additional Configuration window.

Use the fields on the Additional Configuration window (shown in Figure 4.26) to specify installation details:

- Use the Port field to specify the port number that Advanced Server should listen to for connection requests from client applications. The default is 5444.
- If the Locale field is set to [Default Locale], Advanced Server uses the system locale as the working locale. Use the drop-down listbox next to Locale to specify an alternate locale for Advanced Server.
- By default, the Setup wizard installs corresponding sample data for the server dialect specified (Oracle or PostgreSQL). Clear the checkbox next to Install sample tables and procedures if you do not wish to have sample data installed.

After verifying the selections on the Additional Configuration window, click Next to open the Dynatune Dynamic Tuning: Server Utilization window (shown in Figure 4.27).

The Setup wizard facilitates performance tuning via the Dynatune Dynamic Tuning feature. Dynatune functionality allows Advanced Server to make optimal usage of the system resources available on the host machine.

Figure 4.27 -The Server Utilization window.

The edb_dynatune configuration parameter determines how Advanced Server allocates system resources. The radio buttons on the Server Utilization window set the initial value of the edb_dynatune configuration parameter.

- Select Development to set the value of edb_dynatune to 33. A low value dedicates the least amount of the host machine's resources to the database server. This is a good choice for a development machine.
- Select General Purpose to set the value of edb_dynatune to 66. A mid-range value dedicates a moderate amount of system resources to the database server. This would be a good setting for an application server with a fixed number of applications running on the same host as Advanced Server.
- Select Dedicated to set the value of edb_dynatune to 100. A high value dedicates most of the system resources to the database server. This is a good choice for a dedicated server host.

After the installation is complete, you can adjust the value of edb_dynatune by editing the postgresql.conf file, located in the data directory of your Advanced Server Installation. After editing the postgresql.conf file, you must restart the server for the changes to take effect.

Select the appropriate setting for your system, and click Next to continue to the Dynatune Dynamic Tuning: Workload Profile window (shown in Figure 4.28).

Figure 4.28 -The Workload Profile window.

Use the radio buttons on the Workload Profile window to specify the initial value of the edb_dynatune_profile configuration parameter. The edb_dynatune_profile parameter controls performance-tuning aspects based on the type of work that the server performs.

- Select Transaction Processing (OLTP systems) to specify an edb_dynatune_profile value of oltp. Recommended when Advanced Server is supporting heavy online transaction processing workloads.
- Select General Purpose (OLTP and reporting workloads) to specify an edb_dynatune_profile value of mixed. Recommended for servers that provide a mix of transaction processing and data reporting.
- Select Reporting (Complex queries or OLAP workloads) to specify an edb_dynatune_profile value of reporting. Recommended for database servers used for heavy data reporting.

After the installation is complete, you can adjust the value of edb_dynatune_profile by editing the postgresql.conf file, located in the data directory of your Advanced Server installation. After editing the postgresql.conf file, you must restart the server for the changes to take effect.

For more information about edb_dynatune and other performance-related topics, see the *EDB Postgres Advanced Server Guide* available at:

http://www.enterprisedb.com/products-services-training/products/documentation

After selecting the radio button that best describes the use of the system, click Next to continue. The Advanced Configuration window (shown in Figure 4.29) opens.

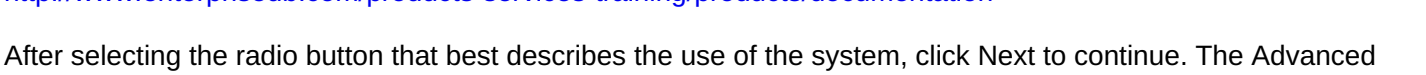

Figure 4.29 –The Update Notification Service selection window.

Check the box to the left of Install Update Notification Service to install the service and enable notifications. When enabled, the Update Notification Service notifies you of any new updates and security patches available for your installation of Advanced Server.

By default, Advanced Server is configured to install and enable the service when the system boots; clear the checkbox to skip the installation, or accept the defaults, and click Next to continue.

The Pre Installation Summary window opens (shown in Figure 4.30).

Figure 4.30 -The Pre Installation Summary.

The Pre Installation Summary provides an overview of the options specified during the Setup process. Review the options before clicking Next; use the Back button to navigate back through the dialogs to update any options.

The Ready to Install window (shown in Figure 4.31) confirms that the installer has the information it needs about your configuration preferences to install Advanced Server.

Figure 4.31 -The Ready to Install window.

Click Next to continue. The Setup wizard confirms the installation progress of Advanced Server via a progress bar (shown in Figure 4.32).

Figure 4.32 –The wizard confirms the installation progress.

If you have elected to add StackBuilder Plus to your installation, the Setup wizard will offer to Launch Stack Builder Plus at exit? (see Figure 4.33).

Figure 4.33 -The Setup wizard offers to Launch StackBuilder Plus at exit.

Clear the StackBuilder Plus checkbox and click Finish to complete the Advanced Server installation, or accept the default and proceed to StackBuilder Plus.

StackBuilder Plus provides a graphical interface that updates, downloads and installs applications and drivers that work with Advanced Server. For more information about StackBuilder Plus, see Section 4.5, *Using StackBuilder Plus*.

Invoking the Installer from the Command Line

The command line options of the Advanced Server installer offer functionality in situations where a graphical installation may not work because of limited resources or privileges. You can:

- Include the --mode unattended option when invoking the installer to perform an installation without user input.
- Include the --mode text option when invoking the installer to perform an installation from the command line.
- Invoke the installer with the --extract-only option to perform a minimal installation when you don’t hold the privileges required to perform a complete installation.

Not all command line options are suitable for all platforms. For a complete reference guide to the command line options, see Section 4.4.4, *Reference - Command Line Options*.

Please note: If you are invoking the installer from the command line to perform a system upgrade, the installer will ignore command line options, and preserve the configuration of the previous installation.

Performing a Text Mode Installation

To specify that the installer should run in text mode, include the --mode text command line option when invoking the installer. Text-mode installations are useful if you need to install on a remote server using ssh tunneling (and have access to a minimal amount of bandwidth), or if you do not have access to a graphical interface.

In text mode, the installer uses a series of command line questions to establish the configuration parameters. Text-mode installations are valid only on Linux systems.

You must assume superuser privileges before performing a text-mode installation. To perform a text-mode installation on a Linux system, navigate to the directory that contains the installation binary file and enter:

```
# ./edb-as10-server-10.x.x.x-linux-x64.run --mode text
```

At any point during the installation process, you can press Ctrl-C to abort the installation.

The text mode installer welcomes you to the Setup Wizard, and introduces the License Agreement. Use the Enter key to page through the license agreement:

> Welcome to the EDB Postgres Advanced Server Setup Wizard.
>
> ---------- Please read the following License Agreement. You must accept the terms of this agreement before continuing with the installation. Press [Enter] to continue:

Press Enter to continue reviewing the license agreement. After displaying the license, the installer prompts you to accept the license:

> Do you accept this license? [y/n]:

After reading the license agreement, enter y to accept the agreement and proceed with the installation. Enter n if you do not accept the license agreement; this will abort the installation.

> Press Enter to proceed.

Next, the installer will prompt you for the User Authentication information associated with your EnterpriseDB user account. There is no charge to register for an EnterpriseDB user account; if you do not have a user account, visit http://www.enterprisedb.com/user-login-registration to register.

> This installation requires a registration with EnterpriseDB.com. Please enter your credentials below. If you do not have an account, Please create one now on https://www.enterprisedb.com/user-login-registration

When prompted, enter the email address of a registered account, and then the corresponding password:

> Email []:
>
> Password :

By default, Advanced Server is installed in /opt/edb/as10:

> Please specify the directory where EDB Postgres Advanced Server will be installed.
>
> Installation Directory [/opt/edb/as10]:

Enter an alternate location, or press Enter to accept the default and continue to the component selection portion of the installation process.

Next, the installer prompts you individually for each component that is to be installed with Advanced Server:

> Select the components you want to install.

Enter Y or press Enter to accept the default value of yes after each component that you wish to include with the installation. Enter n to omit a component from the installation.

> EDB Postgres Advanced Server [Y/n] : pgAdmin 4 [Y/n] : StackBuilder Plus [Y/n] : Command Line Tools [Y/n] :

The Advanced Server components are:

EDB Postgres Advanced Server

Press Y or enter after the EDB Postgres Advanced Server option to install Advanced Server 10.

pgAdmin 4

pgAdmin 4 provides a powerful graphical interface for database management and monitoring.

StackBuilder Plus

The StackBuilder Plus utility is a graphical tool that can update installed products, or download and add supporting modules (and the resulting dependencies) after your Advanced Server setup and installation completes. See Section 4.5 for more information about StackBuilder Plus.

Command Line Tools

The Command Line Tools option installs command line tools and supporting client libraries including:

- libpq
- psql
- EDB*Loader
- ecpgPlus
- pg_basebackup, pg_dump, and pg_restore
- pg_bench
- and more.

This option is required if you are installing Advanced Server or pgAdmin 4.

After selecting components for installation, confirm that the list is correct by entering Y; enter n to iterate through the list of components a second time.

> Is the selection above correct? [Y/n]:

Next, the installer prompts you to specify the location of the Additional Directories required by Advanced Server. The default data directory is /opt/edb/as10/data. You can specify an alternate location, or press Enter to accept the default and continue.

> Please select a directory under which to store your data. Data Directory [/opt/edb/as10/data]:

The default location of the Advanced Server Write-Ahead Log (WAL) Directory is /opt/edb/as10/data/pg_wal. Press Enter to accept the default location and continue, or specify an alternate location.

> Please select a directory under which to store your Write-Ahead Logs. Write-Ahead Log (WAL) Directory [/opt/edb/as10/data/pg_wal]:

Advanced Server uses write-ahead logs to help ensure transaction safety and speed transaction processing; when you make a change to a table, the change is stored in shared memory and a record of the change is written to the write-ahead log. When you COMMIT a transaction, Advance Server writes contents of the write-ahead log to disk.

Next, the installer prompts you to select an Advanced Server Dialect:

> EDB Postgres Advanced Server can be configured in one of two "Dialects" - 1) Compatible with Oracle or 2) Compatible with Postgres.
>
> If you select Compatible with Oracle, Advanced Server will be configured with appropriate data type conversions, time and date formats, Oracle-styled operators, dictionary views and more. This makes it easier to migrate or write new applications that are more compatible with the Oracle database.
>
> If you select Compatible with Postgres, Advanced Server will be configured with standard PostgeSQL data types, time/date formats and operators.

> Advanced Server Dialect
>
> [1] Compatible with Oracle
>
> [2] Compatible with PostgreSQL
>
> Please choose an option [1] :

The configuration mode specifies the server dialect with which Advanced Server will be compatible; you can choose between Compatible with Oracle and Compatible with PostgreSQL installation modes.

Compatible with Oracle Mode

Installing Advanced Server in Compatible with Oracle mode provides the following functionality:

- Data dictionary views and data type conversions compatible with Oracle databases.
- Date values displayed in a format compatible with Oracle syntax.
- Oracle-styled concatenation rules (if you concatenate a string value with a NULL value, the returned value is the value of the string).
- Schemas (dbo and sys) compatible with Oracle databases added to the SEARCH_PATH.
- Support for the following Oracle built-in packages:

Package	Functionality Compatible with Oracle Databases
dbms_alert	Provides the ability to register for, send and receive alerts.
dbms_aq	Provides queueing functionality for Advanced Server.
dbms_aqadm	Provides supporting functionality for dbms_aq.
dbms_crypto	Provides a way to encrypt or decrypt RAW, BLOB or CLOB data.
dbms_job	Implements job-scheduling functionality.
dbms_lob	Provides the ability to manage large objects.
dbms_lock	Provides support for the DBMS_LOCK.SLEEP procedure.
dbms_mview	Provides a way to manage and refresh materialized views.
dbms_output	Provides the ability to display a message on the client.
dbms_pipe	Provides the ability to send a message from one session and read it in another session.
dbms_profiler	Collects and stores performance data about PL/pgSQL and SPL statements.
dbms_random	Provides a way to generate random numbers.
dbms_rls	Implements row level security.
dbms_scheduler	Provides a way to create and manage Oracle-style jobs.
dbms_session	A partial implementation that provides support for DBMS_SESSION.SET_ROLE.
dbms_sql	Implements use of Dynamic SQL
dbms_utility	Provides a collection of misc functions and procedures.
utl_encode	Provides a way to encode or decode data.
utl_file	Provides a way for a function, procedure or anonymous block to interact with files stored in the server's file system.
utl_http	Provides a way to use HTTP or HTTPS to retrieve information found at a URL.
utl_mail	Provides a simplified interface for sending email and attachments.
utl_raw	Provides a way to manipulate or retrieve the length of raw data types.
utl_smtp	Implements smtp email functions.
utl_url	Provides a way to escape illegal and reserved characters in a URL.

This is not a comprehensive list of the compatibility features for Oracle included when Advanced Server is installed in Compatible with Oracle mode; more information about Advanced Server is available in the *Database Compatibility for Oracle Developer's Guide* available at:

http://www.enterprisedb.com/products-services-training/products/documentation/enterpriseedition

If you choose to install in Compatible with Oracle mode, the Advanced Server superuser name is enterprisedb.

Compatible with PostgreSQL Mode

When installed in Compatible with PostgreSQL mode, Advanced Server exhibits complete compatibility with Postgres version 10. For more information about PostgreSQL functionality, visit the official PostgreSQL website at:

http://www.postgresql.org

If you choose to install in Compatible with PostgreSQL mode, the Advanced Server superuser name is postgres.

Press Enter to accept the default configuration mode (Compatible with Oracle) and continue; enter 2 and press Enter to install in Compatible with PostgreSQL mode.

Next, the installer prompts you for a database superuser password:

> Please provide a password for the database superuser (enterprisedb). A locked Unix user account (enterprisedb) will be created if not present.
>
> Password :
>
> Retype Password :

Advanced Server uses the password specified for the database superuser. The password must conform to any security policies existing on the Advanced Server host.

After entering a password in the Password field, confirm the password and press Enter to continue.

The installer prompts you for Additional Configuration information:

> Additional Configuration

When prompted, enter the Port that the Advanced Server service will monitor for connections. By default, Advanced Server chooses the first available port after port number 5444:

> Please select the port number the server should listen on.
>
> Port [5444]:

Specify a Locale by entering a locale number from the list shown. Accept the Default locale value to instruct the installer to use the system locale as the server locale.

> Select the locale to be used by the new database cluster.
>
> Locale
>
> [1] [Default locale] [2] C [3] POSIX
>
> Please choose an option [1] :

When prompted, enter Y (or press Enter to accept the default value) to install the sample tables and procedures for the database dialect specified by the compatibility mode (Oracle or PostgreSQL):

> Would you like to install sample tables and procedures? Install sample tables and procedures. [Y/n]:

Dynatune functionality allows Advanced Server to make optimal usage of the system resources available on the host machine. To facilitate performance tuning through Dynatune, the installer prompts you first for Server Utilization information:

> Dynatune Dynamic Tuning: Server Utilization
>
> Please select the type of server to determine the amount of system resources that may be utilized:

> [1] Development (e.g. a developer's laptop)
>
> [2] General Purpose (e.g. a web or application server)
>
> [3] Dedicated (a server running only EDB Postgres)
>
> Please choose an option [2] :

The edb_dynatune configuration parameter determines how Advanced Server allocates system resources. Your selection will establish the initial value of edb_dynatune.

- Specify Development to set the value of edb_dynatune to 33. A low value dedicates the least amount of the host machine's resources to the database server. This is a good choice for a development machine.
- Specify General Purpose to set the value of edb_dynatune to 66. A mid-range value dedicates a moderate amount of system resources to the database server. This would be a good setting for an application server with a fixed number of applications running on the same host as Advanced Server.
- Specify Dedicated to set the value of edb_dynatune to 100. A high value dedicates most of the system resources to the database server. This is a good choice for a dedicated server host.

Enter a value of 1, 2, or 3, or accept the default value of 2 (to indicate that the server will be used for General Purpose processing) and press Enter to continue.

Next, the Advanced Server installer prompts for information about the type of workload the system will be processing:

> Dynatune Dynamic Tuning:
>
> Workload Profile
>
> Please select the type of workload this server will be used for:
>
> [1] Transaction Processing (OLTP systems)
>
> [2] General Purpose (OLTP and reporting workloads)
>
> [3] Reporting (Complex queries or OLAP workloads)
>
> Please choose an option [1] :

The installer uses the Workload Profile to establish the initial value of the edb_dynatune_profile configuration parameter. The edb_dynatune_profile parameter controls performance-tuning aspects based on the type of work that the server performs.

- Enter 1 to indicate Transaction Processing (OLTP systems) and set the value of edb_dynatune_profile to oltp. Recommended when Advanced Server is supporting heavy online transaction processing workloads.
- Enter 2 to indicate General Purpose (OLTP and reporting workloads) and set the value of edb_dynatune_profile to mixed. Recommended for servers that provide a mix of transaction processing and data reporting.
- Enter 3 to indicate Reporting (Complex queries or OLAP workloads) and set the value of edb_dynatune_profile to reporting. Recommended for database servers used for heavy data reporting.

After choosing a Workload Profile, press Enter to continue.

After the installation is complete, you can adjust the values of edb_dynatune and edb_dynatune_profile by editing the postgresql.conf file, located in the data directory of your Advanced Server installation, and restarting the server.

For more information about edb_dynatune and other performance-related topics, see the *EDB Postgres Advanced Server Guide* available at:

http://www.enterprisedb.com/products-services-training/products/documentation

Update Notification Service

If enabled, the Update Notification Service notifies you of any available updates and security patches for your installation of Advanced Server.

```
Update Notification Service

Update Notification Service [Y/n]:
```

By default, the installer specifies that Advanced Server should enable the notification service when the system boots; specify n to disable the notification service, or accept the default, and press Enter to continue to the Pre Installation Summary:

```
Pre Installation Summary

Following settings will be used for installation:

Installation Directory: /opt/edb/as10

Data Directory: /opt/edb/as10/data

WAL Directory: /opt/edb/as10/data/pg_wal

Database Port: 5444

Database Superuser: enterprisedb

Operating System Account: enterprisedb

Database Service: edb-as-10

Command Line Tools Installation Directory: /opt/edb/as10

pgAdmin4 Installation Directory: /opt/edb/as10/pgAdmin4

StackBuilderPlus Installation Directory: /opt/edb/as10/stackbuildeplus

Press [Enter] to continue:
```

The Pre Installation Summary lists the options specified during the installation setup process; review the listing and press Enter to continue; press Enter again to start the installation process.

```
Setup is now ready to begin installing EDB Postgres Advanced Server on your computer.

Do you want to continue? [Y/n]:
```

The installer extracts the Advanced Server files and proceeds with the installation:

```
Please wait while Setup installs EDB Postgres Advanced Server on your computer.

Installing EDB Postgres Advanced Server

0% ______________ 50% ______________ 100%

##########################################
```

The installer informs you when the installation is complete.

```
Setup has finished installing EDB Postgres Advanced Server on your computer.
```

Performing an Unattended Installation

To specify that the installer should run without user interaction, include the --mode unattended command line option. In unattended mode, the installer uses one of the following sources for configuration parameters:

- command line options (specified when invoking the installer)
- parameters specified in an option file
- Advanced Server installation defaults

You can embed the non-interactive Advanced Server installer within another application installer; during the installation process, a progress bar allows the user to view the progression of the installation (shown in Figure 4.34).

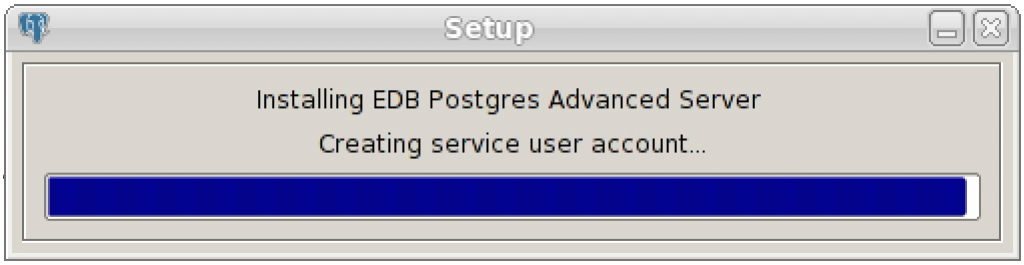

Figure 4.34 –The installation progress bar.

You must have superuser privileges to install Advanced Server using the --mode unattended option on a Linux system. On a Windows system, administrative privileges are required. If you are using the --mode unattended option to install Advanced Server with another installer, the calling installer must be invoked with superuser or administrative privileges.

On Linux

To install in unattended mode on a Linux machine, navigate to the directory that contains the Advanced Server installer and enter:

./edb-as10-server-10.*x.x*-x-linux-x64.run --mode unattended

--superpassword *database_superuser_password* --webusername *edb_user_name@email.com* --webpassword *edb_user_password*

The --superpassword option specifies a password for the database superuser. If you omit the option, the database superuser password defaults to enterprisedb. The default password can be easily guessed by a potential intruder; be sure to provide a stronger password with the --superpassword option.

You must include the --webusername and --webpassword options to specify the identity of a registered EnterpriseDB user. There is no charge to register for an EnterpriseDB user account; if you do not have an account, you can create one at:

http://www.enterprisedb.com/user-login-registration

You can control configuration parameters for Advanced Server by specifying options at the command line, or by including the parameters (in option=value pairs) in a configuration file. A sample configuration file might include:

```
mode=unattended prefix=/opt/edb/as10 datadir=/opt/edb/as10/data serverport=5444
webusername=edb_user_name@email.com webpassword=edb_user_password
```

Then, when you invoke the installer, include the --optionfile parameter, and the complete path to the configuration parameter file:

./edb-as10-server-10.x.x-x-linux-x64.run --optionfile /$HOME/config_param

For more information about the command line options supported during an unattended installation, see Section 4.4.4, *Reference - Command Line Options*.

On Windows

To start the installer in unattended mode on a Windows system, navigate to the directory that contains the executable file, and enter:

edb-as10-server-10.x.*x-x*-windows-x64.exe --mode unattended --superpassword *database_superuser_password* --servicepassword *system_password* --webusername *edb_user_name@email.com* --webpassword *edb_user_password*

When invoking the installer, include the --servicepassword option to specify an operating system password for the user installing Advanced Server.

Use the --superpassword option to specify a password that conforms to the password security policies defined on the host; enforced password policies on your system may not accept the default password (enterprisedb).

Use the --webusername and --webpassword options to specify the identity of a registered EnterpriseDB user; if you do not have an account, you can create one at:

http://www.enterprisedb.com/user-login-registration

Performing an Installation with Limited Privileges

To perform an abbreviated installation of Advanced Server without access to root or administrative privileges, invoke the installer from the command line and include the --extract-only option. Invoking the installer with the --extract-only option extracts the binary files in an unaltered form, allowing you to experiment with a minimal installation of Advanced Server.

If you invoke the installer with the --extract-only options, you can either manually create a cluster and start the service, or run the installation script. To manually create the cluster, you must:

- Initialize the cluster
- Configure the cluster
- Start and stop the service with pg_ctl

For more information about the initdb and pg_ctl commands, please see the PostgreSQL Core Documentation at:

https://www.postgresql.org/docs/10/static/app-initdb.html

https://www.postgresql.org/docs/10/static/app-pg-ctl.html

If you include the --extract-only option when you invoke the installer, the installer steps through a shortened form of the Setup wizard. During the brief installation process, the installer generates an installation script that can be later used to complete a more complete installation. To invoke the installation script, you must have superuser privileges on Linux or administrative privileges on Windows.

The installation script:

- Initializes the database cluster if the cluster is empty.
- Configures the server to start at boot-time.
- Establishes initial values for Dynatune (dynamic tuning) variables.

The scripted Advanced Server installation does not create menu shortcuts or access to EDB Postgres StackBuilder Plus, and no modifications are made to registry files. The Advanced Server Update Monitor will not detect components installed by the scripted installation, and will not issue alerts for available updates to those components.

To perform a limited installation and generate an installation script, download and unpack the Advanced Server installer. Navigate into the directory that contains the installer, and invoke the installer with the command:

On Linux:

./edb-as10-server-10.x*.x-x*-linux-x64.run --extract-only yes

On Windows:

edb-as10-server-10.x*.x-x*-windows.exe --extract-only yes

A dialog opens, prompting you to choose an installation language. Select a language for the installation from the drop-down listbox, and click OK to continue. The Setup Wizard opens (shown in Figure 4.35).

Figure 4.35 -The Welcome window.

Click Next to continue to the Advanced Server License Agreement (shown in Figure 4.36).

Figure 4.36 -The Advanced Server license agreement.

After reading the license agreement, select the appropriate radio button and click Next to continue to the User Authentication window (shown in Figure 4.37).

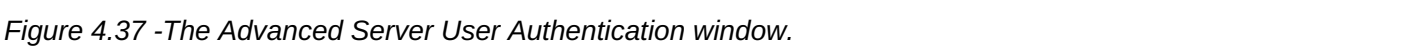

Figure 4.37 -The Advanced Server User Authentication window.

Before continuing, you must provide the email address and password associated with your EnterpriseDB user account. Registration is free; if you do not have a user account, click the link provided to open a web browser, and register your user information.

Enter the email address of a registered account in the Email field, and the corresponding password in the Password field, and click Next to continue.

Figure 4.38 -Specify an installation directory.

On Linux, the default Advanced Server installation directory is:

/opt/edb/as10

On Windows, the default Advanced Server installation directory is:

C:\Program Files\edb\as10

You can accept the default installation location, and click Next to continue to the Ready to Install window (shown in Figure 4.39), or optionally click the File Browser icon to choose an alternate installation directory.

Figure 4.39 -The Setup wizard is ready to install Advanced Server.

Click Next to proceed with the Advanced Server installation. During the installation, progress bars and popups mark the installation progress. The installer notifies you when the installation is complete (see Figure 4.40).

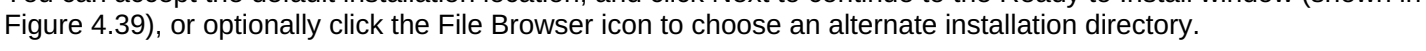

Figure 4.40 -The Advanced Server installation is complete.

After completing the minimal installation, you can execute a script to initialize a cluster and start the service. The script is (by default) located in the following directories:

On Linux:

/opt/edb

On Windows:

C:\Program Files\edb

To execute the installation script, open a command line and assume superuser or administrative privileges. Navigate to the directory that contains the script, and execute the command:

On Linux:

./runAsRoot.sh

On Windows:

cscript runAsAdmin.vbs

The installation script executes at the command line, prompting you for installation configuration information. The default configuration value is displayed in square braces immediately before each prompt; update the default value or press Enter to accept the default value and continue.

Example

The following dialog is an example of a scripted installation on a Linux system. The actual installation dialog will vary by platform and reflect the options specified during the installation.

\====================== INSTALLATION DIRECTORY \====================== Please enter the installation directory [/opt/edb] :

The installation directory is the directory where Advanced Server is installed.

\============== DATA DIRECTORY \============== NOTE: If data directory exists and postgresql.conf file exists in that directory, we will not initialize the cluster.

Please enter the data directory path: [/opt/edb/as10/data] :

The data directory is the directory where Advanced Server data is stored.

\============= WAL DIRECTORY \============= Please enter the Write-Ahead Log (WAL) directory path: [/opt/edb/as10/data/pg_wal] :

The WAL directory is where the write-ahead log will be written.

\============= DATABASE MODE \============= Please enter database mode: [oracle] :

Database mode specifies the database dialect with which the Advanced Server installation is compatible. The optional values are oracle or postgresql.

Compatible with Oracle Mode

Specify oracle mode to include the following functionality:

- Data dictionary views and data type conversions compatible with Oracle databases.
- Date values displayed in a format compatible with Oracle syntax.
- Oracle-styled concatenation rules (if you concatenate a string value with a NULL value, the returned value is the value of the string).
- Schemas (dbo and sys) compatible with Oracle databases added to the SEARCH_PATH.
- Support for the following Oracle built-in packages.

This is not a comprehensive list of the compatibility features for Oracle included when Advanced Server is installed in Compatible with Oracle mode; more information about Advanced Server is available in the *Database Compatibility for Oracle Developer's Guide* available at:

http://www.enterprisedb.com/products-services-training/products/documentation

If you choose to install in Compatible with Oracle mode, the Advanced Server superuser name is enterprisedb.

Compatible with PostgreSQL Mode

Specify postgresql to install Advanced Server with complete compatibility with Postgres version 10.

For more information about PostgreSQL functionality, see the PostgreSQL Core Documentation available at:

http://www.enterprisedb.com/products-services-training/products/documentation

If you choose to install in Compatible with PostgreSQL mode, the Advanced Server superuser name is postgres.

\==== PORT \==== NOTE: We will not be able to examine, if port is currently used by other application.

Please enter port: [5444] :

Specify a port number for the Advanced Server listener to listen on.

\====== LOCALE \====== Please enter the locale: [DEFAULT] :

Specify a locale for the Advanced Server installation. If you accept the DEFAULT value, the locale defaults to the locale of the host system.

\============= SAMPLE TABLES \============= Install sample tables and procedures? (Y/n): [Y] :

Press Return, or enter Y to accept the default, and install the sample tables and procedures; enter an n and press Return to skip this step.

\========================== DATABASE SUPERUSER PASSWORD \========================== Please provide password for the super-user(enterprisedb): [ ] : Please re-type password for the super-user(enterprisedb): [ ] :

Specify and confirm a password for the database superuser. By default, the database superuser is named enterprisedb. (On Windows, there is no password validation if you are logged in as an administrator, but you may be prompted to supply a service account password.)

\================= SERVER UTILIZATION \================= Please enter the server utilization: [66] :

Specify a value between 1 and 100.

The server utilization value is used as an initial value for the edb_dynatune configuration parameter. edb_dynatune determines how Advanced Server allocates system resources.

- A low value dedicates the least amount of the host machine's resources to the database server; a low value is a good choice for a development machine.
- A mid-range value dedicates a moderate amount of system resources to the database server. A mid-range value is a good setting for an application server with a fixed number of applications running on the same host as Advanced Server.
- A high value dedicates most of the system resources to the database server. This is a good choice for a dedicated server host.

After the installation is complete, you can adjust the value of edb_dynatune by editing the postgresql.conf file, located in the data directory of your Advanced Server installation. After editing the postgresql.conf file, you must restart the server for the changes to take effect.

\================ WORKLOAD PROFILE \================ Please enter the workload profile: [oltp] :

The workload profile value is used as an initial value for the edb_dynatune_profile configuration parameter. edb_dynatune_profile controls performance-tuning based on the type of work that the server performs.

- Specify oltp if the server will be supporting heavy online transaction workloads.
- Specify mixed if the server will provide a mix of transaction processing and data reporting.

- Specify reporting if the database server will be used for heavy data reporting.

After the installation is complete, you can adjust the value of edb_dynatune_profile by editing the postgresql.conf file, located in the data directory of your Advanced Server installation, and restarting the server.

Before continuing with the installation, the installer displays the selected options and initializes the database cluster in preparation for the installation of individual components. When the installer has prepared the system for the installation, the installation begins. Before installing a component, the installer prompts you to select modules for installation. With each component, onscreen warnings may alert you to unresolved dependencies.

Please note that the available components are different for each platform, and the prompts that follow may vary.

The installer saves the specified configuration values in the following file:

'/opt/edb/.rar_options_*xxxxx*'

After continued processing, the Advanced Server installation is complete.

Reference - Command Line Options

You can optionally include the following parameters for an Advanced Server installation on the command line, or in a configuration file when invoking the Advanced Server installer.

--create_samples { yes | no }

Use the --create_samples option to specify whether the installer should create the sample tables and procedures for the database dialect specified with the --databasemode parameter. The default is yes.

--databasemode { oracle | postgresql }

Use the --databasemode parameter to specify a database dialect. The default is oracle.

--datadir *data_directory*

Use the --datadir parameter to specify a location for the cluster's data directory. *data_directory* is the name of the directory; include the complete path to the desired directory.

--debuglevel { 0 | 1 | 2 | 3 | 4 }

Use the --debuglevel parameter to set the level of detail written to the *debug_log* file (see --debugtrace). Higher values produce more detail in a longer trace file. The default is 2.

--debugtrace *debug_log*

Use the --debugtrace parameter to troubleshoot installation problems. *debug_log* is the name of the file that contains installation troubleshooting details.

--disable-components component_list

Use the --disable-components parameter to specify a list of Advanced Server components to exclude from the installation. By default, *component_list* contains '' (the empty string). *component_list* is a comma-separated list containing one or more of the following components:

dbserver

> EDB Postgres Advanced Server 10.

pgadmin4 (Linux and Windows only.)

> The EDB Postgres pgAdmin 4 provides a powerful graphical interface for database management and

monitoring.

--enable_acledit { 1 | 0 }

The --enable_acledit 1 option instructs the installer to grant permission to the user specified by the --serviceaccount option to access the Advanced Server binaries and data directory. By default, this option is disabled if --enable_acledit 0 is specified or if the --enable_acledit option is completely omitted. **Note:** Specification of this option is valid only when installing on Windows. This option cannot be specified when installing on Linux. The --enable_acledit 1 option particularly should be specified when a *discretionary access control list* (DACL) needs to be set for allowing access to objects on a Windows host on which Advanced Server is to be installed. See the following for information on a DACL:

https://msdn.microsoft.com/en-us/library/windows/desktop/aa446597(v=vs.85).aspx

In order to perform future operations such as upgrading Advanced Server, access to the data directory must exist for the service account user specified by the --serviceaccount option. By specifying the --enable_acledit 1 option, access to the data directory by the service account user is provided.

--enable-components *component_list*

Although this option is listed when you run the installer with the --help option, the --enable-components parameter has absolutely no effect on which components are installed. All components will be installed regardless of what is specified in *component_list*. In order to install only specific, selected components, you must use the --disable-components parameter previously described in this section to list the components you do not want to install.

--extract-only { yes | no }

Include the --extract-only parameter to indicate that the installer should extract the Advanced Server binaries without performing a complete installation. Superuser privileges are not required for the --extract-only option. The default value is no.

--help

Include the --help parameter to view a list of the optional parameters.

--installer-language { en | ja | zh_CN | zh_TW | ko }

Use the --installer-language parameter to specify an installation language for Advanced Server. The default is en.

en specifies English.

ja specifies Japanese

zh_CN specifies Chinese Simplified.

zh_TW specifies Traditional Chinese.

ko specifies Korean.

--install_runtimes { yes | no } (Windows only.)

Include --install_runtimes to specify whether the installer should install the Microsoft Visual C++ runtime libraries. Default is yes.

--locale *locale*

Specifies the locale for the Advanced Server cluster. By default, the installer will use to the locale detected by initdb.

--mode {qt | gtk | xwindow | text | unattended}

Use the --mode parameter to specify an installation mode. The following modes are supported:

qt - Specify qt to tell the installer to use the Qt graphical toolkit

gtk - Specify gtk to tell the installer to use the GTK graphical toolkit.

xwindow - Specify xwindow to tell the installer to use the X Window graphical toolkit.

text - Specify text to perform a text mode installation in a console window. This is a Linux-only option.

unattended - Specify unattended to specify that the installer should perform an installation that requires no user input during the installation process.

--optionfile *config_file*

Use the --optionfile parameter to specify the name of a file that contains the installation configuration parameters. *config_file* must specify the complete path to the configuration parameter file.

--prefix *installation_dir*/as9.*x*

Use the --prefix parameter to specify an installation directory for Advanced Server. The installer will append a version-specific sub-directory (i.e. as10) to the specified directory. By default, on a Linux system, Advanced Server is installed in:

/opt/edb/as10

The default installation directory on a Windows system is:

C:\Program Files\edb\as10

--productkey *product_key*

Use the --productkey parameter to specify a value for the product key.

The --productkey parameter is only required when the specified system locale is Japanese, Chinese or Korean.

--serverport *port_number*

Use the --serverport parameter to specify a listener port number for Advanced Server.

If you are installing Advanced Server in unattended mode, and do not specify a value using the --serverport parameter, the installer will use port 5444, or the first available port after port 5444 as the default listener port.

--server_utilization {33 | 66 | 100}

Use the --server_utilization parameter to specify a value for the edb_dynatune configuration parameter. The edb_dynatune configuration parameter determines how Advanced Server allocates system resources.

- A value of 33 is appropriate for a system used for development. A low value dedicates the least amount of the host machine’s resources to the database server.
- A value of 66 is appropriate for an application server with a fixed number of applications. A mid-range value dedicates a moderate amount of system resources to the database server. The default value is 66.
- A value of 100 is appropriate for a host machine that is dedicated to running Advanced Server. A high value dedicates most of the system resources to the database server.

When the installation is complete, you can adjust the value of edb_dynatune by editing the postgresql.conf file, located in the data directory of your Advanced Server installation. After editing the postgresql.conf file, you must restart the server for the changes to take effect.

--serviceaccount *user_account_name*

Use the --serviceaccount parameter to specify the name of the user account that owns the server process.

- If --databasemode is set to oracle (the default), the default value of --serviceaccount is enterprisedb.
- If --databasemode is postgresql, the default value of --serviceaccount is set to postgres.

Please note that for security reasons, the --serviceaccount parameter must specify the name of an account that does not hold administrator privileges.

If you specify both the --serviceaccount option and the --enable_acledit 1 option when invoking the installer, the database service and pgAgent will use the same service account, thereby having the required permissions to access the Advanced Server binaries and data directory. **Note:** For installing on Windows hosts, see the --enable_acledit option in this section for additional information relevant to a Windows environment. **Note:** Specification of the --enable_acledit option is permitted only when installing on Windows. The --enable_acledit option cannot be specified when installing on Linux.

Please note that on Windows hosts, if you do not include the --serviceaccount option when invoking the installer, the NetworkService account will own the database service, and the pgAgent service will be owned by either enterprisedb or postgres (depending on the installation mode).

--servicename *service_name*

Use the --servicename parameter to specify the name of the Advanced Server service. The default is edb-as-10.

--servicepassword *user_password* (Windows only)

Use --servicepassword to specify the OS system password. If unspecified, the value of --servicepassword defaults to the value of --superpassword.

--superaccount *super_user_name*

Use the --superaccount parameter to specify the user name of the database superuser.

- If --databasemode is set to oracle (the default), the default value of --superaccount is enterprisedb.
- If --databasemode is set to postgresql, the default value of --superaccount is set to postgres

--superpassword *superuser_password*

Use --superpassword to specify the database superuser password. If you are installing in non-interactive mode, --superpassword defaults to enterprisedb.

--unattendedmodeui { none | minimal | minimalWithDialogs }

Use the --unattendedmodeui parameter to specify installer behavior during an unattended installation.

Include --unattendedmodeui none to specify that the installer should not display progress bars during the Advanced Server installation.

Include --unattendedmodeui minimal to specify that the installer should display progress bars during the installation process. This is the default behavior.

Include --unattendedmodeui minimalWithDialogs to specify that the installer should display progress bars and report any errors encountered during the installation process (in additional dialogs).

--version

Include the --version parameter to retrieve version information about the installer:

EDB Postgres Advanced Server 10 --- Built on 2017-06-15 00:04:00 IB: 15.10.1-201511121057

--webusername *{registered_username}*

You must specify the name of a registered user and password when performing an installation of EDB Postgres Advanced Server 10. Use the --webusername parameter to specify the name of the registered EnterpriseDB user that is performing the installation.

registered_username must be an email address.

If you do not have a *registered user name*, visit the EnterpriseDB website at:

http://www.enterprisedb.com/user-login-registration

--webpassword *{associated_password}*

Use the --webpassword parameter to specify the password associated with the registered EnterpriseDB user that is performing the installation.

--workload_profile {oltp | mixed | reporting}

Use the --workload_profile parameter to specify an initial value for the edb_dynatune_profile configuration parameter. edb_dynatune_profile controls aspects of performance-tuning based on the type of work that the server performs.

- Specify oltp if the Advanced Server installation will be used to support heavy online transaction processing workloads.
- The default value is oltp.
- Specify mixed if Advanced Server will provide a mix of transaction processing and data reporting.
- Specify reporting if Advanced Server will be used for heavy data reporting.

After the installation is complete, you can adjust the value of edb_dynatune_profile by editing the postgresql.conf file, located in the data directory of your Advanced Server installation. After editing the postgresql.conf file, you must restart the server for the changes to take effect.

For more information about edb_dynatune and other performance-related topics, see the *EDB Postgres Advanced Server Guide* available at:

http://www.enterprisedb.com/products-services-training/products/documentation/enterpriseedition

--xlogdir *directory_name* (Linux only.)

Use the --xlogdir parameter to specify a location for the write-ahead log. The default value is /opt/edb/as10/data/pg_wal.

Using StackBuilder Plus

The StackBuilder Plus utility provides a graphical interface that simplifies the process of updating, downloading and installing modules that complement your Advanced Server installation. When you install a module with StackBuilder Plus, StackBuilder Plus automatically resolves any software dependencies.

Please note: If your installation resides on a Linux system, you must install the redhat-lsb package before invoking StackBuilder Plus. For more information, see Section 4.1.

You can invoke StackBuilder Plus at any time after the installation has completed by selecting the StackBuilder Plus menu option from the EDB Postgres Advanced Server 10 menu (Linux) or from the Apps menu (Windows). Enter your system password (if prompted), and the StackBuilder Plus welcome window opens (shown in Figure 4.41).

Figure 4.41 -The StackBuilder Plus welcome window.

Use the drop-down listbox on the welcome window to select your Advanced Server installation.

StackBuilder Plus requires Internet access; if your installation of Advanced Server resides behind a firewall (with restricted Internet access), StackBuilder Plus can download program installers through a proxy server. The module provider determines if the module can be accessed through an HTTP proxy or an FTP proxy; currently, all updates are transferred via an HTTP proxy and the FTP proxy information is not used.

If the selected Advanced Server installation has restricted Internet access, use the Proxy Servers button on the Welcome window to open the Proxy servers dialog (shown in Figure 4.42).

Figure 4.42 –The Proxy servers dialog.

Enter the IP address and port number of the proxy server in the HTTP proxy on the Proxy servers dialog. Currently, all StackBuilder Plus modules are distributed via HTTP proxy (FTP proxy information is ignored). Click OK to continue.

Figure 4.43 –The StackBuilder Plus module selection window.

The tree control on the StackBuilder Plus module selection window (shown in Figure 4.43) displays a node for each module category.

To add a new component to the selected Advanced Server installation or to upgrade a component, check the box to the left of the module name and click Next. A window opens, requesting your EnterpriseDB registration information (as shown in Figure 4.44).

Figure 4.44 –The User Authenication window.

Before downloading and installing modules and updates with StackBuilder Plus, you must enter the user information associated with your EnterpriseDB account. If you do not have an EnterpriseDB user account, click the link provided to open a web browser, and enter your user information.

Enter the email address of a registered account in the Email field, and the corresponding password in the Password field, and click Next to continue. The next dialog confirms the packages selected (Figure 4.45).

Figure 4.45 -A summary window displays a list of selected packages.

By default, the selected package installers are downloaded to:

On Windows:

C:\Users\Administrator

On Linux:

```
/root
```

You can change the directory; use the button (...) to the right of the Download directory field to open a file selector, and choose an alternate location to store the downloaded installers. Click Next to connect to the server and download the required installation files.

When the download completes, a window opens that confirms the installation files have been downloaded and are ready for installation (see Figure 4.46).

Figure 4.46 -Confirmation that the download process is complete.

You can check the box next to Skip Installation, and select Next to exit StackBuilder Plus without installing the downloaded files, or leave the box unchecked and click Next to start the installation process.

Each downloaded installer has different requirements. As the installers execute, they may prompt you to confirm acceptance of license agreements, to enter passwords, and enter configuration information.

During the installation process, you may be prompted by one (or more) of the installers to restart your system. Select No or Restart Later until all installations are completed. When the last installation has completed, reboot the

system to apply all of the updates.

You may occasionally encounter packages that don't install successfully. If a package fails to install, StackBuilder Plus will alert you to the installation error with a popup dialog, and write a message to the log file at:

> On Windows: %TEMP%
>
> On Linux: /root

Figure 4.47 -StackBuilder Plus confirms the completed installation.

When the installation is complete, StackBuilder Plus will alert you to the success or failure of the installations of the requested packages (see Figure 4.47). If you were prompted by an installer to restart your computer, reboot now.

The following table lists some of the modules supported by StackBuilder Plus. Please note that the list is subject to change and varies by platform.

Category and Module Name	Description
Add-ons, tools and utilities	
EnterpriseDB Migration Toolkit	Migration Toolkit is a command line tool that facilitates migration from Oracle databases into Advanced Server
Infinite Cache	Infinite Cache (for Linux only) allows you to utilize memory on other computers connected to your network to increase the amount of memory in the shared buffer cache.
PgBouncer	Connection pooler for Postgres Server, packaged by EnterpriseDB.
StackBuilder Plus	An advanced application stack builder that provides an easy interface for downloading and installing Advanced Server updates and modules.
pgAdmin 4	A full-featured graphical client that can manage multiple databases.
pgAgent	pgAgent is a job scheduling agent for Postgres, capable of running multi-step batch/shell and SQL tasks on complex schedules
pgPool-II	pgPool-II provides load balancing, connection pooling, high availability, and connection limits for Advanced Server databases.
Database Drivers	
EnterpriseDB Connectors	A collection of drivers. Includes .NET, ODBC, JDBC and libpq drivers for Advanced Server
Database Server	
Advanced Server	The EDB Postgres Advanced Server database server.
EnterpriseDB Tools	
Postgres Enterprise Manager Agent	The PEM Agent is responsible for executing tasks and reporting statistics from the host and monitored Postgres instances to the PEM Server.
Postgres Enterprise Manager Client	The PEM Client is a full-featured graphical interface that allows you to schedule tasks and report statistics for the host and monitored instances.
Postgres Enterprise Manager Server	The PEM Server is used as the data repository for monitoring data and as a server to which the agents and client connect.
Replication Server	Replication Server is an asynchronous, master-to-standby replication system enabling replication of tables from an Oracle or SQL Server database to an Advanced Server database.

Category and Module Name	Description
Replication Solutions	
Slony Replication	Slony is a master to multiple standbys replication system that supports cascading and failover. Packaged by EnterpriseDB.
Spatial Extensions	
PostGIS	PostGIS enables Advanced Server to store spatial data for geographic information systems (GIS).
Web Development	
ApachePHP	A distribution of the Apache webserver and PHP, preconfigured for use with Advanced Server. Packaged by EnterpriseDB.
PEM-HTTPD	A pre-configured Apache webserver for use with PostgreSQL. Packaged by EnterpriseDB.

1. ***Using the Update Monitor***

The Update Monitor utility polls the EnterpriseDB website and alerts you to security updates and enhancements as they become available for Advanced Server 10. Update Monitor is automatically installed and invoked with Advanced Server.

When Update Monitor is actively monitoring, the Postgres elephant icon is displayed in the system tray (see Figure 4.48).

Figure 4.48 -The Update Monitor icon.

If you have installed more than one version of Advanced Server, Update Monitor watches for updates and alerts for all installed versions. When Update Monitor finds an update or alert, it displays an alert symbol to let you know that an update or alert is available for one of the Advanced Server installations (see Figure 4.49).

Figure 4.49 -The Update Monitor icon displays an alert.

Right click on the symbol to open the context menu (shown in Figure 4.50).

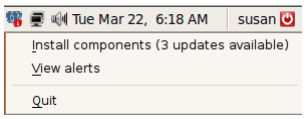

Figure 4.50 -The Update Monitor context menu.

If updates are available for your Advanced Server installation, the update count is displayed after the View alerts menu item. Click Install components to start the installation process.

A system dialog opens, prompting you to enter your password (Figure 4.51).

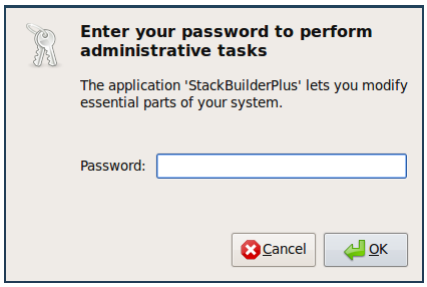

Figure 4.51 - Enter a superuser password.

Enter a superuser password, and click OK to continue. StackBuilder Plus opens (shown in Figure 4.52).

Figure 4.52 - The StackBuilder Plus welcome window.

The StackBuilder Plus wizard walks you through installing the latest versions of the Advanced Server component software; see Section 4.5, *Using StackBuilder Plus* for more information about the update process.

When the update is complete and there are no new updates available, the Update Monitor icon returns to a non-alerted state.

Update Monitor also monitors the EnterpriseDB website for alerts. If an alert is available for your Advanced Server installation, the Update Monitor icon displays an alert symbol. Right-click on the icon to access the context menu, where the alert count is displayed next to the View alerts menu item. Choose the View alerts option to display the EnterpriseDB Advanced Server Alerts window (see Figure 4.53).

Figure 4.53 - An EnterpriseDB Technical alert.

- The EnterpriseDB Advanced Server Alerts window displays helpful hyperlinks that can direct you to more information relevant to the alert.
- Use the Run StackBuilder Plus button to open StackBuilder Plus from the alert to run applicable updates for your current Advanced Server installation.

1. ***Installation Troubleshooting***

Difficulty Displaying Java-based Applications

If you encounter difficulty displaying Java-based server features (controls or text not being displayed correctly, or blank windows), upgrading to the latest libxcb-xlib libraries should correct the problem on most Linux distributions. Please visit the following link for other possible work-arounds:

http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=6532373

--mode unattended Authentication Errors

Authentication errors from component modules during unattended installations may indicate that the specified values of --servicepassword or --superpassword may be incorrect.

Errors During an Advanced Server Installation on Windows

If you encounter an error during the installation process on a Windows system, exit the installation, and ensure that your version of Windows is up-to-date. After applying any outstanding operating system updates, re-invoke the Advanced Server installer.

Applications Fail to Launch During an Advanced Server Installation on Linux

If applications fail to launch (such as StackBuilder Plus or your web browser) during the installation process on a Linux system, verify that the xdg-open program is on your system. If xdg-open is missing, install the xdg-utils package.

If you are using the GNOME desktop, load the root profile before running the Advanced Server installation script. To load the root profile, issue the command, su - root instead of su root before installing Advanced Server.

Configuration File Editors Close Spontaneously

If you are using a Linux system with the gnome console, a bug in the gnome shell may cause configuration file editors accessed via the Expert Configuration menu (under the Advanced Server Application menu) to close

spontaneously. To correct this error, open a terminal window and enter:

dconf write /org/gnome/settings-daemon/plugins/cursor/active false

Please note that each time you reboot your system, you must invoke the command, resetting the value.

The Installation Fails to Complete Due to Existing data Directory Contents

If an installation fails to complete due to existing content in the data directory, the server will write an error message to the server logs:

A data directory is neither empty, or a recognisable data directory.

If you encounter a similar message, you should confirm that the data directory is empty; the presence of files (including the system-generated lost+found folder) will prevent the installation from completing. Either remove the files from the data directory, or specify a different location for the data directory before re-invoking the installer to complete the installation.

1.3 Managing an Advanced Server Installation

Unless otherwise noted, the commands and paths noted in the following section assume that you have performed an installation with the interactive installer.

Starting and Stopping Advanced Server and Supporting Components

A service is a program that runs in the background and requires no user interaction (in fact, a service provides no user interface); a service can be configured to start at boot time, or manually on demand. Services are best controlled using the platform-specific operating system service control utility. Many of the Advanced Server supporting components are services.

The following table lists the names of the services that control Advanced Server and services that control Advanced Server supporting components:

Advanced Server Component Name	Linux Service Name	Windows Service Name
Advanced Server	edb-as-10	edb-as-10
Infinite Cache	edb-icache	N/A
pgAgent	edb-pgagent-10	EDB Postgres Advanced Server 10 Scheduling Agent
PgBouncer	edb-pgbouncer-1.7	edb-pgbouncer-1.7
pgPool-II	edb-pgpool-3.5	N/A
Slony	edb-slony-replication-10	edb-slony-replication-10

Advanced Server's database server, and the services of Advanced Server's supporting components can be controlled at the command line or through operating system-specific graphical interfaces.

Controlling a Service on Linux

The commands that control the Advanced Server service on a Linux platform are version specific.

Controlling a Service on CentOS or RHEL 7.x

If your installation of Advanced Server resides on version 7.x of RHEL and CentOS, you must use the systemctl command to control the Advanced Server service and supporting components.

The systemctl command must be in your search path and must be invoked with superuser privileges. To use the command, open a command line, and enter:

systemctl *action service_name*

Where:

action

action specifies the action taken by the service command. Specify:

- start to start the service.
- stop to stop the service.
- restart to stop and then start the service.
- status to discover the current status of the service.

service_name

service_name specifies the name of the service.

Controlling a Service on CentOS or RHEL 6.x

On version 6.x of RHEL or CentOS Linux, you can control a service at the command line with the service command. The service command can be used to manage an Advanced Server cluster, as well as the services of component software installed with Advanced Server.

Using the service command to change the status of a service allows the Linux service controller to keep track of the server status (the pg_ctl command does not alert the service controller to changes in the status of a server). The command must be in your search path and must be invoked with superuser privileges. Open a command line, and issue the command:

service *service_name action*

The Linux service command invokes a script (with the same name as the service) that resides in /etc/init.d. If your Linux distribution does not support the service command, you can call the script directly by entering:

/etc/init.d/*service_name action*

Where:

service_name

service_name specifies the name of the service.

action

action specifies the action taken by the service command. Specify:

- start to start the service.
- stop to stop the service.
- condstop to stop the service without displaying a notice if the server is already stopped.

- restart to stop and then start the service.
- condrestart to restart the service without displaying a notice if the server is already stopped.

- try-restart to restart the service without displaying a notice if the server is already stopped.
- status to discover the current status of the service.

Using pg_ctl to Control Advanced Server

You can use the pg_ctl utility to control an Advanced Server service from the command line on any platform. pg_ctl allows you to start, stop, or restart the Advanced Server database server, reload the configuration parameters, or display the status of a running server. To invoke the utility, assume the identity of the cluster owner, navigate into the home directory of Advanced Server, and issue the command:

./bin/pg_ctl -D *data_directory action*

data_directory

data_directory is the location of the data controlled by the Advanced Server cluster.

action

action specifies the action taken by the pg_ctl utility. Specify:

- start to start the service.
- stop to stop the service.
- restart to stop and then start the service.
- reload sends the server a SIGHUP signal, reloading configuration parameters
- status to discover the current status of the service.

For more information about using the pg_ctl utility, or the command line options available, please see the official PostgreSQL Core Documentation available at:

https://www.postgresql.org/docs/10/static/app-pg-ctl.html

Choosing Between pg_ctl and the service Command

You can use the pg_ctl utility to manage the status of an Advanced Server cluster, but it is important to note that pg_ctl does not alert the operating system service controller to changes in the status of a server, so it is beneficial to use the service command whenever possible.

Note that when you invoke the installer with the --extract-only option, the installer does not create a service, it merely unpacks the server. If you have installed Advanced Server by invoking the installer with the --extract-only option, you must use the pg_ctl command to control the server.

Using the edbstart and edbstop Utilities

Note: edbstart and edbstop functionality is supported only on Linux hosts that are running Advanced Server installations performed with the Interactive installer. RPM installations do not support edbstart and edbstop.

While the autostart scripts created during an Advanced Server installation control a single database cluster, the edbstart and edbstop utilities can control multiple database clusters on the same host, with a single configuration file.

The edbstart and edbstop utilities use a file named edbtab (described below) to determine which instances of Advanced Server should start when the operating system boots, and stop when the host is shut down.

Before using the edbstart or edbstop utilities, you should disable the Advanced Server autostart scripts. The commands that disable the scripts are platform specific;

on Fedora/Redhat:

chkconfig --level 2345 edb-as-10 off

on Debian/Ubuntu:

update-rc.d edb-as-10 disable

After stopping the Advanced Server service, use an editor to create a file named edbtab in the /etc directory; you can copy the sample file located in:

/opt/edb/as10/scripts/server/autostart

Edit the edbtab file, specifying which Advanced Server clusters that the edbstart and edbstop programs will control, and indicating if the cluster should be automatically started and stopped.

Each edbtab file entry should take the form:

edb_home directory:*edb_data_directory*:N|Y

edb_home_directory

edb_home_directory specifies the home directory of the Advanced Server installation that the edbstart/edbstop utilities will control.

edb_data_directory

edb_data_directory specifies the data directory of the database cluster that the edbstart/edbstop utilities will control. *edb_data_directory* is the same as the value of $PGDATA for a specified cluster.

N|Y

Y specifies that edbstart and edbstop should control the service; N specifies that the user will control the service manually.

Include a separate entry in the edbtab file for each Advanced Server cluster that you wish to control with the edbstart and edbstop utilities.

After editing the edbtab file, copy the edb_autostart script to /etc/init.d. By default, the edb_autostart script is located in:

/opt/edb/as10/scripts/server/autostart

Copy the edbstart and edbstop scripts to $EDBHOME. Make the scripts executable with the following command:

chmod +x edbstart

chmod +x edbstop

chmod +x edbstart edbstop /etc/init.d/edb_autostart

Enable the edb_autostart service with the commands:

chkconfig --level 2345 edb_autostart on

chkconfig --add edb_autostart

For the service to take effect, you must restart your system.

Manually Controlling the Server with edbstart and edbstop

You can use edbstart and edbstop at the command line to manually control all of the clusters specified in the edbtab file, or to control an individual cluster. Call edbstart without an argument to start all of the clusters listed

within the edbtab file; invoke edbstop without an argument to stop all of the clusters listed in the edbtab file. You can control an individual cluster by specifying the cluster's data directory as an argument. The following command starts a cluster:

edbstart /opt/edb/as10/data

While the following command stops a cluster:

edbstop /opt/edb/as10/data

Configuring Component Services to AutoStart at System Reboot

After installing, configuring and starting the services of Advanced Server supporting components on a Linux system, you must manually configure your system to autostart the service when your system reboots. To configure a service to autostart on a Linux system, open a command line, assume superuser privileges, and enter the following command.

On a Redhat-compatible Linux system:

/sbin/chkconfig *service_name* on

On a Debian-compatible Linux system, use the command:

/usr/sbin/update-rc.d *service_name* enable

Where *service_name* specifies the name of the service.

Please note that if you are using a Windows system, the Slony service will be configured to autostart by default. On Windows, you can use the Service Properties dialog to control the service startup type. For more information about controlling a service on Windows, see Section 5.3.

Controlling a Service on Windows

The Windows operating system includes a graphical service controller that offers graphical control of Advanced Server and the services associated with Advanced Server components. The Windows Services utility can be accessed through the Administrative Tools section of the Control Panel, or by navigating through the Apps menu to Run; when the Run dialog opens, enter services.msc and click OK.

Figure 5.1 - The Advanced Server service in the Windows Services window.

When the Services window opens, use the scroll bar to move through the listed services to highlight edb-as-10 (see Figure 5.1):

- Use the Stop the service option to stop the instance of Advanced Server. Please note that any user (or client application) connected to the Advanced Server instance will be abruptly disconnected if you stop the service.
- Use the Start the service option to start the Advanced Server service.
- Use the Pause the service option to tell Advanced Server to reload the server configuration parameters without disrupting user sessions for many of the configuration parameters. See Section 6, *Configuring Advanced Server* for more information about the parameters that can be updated with a server reload.

  Please Note: A limitation in Windows may cause Advanced Server to generate an error message after performing a parameter reload. To confirm that the reload command has successfully updated the parameters, query the pg_settings table to verify that the change has taken effect.

- Use the Restart the service option to stop and then start the Advanced Server. Please note that any user sessions will be terminated when you stop the service. This option is useful to reset server parameters that only take effect on server start.

Controlling Server Startup Behavior on Windows

You can use the Windows Services utility to control the startup behavior of the server. To alter the startup properties of a server, navigate through the Control Panel to the Services window, or navigate through the Apps menu to Run; when the Run dialog opens, enter services.msc and click OK.

Right click on the name of the service you wish to change and select Properties from the context menu to open the Properties dialog.

Use the drop-down listbox in the Startup type field (shown in Figure 5.2) to specify how the Advanced Server service will behave when the host starts.

Figure 5.2 - Specifying Advanced Server's startup behavior.

- Specify Automatic (Delayed Start) to instruct the service controller to start after boot.
- Specify Automatic to instruct the service controller to start and stop the server whenever the system starts or stops.
- Specify Manual to instruct the service controller that the server must be started manually.
- Specify Disabled to instruct the service controller to disable the service; after disabling the service, you must stop the service or restart the server to make the change take effect. Once disabled, the server's status cannot be changed until Startup type is reset to Automatic (Delayed Start), Automatic or Manual.

1.4 Configuring Advanced Server

Unless otherwise noted, the commands and paths noted in the following section assume that you have performed an installation with the interactive installer.

You can easily update parameters that determine the behavior of Advanced Server and supporting components by modifying the following configuration files:

- The postgresql.conf file determines the initial values of Advanced Server configuration parameters.
- The pg_hba.conf file specifies your preferences for network authentication and authorization.
- The pg_ident.conf file maps operating system identities (user names) to Advanced Server identities (roles) when using ident-based authentication.

You can use your editor of choice to open a configuration file, or navigate through a menu to open the file:

- On a Windows system, a link to each configuration file is available on the Apps menu.
- To update configuration files in Linux, navigate through the EDB Postgres menu selection on the Applications menu to the Advanced Server 10 menu; use the Expert Configuration menu to select the configuration file that you would like to edit (see Figure 6.1).

Figure 6.1 -Accessing the configuration files through the Applications menu.

Modifying the postgresql.conf File

Configuration parameters in the postgresql.conf file specify server behavior with regards to auditing, authentication, encryption, and other behaviors. The postgresql.conf file resides in the data directory under your Advanced Server installation.

Figure 6.2 - The postgresql.conf file.

Parameters that are preceded by a pound sign (#) are set to their default value (as shown in the parameter setting). 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.

Within the postgresql.conf file, some parameters contain comments that indicate change requires restart (see Figure 6.2). To view a list of the parameters that require a server restart, execute the following query at the EDB-PSQL command line (see Figure 6.3):

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


Figure 6.3 - Configuration parameters that require a server restart.

If you are changing a parameter that requires a server restart, see Section 5.1, *Starting and Stopping Advanced Server* for information about restarting Advanced Server.

On a Linux system, you can reload the system configuration parameter values by navigating through the EDB Postgres menu to the Advanced Server 10 menu; then, navigate through the Expert Configuration menu, selecting Reload Configuration. Reloading the configuration parameters does not require Advanced Server users to log out of their current Advanced Server sessions.

On a Windows system, you will find the Reload Configuration menu selection on the Apps menu.

Modifying the pg_hba.conf File

Entries in the pg_hba.conf file specify the authentication method or methods that the server will use when authenticating connecting clients. Before connecting to the server, you may be required to modify the authentication properties specified in the pg_hba.conf file.

When you invoke the initdb utility to create a cluster, initdb creates a pg_hba.conf file for that cluster that specifies the type of authentication required from connecting clients.

The default authentication configuration specified in the pg_hba.conf file is:

```
# TYPE DATABASE USER ADDRESS METHOD

# "local" is for Unix domain socket connections only

local all all md5

# IPv4 local connections:

host all all 127.0.0.1/32 md5

# IPv6 local connections:

host all all ::1/128 md5
```

Allow replication connections from localhost, by a user with the

replication privilege.

#local replication enterprisedb md5

#host replication enterprisedb 127.0.0.1/32 md5

#host replication enterprisedb ::1/128 md5

Appropriate authetication methods provide protection and security. Please consult the PostgreSQL documentation for details about authentication options:

https://www.postgresql.org/docs/10/static/auth-methods.html

To modify the pg_hba.conf file, open the file with your choice of editor. After modifying the authentication settings in the pg_hba.conf file, use the services utility (Windows), or use the following command to restart the server and apply the changes:

On Linux 6.x:

service edb-as-10 restart

On Linux 7.x:

systemctl restart edb-as-10

For more information about modifying the pg_hba.conf file, see the PostgreSQL Core Documentation at:

https://www.postgresql.org/docs/10/static/auth-pg-hba-conf.html

Setting Advanced Server Environment Variables

The graphical installers provide a script that simplifies the task of setting environment variables, allowing a user to more easily invoke client applications at the command line. The script sets the environment variables for your current shell session; when your shell session ends, the environment variables are destroyed. You may wish to invoke pgplus_env or pg_env from your system-wide shell startup script, so that environment variables are automatically defined for each shell session.

The pgplus_env script is created during the Advanced Server installation process and reflects the choices made during installation. To invoke the script, open a command line and enter:

On Linux:

source /opt/edb/as10/pgplus_env.sh

On Windows:

C:\Program Files\edb\10AS\pgplus_env.bat

As the pgplus_env.sh script executes (on Linux), it sets the following environment variables:

```
export PATH=/opt/edb/as10/bin:$PATH

export EDBHOME=/opt/edb/as10

export PGDATA=/opt/edb/as10/data

export PGDATABASE=edb
```

```
# export PGUSER=enterprisedb
  export PGPORT=5444
  export PGLOCALEDIR=/opt/edb/as10/share/locale
```

As the pgplus_env.bat script executes (on Windows), it sets the following environment variables:

```
PATH="C:\Program Files\edb\as10\bin";%PATH%
EDBHOME=C:\Program Files\edb\as10
PGDATA=C:\Program Files\edb\as10\data
PGDATABASE=edb
REM @SET PGUSER=enterprisedb
PGPORT=5444
PGLOCALEDIR=C:\Program Files\edb\as10\share\locale
```

If you have used an installer created by EnterpriseDB to install PostgreSQL, the pg_env script performs the same function. To invoke the pg_env script, open a command line, and enter:

On Linux:

```
source /opt/PostgreSQL/10/pg_env.sh
```

On Windows:

```
C:\Progra~1\PostgreSQL\10\pg_env.bat
```

As the pg_env.sh script executes (on Linux), it sets the following environment variables:

```
PATH=/home/opt/PostgreSQL/10/bin:$PATH
PGDATA=/home/opt/PostgreSQL/10/data
PGDATABASE=postgres
PGUSER=postgres
PGPORT=5432
PGLOCALEDIR=/home/opt/PostgreSQL/10/share/locale
MANPATH=$MANPATH:/home/opt/PostgreSQL/10/share/man
```

As the pg_env.bat script executes (on Windows), it sets the following environment variables:

```
PATH="C:\Program Files\PostgreSQL\10\bin";%PATH%
PGDATA=C:\Program Files\PostgreSQL\10\data
PGDATABASE=postgres
PGUSER=postgres
PGPORT=5432
PGLOCALEDIR=C:\Program Files\PostgreSQL\10\share\locale
```

Connecting to Advanced Server with psql

psql is a command line client application that allows you to execute SQL commands and view the results. To open the edb-psql client, the client must be in your search path. The executable resides in the bin directory, under your Advanced Server installation.

On Linux:

/opt/edb/as10/bin/psql

On Windows:

C:\Program Files\edb\as10\bin\psql

Use the following command and command options to start the psql client:

psql -d edb -U enterprisedb

Figure 6.4 - Connecting to the server.

Where:

-d specifies the database to which psql will connect;

-U specifies the identity of the database user that will be used for the session.

If you have performed an installation with the interactive installer, you can access the psql client through the Applications or Start menu. Navigate through the EDB Postgres menu to the Advanced Server 10 menu; then, navigate through the Run SQL Command Line menu, selecting EDB-PSQL. When the Terminal window opens, provide connection information for your session.

For more information about using the command line client, please refer to the PostgreSQL Core Documentation at:

https://www.postgresql.org/docs/10/static/app-psql.html

Connecting to Advanced Server with the pgAdmin 4 Client

pgAdmin 4 provides an interactive graphical interface that you can use to manage your database and database objects. Easy-to-use dialogs and online help simplify tasks such as object creation, role management, and granting or revoking privileges. The tabbed browser panel provides quick access to information about the object currently selected in the pgAdmin tree control.

To open pgAdmin, use the Linux Applications or Windows Start menu to access the EDB Postgres menu; navigate through the Advanced Server 10 menu to select pgAdmin. The client opens as shown in Figure 6.5.

Figure 6.5 – The pgAdmin 4 client.

To connect to the Advanced Server database server, expand the server node of the Browser tree control, and right click on the EDB Postgres Advanced Server node. When the context menu opens, select Connect Server. The Connect to Server dialog opens (see Figure 6.6).

Figure 6.6 – The pgAdmin 4 client.

Provide the password associated with the database superuser in the Password field, and click OK to connect.

![image](./images/image60.png)

Figure 6.7 – The pgAdmin client.

When the client connects (see Figure 6.7), you can use the Browser tree control to retrieve information about existing database objects, or to create new objects. For more information about using the pgAdmin client, use the Help drop-down menu to access the online help files.

1.5 Limitations

- The pg_upgrade utility cannot upgrade a partitioned table if a foreign key refers to the partitioned table.
- If you are upgrading from the version 9.4 server or a lower version of Advanced Server, and you use partitioned tables that include a SUBPARTITION BY clause, you must use pg_dump and pg_restore to upgrade an existing Advanced Server installation to a later version of Advanced Server. To upgrade, you must:

1. Use pg_dump to preserve the content of the subpartitioned table.
2. Drop the table from the Advanced Server 9.4 database or a lower version of Advanced Server database.
3. Use pg_upgrade to upgrade the rest of the Advanced Server database to a more recent version.
4. Use pg_restore to restore the subpartitioned table to the latest upgraded Advanced Server database.

1.6 Upgrading an Installation With pg_upgrade

While minor upgrades between versions are fairly simple, and require only the installation of new executables, past major version upgrades have been both expensive and time consuming. pg_upgrade facilitates migration between any version of Advanced Server (version 9.0 or later), and any subsequent release of Advanced Server that is supported on the same platform.

Without pg_upgrade, to migrate from an earlier version of Advanced Server to Advanced Server 10, you must export all of your data using pg_dump, install the new release, run initdb to create a new cluster, and then import your old data. If you have a significant amount of data, that can take a considerable amount of time and planning. You may also have to use additional storage to temporarily accommodate both the original data and the exported data.

pg_upgrade can reduce both the amount of time required and the disk space required for many major-version upgrades.

The pg_upgrade utility performs an in-place transfer of existing data between Advanced Server and any subsequent version.

Several factors determine if an in-place upgrade is practical:

- The on-disk representation of user-defined tables must not change between the original version and the upgraded version.
- The on-disk representation of data types must not change between the original version and the upgraded version.
- To upgrade between major versions of Advanced Server with pg_upgrade, both versions must share a common binary representation for each data type. Therefore, you cannot use pg_upgrade to migrate from a 32-bit to a 64-bit Linux platform.

Before performing a version upgrade, pg_upgrade will verify that the two clusters (the old cluster and the new cluster) are compatible.

If the upgrade involves a change in the on-disk representation of database objects or data, or involves a change in the binary representation of data types, pg_upgrade will be unable to perform the upgrade; to upgrade, you will have to pg_dump the old data and then import that data into the new cluster.

The pg_upgrade executable is distributed with Advanced Server 10, and is installed as part of the Database Server component; no additional installation or configuration steps are required.

Performing an Upgrade - Overview

To upgrade an earlier version of Advanced Server to the current version, you must:

- Install the current version of Advanced Server. The new installation must contain the same supporting server components as the old installation.
- Empty the target database or create a new target cluster with initdb.
- Place the pg_hba.conf file for both databases in trust authentication mode (to avoid authentication conflicts.
- Shut down the old and new Advanced Server services.
- Invoke the pg_upgrade utility.

When pg_upgrade starts, it performs a compatibility check to ensure that all required executables are present and contain the expected version numbers. The verification process also checks the old and new $PGDATA directories to ensure that the expected files and subdirectories are in place. If the verification process succeeds, pg_upgrade starts the old postmaster and runs pg_dumpall --schema-only to capture the metadata contained in the old cluster. The script produced by pg_dumpall is used in a later step to recreate all user-defined objects in the new cluster.

Note that the script produced by pg_dumpall recreates only user-defined objects and not system-defined objects. The new cluster will *already* contain the system-defined objects created by the latest version of Advanced Server.

After extracting the metadata from the old cluster, pg_upgrade performs the bookkeeping tasks required to sync the new cluster with the existing data.

pg_upgrade runs the pg_dumpall script against the new cluster to create (empty) database objects of the same shape and type as those found in the old cluster. Then, pg_upgrade links or copies each table and index from the old cluster to the new cluster.

Linking versus Copying

When invoking pg_upgrade, you can use a command-line option to specify whether pg_upgrade should *copy* or *link* each table and index in the old cluster to the new cluster.

Linking is much faster because pg_upgrade simply creates a second name (a hard link) for each file in the cluster; linking also requires no extra workspace because pg_upgrade does not make a copy of the original data. When linking the old cluster and the new cluster, the old and new clusters share the data; note that after starting the new cluster, your data can no longer be used with the previous version of Advanced Server.

If you choose to copy data from the old cluster to the new cluster, pg_upgrade will still reduce the amount of time required to perform an upgrade compared to the traditional dump/restore procedure. pg_upgrade uses a file-at-a-time mechanism to copy data files from the old cluster to the new cluster (versus the row-by-row mechanism used by dump/restore). When you use pg_upgrade, you avoid building indexes in the new cluster; each index is simply copied from the old cluster to the new cluster. Finally, using a dump/restore procedure to upgrade requires a great deal of workspace to hold the intermediate text-based dump of all of your data, while pg_upgrade requires very little extra workspace.

Data that is stored in user-defined tablespaces is not copied to the new cluster; it stays in the same location in the file system, but is copied into a subdirectory whose name reflects the version number of the new cluster. To manually relocate files that are stored in a tablespace after upgrading, move the files to the new location and

update the symbolic links (located in the pg_tblspc directory under your cluster's data directory) to point to the files.

Invoking pg_upgrade

When invoking pg_upgrade, you must specify the location of the old and new cluster's PGDATA and executable (/bin) directories, as well as the name of the Advanced Server superuser, and the ports on which the installations are listening. A typical call to invoke pg_upgrade to migrate from Advanced Server 9.6 to Advanced Server 10 takes the form:

> pg_upgrade --old-datadir *path_to_9.6_data_directory* --new-datadir *path_to_10_data_directory* --user *superuser_name* --old-bindir *path_to_9.6_bin_directory* --new-bindir *path_to_10_bin_directory* --old-port *9.6_port* --new-port *10_port*

Where:

--old-datadir *path_to_9.6_data_directory*

> Use the --old-datadir option to specify the complete path to the data directory within the Advanced Server 9.6 installation.

--new-datadir *path_to_10_data_directory*

> Use the --new-datadir option to specify the complete path to the data directory within the Advanced Server 10 installation.

--username *superuser_name*

> Include the --username option to specify the name of the Advanced Server superuser. The superuser name should be the same in both versions of Advanced Server. By default, when Advanced Server is installed in Oracle mode, the superuser is named enterprisedb. If installed in PostgreSQL mode, the superuser is named postgres.
>
> If the Advanced Server superuser name is not the same in both clusters, the clusters will not pass the pg_upgrade consistency check.

--old-bindir *path_to_9.6_bin_directory*

> Use the --old-bindir option to specify the complete path to the bin directory in the Advanced Server 9.6 installation.

--new-bindir *path_to_10_bin_directory*

> Use the --new-bindir option to specify the complete path to the bin directory in the Advanced Server 10 installation.

--old-port *9.6_port*

> Include the --old-port option to specify the port on which Advanced Server 9.6 listens for connections.

--new-port *10_port*

> Include the --new-port option to specify the port on which Advanced Server 10 listens for connections.

Command Line Options - Reference

pg_upgrade accepts the following command line options; each option is available in a long form or a short form:

-b *path_to_old_bin_directory* --old-bindir *path_to_old_bin_directory*

Use the -b or --old-bindir keyword to specify the location of the old cluster's executable directory.

-B *path_to_new_bin_directory* --new-bindir *path_to_new_bin_directory*

Use the -B or --new-bindir keyword to specify the location of the new cluster's executable directory.

-c --check

Include the -c or --check keyword to specify that pg_upgrade should perform a consistency check on the old and new cluster without performing a version upgrade.

-d *path_to_old_data_directory* --old-datadir *path_to_old_data_directory*

Use the -d or --old-datadir keyword to specify the location of the old cluster's data directory.

-D *path_to_new_data_directory* --new-datadir *path_to_new_data_directory*

Use the -D or --new-datadir keyword to specify the location of the new cluster's data directory.

Please note: Data that is stored in user-defined tablespaces is not copied to the new cluster; it stays in the same location in the file system, but is copied into a subdirectory whose name reflects the version number of the new cluster. To manually relocate files that are stored in a tablespace after upgrading, you must move the files to the new location and update the symbolic links (located in the pg_tblspc directory under your cluster's data directory) to point to the files.

-j --jobs

Include the -j or --jobs keyword to specify the number of simultaneous processes or threads to use during the upgrade.

-k --link

Include the -k or --link keyword to create a hard link from the new cluster to the old cluster. See Section 8.1.1, *Linking versus Copying* for more information about using a symbolic link.

-o *options* --old-options *options*

Use the -o or --old-options keyword to specify options that will be passed to the old postgres command. Enclose options in single or double quotes to ensure that they are passed as a group.

-O *options* --new-options *options*

Use the -O or --new-options keyword to specify options to be passed to the new postgres command. Enclose options in single or double quotes to ensure that they are passed as a group.

-p *old_port_number* --old-port *old_port_number*

Include the -p or --old-port keyword to specify the port number of the Advanced Server installation that you are upgrading.

-P *new_port_number* --new-port *new_port_number*

Include the -P or --new-port keyword to specify the port number of the new Advanced Server installation.

Please note: If the original Advanced Server installation is using port number 5444 when you invoke the Advanced Server 10 installer, the installer will recommend using listener port 5445 for the new installation of Advanced Server.

-r --retain

During the upgrade process, pg_upgrade creates four append-only log files; when the upgrade is completed,

> pg_upgrade deletes these files. Include the -r or --retain option to specify that the server should retain the pg_upgrade log files.

-U *user_name* --username *user_name*

> Include the -U or --username keyword to specify the name of the Advanced Server database superuser. The same superuser must exist in both clusters.

-v --verbose

> Include the -v or --verbose keyword to enable verbose output during the upgrade process.

-V --version

> Use the -V or --version keyword to display version information for pg_upgrade.

-? -h --help

> Use -?, -h or --help options to display pg_upgrade help information.

Upgrading to Advanced Server 10 – Step-by-Step

You can use pg_upgrade to upgrade from an existing installation of Advanced Server into the cluster built by the Advanced Server 10 installer or into an alternate cluster created using the initdb command. In this section, we will provide the details of upgrading into the cluster provided by the installer.

The basic steps to perform an upgrade into an empty cluster created with the initdb command are the same as the steps to upgrade into the cluster created by the Advanced Server 10 installer, but you can omit Step 2 (*Empty the edb database*), and substitute the location of the alternate cluster when specifying a target cluster for the upgrade.

If a problem occurs during the upgrade process, you can revert to the previous version. See Section 8.5, *Reverting to the Old Cluster* for detailed information about this process.

You must be an operating system superuser or Windows Administrator to perform an Advanced Server upgrade.

Step 1 - Install the New Server

Install Advanced Server 10, specifying the same non-server components that were installed during the previous Advanced Server installation. Please note that the new cluster and the old cluster must reside in different directories.

Step 2 - Empty the target database

The target cluster must not contain any data; you can create an empty cluster using the initdb command, or you can empty a database that was created during the installation of Advanced Server 10. If you have installed Advanced Server in PostgreSQL mode, the installer creates a single database named postgres; if you have installed Advanced Server in Oracle mode, it creates a database named postgres and a database named edb.

The easiest way to empty the target database is to drop the database and then create a new database. Before invoking the DROP DATABASE command, you must disconnect any users and halt any services that are currently using the database.

On Windows, navigate through the Control Panel to the Services manager; highlight each service in the Services list, and select Stop.

On Linux, open a terminal window, assume superuser privileges, and manually stop each service; for example, if you are on Linux 6.x, invoke the command:

> service edb-pgagent-10 stop

to stop the pgAgent service.

After stopping any services that are currently connected to Advanced Server, you can use the EDB-PSQL command line client to drop and create a database. When the client opens, connect to the template1 database as the database superuser; if prompted, provide authentication information. Then, use the following command to drop your database:

> DROP DATABASE *database_name*;

Where *database_name* is the name of the database.

Then, create an empty database based on the contents of the template1 database (see Figure 8.1):

> CREATE DATABASE *database_name*;

Step 3 - Set both servers in trust mode

During the upgrade process, pg_upgrade will connect to the old and new servers several times; to make the connection process easier, you can edit the pg_hba.conf file, setting the authentication mode to trust. To modify the pg_hba.conf file, navigate through the Start menu to the EDB Postgres menu; to the Advanced Server menu, and open the Expert Configuration menu; select the Edit pg_hba.conf menu option to open the pg_hba.conf file.

You should allow trust authentication for the previous Advanced Server installation, and Advanced Server 10 servers. Edit the pg_hba.conf file for both installations of Advanced Server as shown in Figure 7.1.

Figure 7.1 - Configuring Advanced Server to use trust authentication.

After editing each file, save the file and exit the editor.

If the system is required to maintain md5 authentication mode during the upgrade process, you can specify user passwords for the database superuser in a password file (pgpass.conf on Windows, .pgpass on Linux). For more information about configuring a password file, see the PostgreSQL Core Documentation, available through:

https://www.postgresql.org/docs/10/static/libpq-pgpass.html

Step 4 - Stop All Component Services and Servers

Before you invoke pg_upgrade, you must stop any services that belong to the original Advanced Server installation, Advanced Server 10 or the supporting components. This ensures that a service will not attempt to access either cluster during the upgrade process.

The services that are most likely to be running in your installation are:

Service:	On Linux:	On Windows
Postgres Plus Advanced Server 9.0	ppas-9.0	ppas-9.0
Postgres Plus Advanced Server 9.1	ppas-9.1	ppas-9.1
Postgres Plus Advanced Server 9.2	ppas-9.2	ppas-9.2
Postgres Plus Advanced Server 9.3	ppas-9.3	ppas-9.3
Postgres Plus Advanced Server 9.4	ppas-9.4	ppas-9.4
Postgres Plus Advanced Server 9.5	ppas-9.5	ppas-9.5
EnterpriseDB Postgres Advanced Server 9.6	edb-as-9.6	edb-as-9.6

Service:	On Linux:	On Windows
EnterpriseDB Postgres Advanced Server 10	edb-as-10	edb-as-10
Advanced Server 9.0 Scheduling Agent	ppasAgent-90	Postgres Plus Advanced Server 90 Scheduling Agent
Advanced Server 9.1 Scheduling Agent	ppasAgent-91	Postgres Plus Advanced Server 91 Scheduling Agent
Advanced Server 9.2 Scheduling Agent	ppas-agent-9.2	Postgres Plus Advanced Server 9.2 Scheduling Agent
Advanced Server 9.3 Scheduling Agent	ppas-agent-9.3	Postgres Plus Advanced Server 9.3 Scheduling Agent
Advanced Server 9.4 Scheduling Agent	ppas-agent-9.4	Postgres Plus Advanced Server 9.4 Scheduling Agent
Advanced Server 9.5 Scheduling Agent	ppas-agent-9.5	Postgres Plus Advanced Server 9.5 Scheduling Agent
Advanced Server 9.6 Scheduling Agent (pgAgent)	edb-pgagent-9.6	EnterpriseDB Postgres Advanced Server 9.6 Scheduling Agent
Infinite Cache 9.2	ppas-infinitecache-9.2	N/A
Infinite Cache 9.3	ppas-infinitecache-9.3	N/A
Infinite Cache 9.4	ppas-infinitecache	N/A
Infinite Cache 9.5	ppas-infinitecache	N/A
Infinite Cache 9.6	edb-icache	N/A
Infinite Cache 10	edb-icache	N/A
PgBouncer 9.0	pgbouncer-90	pgbouncer-90
PgBouncer 9.1	pgbouncer-91	pgbouncer-91
PgBouncer 9.2	pgbouncer-9.2	pgbouncer-9.2
PgBouncer 9.3	pgbouncer-9.3	pgbouncer-9.3
PgBouncer	Pgbouncer	pgbouncer
PgBouncer 1.6	ppas-pgbouncer-1.6 or ppas-pgbouncer16	ppas-pgbouncer-1.6
PgBouncer 1.7	edb-pgbouncer-1.7	edb-pgbouncer-1.7
PgPool 9.2	ppas-pgpool-9.2	N/A
PgPool 9.3	ppas-pgpool-9.3	N/A
PgPool	ppas-pgpool	N/A
PgPool 3.4	ppas-pgpool-3.4 or ppas-pgpool34 or	N/A
pgPool-II	edb-pgpool-3.5	N/A
Slony 9.2	ppas-replication-9.2	ppas-replication-9.2
Slony 9.3	ppas-replication-9.3	ppas-replication-9.3
Slony 9.4	ppas-replication-9.4	ppas-replication-9.4
Slony 9.5	ppas-replication-9.5	ppas-replication-9.5
Slony 9.6	edb-slony-replication-9.6	edb-slony-replication-9.6
xDB Publication Server 9.0	edb-xdbpubserver-90	Publication Service 90
xDB Publication Server 9.1	edb-xdbpubserver-91	Publication Service 91
xDB Subscription Server	edb-xdbsubserver-90	Subscription Service 90
xDB Subscription Server	edb-xdbsubserver-91	Subscription Service 91
EDB Replication Server v6.x	edb-xdbpubserver	Publication Service for xDB Replication Server
EDB Subscription Server v6.x	edb-xdbsubserver	Subscription Service for xDB Replication Server

To stop a service on Windows:

Open the Services applet; highlight each Advanced Server or supporting component service displayed in the list, and select Stop.

To stop a service on Linux:

Open a terminal window and manually stop each service at the command line.

Step 5 for Linux only - Assume the identity of the cluster owner

If you are using Linux, assume the identity of the Advanced Server cluster owner. (The following example assumes Advanced Server was installed in the default, compatibility with Oracle database mode, thus assigning enterprisedb as the cluster owner. If installed in compatibility with PostgreSQL database mode, postgres is the cluster owner.)

su - enterprisedb

Enter the Advanced Server cluster owner password if prompted. Then, set the path to include the location of the pg_upgrade executable:

```
export PATH=$PATH:/opt/edb/as10/bin
```

During the upgrade process, pg_upgrade writes a file to the current working directory of the enterprisedb user; you must invoke pg_upgrade from a directory where the enterprisedb user has write privileges. After performing the above commands, navigate to a directory in which the enterprisedb user has sufficient privileges to write a file.

```
cd /tmp
```

Proceed to Step 6.

Step 5 for Windows only - Assume the identity of the cluster owner

If you are using Windows, open a terminal window, assume the identity of the Advanced Server cluster owner and set the path to the pg_upgrade executable.

If the --serviceaccount *service_account_user* parameter was specified during the initial installation of Advanced Server, then *service_account_user* is the Advanced Server cluster owner and is the user to be given with the RUNAS command.

```
RUNAS /USER:service_account_user "CMD.EXE" SET PATH=%PATH%;C:\Program Files\edb\as10\bin
```

During the upgrade process, pg_upgrade writes a file to the current working directory of the service account user; you must invoke pg_upgrade from a directory where the service account user has write privileges. After performing the above commands, navigate to a directory in which the service account user has sufficient privileges to write a file.

```
cd %TEMP%
```

Proceed to Step 6.

If the --serviceaccount parameter was omitted during the initial installation of Advanced Server, then the default owner of the Advanced Server service and the database cluster is NT AUTHORITY\NetworkService.

When NT AUTHORITY\NetworkService is the service account user, the RUNAS command may not be usable as it prompts for a password and the NT AUTHORITY\NetworkService account is not assigned a password. Thus, there is typically a failure with an error message such as, “Unable to acquire user password”.

Under this circumstance a Windows utility program named PsExec must be used to run CMD.EXE as the service account NT AUTHORITY\NetworkService.

The PsExec program must be obtained by downloading PsTools, which is available at the following site:

https://technet.microsoft.com/en-us/sysinternals/bb897553.aspx

You can then use the following command to run CMD.EXE as NT AUTHORITY\NetworkService, and then set the path to the pg_upgrade executable.

```
psexec.exe -u "NT AUTHORITY\NetworkService" CMD.EXE SET PATH=%PATH%;C:\Program Files\edb\as10\bin
```

During the upgrade process, pg_upgrade writes a file to the current working directory of the service account user; you must invoke pg_upgrade from a directory where the service account user has write privileges. After performing the above commands, navigate to a directory in which the service account user has sufficient privileges to write a file.

```
cd %TEMP%
```

Proceed with Step 6.

Step 6 - Perform a consistency check

Before attempting an upgrade, perform a consistency check to assure that the old and new clusters are compatible and properly configured. Include the --check option to instruct pg_upgrade to perform the consistency check.

The following example demonstrates invoking pg_upgrade to perform a consistency check on Linux:

```
pg_upgrade -d /opt/PostgresPlus/9.6AS/data -D /opt/edb/as10/data -U enterprisedb -b /opt/PostgresPlus/9.6AS/bin -B /opt/edb/as10/bin -p 5444 -P 5445 --check
```

If the command is successful, it will return *Clusters are compatible*.

If you are using Windows, you must quote any directory names that contain a space:

```
pg_upgrade.exe -d "C:\Program Files\ PostgresPlus\9.6AS \data" -D "C:\Program Files\edb\as10\data" -U enterprisedb -b "C:\Program Files\PostgresPlus\9.6AS\bin" -B "C:\Program Files\edb\as10\bin" -p 5444 -P 5445 --check
```

During the consistency checking process, pg_upgrade will log any discrepancies that it finds to a file located in the directory from which pg_upgrade was invoked. When the consistency check completes, review the file to identify any missing components or upgrade conflicts. You must resolve any conflicts before invoking pg_upgrade to perform a version upgrade.

If pg_upgrade alerts you to a missing component, you can use StackBuilder Plus to add the component that contains the component. Before using StackBuilder Plus, you must restart the Advanced Server 10 service. After restarting the service, open StackBuilder Plus by navigating through the Start menu to the Advanced Server 10 menu, and selecting StackBuilder Plus. Follow the onscreen advice of the StackBuilder Plus wizard to download and install the missing components.

For more information about using StackBuilder Plus, please see Section 4.5, *Using StackBuilder Plus*.

When pg_upgrade has confirmed that the clusters are compatible, you can perform a version upgrade.

Step 7 - Run pg_upgrade

After confirming that the clusters are compatible, you can invoke pg_upgrade to upgrade the old cluster to the new version of Advanced Server.

On Linux:

```
pg_upgrade -d /opt/PostgresPlus/9.6AS/data -D /opt/edb/as10/data -U enterprisedb -b /opt/PostgresPlus/9.6AS/bin -B /opt/edb/as10/bin -p 5444 -P 5445
```

On Windows:

```
pg_upgrade.exe -d "C:\Program Files\PostgresPlus\9.6AS\data" -D "C:\Program Files\edb\as10\data" -U
```

```
enterprisedb -b "C:\Program Files\PostgresPlus\9.6AS\bin" -B "C:\Program Files\edb\as10\bin" -p 5444 -P 5445
```

pg_upgrade will display the progress of the upgrade onscreen:

```
$ pg_upgrade -d /opt/edb/as10/data -D /opt/edb/as10/data -U enterprisedb -b /opt/edb/as10/bin -B /opt/edb/as10/bin -p 5444 -P 5445

Performing Consistency Checks

-----------------------------

Checking current, bin, and data directories ok

Checking cluster versions ok

Checking database user is a superuser ok

Checking for prepared transactions ok

Checking for reg* system OID user data types ok

Checking for contrib/isn with bigint-passing mismatch ok

Creating catalog dump ok

Checking for presence of required libraries ok

Checking database user is a superuser ok

Checking for prepared transactions ok

If pg_upgrade fails after this point, you must re-initdb the

new cluster before continuing.

Performing Upgrade

------------------

Analyzing all rows in the new cluster ok

Freezing all rows on the new cluster ok

Deleting files from new pg_clog ok

Copying old pg_clog to new server ok

Setting next transaction ID for new cluster ok

Resetting WAL archives ok

Setting frozenxid counters in new cluster ok

Creating databases in the new cluster ok

Adding support functions to new cluster ok

Restoring database schema to new cluster ok

Removing support functions from new cluster ok

Copying user relation files
```

ok

Setting next OID for new cluster ok

Creating script to analyze new cluster ok

Creating script to delete old cluster ok

Upgrade Complete

Optimizer statistics are not transferred by pg_upgrade so,

once you start the new server, consider running:

analyze_new_cluster.sh

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

delete_old_cluster.sh

While pg_upgrade runs, it may generate SQL scripts that handle special circumstances that it has encountered during your upgrade. For example, if the old cluster contains large objects, you may need to invoke a script that defines the default permissions for the objects in the new cluster. When performing the pre-upgrade consistency check pg_upgrade will alert you to any script that you may be required to run manually.

You must invoke the scripts after pg_upgrade completes. To invoke the scripts, connect to the new cluster as a database superuser with the EDB-PSQL command line client, and invoke each script using the \i option:

```
\i complete_path_to_script/script.sql
```

It is generally unsafe to access tables referenced in rebuild scripts until the rebuild scripts have completed; accessing the tables could yield incorrect results or poor performance. Tables not referenced in rebuild scripts can be accessed immediately.

Please Note: If pg_upgrade fails to complete the upgrade process, the old cluster will be unchanged, except that $PGDATA/global/pg_control is renamed to pg_control.old and each tablespace is renamed to *tablespace*.old. To revert to the pre-invocation state:

1. Delete any tablespace directories created by the new cluster.

2. Rename $PGDATA/global/pg_control, removing the .old suffix.

3. Rename the old cluster tablespace directory names, removing the .old suffix.

4. Remove any database objects (from the new cluster) that may have been moved before the upgrade failed.

After performing these steps, resolve any upgrade conflicts encountered before attempting the upgrade again.

When the upgrade is complete, pg_upgrade may also recommend vacuuming the new cluster, and will provide a script that allows you to delete the old cluster.

Before removing the old cluster, ensure that the cluster has been upgraded as expected, and that you have preserved a backup of the cluster in case you need to revert to a previous version.

Step 8 - Restore the authentication settings in the pg_hba.conf file

If you modified the pg_hba.conf file to permit trust authentication, update the contents of the pg_hba.conf file to reflect your preferred authentication settings.

Step 9 - Move and identify user-defined tablespaces (*Optional*)

If you have data stored in a user-defined tablespace, you must manually relocate tablespace files after upgrading; move the files to the new location and update the symbolic links (located in the pg_tblspc directory under your cluster's data directory) to point to the files.

pg_upgrade Troubleshooting

The troubleshooting tips in this section address problems you may encounter when using pg_upgrade.

Upgrade Error - There seems to be a postmaster servicing the cluster

If pg_upgrade reports that a postmaster is servicing the cluster, please stop all Advanced Server services and try the upgrade again.

Upgrade Error - fe_sendauth: no password supplied

If pg_upgrade reports an authentication error that references a missing password, please modify the pg_hba.conf files in the old and new cluster to enable trust authentication, or configure the system to use a pgpass.conf file.

Upgrade Error - New cluster is not empty; exiting

If pg_upgrade reports that the new cluster is not empty, please empty the new cluster. **T**he target cluster may not contain any user-defined databases.

Upgrade Error - Failed to load library

If the original Advanced Server cluster included libraries that are not included in the Advanced Server 10 cluster, pg_upgrade will alert you to the missing component during the consistency check by writing an entry to the loadable_libraries.txt file in the directory from which you invoked pg_upgrade. Generally, for missing libraries that are not part of a major component upgrade, perform the following steps:

1. Restart the Advanced Server service.

Use StackBuilder Plus to download and install the missing module as described in Chapter 4, *Using StackBuilder Plus*. Then:

2. Stop the Advanced Server service.

3. Resume the upgrade process: invoke pg_upgrade to perform consistency checking.

4. When you have resolved any remaining problems noted in the consistency checks, invoke pg_upgrade to perform the data migration from the old cluster to the new cluster.

Reverting to the Old Cluster

The method used to revert to a previous cluster varies with the options specified when invoking pg_upgrade:

- If you specified the --check option when invoking pg_upgrade, an upgrade has not been performed, and no modifications have been made to the old cluster; you can re-use the old cluster at any time.
- If you included the --link option when invoking pg_upgrade, the data files are shared between the old and new

cluster after the upgrade completes. If you have started the server that is servicing the new cluster, the new server has written to those shared files and it is unsafe to use the old cluster.
- If you ran pg_upgrade without the --link specification or have not started the new server, the old cluster is unchanged, except that the .old suffix has been appended to the $PGDATA/global/pg_control and tablespace directories.
- To reuse the old cluster, delete the tablespace directories created by the new cluster and remove the .old suffix from $PGDATA/global/pg_control and the old cluster tablespace directory names and restart the server that services the old cluster.

1.7 Uninstalling Advanced Server

Note that after uninstalling Advanced Server, the cluster data files remain intact and the service user persists. You may manually remove the cluster data and service user from the system.

Uninstalling an RPM Package

You can use variations of the rpm or yum command to remove installed packages. Note that removing a package does not damage the Advanced Server data directory.

Include the -e option when invoking the rpm command to remove an installed package; the command syntax is:

```
rpm -e package_name
```

Where *package_name* is the name of the package that you would like to remove.

You can use the yum remove command to remove a package installed by yum. To remove a package, open a terminal window, assume superuser privileges, and enter the command:

```
yum remove package_name
```

Where *package_name* is the name of the package that you would like to remove.

Note: yum and RPM will not remove a package that is required by another package. If you attempt to remove a package that satisfies a package dependency, yum or RPM will provide a warning.

Using Advanced Server Uninstallers at the Command Line

The Advanced Server interactive installer creates an uninstaller that you can use to remove Advanced Server or components bundled with the installer (pgAdmin 4, StackBuilder Plus, or the command line tools). If you uninstall an Advanced Server component, the remainder of the Advanced Server installation will remain intact.

On Linux, the uninstaller is created in /opt/edb/as10. To open the uninstaller, assume superuser privileges, navigate into the directory that contains the uninstaller, and enter:

./uninstall-edb-as10-server

On Windows, the uninstaller is created in C:\Program Files\edb\as10. To open the uninstaller, assume superuser privileges, navigate into the directory that contains the uninstaller, and enter:

uninstall-edb-as10-server.exe

The uninstaller opens as shown in Figure 8.1.

![image](./images/image62.png)

Figure 8.1 – The Advanced Server uninstaller.

You can remove the Entire application (the default), or select the radio button next to Individual components to select components for removal; click Next.

![image](./images/image63.png)

Figure 8.2 – Select components for uninstallation.

Check the box to the left of a component name to select a component for removal and click Next to continue (see Figure 8.2).

![image](./images/image64.png)

Figure 8.3 – Acknowledge that dependent components are removed first.

If you choose to remove components that are dependent on Advanced Server, those components will be removed first; click Yes to acknowledge that you wish to continue (see Figure 8.3).

When the uninstaller completes, a popup confirms that the data directory and service account have not been removed (see Figure 8.4).

![image](./images/image65.png)

Figure 8.4 - A dialog confirms that the data directory and service user have not been removed.

When the uninstallation is complete, an Info dialog opens to confirm that Advanced Server (and/or its components) has been removed (see Figure 8.5).

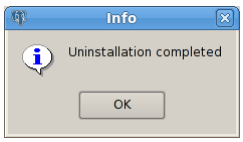

Figure 8.5 - The uninstallation is complete.

1.8 Introduction

The EDB Postgres Advanced Server Installation Guide is a comprehensive guide to installing EDB Postgres Advanced Server (Advanced Server). In this guide you will find detailed information about:

- Software prerequisites for Advanced Server 10.
- Using a package manager to install and update Advanced Server and its supporting components or utilities.
- Installation options available through the interactive setup wizard on Linux and Windows.
- Managing an Advanced Server installation.
- Configuring an Advanced Server installation.
- Using pg_upgrade to upgrade from an earlier version of Advanced Server to Advanced Server 10.
- Uninstalling Advanced Server and its components.

Typographical Conventions Used in this Guide

Certain typographical conventions are used in this manual to clarify the meaning and usage of various commands,

statements, programs, examples, etc. This section provides a summary of these conventions.

In the following descriptions, a *term* refers to any word or group of words that are language keywords, user-supplied values, literals, etc. A term's exact meaning depends upon the context in which it is used.

- *Italic font* introduces a new term, typically in the sentence that defines it for the first time.
- Fixed-width (mono-spaced) font is used for terms that must be given literally such as SQL commands, specific table and column names used in the examples, programming language keywords, etc. For example, SELECT * FROM emp;
- *Italic fixed-width font* is used for terms for which the user must substitute values in actual usage. For example, DELETE FROM *table_name*;

- A vertical pipe | denotes a choice between the terms on either side of the pipe. A vertical pipe is used to separate two or more alternative terms within square brackets (optional choices) or braces (one mandatory choice).

- Square brackets [ ] denote that one or none of the enclosed terms may be substituted. For example, [a | b] means choose one of "a" or "b" or neither of the two.

- Braces {} denote that exactly one of the enclosed alternatives must be specified. For example, { a | b } means exactly one of "a" or "b" must be specified.

- Ellipses ... denote that the preceding term may be repeated. For example, [a | b] ... means that you may have the sequence, "b a a b a".

2 EDB Postgres Advanced Server v10.0 Features

EnterpriseDB features added for Advanced Server Version 10 include:

- Improved Auditing:
- Configurable by DDL and DML statement types
- Configurable by User and Database
- Audit logs can now be directed to syslog
- EDB Clone Schema:
- Allows you to make a copy of an existing schema
- Customizable WAL Segment Size:
- Option for initdb
- Provides the capability to specify the WAL Segment size
- Automatic Prewarm
- The14th Generation of compatibility with the Oracle Database including the following:
- Support for IN OUT Parameters for EXECUTE IMMEDIATE
- OCI Aggregate Pushdown

For a summary of the newly added features that are compatible with Oracle databases, see the following documents:

- Database Compatibility for Oracle Developer's Guide
- Database Compatibility for Oracle Developers Reference Guide
- Database Compatibility for Oracle Developers Tools and Utilities Guide
- Database Compatibility for Oracle Developers Built-in Package Guide

This version also integrates all of the PostgreSQL v10.0 features, including:

- Declarative Partitioning
- Replaces the use of table inheritance
- Provides better performance
- Support for list and range partitioning

- Existing EPAS Partitioning has been fully integrated
- Logical Replication
- Based on Logical Decoding
- Adds configurable replication at table level granularity
- Parallel Query Phase 2
- Parallel Index Scan
- Parallel Bitmap Heap Scan
- Parallel Merge Join
- Parallel subplans
- Extended language support for Parallel query
- SCRAM Authentication
- Durable Hash Indexes
- Executor Speedups
- ICU Collation Support
- FDW Aggregate Pushdown
- Improved Wait Event visibility
- Extended Statistics
- Support NEW and OLD tuplestores in AFTER triggers

Platform Support and System Requirements

EDB Postgres Advanced Server v10.0 supports 64 bit Linux and Windows server platforms. The Advanced Server 10 RPM packages are supported on the following platforms:

64 bit Linux:

- Red Hat Enterprise Linux (x86_64) 6.x and 7.x
- CentOS (x86_64) 6.x and 7.x
- PPC-LE 8 running RHEL or CentOS 7.x

The Advanced Server 10 graphical (or interactive) installers are supported on the following platforms:

64 bit Linux:

- Red Hat Enterprise Linux 6.x and 7.x
- CentOS 6.x and 7.x
- Oracle Enterprise Linux 6.x and 7.x
- Ubuntu 14.04 LTS and 16.04 LTS
- Debian 7 and 8
- SELinux Enterprise 12.x

64 bit Windows:

- Windows Server 2016
- Windows Server 2012 R2 Server

Note: Connectors Installer will be supported on Windows 7, 8, & 10

Details on supported platforms is available on the EnterpriseDB website:

http://www.enterprisedb.com/ppas-platform-support

New platforms will be added as they become available. The next scheduled platform for v10.0 will be SLES 12.0.

Installers and Documentation

EDB Postgres Advanced Server v10.1.5 is packaged and delivered as a series of interactive installers available on the EnterpriseDB website. Visit:

https://www.enterprisedb.com/advanced-downloads

RPM Packages are available for download from:

http://yum.enterprisedb.com/

Documentation is provided on the EnterpriseDB website. Visit:

https://www.enterprisedb.com/resources/product-documentation

Test Matrix

The following components have been tested with EBD Postgres Advanced Server V10.0

- Procedural Language Packs – PL/Perl 5.24, PL/Python 3.4, PL/TCL 8.6
- pgAgent 3.4.1
- Slony 2.2.6
- Connectors 10.0.0
- JDBC 9.4-1208, ODBC 9.05.0400, .NET 3.0.5, OCL 9.6.0.0
- pgAdmin 4 Client 2.0
- pgBouncer 1.7.2.1
- pgPool-II & pgPool-II Extensions 3.6.7
- MTK 51.0.0
- EDBPlus 36.0.0
- PostGIS support will be added when PostGIS versions supporting both 9.6 and 10.1 are made available

Incompatibilities

PostgeSQL 10.0 contains a number of changes that may affect compatibility with previous releases. They are published in the *PostgreSQL 10.0 Release Notes* - https://www.postgresql.org/docs/10/static/release-10.html - and listed here for convenience.

- Hash indexes must be rebuilt after pg_upgrade-ing from any previous major PostgreSQL version (Mithun Cy, Robert Haas, Amit Kapila)

> Major hash index improvements necessitated this requirement. pg_upgrade will create a script to assist with this.

- Rename write-ahead log directory pg_xlog to pg_wal, and rename transaction status directory pg_clog to pg_xact (Michael Paquier)

> Users have occasionally thought that these directories contained only inessential log files, and proceeded to remove write-ahead log files or transaction status files manually, causing irrecoverable data loss. These name changes are intended to discourage such errors in future.

- Rename SQL functions, tools, and options that reference “xlog” to “wal” (Robert Haas)

> For example, pg_switch_xlog() becomes pg_switch_wal(), pg_receivexlog becomes pg_receivewal, and --xlogdir becomes --waldir. This is for consistency with the change of the pg_xlog directory name; in general, the “xlog” terminology is no longer used in any user-facing places.

- Rename WAL-related functions and views to use lsn instead of location (David Rowley)

> There was previously an inconsistent mixture of the two terminologies.

- Change the implementation of set-returning functions appearing in a query's SELECT list (Andres Freund)

> Set-returning functions are now evaluated before evaluation of scalar expressions in the SELECT list, much

> as though they had been placed in a LATERAL FROM-clause item. This allows saner semantics for cases where multiple set-returning functions are present. If they return different numbers of rows, the shorter results are extended to match the longest result by adding nulls. Previously the results were cycled until they all terminated at the same time, producing a number of rows equal to the least common multiple of the functions' periods. In addition, set-returning functions are now disallowed within CASE and COALESCE constructs.
>
> - When ALTER TABLE ... ADD PRIMARY KEY marks columns NOT NULL, that change now propagates to inheritance child tables as well (Michael Paquier)
> - Prevent statement-level triggers from firing more than once per statement (Tom Lane)

Cases involving writable CTEs updating the same table updated by the containing statement, or by another writable CTE, fired BEFORE STATEMENT or AFTER STATEMENT triggers more than once. Also, if there were statement-level triggers on a table affected by a foreign key enforcement action (such as ON DELETE CASCADE), they could fire more than once per outer SQL statement. This is contrary to the SQL standard, so change it.

- Move sequences' metadata fields into a new pg_sequence system catalog (Peter Eisentraut)

A sequence relation now stores only the fields that can be modified by nextval(), that is last_value, log_cnt, and is_called. Other sequence properties, such as the starting value and increment, are kept in a corresponding row of the pg_sequence catalog. ALTER SEQUENCE updates are now fully transactional, implying that the sequence is locked until commit. The nextval() and setval() functions remain nontransactional.

The main incompatibility introduced by this change is that selecting from a sequence relation now returns only the three fields named above. To obtain the sequence's other properties, applications must look into pg_sequence. The new system view pg_sequence can also be used for this purpose; it provides column names that are more compatible with existing code.

The output of psql's \d command for a sequence has been redesigned, too.

- Make pg_basebackup stream the WAL needed to restore the backup by default (Magnus Hagander)

This changes pg_basebackup's -X/--xlog-method default to stream. An option value none has been added to reproduce the old behavior. The pg_basebackup option -x has been removed (instead, use -X fetch).

- Change how logical replication uses pg_hba.conf (Peter Eisentraut)

In previous releases, a logical replication connection required the replication keyword in the database column. As of this release, logical replication matches a normal entry with a database name or keywords such as all. Physical replication continues to use the replication keyword. Since built-in logical replication is new in this release, this change only affects users of third-party logical replication plugins.

- Make all pg_ctl actions wait for completion by default (Peter Eisentraut)

Previously some pg_ctl actions didn't wait for completion, and required the use of -w to do so.

- Change the default value of the log_directory server parameter from pg_log to log (Andreas Karlsson)
- Add configuration option ssl_dh_params_file to specify file name for custom OpenSSL DH parameters (Heikki Linnakangas)

This replaces the hardcoded, undocumented file name dh1024.pem. Note that dh1024.pem is no longer examined by default; you must set this option if you want to use custom DH parameters.

- Increase the size of the default DH parameters used for OpenSSL ephemeral DH ciphers to 2048 bits (Heikki Linnakangas)

The size of the compiled-in DH parameters has been increased from 1024 to 2048 bits, making DH key exchange more resistant to brute-force attacks. However, some old SSL implementations, notably some revisions of Java Runtime Environment version 6, will not accept DH parameters longer than 1024 bits, and hence will not be able to connect over SSL. If it's necessary to support such old clients, you can use custom 1024-bit DH parameters instead of the compiled-in defaults.

- Remove the ability to store unencrypted passwords on the server (Heikki Linnakangas)

The password_encryption server parameter no longer supports off or plain. The UNENCRYPTED option is no longer supported in CREATE/ALTER USER ... PASSSWORD. Similarly, the --unencrypted option has been removed from createuser. Unencrypted passwords migrated from older versions will be stored encrypted in this release. The default setting for password_encryption is still md5.

- Add min_parallel_table_scan_size and min_parallel_index_scan_size server parameters to control parallel queries (Amit Kapila, Robert Haas)

These replace min_parallel_relation_size, which was found to be too generic.

- Don't downcase unquoted text within shared_preload_libraries and related server parameters (QL Zhuo)

These settings are really lists of file names, but they were previously treated as lists of SQL identifiers, which have different parsing rules.

- Remove sql_inheritance server parameter (Robert Haas)

Changing this setting from the default value caused queries referencing parent tables to not include child tables. The SQL standard requires them to be included, however, and this has been the default since PostgreSQL 7.1.

- Allow multi-dimensional arrays to be passed into PL/Python functions, and returned as nested Python lists (Alexey Grishchenko, Dave Cramer, Heikki Linnakangas)

This feature requires a backwards-incompatible change to the handling of arrays of composite types in PL/Python. Previously, you could return an array of composite values by writing, e.g., [[col1, col2], [col1, col2]]; but now that is interpreted as a two-dimensional array. Composite types in arrays must now be written as Python tuples, not lists, to resolve the ambiguity; that is, write [(col1, col2), (col1, col2)] instead.

- Remove PL/Tcl's “module” auto-loading facility (Tom Lane)

This functionality has been replaced by new server parameters pltcl.start_proc and pltclu.start_proc which are easier to use and more similar to features available in other PLs.

- Remove pg_dump/pg_dumpall support for dumping from pre-8.0 servers (Tom Lane)

Users needing to dump from pre-8.0 servers will need to use dump programs from PostgreSQL 9.6 or earlier. The resulting output should still load successfully into newer servers.

- Remove support for floating-point timestamps and intervals (Tom Lane)

This removes configure's --disable-integer-datetimes option. Floating-point timestamps have few advantages and have not been the default since PostgreSQL 8.3.

- Remove server support for client/server protocol version 1.0 (Tom Lane)

This protocol hasn't had client support since PostgreSQL 6.3.

- Remove contrib/tsearch2 module (Robert Haas)

This module provided compatibility with the version of full text search that shipped in pre-8.3 PostgreSQL releases.

- Remove createlang and droplang command-line applications (Peter Eisentraut)

These had been deprecated since PostgreSQL 9.1. Instead, use CREATE EXTENSION and DROP EXTENSION directly.

- Remove support for version-0 function calling conventions (Andres Freund)

Extensions providing C-coded functions must now conform to version 1 calling conventions. Version 0 has been deprecated since 2001.

Deprecated features

Please note that the following items will be deprecated and will no longer be provided in EDB Postgres Advanced Server 11:

- Linux Graphical Installers
- Infinite Cache

How to Report Problems

To report any issues you are having please contact EnterpriseDB's technical support staff:

- Email: support@enterprisedb.com
- Phone: +1-732-331-1320 or 1-800-235-5891 (US Only)

2.1 Introduction

With this release of EDB Postgres Advanced Server 10.0, EnterpriseDB continues its leadership as the only worldwide company to deliver innovative and low cost open source derived database solutions with commercial quality, ease of use, compatibility, scalability, and performance for small or large-scale enterprises.

EDB Postgres Advanced Server 10.0 is built on the open source PostgreSQL 10.0, which introduces an impressive number of improvements that enable databases to scale up and scale out in more efficient ways. PostgreSQL 10.0 introduces Native Partitioning, Logical Replication, SCRAM Authentication, additional Parallel Query capabilities as well as a host of other new features and capabilities.

EDB Postgres Advanced Server 10.0 adds a number of new features that will delight developers and DBAs alike, including:

- Enhanced Partitioning features such as Hash Partitioning, Partition Management and additional Partition Pruning for performance.
- Customizable WAL Segment Size for performance optimization.
- New EDB Audit features.
- Additional Oracle compatibility.

These release notes are applicable to the 10.1.5 release on November 14, 2017.

3 The SQL Language

The following sections describe the subset of the Advanced Server SQL language compatible with Oracle databases. The following SQL syntax, commands, data types, and functions work in both EDB Postgres Advanced Server and Oracle.

The Advanced Server documentation set includes syntax and commands for extended functionality (functionality that does not provide database compatibility for Oracle or support Oracle-styled applications) that is not included in this guide.

This section is organized into the following sections:

- General discussion of Advanced Server SQL syntax and language elements
- Data types
- Summary of SQL commands
- Built-in functions

SQL Syntax

This section describes the general syntax of SQL. It forms the foundation for understanding the following chapters that include detail about how the SQL commands are applied to define and modify data.

Lexical Structure

SQL input consists of a sequence of commands. A *command* is composed of a sequence of *tokens*, terminated by a semicolon (;). The end of the input stream also terminates a command. Which tokens are valid depends on the syntax of the particular command.

A token can be a *key word*, an *identifier*, a *quoted identifier*, a *literal* (or *constant*), or a special character symbol. Tokens are normally separated by *whitespace* (space, tab, new line), but need not be if there is no ambiguity (which is generally only the case if a special character is adjacent to some other token type).

Additionally, *comments* can occur in SQL input. They are not tokens - they are effectively equivalent to whitespace.

For example, the following is (syntactically) valid SQL input:

```
SELECT * FROM MY_TABLE;

UPDATE MY_TABLE SET A = 5;

INSERT INTO MY_TABLE VALUES (3, 'hi there');
```

This is a sequence of three commands, one per line (although this is not required; more than one command can be on a line, and commands can usually be split across lines).

The SQL syntax is not very consistent regarding what tokens identify commands and which are operands or parameters. The first few tokens are generally the command name, so in the above example we would usually speak of a SELECT, an UPDATE, and an INSERT command. But for instance the UPDATE command always requires a SET token to appear in a certain position, and this particular variation of INSERT also requires a VALUES token in order to be complete. The precise syntax rules for each command are described in Section 2.3.

Identifiers and Key Words

Tokens such as SELECT, UPDATE, or VALUES in the example above are examples of *key words*, that is, words that have a fixed meaning in the SQL language. The tokens MY_TABLE and A are examples of *identifiers*. They identify names of tables, columns, or other database objects, depending on the command they are used in. Therefore they are sometimes simply called, “*names*”. Key words and identifiers have the same *lexical structure*, meaning that one cannot know whether a token is an identifier or a key word without knowing the language.

SQL identifiers and key words must begin with a letter (a-z or A-Z). Subsequent characters in an identifier or key word can be letters, underscores, digits (0-9), dollar signs ($), or number signs (#).

Identifier and key word names are case insensitive. Therefore

```
UPDATE MY_TABLE SET A = 5;
```

can equivalently be written as:

uPDaTE my_TabLE SeT a = 5;

A convention often used is to write key words in upper case and names in lower case, e.g.,

UPDATE my_table SET a = 5;

There is a second kind of identifier: the *delimited identifier* or *quoted identifier*. It is formed by enclosing an arbitrary sequence of characters in double-quotes ("). A delimited identifier is always an identifier, never a key word. So "select" could be used to refer to a column or table named "select", whereas an unquoted select would be taken as a key word and would therefore provoke a parse error when used where a table or column name is expected. The example can be written with quoted identifiers like this:

UPDATE "my_table" SET "a" = 5;

Quoted identifiers can contain any character, except the character with the numeric code zero.

To include a double quote, use two double quotes. This allows you to construct table or column names that would otherwise not be possible (such as ones containing spaces or ampersands). The length limitation still applies.

Quoting an identifier also makes it case-sensitive, whereas unquoted names are always folded to lower case. For example, the identifiers FOO, foo, and "foo" are considered the same by Advanced Server, but "Foo" and "FOO" are different from these three and each other. The folding of unquoted names to lower case is not compatible with Oracle databases. In Oracle syntax, unquoted names are folded to upper case: for example, foo is equivalent to "FOO" not "foo". If you want to write portable applications you are advised to always quote a particular name or never quote it.

Constants

The kinds of implicitly-typed constants in Advanced Server are *strings* and *numbers*. Constants can also be specified with explicit types, which can enable more accurate representation and more efficient handling by the system. These alternatives are discussed in the following subsections.

String Constants

A *string constant* in SQL is an arbitrary sequence of characters bounded by single quotes ('), for example 'This is a string'. To include a single-quote character within a string constant, write two adjacent single quotes, e.g. 'Dianne''s horse'. Note that this is not the same as a double-quote character (").

Numeric Constants

Numeric constants are accepted in these general forms:

digits

digits.[*digits*][e[+-]*digits*]

[*digits*].*digits*[e[+-]*digits*]

*digits*e[+-]*digits*

where *digits* is one or more decimal digits (0 through 9). At least one digit must be before or after the decimal point, if one is used. At least one digit must follow the exponent marker (e), if one is present. There may not be any spaces or other characters embedded in the constant. Note that any leading plus or minus sign is not actually considered part of the constant; it is an operator applied to the constant.

These are some examples of valid numeric constants:

42 3.5 4. .001 5e2 1.925e-3

A numeric constant that contains neither a decimal point nor an exponent is initially presumed to be type INTEGER if its value fits in type INTEGER (32 bits); otherwise it is presumed to be type BIGINT if its value fits in type BIGINT (64 bits); otherwise it is taken to be type NUMBER. Constants that contain decimal points and/or exponents are always initially presumed to be type NUMBER.

The initially assigned data type of a numeric constant is just a starting point for the type resolution algorithms. In most cases the constant will be automatically coerced to the most appropriate type depending on context. When necessary, you can force a numeric value to be interpreted as a specific data type by casting it as described in the following section.

Constants of Other Types

A constant of an arbitrary type can be entered using the following notation:

CAST('*string*' AS *type*)

The string constant's text is passed to the input conversion routine for the type called *type*. The result is a constant of the indicated type. The explicit type cast may be omitted if there is no ambiguity as to the type the constant must be (for example, when it is assigned directly to a table column), in which case it is automatically coerced.

CAST can also be used to specify runtime type conversions of arbitrary expressions.

Comments

A comment is an arbitrary sequence of characters beginning with double dashes and extending to the end of the line, e.g.:

-- This is a standard SQL comment

Alternatively, C-style block comments can be used:

/* multiline comment

* block

*/

where the comment begins with /* and extends to the matching occurrence of */.

Data Types

The following table shows the built-in general-purpose data types.

Table - Data Types

Name	Alias	Description
BLOB	LONG RAW, RAW(*n*), BYTEA	Binary data
BOOLEAN		Logical Boolean (true/false)
CHAR [(*n*)]	CHARACTER [(*n*)]	Fixed-length character string of n characters
CLOB	LONG, LONG VARCHAR	Long character string
DATE	TIMESTAMP	Date and time to the second

DOUBLE PRECISION	FLOAT, FLOAT(25) – FLOAT(53)	Double precision floating-point number
INTEGER	INT, BINARY_INTEGER, PLS_INTEGER	Signed four-byte integer
NUMBER	DEC, DECIMAL, NUMERIC	Exact numeric with optional decimal places
NUMBER(*p* [, *s*])	DEC(p [, *s*]), DECIMAL(*p* [, *s*]), NUMERIC(*p* [, *s*])	Exact numeric of maximum precision, *p*, and optional scale, *s*
REAL	FLOAT(1) – FLOAT(24)	Single precision floating-point number
TIMESTAMP [(*p*)]		Date and time with optional, fractional second precision, *p*
TIMESTAMP [(*p*)] WITH TIME ZONE		Date and time with optional, fractional second precision, *p*, and with time zone
VARCHAR2(*n*)	CHAR VARYING(*n*), CHARACTER VARYING(*n*), VARCHAR(*n*)	Variable-length character string with a maximum length of *n* characters
XMLTYPE		XML data

Numeric Types

Numeric types consist of four-byte integers, four-byte and eight-byte floating-point numbers, and fixed-precision decimals. The following table lists the available types.

Table - Numeric Types

Name	Storage Size	Description	Range
BINARY_INTEGER	4 bytes	Signed integer, Alias for INTEGER	-2,147,483,648 to +2,147,483,647
DOUBLE PRECISION	8 bytes	Variable-precision, inexact	15 decimal digits precision
INTEGER	4 bytes	Usual choice for integer	-2,147,483,648 to +2,147,483,647
NUMBER	Variable	User-specified precision, exact	Up to 1000 digits of precision
NUMBER(*p* [, *s*])	Variable	Exact numeric of maximum precision, *p*, and optional scale, *s*	Up to 1000 digits of precision
PLS_INTEGER	4 bytes	Signed integer, Alias for INTEGER	-2,147,483,648 to +2,147,483,647
REAL	4 bytes	Variable-precision, inexact	6 decimal digits precision
ROWID	8 bytes	Signed 8 bit integer.	-9223372036854775808 to 9223372036854775807

The following sections describe the types in detail.

Integer Types

The type, INTEGER, stores whole numbers (without fractional components) between the values of -2,147,483,648 and +2,147,483,647. Attempts to store values outside of the allowed range will result in an error.

Columns of the ROWID type holds fixed-length binary data that describes the physical address of a record. ROWID is an unsigned, four-byte INTEGER that stores whole numbers (without fractional components) between the values of 0 and 4,294,967,295. Attempts to store values outside of the allowed range will result in an error.

Arbitrary Precision Numbers

The type, NUMBER, can store practically an unlimited number of digits of precision and perform calculations exactly. It is especially recommended for storing monetary amounts and other quantities where exactness is required. However, the NUMBER type is very slow compared to the floating-point types described in the next section.

In what follows we use these terms: The *scale* of a NUMBER is the count of decimal digits in the fractional part, to the right of the decimal point. The *precision* of a NUMBER is the total count of significant digits in the whole number, that is, the number of digits to both sides of the decimal point. So the number 23.5141 has a precision of 6 and a scale of 4. Integers can be considered to have a scale of zero.

Both the precision and the scale of the NUMBER type can be configured. To declare a column of type NUMBER use the syntax

NUMBER(*precision*, *scale*)

The precision must be positive, the scale zero or positive. Alternatively,

NUMBER(*precision*)

selects a scale of 0. Specifying NUMBER without any precision or scale creates a column in which numeric values of any precision and scale can be stored, up to the implementation limit on precision. A column of this kind will not coerce input values to any particular scale, whereas NUMBER columns with a declared scale will coerce input values to that scale. (The SQL standard requires a default scale of 0, i.e., coercion to integer precision. For maximum portability, it is best to specify the precision and scale explicitly.)

If the precision or scale of a value is greater than the declared precision or scale of a column, the system will attempt to round the value. If the value cannot be rounded so as to satisfy the declared limits, an error is raised.

Floating-Point Types

The data types REAL and DOUBLE PRECISION are *inexact*, variable-precision numeric types. In practice, these types are usually implementations of IEEE Standard 754 for Binary Floating-Point Arithmetic (single and double precision, respectively), to the extent that the underlying processor, operating system, and compiler support it.

Inexact means that some values cannot be converted exactly to the internal format and are stored as approximations, so that storing and printing back out a value may show slight discrepancies. Managing these errors and how they propagate through calculations is the subject of an entire branch of mathematics and computer science and will not be discussed further here, except for the following points:

If you require exact storage and calculations (such as for monetary amounts), use the NUMBER type instead.

If you want to do complicated calculations with these types for anything important, especially if you rely on certain behavior in boundary cases (infinity, underflow), you should evaluate the implementation carefully.

Comparing two floating-point values for equality may or may not work as expected.

On most platforms, the REAL type has a range of at least 1E-37 to 1E+37 with a precision of at least 6 decimal digits. The DOUBLE PRECISION type typically has a range of around 1E-307 to 1E+308 with a precision of at least 15 digits. Values that are too large or too small will cause an error. Rounding may take place if the precision of an input number is too high. Numbers too close to zero that are not representable as distinct from zero will cause an underflow error.

Advanced Server also supports the SQL standard notations FLOAT and FLOAT(*p*) for specifying inexact numeric types. Here, *p* specifies the minimum acceptable precision in binary digits. Advanced Server accepts FLOAT(1) to

FLOAT(24) as selecting the REAL type, while FLOAT(25) to FLOAT(53) as selecting DOUBLE PRECISION. Values of *p* outside the allowed range draw an error. FLOAT with no precision specified is taken to mean DOUBLE PRECISION.

Character Types

The following table lists the general-purpose character types available in Advanced Server.

Table - Character Types

Name	Description
CHAR[(*n*)]	Fixed-length character string, blank-padded to the size specified by *n*
CLOB	Large variable-length up to 1 GB
LONG	Variable unlimited length.
NVARCHAR(*n*)	Variable-length national character string, with limit.
NVARCHAR2(*n*)	Variable-length national character string, with limit.
STRING	Alias for VARCHAR2.
VARCHAR(*n*)	Variable-length character string, with limit (considered deprecated, but supported for compatibility)
VARCHAR2(*n*)	Variable-length character string, with limit

Where *n* is a positive integer; these types can store strings up to *n* characters in length. An attempt to assign a value that exceeds the length of *n* will result in an error, unless the excess characters are all spaces, in which case the string will be truncated to the maximum length.

CHAR

If you do not specify a value for *n*, *n* will default to 1. If the string to be assigned is shorter than *n*, values of type CHAR will be space-padded to the specified width (*n*), and will be stored and displayed that way.

Padding spaces are treated as semantically insignificant. That is, trailing spaces are disregarded when comparing two values of type CHAR, and they will be removed when converting a CHAR value to one of the other string types.

If you explicitly cast an over-length value to a CHAR(*n*) type, the value will be truncated to *n* characters without raising an error (as specified by the SQL standard).

VARCHAR, VARCHAR2, NVARCHAR and NVARCHAR2

If the string to be assigned is shorter than *n*, values of type VARCHAR, VARCHAR2, NVARCHAR and NVARCHAR2 will store the shorter string without padding.

Note that trailing spaces *are* semantically significant in VARCHAR values.

If you explicitly cast a value to a VARCHAR type, an over-length value will be truncated to *n* characters without raising an error (as specified by the SQL standard).

CLOB

You can store a large character string in a CLOB type. CLOB is semantically equivalent to VARCHAR2 except no length limit is specified. Generally, you should use a CLOB type if the maximum string length is not known.

The longest possible character string that can be stored in a CLOB type is about 1 GB.

The storage requirement for data of these types is the actual string plus 1 byte if the string is less than 127 bytes, or 4 bytes if the string is 127 bytes or greater. In the case of CHAR, the padding also requires storage. Long strings are compressed by the system automatically, so the physical requirement on disk may be less. Long

values are stored in background tables so they do not interfere with rapid access to the shorter column values.

The database character set determines the character set used to store textual values.

Binary Data

The following data types allows storage of binary strings.

Table - Binary Large Object

Name	Storage Size	Description
BINARY	The length of the binary string.	Fixed-length binary string, with a length between 1 and 8300.
BLOB	The actual binary string plus 1 byte if the binary string is less than 127 bytes, or 4 bytes if the binary string is 127 bytes or greater.	Variable-length binary string, with a maximum size of 1 GB.
VARBINARY	The length of the binary string	Variable-length binary string, with a length between 1 and 8300.

A binary string is a sequence of octets (or bytes). Binary strings are distinguished from characters strings by two characteristics: First, binary strings specifically allow storing octets of value zero and other "non-printable" octets (defined as octets outside the range 32 to 126). Second, operations on binary strings process the actual bytes, whereas the encoding and processing of character strings depends on locale settings.

Date/Time Types

The following discussion of the date/time types assumes that the configuration parameter, edb_redwood_date, has been set to true whenever a table is created or altered.

Advanced Server supports the date/time types shown in the following table.

Table - Date/Time Types

Name	Storage Size	Description	Low Value	High Value	Resolution
DATE	8 bytes	Date and time	4713 BC	5874897 AD	1 second
INTERVAL DAY TO SECOND [(*p*)]	12 bytes	Period of time	-178000000 years	178000000 years	1 microsecond / 14 digits
INTERVAL YEAR TO MONTH	12 bytes	Period of time	-178000000 years	178000000 years	1 microsecond / 14 digits
TIMESTAMP [(*p*)]	8 bytes	Date and time	4713 BC	5874897 AD	1 microsecond
TIMESTAMP [(*p*)] WITH TIME ZONE	8 bytes	Date and time with time zone	4713 BC	5874897 AD	1 microsecond

When DATE appears as the data type of a column in the data definition language (DDL) commands, CREATE TABLE or ALTER TABLE, it is translated to TIMESTAMP at the time the table definition is stored in the database. Thus, a time component will also be stored in the column along with the date.

When DATE appears as a data type of a variable in an SPL declaration section, or the data type of a formal parameter in an SPL procedure or an SPL function, or the return type of an SPL function, it is always translated to TIMESTAMP and thus can handle a time component if present.

TIMESTAMP accepts an optional precision value *p* which specifies the number of fractional digits retained in the seconds field. The allowed range of *p* is from 0 to 6 with the default being 6.

When TIMESTAMP values are stored as double precision floating-point numbers (currently the default), the effective limit of precision may be less than 6. TIMESTAMP values are stored as seconds before or after midnight 2000-01-01. Microsecond precision is achieved for dates within a few years of 2000-01-01, but the precision degrades for dates further away. When TIMESTAMP values are stored as eight-byte integers (a compile-time option), microsecond precision is available over the full range of values. However eight-byte integer timestamps have a more limited range of dates than shown above: from 4713 BC up to 294276 AD.

TIMESTAMP (*p*) WITH TIME ZONE is similar to TIMESTAMP (*p*), but includes the time zone as well.

INTERVAL Types

INTERVAL values specify a period of time. Values of INTERVAL type are composed of fields that describe the value of the data. The following table lists the fields allowed in an INTERVAL type:

Table - Interval Types

Field Name	INTERVAL Values Allowed
YEAR	Integer value (positive or negative)
MONTH	0 through 11
DAY	Integer value (positive or negative)
HOUR	0 through 23
MINUTE	0 through 59
SECOND	0 through 59.9(*p*) where 9(*p*) is the precision of fractional seconds

The fields must be presented in descending order – from YEARS to MONTHS, and from DAYS to HOURS, MINUTES and then SECONDS.

Advanced Server supports two INTERVAL types compatible with Oracle databases.

The first variation supported by Advanced Server is INTERVAL DAY TO SECOND [(*p*)]. INTERVAL DAY TO SECOND [(*p*)] stores a time interval in days, hours, minutes and seconds.

> *p* specifies the precision of the second field.

Advanced Server interprets the value:

```
INTERVAL '1 2:34:5.678' DAY TO SECOND(3)
```

as 1 day, 2 hours, 34 minutes, 5 seconds and 678 thousandths of a second.

Advanced Server interprets the value:

```
INTERVAL '1 23' DAY TO HOUR
```

as 1 day and 23 hours.

Advanced Server interprets the value:

```
INTERVAL '2:34' HOUR TO MINUTE
```

as 2 hours and 34 minutes.

Advanced Server interprets the value:

```
INTERVAL '2:34:56.129' HOUR TO SECOND(2)
```

as 2 hours, 34 minutes, 56 seconds and 13 thousandths of a second. Note that the fractional second is rounded up to 13 because of the specified precision.

The second variation supported by Advanced Server that is compatible with Oracle databases is INTERVAL YEAR TO MONTH. This variation stores a time interval in years and months.

Advanced Server interprets the value:

```
INTERVAL '12-3' YEAR TO MONTH
```

as 12 years and 3 months.

Advanced Server interprets the value:

```
INTERVAL '456' YEAR(2)
```

as 12 years and 3 months.

Advanced Server interprets the value:

```
INTERVAL '300' MONTH
```

as 25 years.

Date/Time Input

Date and time input is accepted in ISO 8601 SQL-compatible format, the Oracle default dd-MON-yy format, as well as a number of other formats provided that there is no ambiguity as to which component is the year, month, and day. However, use of the TO_DATE function is strongly recommended to avoid ambiguities.

Any date or time literal input needs to be enclosed in single quotes, like text strings. The following SQL standard syntax is also accepted:

type '*value*'

type is either DATE or TIMESTAMP.

value is a date/time text string.

Dates

The following table shows some possible input formats for dates, all of which equate to January 8, 1999.

Table - Date Input

Example
January 8, 1999
1999-01-08
1999-Jan-08
Jan-08-1999
08-Jan-1999
08-Jan-99
Jan-08-99

19990108

990108

The date values can be assigned to a DATE or TIMESTAMP column or variable. The hour, minute, and seconds fields will be set to zero if the date value is not appended with a time value.

Times

Some examples of the time component of a date or time stamp are shown in the following table.

Table - Time Input

Example	Description
04:05:06.789	ISO 8601
04:05:06	ISO 8601
04:05	ISO 8601
040506	ISO 8601
04:05 AM	Same as 04:05; AM does not affect value
04:05 PM	Same as 16:05; input hour must be \<= 12

Time Stamps

Valid input for time stamps consists of a concatenation of a date and a time. The date portion of the time stamp can be formatted according to any of the examples shown in Table 2-7. The time portion of the time stamp can be formatted according to any of examples shown in Table 2-8.

The following is an example of a time stamp which follows the Oracle default format.

08-JAN-99 04:05:06

The following is an example of a time stamp which follows the ISO 8601 standard.

1999-01-08 04:05:06

Date/Time Output

The default output format of the date/time types will be either (dd-MON-yy) referred to as the *Redwood date style*, compatible with Oracle databases, or (yyyy-mm-dd) referred to as the ISO 8601 format, depending upon the application interface to the database. Applications that use JDBC such as SQL Interactive always present the date in ISO 8601 form. Other applications such as PSQL present the date in Redwood form.

The following table shows examples of the output formats for the two styles, Redwood and ISO 8601.

Table - Date/Time Output Styles

Description	Example
Redwood style	31-DEC-05 07:37:16
ISO 8601/SQL standard	1997-12-17 07:37:16

Internals

Advanced Server uses Julian dates for all date/time calculations. Julian dates correctly predict or calculate any

date after 4713 BC based on the assumption that the length of the year is 365.2425 days.

Boolean Type

Advanced Server provides the standard SQL type BOOLEAN. BOOLEAN can have one of only two states: TRUE or FALSE. A third state, UNKNOWN, is represented by the SQL NULL value.

Table - Boolean Type

Name	Storage Size	Description
BOOLEAN	1 byte	Logical Boolean (true/false)

The valid literal value for representing the true state is TRUE. The valid literal for representing the false state is FALSE.

XML Type

The XMLTYPE data type is used to store XML data. Its advantage over storing XML data in a character field is that it checks the input values for well-formedness, and there are support functions to perform type-safe operations on it.

The XML type can store well-formed “documents”, as defined by the XML standard, as well as “content” fragments, which are defined by the production XMLDecl? content in the XML standard. Roughly, this means that content fragments can have more than one top-level element or character node.

Note: Oracle does not support the storage of content fragments in XMLTYPE columns.

The following example shows the creation and insertion of a row into a table with an XMLTYPE column.

```
CREATE TABLE books (

content XMLTYPE

);

INSERT INTO books VALUES (XMLPARSE (DOCUMENT '\<?xml version="1.0"?>\<book>\<title>Manual\</title>\
<chapter>...\</chapter>\</book>'));

SELECT * FROM books;

content

----------------------------------------------------------

\<book>\<title>Manual\</title>\<chapter>...\</chapter>\</book>

(1 row)
```

SQL Commands

This section provides a summary of the SQL commands compatible with Oracle databases that are supported by Advanced Server. The SQL commands in this section will work on both an Oracle database and an Advanced Server database.

Note the following points:

- Advanced Server supports other commands that are not listed here. These commands may have no Oracle equivalent or they may provide the similar or same functionality as an Oracle SQL command, but with different syntax.
- The SQL commands in this section do not necessarily represent the full syntax, options, and functionality available for each command. In most cases, syntax, options, and functionality that are not compatible with Oracle databases have been omitted from the command description and syntax.
- The Advanced Server documentation set documents command functionality that may not be compatible with Oracle databases.

ALTER INDEX

Name

ALTER INDEX -- modify an existing index.

Synopsis

Advanced Server supports two variations of the ALTER INDEX command compatible with Oracle databases. Use the first variation to rename an index:

ALTER INDEX *name* RENAME TO *new_name*

Use the second variation of the ALTER INDEX command to rebuild an index:

ALTER INDEX *name* REBUILD

Description

ALTER INDEX changes the definition of an existing index. The RENAME clause changes the name of the index. The REBUILD clause reconstructs an index, replacing the old copy of the index with an updated version based on the index's table.

The REBUILD clause invokes the PostgreSQL REINDEX command; for more information about using the REBUILD clause, see the PostgreSQL core documentation at:

https://www.postgresql.org/docs/10/static/sql-reindex.html

ALTER INDEX has no effect on stored data.

Parameters

name

The name (possibly schema-qualified) of an existing index.

new_name

New name for the index.

Examples

To change the name of an index from name_idx to empname_idx:

```
ALTER INDEX name_idx RENAME TO empname_idx;
```

To rebuild an index named empname_idx:

```
ALTER INDEX empname_idx REBUILD;
```

See Also

CREATE INDEX, DROP INDEX

ALTER PROCEDURE

Name

ALTER PROCEDURE

Synopsis

ALTER PROCEDURE *procedure_name options* [RESTRICT]

Description

Use the ALTER PROCEDURE statement to specify that a procedure is a SECURITY INVOKER or SECURITY DEFINER.

Parameters

procedure_name

procedure_name specifies the (possibly schema-qualified) name of a stored procedure.

options may be:

[EXTERNAL] SECURITY DEFINER

Specify SECURITY DEFINER to instruct the server to execute the procedure with the privileges of the user that created the procedure. The EXTERNAL keyword is accepted for compatibility, but ignored.

[EXTERNAL] SECURITY INVOKER

Specify SECURITY INVOKER to instruct the server to execute the procedure with the privileges of the user that is invoking the procedure. The EXTERNAL keyword is accepted for compatibility, but ignored.

The RESTRICT keyword is accepted for compatibility, but ignored.

Examples

The following command specifies that the update_balance procedure should execute with the privileges of the user invoking the procedure:

ALTER PROCEDURE update_balance SECURITY INVOKER;

ALTER PROFILE

Name

ALTER PROFILE – alter an existing profile

Synopsis

ALTER PROFILE *profile_name* RENAME TO *new_name*;

ALTER PROFILE *profile_name* LIMIT {*parameter value*}[...];

Description

Use the ALTER PROFILE command to modify a user-defined profile; Advanced Server supports two forms of the command:

- Use ALTER PROFILE…RENAME TO to change the name of a profile.
- Use ALTER PROFILE…LIMIT to modify the limits associated with a profile.

Include the LIMIT clause and one or more space-delimited *parameter*/*value* pairs to specify the rules enforced by Advanced Server, or use ALTER PROFILE…RENAME TO to change the name of a profile.

Parameters

profile_name

The name of the profile.

new_name

> *new_name* specifies the new name of the profile.

parameter

> *parameter* specifies the attribute limited by the profile.

value

> *value* specifies the parameter limit.

Advanced Server supports the *value* shown below for each *parameter*:

> FAILED_LOGIN_ATTEMPTS specifies the number of failed login attempts that a user may make before the server locks the user out of their account 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 may make an unlimited number of failed login attempts.

> PASSWORD_LOCK_TIME specifies the length of time that must pass before the server unlocks an account that has been locked because of 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 it is manually unlocked by a database superuser.

> PASSWORD_LIFE_TIME specifies the number of days that the current password may be used before the user is prompted to provide a new password. Include the PASSWORD_GRACE_TIME clause when using the PASSWORD_LIFE_TIME clause to specify the number of days that will pass after the password expires before connections by the role are rejected. If PASSWORD_GRACE_TIME is not specified, the password will expire on the day specified by the default value of PASSWORD_GRACE_TIME, and the user will not be allowed to execute any command until a new password is provided. 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 does not have an expiration date.

PASSWORD_GRACE_TIME specifies the length of the grace period after a password expires until the user is forced to change their password. When the grace period expires, a user will be allowed to connect, but will not be allowed to 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_REUSE_TIME specifies the number of days a user must wait before re-using a password. The PASSWORD_REUSE_TIME and PASSWORD_REUSE_MAX parameters are intended to be used together. If you specify a finite value for one of these parameters while the other is UNLIMITED, old passwords can never be reused. If both parameters are set to UNLIMITED there are no restrictions on password reuse. 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_REUSE_TIME specified in the DEFAULT profile.
- UNLIMITED – The password can be re-used without restrictions.

PASSWORD_REUSE_MAX specifies the number of password changes that must occur before a password can be reused. The PASSWORD_REUSE_TIME and PASSWORD_REUSE_MAX parameters are intended to be used together. If you specify a finite value for one of these parameters while the other is UNLIMITED, old passwords can never be reused. If both parameters are set to UNLIMITED there are no restrictions on password reuse. Supported values are:

- An INTEGER value greater than or equal to 0.
- DEFAULT - the value of PASSWORD_REUSE_MAX specified in the DEFAULT profile.
- UNLIMITED – The password can be re-used without restrictions.

PASSWORD_VERIFY_FUNCTION specifies password complexity. Supported values are:

- The name of a PL/SQL function.
- DEFAULT - the value of PASSWORD_VERIFY_FUNCTION specified in the DEFAULT profile.
- NULL

Examples

The following example modifies a profile named acctg_profile:

```
ALTER PROFILE acctg_profile LIMIT FAILED_LOGIN_ATTEMPTS 3 PASSWORD_LOCK_TIME 1;
```

acctg_profile will count failed connection attempts when a login role attempts to connect to the server. The profile specifies that if a user has not authenticated with the correct password in three attempts, the account will be locked for one day.

The following example changes the name of acctg_profile to payables_profile:

```
ALTER PROFILE acctg_profile RENAME TO payables_profile;
```

ALTER QUEUE

Advanced Server includes extra syntax (not offered by Oracle) with the ALTER QUEUE SQL command. This syntax can be used in association with the DBMS_AQADM package.

Name

ALTER QUEUE -- allows a superuser or a user with the aq_administrator_role privilege to modify the attributes of a queue.

Synopsis

This command is available in four forms. The first form of this command changes the name of a queue.

> ALTER QUEUE *queue_name* RENAME TO *new_name*

Parameters

> *queue_name*
>
> The name (optionally schema-qualified) of an existing queue.
>
> RENAME TO
>
> Include the RENAME TO clause and a new name for the queue to rename the queue.
>
> *new_name*
>
> New name for the queue.

The second form of the ALTER QUEUE command modifies the attributes of the queue:

ALTER QUEUE *queue_name* SET [({ *option_name option_value* } [,SET *option_name*

Parameters

> *queue_name*
>
> The name (optionally schema-qualified) of an existing queue.
>
> Include the SET clause and option_name/option_value pairs to modify the attributes of the queue:
>
> *option_name option_value*
>
> The name of the option or options to be associated with the new queue and the corresponding value of the option. If you provide duplicate option names, the server will return an error.

- If *option_name* is retries, provide an integer that represents the number of times a dequeue may be attempted.
- If *option_name* is retrydelay, provide a double-precision value that represents the delay in seconds.
- If *option_name* is retention, provide a double-precision value that represents the retention time in seconds.

Use the third form of the ALTER QUEUE command to enable or disable enqueuing and/or dequeuing on a particular queue:

> ALTER QUEUE *queue_name* ACCESS { START | STOP } [FOR { enqueue | dequeue }] [NOWAIT]

Parameters

> *queue_name*
>
> The name (optionally schema-qualified) of an existing queue.

ACCESS

Include the ACCESS keyword to enable or disable enqueuing and/or dequeuing on a particular queue.

START | STOP

Use the START and STOP keywords to indicate the desired state of the queue.

FOR enqueue|dequeue

Use the FOR clause to indicate if you are specifying the state of enqueueing or dequeueing activity on the specified queue.

NOWAIT

Include the NOWAIT keyword to specify that the server should not wait for the completion of outstanding transactions before changing the state of the queue. The NOWAIT keyword can only be used when specifying an ACCESS value of STOP. The server will return an error if NOWAIT is specified with an ACCESS value of START.

Use the fourth form to ADD or DROP callback details for a particular queue.

> ALTER QUEUE *queue_name* { ADD | DROP } CALL TO *location_name* [WITH *callback_option*]

Parameters

queue_name

The name (optionally schema-qualified) of an existing queue.

ADD | DROP

Include the ADD or DROP keywords to enable add or remove callback details for a queue.

location_name

location_name specifies the name of the callback procedure.

callback_option

callback_option can be context; specify a RAW value when including this clause.

Example

The following example changes the name of a queue from work_queue_east to work_order:

ALTER QUEUE work_queue_east RENAME TO work_order;

The following example modifies a queue named work_order, setting the number of retries to 100, the delay between retries to 2 seconds, and the length of time that the queue will retain dequeued messages to 10 seconds:

ALTER QUEUE work_order SET (retries 100, retrydelay 2, retention 10);

The following commands enable enqueueing and dequeueing in a queue named work_order:

ALTER QUEUE work_order ACCESS START;

ALTER QUEUE work_order ACCESS START FOR enqueue;

ALTER QUEUE work_order ACCESS START FOR dequeue;

The following commands disable enqueueing and dequeueing in a queue named work_order:

ALTER QUEUE work_order ACCESS STOP NOWAIT;

ALTER QUEUE work_order ACCESS STOP FOR enqueue;

ALTER QUEUE work_order ACCESS STOP FOR dequeue;

See Also

CREATE QUEUE, DROP QUEUE

ALTER QUEUE TABLE

Advanced Server includes extra syntax (not offered by Oracle) with the ALTER QUEUE SQL command. This syntax can be used in association with the DBMS_AQADM package.

Name

ALTER QUEUE TABLE-- modify an existing queue table.

Synopsis

Use ALTER QUEUE TABLE to change the name of an existing queue table:

```
ALTER QUEUE TABLE name RENAME TO new_name
```

Description

ALTER QUEUE TABLE allows a superuser or a user with the aq_administrator_role privilege to change the name of an existing queue table.

Parameters

name

The name (optionally schema-qualified) of an existing queue table.

new_name

New name for the queue table.

Example

To change the name of a queue table from wo_table_east to work_order_table:

ALTER QUEUE TABLE wo_queue_east RENAME TO work_order_table;

See Also

CREATE QUEUE TABLE, DROP QUEUE TABLE

ALTER ROLE... IDENTIFIED BY

Name

ALTER ROLE - change the password associated with a database role

Synopsis

ALTER ROLE *role_name* IDENTIFIED BY *password

- [REPLACE *prev_password*]

Description

A role without the CREATEROLE privilege may use this command to change their own password. An unprivileged role must include the REPLACE clause and their previous password if PASSWORD_VERIFY_FUNCTION is not NULL in their profile. When the REPLACE clause is used by a non-superuser, the server will compare the password provided to the existing password and raise an error if the passwords do not match.

A database superuser can use this command to change the password associated with any role. If a superuser includes the REPLACE clause, the clause is ignored; a non-matching value for the previous password will not throw an error.

If the role for which the password is being changed has the SUPERUSER attribute, then a superuser must issue this command. A role with the CREATEROLE attribute can use this command to change the password associated with a role that is not a superuser.

Parameters

role_name

The name of the role whose password is to be altered.

password

The role's new password.

prev_password

The role's previous password.

Examples

To change a role's password:

ALTER ROLE john IDENTIFIED BY xyRP35z REPLACE 23PJ74a;

ALTER ROLE - Managing Database Link and DBMS_RLS Privileges

Advanced Server includes extra syntax (not offered by Oracle) for the ALTER ROLE command. This syntax can be useful when assigning privileges related to creating and dropping database links compatible with Oracle databases, and fine-grained access control (using DBMS_RLS).

CREATE DATABASE LINK

A user who holds the CREATE DATABASE LINK privilege may create a private database link. The following ALTER ROLE command grants privileges to an Advanced Server role that allow the specified role to create a private database link:

> ALTER ROLE *role_name* WITH [CREATEDBLINK | CREATE DATABASE LINK]

This command is the functional equivalent of:

> GRANT CREATE DATABASE LINK to *role_name*

Use the following command to revoke the privilege:

> ALTER ROLE *role_name* WITH [NOCREATEDBLINK | NO CREATE DATABASE LINK]

Please note: the CREATEDBLINK and NOCREATEDBLINK keywords should be considered deprecated syntax; we recommend using the CREATE DATABASE LINK and NO CREATE DATABASE LINK syntax options.

CREATE PUBLIC DATABASE LINK

A user who holds the CREATE PUBLIC DATABASE LINK privilege may create a public database link. The following ALTER ROLE command grants privileges to an Advanced Server role that allow the specified role to create a public database link:

```
ALTER ROLE role_name WITH [CREATEPUBLICDBLINK | CREATE PUBLIC DATABASE LINK]
```

This command is the functional equivalent of:

```
GRANT CREATE PUBLIC DATABASE LINK to role_name
```

Use the following command to revoke the privilege:

```
ALTER ROLE role_name WITH [NOCREATEPUBLICDBLINK | NO CREATE PUBLIC DATABASE LINK]
```

Please note: the CREATEPUBLICDBLINK and NOCREATEPUBLICDBLINK keywords should be considered deprecated syntax; we recommend using the CREATE PUBLIC DATABASE LINK and NO CREATE PUBLIC DATABASE LINK syntax options.

DROP PUBLIC DATABASE LINK

A user who holds the DROP PUBLIC DATABASE LINK privilege may drop a public database link. The following ALTER ROLE command grants privileges to an Advanced Server role that allow the specified role to drop a public database link:

```
ALTER ROLE role_name WITH [DROPPUBLICDBLINK | DROP PUBLIC DATABASE LINK]
```

This command is the functional equivalent of:

```
GRANT DROP PUBLIC DATABASE LINK to role_name
```

Use the following command to revoke the privilege:

```
ALTER ROLE role_name WITH [NODROPPUBLICDBLINK | NO DROP PUBLIC DATABASE LINK]
```

Please note: the DROPPUBLICDBLINK and NODROPPUBLICDBLINK keywords should be considered deprecated syntax; we recommend using the DROP PUBLIC DATABASE LINK and NO DROP PUBLIC DATABASE LINK syntax options.

EXEMPT ACCESS POLICY

A user who holds the EXEMPT ACCESS POLICY privilege is exempt from fine-grained access control (DBMS_RLS) policies. A user who holds these privileges will be able to view or modify any row in a table constrained by a DBMS_RLS policy. The following ALTER ROLE command grants privileges to an Advanced Server role that exempt the specified role from any defined DBMS_RLS policies:

```
ALTER ROLE role_name WITH [POLICYEXEMPT | EXEMPT ACCESS POLICY]
```

This command is the functional equivalent of:

```
GRANT EXEMPT ACCESS POLICY TO role_name
```

Use the following command to revoke the privilege:

```
ALTER ROLE role_name WITH [NOPOLICYEXEMPT | NO EXEMPT ACCESS POLICY]
```

Please note: the POLICYEXEMPT and NOPOLICYEXEMPT keywords should be considered deprecated syntax; we recommend using the EXEMPT ACCESS POLICY and NO EXEMPT ACCESS POLICY syntax options.

See Also

CREATE ROLE, DROP ROLE, GRANT, REVOKE, SET ROLE

ALTER SEQUENCE

Name

ALTER SEQUENCE -- change the definition of a sequence generator

Synopsis

ALTER SEQUENCE *name* [INCREMENT BY *increment*]

[MINVALUE *minvalue*] [MAXVALUE *maxvalue*]

[CACHE *cache* | NOCACHE] [CYCLE]

Description

ALTER SEQUENCE changes the parameters of an existing sequence generator. Any parameter not specifically set in the ALTER SEQUENCE command retains its prior setting.

Parameters

name

The name (optionally schema-qualified) of a sequence to be altered.

increment

The clause INCREMENT BY *increment* is optional. A positive value will make an ascending sequence, a negative one a descending sequence. If unspecified, the old increment value will be maintained.

minvalue

The optional clause MINVALUE *minvalue* determines the minimum value a sequence can generate. If not specified, the current minimum value will be maintained. Note that the key words, NO MINVALUE, may be used to set this behavior back to the defaults of 1 and $-2^{63}-1$ for ascending and descending sequences, respectively, however, this term is not compatible with Oracle databases.

maxvalue

The optional clause MAXVALUE *maxvalue* determines the maximum value for the sequence. If not specified, the current maximum value will be maintained. Note that the key words, NO MAXVALUE, may be used to set this behavior back to the defaults of $2^{63}-1$ and -1 for ascending and descending sequences, respectively, however, this term is not compatible with Oracle databases.

cache

The optional clause CACHE *cache* specifies how many sequence numbers are to be preallocated and stored in memory for faster access. The minimum value is 1 (only one value can be generated at a time, i.e., NOCACHE). If unspecified, the old cache value will be maintained.

CYCLE

The CYCLE option allows the sequence to wrap around when the *maxvalue* or *minvalue* has been reached by an ascending or descending sequence respectively. If the limit is reached, the next number generated will be the *minvalue* or *maxvalue*, respectively. If not specified, the old cycle behavior will be maintained. Note that the key words, NO CYCLE, may be used to alter the sequence so that it does not recycle, however, this term is not

compatible with Oracle databases.

Notes

To avoid blocking of concurrent transactions that obtain numbers from the same sequence, ALTER SEQUENCE is never rolled back; the changes take effect immediately and are not reversible.

ALTER SEQUENCE will not immediately affect NEXTVAL results in backends, other than the current one, that have pre-allocated (cached) sequence values. They will use up all cached values prior to noticing the changed sequence parameters. The current backend will be affected immediately.

Examples

Change the increment and cache value of sequence, serial.

ALTER SEQUENCE serial INCREMENT BY 2 CACHE 5;

See Also

CREATE SEQUENCE, DROP SEQUENCE

ALTER SESSION

Name

ALTER SESSION -- change a runtime parameter

Synopsis

ALTER SESSION SET *name* = *value*

Description

The ALTER SESSION command changes runtime configuration parameters. ALTER SESSION only affects the value used by the current session. Some of these parameters are provided solely for compatibility with Oracle syntax and have no effect whatsoever on the runtime behavior of Advanced Server. Others will alter a corresponding Advanced Server database server runtime configuration parameter.

Parameters

name

Name of a settable runtime parameter. Available parameters are listed below.

value

New value of parameter.

Configuration Parameters

The following configuration parameters can be modified using the ALTER SESSION command:

NLS_DATE_FORMAT (string)

Sets the display format for date and time values as well as the rules for interpreting ambiguous date input values. Has the same effect as setting the Advanced Server datestyle runtime configuration parameter.

NLS_LANGUAGE (string)

Sets the language in which messages are displayed. Has the same effect as setting the Advanced Server

lc_messages runtime configuration parameter.

NLS_LENGTH_SEMANTICS (string)

Valid values are BYTE and CHAR. The default is BYTE. This parameter is provided for syntax compatibility only and has no effect in the Advanced Server.

OPTIMIZER_MODE (string)

Sets the default optimization mode for queries. Valid values are ALL_ROWS, CHOOSE, FIRST_ROWS, FIRST_ROWS_10, FIRST_ROWS_100, and FIRST_ROWS_1000. The default is CHOOSE. This parameter is implemented in Advanced Server.

QUERY_REWRITE_ENABLED (string)

Valid values are TRUE, FALSE, and FORCE. The default is FALSE. This parameter is provided for syntax compatibility only and has no effect in Advanced Server.

QUERY_REWRITE_INTEGRITY (string)

Valid values are ENFORCED, TRUSTED, and STALE_TOLERATED. The default is ENFORCED. This parameter is provided for syntax compatibility only and has no effect in Advanced Server.

Examples

Set the language to U.S. English in UTF-8 encoding. Note that in this example, the value, en_US.UTF-8, is in the format that must be specified for Advanced Server. This form is not compatible with Oracle databases.

```
ALTER SESSION SET NLS_LANGUAGE = 'en_US.UTF-8';
```

Set the date display format.

```
ALTER SESSION SET NLS_DATE_FORMAT = 'dd/mm/yyyy';
```

ALTER TABLE

Name

ALTER TABLE -- change the definition of a table

Synopsis

ALTER TABLE *name*

action [, ...]

ALTER TABLE *name*

RENAME COLUMN *column* TO *new_column*

ALTER TABLE *name*

RENAME TO *new_name*

where *action* is one of:

ADD *column type* [*column_constraint* [...]]

DROP COLUMN *column*

ADD *table_constraint*

DROP CONSTRAINT *constraint_name* [CASCADE]

Description

ALTER TABLE changes the definition of an existing table. There are several subforms:

ADD *column type*

> This form adds a new column to the table using the same syntax as CREATE TABLE.

DROP COLUMN

This form drops a column from a table. Indexes and table constraints involving the column will be automatically dropped as well.

ADD *table_constraint*

> This form adds a new constraint to a table using the same syntax as CREATE TABLE.

DROP CONSTRAINT

This form drops constraints on a table. Currently, constraints on tables are not required to have unique names, so there may be more than one constraint matching the specified name. All matching constraints will be dropped.

RENAME

The RENAME forms change the name of a table (or an index, sequence, or view) or the name of an individual column in a table. There is no effect on the stored data.

You must own the table to use ALTER TABLE.

Parameters

name

The name (possibly schema-qualified) of an existing table to alter.

column

Name of a new or existing column.

new_column

New name for an existing column.

new_name

New name for the table.

type

Data type of the new column.

table_constraint

New table constraint for the table.

constraint_name

Name of an existing constraint to drop.

CASCADE

Automatically drop objects that depend on the dropped constraint.

Notes

When you invoke ADD COLUMN, all existing rows in the table are initialized with the column's default value (null if no DEFAULT clause is specified). Adding a column with a non-null default will require the entire table to be rewritten. This may take a significant amount of time for a large table; and it will temporarily require double the disk space. Adding a CHECK or NOT NULL constraint requires scanning the table to verify that existing rows meet the constraint.

The DROP COLUMN form does not physically remove the column, but simply makes it invisible to SQL operations. Subsequent insert and update operations in the table will store a null value for the column. Thus, dropping a column is quick but it will not immediately reduce the on-disk size of your table, as the space occupied by the dropped column is not reclaimed. The space will be reclaimed over time as existing rows are updated.

Changing any part of a system catalog table is not permitted. Refer to CREATE TABLE for a further description of valid parameters.

Examples

To add a column of type VARCHAR2 to a table:

```
ALTER TABLE emp ADD address VARCHAR2(30);
```

To drop a column from a table:

```
ALTER TABLE emp DROP COLUMN address;
```

To rename an existing column:

```
ALTER TABLE emp RENAME COLUMN address TO city;
```

To rename an existing table:

```
ALTER TABLE emp RENAME TO employee;
```

To add a check constraint to a table:

```
ALTER TABLE emp ADD CONSTRAINT sal_chk CHECK (sal > 500);
```

To remove a check constraint from a table:

```
ALTER TABLE emp DROP CONSTRAINT sal_chk;
```

See Also

CREATE TABLE, DROP TABLE

ALTER TABLESPACE

Name

ALTER TABLESPACE -- change the definition of a tablespace

Synopsis

ALTER TABLESPACE *name* RENAME TO *newname*

Description

ALTER TABLESPACE changes the definition of a tablespace.

Parameters

name

The name of an existing tablespace.

newname

The new name of the tablespace. The new name cannot begin with pg_, as such names are reserved for system tablespaces.

Examples

Rename tablespace empspace to employee_space:

ALTER TABLESPACE empspace RENAME TO employee_space;

See Also

DROP TABLESPACE

ALTER USER... IDENTIFIED BY

Name

ALTER USER -- change a database user account

Synopsis

ALTER USER *role_name* IDENTIFIED BY *password* REPLACE *prev_password*

Description

A role without the CREATEROLE privilege may use this command to change their own password. An unprivileged role must include the REPLACE clause and their previous password if PASSWORD_VERIFY_FUNCTION is not NULL in their profile. When the REPLACE clause is used by a non-superuser, the server will compare the password provided to the existing password and raise an error if the passwords do not match.

A database superuser can use this command to change the password associated with any role. If a superuser includes the REPLACE clause, the clause is ignored; a non-matching value for the previous password will not throw an error.

If the role for which the password is being changed has the SUPERUSER attribute, then a superuser must issue this command. A role with the CREATEROLE attribute can use this command to change the password associated with a role that is not a superuser.

Parameters

role_name

The name of the role whose password is to be altered.

password

The role's new password.

prev_password

The role's previous password.

Examples

Change a user password:

ALTER USER john IDENTIFIED BY xyRP35z REPLACE 23PJ74a;

See Also

CREATE USER, DROP USER

ALTER USER|ROLE… PROFILE MANAGEMENT CLAUSES

Name

ALTER USER|ROLE

Synopsis

> ALTER USER|ROLE *name* [[WITH] option[…]

> where option can be the following compatible clauses:
>
> PROFILE *profile_name*
>
> | ACCOUNT {LOCK|UNLOCK}
>
> | PASSWORD EXPIRE [AT '*timestamp*']
>
> or option can be the following non-compatible clauses:
>
> | PASSWORD SET AT '*timestamp*'
>
> | LOCK TIME '*timestamp*' | STORE PRIOR PASSWORD {*'password' 'timestamp*} [, ...]

For information about the administrative clauses of the ALTER USER or ALTER ROLE command that are supported by Advanced Server, please see the PostgreSQL core documentation available at:

https://www.postgresql.org/docs/10/static/sql-commands.html

Only a database superuser can use the ALTER USER|ROLE clauses that enforce profile management. The clauses enforce the following behaviors:

> Include the PROFILE clause and a *profile_name* to associate a pre-defined profile with a role, or to change which pre-defined profile is associated with a user.
>
> Include the ACCOUNT clause and the LOCK or UNLOCK keyword to specify that the user account should be placed in a locked or unlocked state.
>
> Include the LOCK TIME '*timestamp*' clause and a date/time value to lock the role at the specified time, and unlock the role at the time indicated by the PASSWORD_LOCK_TIME parameter of the profile assigned to this role. If LOCK TIME is used with the ACCOUNT LOCK clause, the role can only be unlocked by a database superuser with the ACCOUNT UNLOCK clause.
>
> Include the PASSWORD EXPIRE clause with the AT '*timestamp*' keywords to specify a date/time when the password associated with the role will expire. If you omit the AT '*timestamp*' keywords, the password will expire immediately.
>
> Include the PASSWORD SET AT '*timestamp*' keywords to set the password modification date to the time specified.

> Include the STORE PRIOR PASSWORD {*'password' 'timestamp*} [, ...] clause to modify the password history, adding the new password and the time the password was set.

Each login role may only have one profile. To discover the profile that is currently associated with a login role, query the profile column of the DBA_USERS view.

Parameters

name

The name of the role with which the specified profile will be associated.

password

> The password associated with the role.

profile_name

The name of the profile that will be associated with the role.

timestamp

> The date and time at which the clause will be enforced. When specifying a value for *timestamp*, enclose the value in single-quotes.

Notes

For information about the Postgres-compatible clauses of the ALTER USER or ALTER ROLE command, see the PostgreSQL core documentation available at:

https://www.postgresql.org/docs/10/static/sql-alterrole.html

Examples

The following command uses the ALTER USER… PROFILE command to associate a profile named acctg with a user named john:

```
ALTER USER john PROFILE acctg_profile;
```

The following command uses the ALTER ROLE… PROFILE command to associate a profile named acctg with a user named john:

```
ALTER ROLE john PROFILE acctg_profile;
```

CALL

Name

CALL

Synopsis

CALL *procedure_name* '('[*argument_list*]')'

Description

Use the CALL statement to invoke a procedure. To use the CALL statement, you must have EXECUTE privileges on the procedure that the CALL statement is invoking.

Parameters

procedure_name

procedure_name is the (optionally schema-qualified) procedure name.

argument_list

> *argument_list* specifies a comma-separated list of arguments required by the procedure. Note that each member of *argument_list* corresponds to a formal argument expected by the procedure. Each formal argument may be an IN parameter, an OUT parameter, or an INOUT parameter.

Examples

The CALL statement may take one of several forms, depending on the arguments required by the procedure:

CALL update_balance(); CALL update_balance(1,2,3);

COMMENT

Name

COMMENT -- define or change the comment of an object

Synopsis

COMMENT ON

{

TABLE *table_name* |

COLUMN *table_name*.*column_name*

} IS '*text*'

Description

COMMENT stores a comment about a database object. To modify a comment, issue a new COMMENT command for the same object. Only one comment string is stored for each object. To remove a comment, specify the empty string (two consecutive single quotes with no intervening space) for *text*. Comments are automatically dropped when the object is dropped.

Parameters

table_name

The name of the table to be commented. The table name may be schema-qualified.

table_name.*column_name*

The name of a column within *table_name* to be commented. The table name may be schema-qualified.

text

The new comment.

Notes

There is presently no security mechanism for comments: any user connected to a database can see all the comments for objects in that database (although only superusers can change comments for objects that they don’t own). *Do not put security-critical information in comments.*

Examples

Attach a comment to the table emp:

COMMENT ON TABLE emp IS 'Current employee information';

Attach a comment to the empno column of the emp table:

COMMENT ON COLUMN emp.empno IS 'Employee identification number';

Remove these comments:

COMMENT ON TABLE emp IS '';

COMMENT ON COLUMN emp.empno IS '';

COMMIT

Name

COMMIT -- commit the current transaction

Synopsis

COMMIT [WORK]

Description

COMMIT commits the current transaction. All changes made by the transaction become visible to others and are guaranteed to be durable if a crash occurs.

Parameters

WORK

Optional key word - has no effect.

Notes

Use ROLLBACK to abort a transaction. Issuing COMMIT when not inside a transaction does no harm.

Examples

To commit the current transaction and make all changes permanent:

COMMIT;

See Also

ROLLBACK, ROLLBACK TO SAVEPOINT

CREATE DATABASE

Name

CREATE DATABASE -- create a new database

Synopsis

CREATE DATABASE *name*

Description

CREATE DATABASE creates a new database.

To create a database, you must be a superuser or have the special CREATEDB privilege. Normally, the creator becomes the owner of the new database. Non-superusers with CREATEDB privilege can only create databases owned by them.

The new database will be created by cloning the standard system database template1.

Parameters

name

The name of the database to be created.

Notes

CREATE DATABASE cannot be executed inside a transaction block.

Errors along the line of “could not initialize database directory” are most likely related to insufficient permissions on the data directory, a full disk, or other file system problems.

Examples

To create a new database:

CREATE DATABASE employees;

CREATE [PUBLIC] DATABASE LINK

Name

CREATE [PUBLIC] DATABASE LINK -- create a new database link.

Synopsis

CREATE [PUBLIC] DATABASE LINK *name*

CONNECT TO { CURRENT_USER |

username IDENTIFIED BY '*password*'}

USING { postgres_fdw '*fdw_connection_string*' |

[oci] '*oracle_connection_string*' }

Description

CREATE DATABASE LINK creates a new database link. A database link is an object that allows a reference to a table or view in a remote database within a DELETE, INSERT, SELECT or UPDATE command. A database link is referenced by appending *@dblink* to the table or view name referenced in the SQL command where *dblink* is the name of the database link.

Database links can be public or private. A *public database link* is one that can be used by any user. A *private database link* can be used only by the database link’s owner. Specification of the PUBLIC option creates a public database link. If omitted, a private database link is created.

When the CREATE DATABASE LINK command is given, the database link name and the given connection attributes are stored in the Advanced Server system table named, pg_catalog.edb_dblink. When using a given database link, the database containing the edb_dblink entry defining this database link is called the *local database*. The server and database whose connection attributes are defined within the edb_dblink entry is called the *remote database*

A SQL command containing a reference to a database link must be issued while connected to the local database. When the SQL command is executed, the appropriate authentication and connection is made to the remote database to access the table or view to which the *@dblink* reference is appended.

Note: A database link cannot be used to access a remote database within a standby database server. Standby database servers are used for high availability, load balancing, and replication.

For information about high availability, load balancing, and replication for Postgres database servers, see the PostgreSQL core documentation available at:

https://www.postgresql.org/docs/10/static/high-availability.html

Note: For Advanced Server 10, the CREATE DATABASE LINK command is tested against and certified for use with Oracle version 10g Release 2 (10.2), Oracle version 11g Release 2 (11.2), and Oracle version 12c Release 1 (12.1).

Note: The oci_dblink uses transaction isolation level SERIALIZABLE on the Oracle side, which corresponds to PostgreSQL's REPEATABLE READ. This is necessary as a single PostgreSQL statement can lead to multiple Oracle queries and thereby uses a serializable isolation level to provide consistent results.

A serialization failure may occur due to a table modification concurrent with long-running DML transactions (for example ADD, UPDATE, or DELETE statements). If such a failure occurs, the OCI reports ORA-08177: can't serialize access for this transaction, and the application must retry the transaction.

Parameters

PUBLIC

Create a public database link that can be used by any user. If omitted, then the database link is private and can only be used by the database link's owner.

name

The name of the database link.

username

The username to be used for connecting to the remote database.

CURRENT_USER

> Include CURRENT_USER to specify that Advanced Server should use the user mapping associated with the role that is using the link when establishing a connection to the remote server.

password

The password for *username*.

postgres_fdw

Specifies foreign data wrapper postgres_fdw as the connection to a remote Advanced Server database. If postgres_fdw has not been installed on the database, use the CREATE EXTENSION command to install postgres_fdw. For more information, please see the CREATE EXTENSION command in the PostgreSQL Core documentation at:

https://www.postgresql.org/docs/10/static/sql-createextension.html

fdw_connection_string

Specify the connection information for the postgres_fdw foreign data wrapper.

oci

Specifies a connection to a remote Oracle database. This is Advanced Server's default behavior.

oracle_connection_string

Specify the connection information for an oci connection.

Notes

To create a non-public database link you must have the CREATE DATABASE LINK privilege. To create a public database link you must have the CREATE PUBLIC DATABASE LINK privilege.

Setting up an Oracle Instant Client for oci-dblink

In order to use oci-dblink, an Oracle instant client must be downloaded and installed on the host running the Advanced Server database in which the database link is to be created.

An instant client can be downloaded from the following site:

http://www.oracle.com/technetwork/database/features/instant-client/index-097480.html

Oracle Instant Client for Linux

The following instructions apply to Linux hosts running Advanced Server.

Be sure the libaio library (the Linux-native asynchronous I/O facility) has already been installed on the Linux host running Advanced Server.

The libaio library can be installed with the following command:

```
yum install libaio
```

If the Oracle instant client that you've downloaded does not include the file specifically named libclntsh.so without a version number suffix, you must create a symbolic link named libclntsh.so that points to the downloaded version of the library file. Navigate to the instant client directory and execute the following command:

```
ln -s libclntsh.so.version libclntsh.so
```

Where *version* is the version number of the libclntsh.so library. For example:

```
ln -s libclntsh.so.12.1 libclntsh.so
```

When you are executing a SQL command that references a database link to a remote Oracle database, Advanced Server must know where the Oracle instant client library resides on the Advanced Server host.

The LD_LIBRARY_PATH environment variable must include the path to the Oracle client installation directory containing the libclntsh.so file. For example, assuming the installation directory containing libclntsh.so is /tmp/instantclient:

```
export LD_LIBRARY_PATH=/tmp/instantclient:$LD_LIBRARY_PATH
```

Note: This LD_LIBRARY_PATH environment variable setting must be in effect when the pg_ctl utility is executed to start or restart Advanced Server.

If you are running the current session as the user account (for example, enterprisedb) that will directly invoke pg_ctl to start or restart Advanced Server, then be sure to set LD_LIBRARY_PATH before invoking pg_ctl.

You can set LD_LIBRARY_PATH within the .bash_profile file under the home directory of the enterprisedb user

account (that is, set LD_LIBRARY_PATH within file ~enterprisedb/.bash_profile). In this manner, LD_LIBRARY_PATH will be set when you log in as enterprisedb.

If however, you are using a Linux service script with the systemctl or service command to start or restart Advanced Server, LD_LIBRARY_PATH must be set within the service script so it is in effect when the script invokes the pg_ctl utility.

The particular script file that needs to be modified to include the LD_LIBRARY_PATH setting depends upon the Advanced Server version, the Linux system on which it was installed, and whether it was installed with the graphical installer or an RPM package.

See the appropriate version of the *EDB Postgres Advanced Server Installation Guide* to determine the service script that affects the startup environment. The installation guides can be found at the following location:

https://www.enterprisedb.com/resources/product-documentation

Oracle Instant Client for Windows

The following instructions apply to Windows hosts running Advanced Server.

When you are executing a SQL command that references a database link to a remote Oracle database, Advanced Server must know where the Oracle instant client library resides on the Advanced Server host.

Set the Windows PATH system environment variable to include the Oracle client installation directory that contains the oci.dll file.

As an alternative you, can set the value of the oracle_home configuration parameter in the postgresql.conf file. The value specified in the oracle_home configuration parameter will override the Windows PATH environment variable.

To set the oracle_home configuration parameter in the postgresql.conf file, edit the file, adding the following line:

```
oracle_home = 'lib_directory '
```

Substitute the name of the Windows directory that contains oci.dll for *lib_directory*. For example:

```
oracle_home = 'C:/tmp/instantclient_10_2'
```

After setting the PATH environment variable or the oracle_home configuration parameter, you must restart the server for the changes to take effect. Restart the server from the Windows Services console.

Examples

Creating an oci-dblink Database Link

The following example demonstrates using the CREATE DATABASE LINK command to create a database link (named chicago) that connects an instance of Advanced Server to an Oracle server via an oci-dblink connection. The connection information tells Advanced Server to log in to Oracle as user admin, whose password is mypassword. Including the oci option tells Advanced Server that this is an oci-dblink connection; the connection string, '//127.0.0.1/acctg' specifies the server address and name of the database.

```
CREATE DATABASE LINK chicago

CONNECT TO admin IDENTIFIED BY 'mypassword'

USING oci '//127.0.0.1/acctg';
```

Note: You can specify a hostname in the connection string (in place of an IP address).

Creating a postgres_fdw Database Link

The following example demonstrates using the CREATE DATABASE LINK command to create a database link (named bedford) that connects an instance of Advanced Server to another Advanced Server instance via a

postgres_fdw foreign data wrapper connection. The connection information tells Advanced Server to log in as user admin, whose password is mypassword. Including the postgres_fdw option tells Advanced Server that this is a postgres_fdw connection; the connection string, 'host=127.0.0.1 port=5444 dbname=marketing' specifies the server address and name of the database.

CREATE DATABASE LINK bedford

CONNECT TO admin IDENTIFIED BY 'mypassword'

USING postgres_fdw 'host=127.0.0.1 port=5444 dbname=marketing';

Note: You can specify a hostname in the connection string (in place of an IP address).

Using a Database Link

The following examples demonstrate using a database link with Advanced Server to connect to an Oracle database. The examples assume that a copy of the Advanced Server sample application's emp table has been created in an Oracle database and a second Advanced Server database cluster with the sample application is accepting connections at port 5443.

Create a public database link named, oralink, to an Oracle database named, xe, located at 127.0.0.1 on port 1521. Connect to the Oracle database with username, edb, and password, password.

CREATE PUBLIC DATABASE LINK oralink CONNECT TO edb IDENTIFIED BY 'password' USING '//127.0.0.1:1521/xe';

Issue a SELECT command on the emp table in the Oracle database using database link, oralink.

SELECT * FROM emp@oralink;

empno | ename | job | mgr | hiredate | sal | comm | deptno

-------+--------+-----------+------+--------------------+------+------+--------

7369 | SMITH | CLERK | 7902 | 17-DEC-80 00:00:00 | 800 | | 20

7499 | ALLEN | SALESMAN | 7698 | 20-FEB-81 00:00:00 | 1600 | 300 | 30

7521 | WARD | SALESMAN | 7698 | 22-FEB-81 00:00:00 | 1250 | 500 | 30

7566 | JONES | MANAGER | 7839 | 02-APR-81 00:00:00 | 2975 | | 20

7654 | MARTIN | SALESMAN | 7698 | 28-SEP-81 00:00:00 | 1250 | 1400 | 30

7698 | BLAKE | MANAGER | 7839 | 01-MAY-81 00:00:00 | 2850 | | 30

7782 | CLARK | MANAGER | 7839 | 09-JUN-81 00:00:00 | 2450 | | 10

7788 | SCOTT | ANALYST | 7566 | 19-APR-87 00:00:00 | 3000 | | 20

7839 | KING | PRESIDENT | | 17-NOV-81 00:00:00 | 5000 | | 10

7844 | TURNER | SALESMAN | 7698 | 08-SEP-81 00:00:00 | 1500 | 0 | 30

7876 | ADAMS | CLERK | 7788 | 23-MAY-87 00:00:00 | 1100 | | 20

7900 | JAMES | CLERK | 7698 | 03-DEC-81 00:00:00 | 950 | | 30

7902 | FORD | ANALYST | 7566 | 03-DEC-81 00:00:00 | 3000 | | 20

7934 | MILLER | CLERK | 7782 | 23-JAN-82 00:00:00 | 1300 | | 10

(14 rows)

Create a private database link named, fdwlink, to the Advanced Server database named, edb, located on host 192.168.2.22 running on port 5444. Connect to the Advanced Server database with username, enterprisedb, and password, password.

CREATE DATABASE LINK fdwlink CONNECT TO enterprisedb IDENTIFIED BY 'password' USING postgres_fdw 'host=192.168.2.22 port=5444 dbname=edb';

Display attributes of database links, oralink and fdwlink, from the local edb_dblink system table:

SELECT lnkname, lnkuser, lnkconnstr FROM pg_catalog.edb_dblink;

lnkname | lnkuser | lnkconnstr

---------+--------------+--

oralink | edb | //127.0.0.1:1521/xe

fdwlink | enterprisedb |

(2 rows)

Perform a join of the emp table from the Oracle database with the dept table from the Advanced Server database:

SELECT d.deptno, d.dname, e.empno, e.ename, e.job, e.sal, e.comm FROM emp@oralink e, dept@fdwlink d WHERE e.deptno = d.deptno ORDER BY 1, 3;

deptno | dname | empno | ename | job | sal | comm

--------+------------+-------+--------+-----------+------+------

10 | ACCOUNTING | 7782 | CLARK | MANAGER | 2450 |

10 | ACCOUNTING | 7839 | KING | PRESIDENT | 5000 |

10 | ACCOUNTING | 7934 | MILLER | CLERK | 1300 |

20 | RESEARCH | 7369 | SMITH | CLERK | 800 |

20 | RESEARCH | 7566 | JONES | MANAGER | 2975 |

20 | RESEARCH | 7788 | SCOTT | ANALYST | 3000 |

20 | RESEARCH | 7876 | ADAMS | CLERK | 1100 |

20 | RESEARCH | 7902 | FORD | ANALYST | 3000 |

30 | SALES | 7499 | ALLEN | SALESMAN | 1600 | 300

30 | SALES | 7521 | WARD | SALESMAN | 1250 | 500

30 | SALES | 7654 | MARTIN | SALESMAN | 1250 | 1400

30 | SALES | 7698 | BLAKE | MANAGER | 2850 |

30 | SALES | 7844 | TURNER | SALESMAN | 1500 | 0

30 | SALES | 7900 | JAMES | CLERK | 950 |

(14 rows)

Pushdown for an oci Database Link

When the oci-dblink is used to execute SQL statements on a remote Oracle database, there are certain

circumstances where pushdown of the processing occurs on the foreign server.

Pushdown refers to the occurrence of processing on the foreign (that is, the remote) server instead of the local client where the SQL statement was issued. Pushdown can result in performance improvement since the data is processed on the remote server before being returned to the local client.

Pushdown applies to statements with the standard SQL join operations (inner join, left outer join, right outer join, and full outer join). Pushdown still occurs even when a sort is specified on the resulting data set.

In order for pushdown to occur, certain basic conditions must be met. The tables involved in the join operation must belong to the same foreign server and use the identical connection information to the foreign server (that is, the same database link defined with the CREATE DATABASE LINK command).

In order to determine if pushdown is to be used for a SQL statement, display the execution plan by using the EXPLAIN command.

For information about the EXPLAIN command, please see the PostgreSQL Core documentation at:

https://www.postgresql.org/docs/10/static/sql-explain.html

The following examples use the database link created as shown by the following:

CREATE PUBLIC DATABASE LINK oralink CONNECT TO edb IDENTIFIED BY 'password' USING '//192.168.2.23:1521/xe';

The following example shows the execution plan of an inner join:

EXPLAIN (verbose,costs off) SELECT d.deptno, d.dname, e.empno, e.ename FROM dept@oralink d, emp@oralink e WHERE d.deptno = e.deptno ORDER BY 1, 3;

QUERY PLAN

Foreign Scan

Output: d.deptno, d.dname, e.empno, e.ename

Relations: (_dblink_dept_1 d) INNER JOIN (_dblink_emp_2 e)

Remote Query: SELECT r1.deptno, r1.dname, r2.empno, r2.ename FROM (dept r1 INNER JOIN emp r2 ON ((r1.deptno = r2.deptno))) ORDER BY r1.deptno ASC NULLS LAST, r2.empno ASC NULLS LAST

(4 rows)

Note that the INNER JOIN operation occurs under the Foreign Scan section. The output of this join is the following:

deptno | dname | empno | ename

--------+------------+-------+--------

10 | ACCOUNTING | 7782 | CLARK

10 | ACCOUNTING | 7839 | KING

10 | ACCOUNTING | 7934 | MILLER

20 | RESEARCH | 7369 | SMITH

20 | RESEARCH | 7566 | JONES

20 | RESEARCH | 7788 | SCOTT

20 | RESEARCH | 7876 | ADAMS

20 | RESEARCH | 7902 | FORD

30 | SALES | 7499 | ALLEN

30 | SALES | 7521 | WARD

30 | SALES | 7654 | MARTIN

30 | SALES | 7698 | BLAKE

30 | SALES | 7844 | TURNER

30 | SALES | 7900 | JAMES

(14 rows)

The following shows the execution plan of a left outer join:

```
EXPLAIN (verbose,costs off) SELECT d.deptno, d.dname, e.empno, e.ename FROM dept@oralink d LEFT OUTER JOIN emp@oralink e ON d.deptno = e.deptno ORDER BY 1, 3;

QUERY PLAN

------------------------------------------------------------------------------

Foreign Scan

Output: d.deptno, d.dname, e.empno, e.ename

Relations: (_dblink_dept_1 d) LEFT JOIN (_dblink_emp_2 e)

Remote Query: SELECT r1.deptno, r1.dname, r2.empno, r2.ename FROM (dept r1 LEFT JOIN emp r2 ON ((r1.deptno = r2.deptno))) ORDER BY r1.deptno ASC NULLS LAST, r2.empno ASC NULLS LAST

(4 rows)
```

The output of this join is the following:

```
deptno | dname | empno | ename

--------+------------+-------+--------

10 | ACCOUNTING | 7782 | CLARK

10 | ACCOUNTING | 7839 | KING

10 | ACCOUNTING | 7934 | MILLER

20 | RESEARCH | 7369 | SMITH

20 | RESEARCH | 7566 | JONES

20 | RESEARCH | 7788 | SCOTT

20 | RESEARCH | 7876 | ADAMS

20 | RESEARCH | 7902 | FORD

30 | SALES | 7499 | ALLEN

30 | SALES | 7521 | WARD
```

30 | SALES | 7654 | MARTIN

30 | SALES | 7698 | BLAKE

30 | SALES | 7844 | TURNER

30 | SALES | 7900 | JAMES

40 | OPERATIONS | |

(15 rows)

The following example shows a case where the entire processing is not pushed down because the emp joined table resides locally instead of on the same foreign server.

EXPLAIN (verbose,costs off) SELECT d.deptno, d.dname, e.empno, e.ename FROM dept@oralink d LEFT OUTER JOIN emp e ON d.deptno = e.deptno ORDER BY 1, 3;

QUERY PLAN

Sort

Output: d.deptno, d.dname, e.empno, e.ename

Sort Key: d.deptno, e.empno

-> Hash Left Join

Output: d.deptno, d.dname, e.empno, e.ename

Hash Cond: (d.deptno = e.deptno)

-> Foreign Scan on _dblink_dept_1 d

Output: d.deptno, d.dname, d.loc

Remote Query: SELECT deptno, dname, NULL FROM dept

-> Hash

Output: e.empno, e.ename, e.deptno

-> Seq Scan on public.emp e

Output: e.empno, e.ename, e.deptno

(13 rows)

The output of this join is the same as the previous left outer join example.

Creating a Foreign Table from a Database Link

Note: The procedure described in this section is not compatible with Oracle databases.

After you have created a database link, you can create a foreign table based upon this database link. The foreign table can then be used to access the remote table referencing it with the foreign table name instead of using the database link syntax. Using the database link requires appending *@dblink* to the table or view name referenced in the SQL command where *dblink* is the name of the database link.

This technique can be used for either an oci-dblink connection for remote Oracle access, or a postgres_fdw connection for remote Postgres access.

The following example shows the creation of a foreign table to access a remote Oracle table.

First, create a database link as previously described. The following is the creation of a database link named oralink for connecting to the Oracle database.

```
CREATE PUBLIC DATABASE LINK oralink CONNECT TO edb IDENTIFIED BY 'password' USING
'//127.0.0.1:1521/xe';
```

The following query shows the database link:

```
SELECT lnkname, lnkuser, lnkconnstr FROM pg_catalog.edb_dblink;

lnkname | lnkuser | lnkconnstr

---------+---------+--------------------

oralink | edb | //127.0.0.1:1521/xe

(1 row)
```

When you create the database link, Advanced Server creates a corresponding foreign server. The following query displays the foreign server:

```
SELECT srvname, srvowner, srvfdw, srvtype, srvoptions FROM pg_foreign_server;

srvname | srvowner | srvfdw | srvtype | srvoptions

---------+----------+--------+---------+-------------------------------

oralink | 10 | 14005 | | {connstr=//127.0.0.1:1521/xe}

(1 row)
```

For more information about foreign servers, please see the CREATE SERVER command in the PostgreSQL Core documentation at:

https://www.postgresql.org/docs/10/static/sql-createserver.html

Create the foreign table as shown by the following:

```
CREATE FOREIGN TABLE emp_ora (

empno NUMERIC(4),

ename VARCHAR(10),

job VARCHAR(9),

mgr NUMERIC(4),

hiredate TIMESTAMP WITHOUT TIME ZONE,

sal NUMERIC(7,2),

comm NUMERIC(7,2),

deptno NUMERIC(2)

)

SERVER oralink

OPTIONS (table_name 'emp', schema_name 'edb'
```

);

Note the following in the CREATE FOREIGN TABLE command:

- The name specified in the SERVER clause at the end of the CREATE FOREIGN TABLE command is the name of the foreign server, which is oralink in this example as displayed in the srvname column from the query on pg_foreign_server.
- The table name and schema name are specified in the OPTIONS clause by the table and schema options.
- The column names specified in the CREATE FOREIGN TABLE command must match the column names in the remote table.
- Generally, CONSTRAINT clauses may not be accepted or enforced on the foreign table as they are assumed to have been defined on the remote table.

For more information about the CREATE FOREIGN TABLE command, please see the PostgreSQL Core documentation at:

https://www.postgresql.org/docs/10/static/sql-createforeigntable.html

The following is a query on the foreign table:

SELECT * FROM emp_ora;

empno | ename | job | mgr | hiredate | sal | comm | deptno

-------+--------+-----------+------+--------------------+---------+---------+--------

7369 | SMITH | CLERK | 7902 | 17-DEC-80 00:00:00 | 800.00 | | 20

7499 | ALLEN | SALESMAN | 7698 | 20-FEB-81 00:00:00 | 1600.00 | 300.00 | 30

7521 | WARD | SALESMAN | 7698 | 22-FEB-81 00:00:00 | 1250.00 | 500.00 | 30

7566 | JONES | MANAGER | 7839 | 02-APR-81 00:00:00 | 2975.00 | | 20

7654 | MARTIN | SALESMAN | 7698 | 28-SEP-81 00:00:00 | 1250.00 | 1400.00 | 30

7698 | BLAKE | MANAGER | 7839 | 01-MAY-81 00:00:00 | 2850.00 | | 30

7782 | CLARK | MANAGER | 7839 | 09-JUN-81 00:00:00 | 2450.00 | | 10

7788 | SCOTT | ANALYST | 7566 | 19-APR-87 00:00:00 | 3000.00 | | 20

7839 | KING | PRESIDENT | | 17-NOV-81 00:00:00 | 5000.00 | | 10

7844 | TURNER | SALESMAN | 7698 | 08-SEP-81 00:00:00 | 1500.00 | 0.00 | 30

7876 | ADAMS | CLERK | 7788 | 23-MAY-87 00:00:00 | 1100.00 | | 20

7900 | JAMES | CLERK | 7698 | 03-DEC-81 00:00:00 | 950.00 | | 30

7902 | FORD | ANALYST | 7566 | 03-DEC-81 00:00:00 | 3000.00 | | 20

7934 | MILLER | CLERK | 7782 | 23-JAN-82 00:00:00 | 1300.00 | | 10

(14 rows)

In contrast, the following is a query on the same remote table, but using the database link instead of the foreign table:

SELECT * FROM emp@oralink;

empno | ename | job | mgr | hiredate | sal | comm | deptno

-------+--------+-----------+------+-------------------+------+------+--------

7369 | SMITH | CLERK | 7902 | 17-DEC-80 00:00:00 | 800 | | 20

7499 | ALLEN | SALESMAN | 7698 | 20-FEB-81 00:00:00 | 1600 | 300 | 30

7521 | WARD | SALESMAN | 7698 | 22-FEB-81 00:00:00 | 1250 | 500 | 30

7566 | JONES | MANAGER | 7839 | 02-APR-81 00:00:00 | 2975 | | 20

7654 | MARTIN | SALESMAN | 7698 | 28-SEP-81 00:00:00 | 1250 | 1400 | 30

7698 | BLAKE | MANAGER | 7839 | 01-MAY-81 00:00:00 | 2850 | | 30

7782 | CLARK | MANAGER | 7839 | 09-JUN-81 00:00:00 | 2450 | | 10

7788 | SCOTT | ANALYST | 7566 | 19-APR-87 00:00:00 | 3000 | | 20

7839 | KING | PRESIDENT | | 17-NOV-81 00:00:00 | 5000 | | 10

7844 | TURNER | SALESMAN | 7698 | 08-SEP-81 00:00:00 | 1500 | 0 | 30

7876 | ADAMS | CLERK | 7788 | 23-MAY-87 00:00:00 | 1100 | | 20

7900 | JAMES | CLERK | 7698 | 03-DEC-81 00:00:00 | 950 | | 30

7902 | FORD | ANALYST | 7566 | 03-DEC-81 00:00:00 | 3000 | | 20

7934 | MILLER | CLERK | 7782 | 23-JAN-82 00:00:00 | 1300 | | 10

(14 rows)

Note: For backward compatibility reasons, it is still possible to write USING libpq rather than USING postgres_fdw. However, the libpq connector is missing many important optimizations which are present in the postgres_fdw connector. Therefore, the postgres_fdw connector should be used whenever possible. The libpq option is deprecated and may be removed entirely in a future Advanced Server release.

See Also

DROP DATABASE LINK

CREATE DIRECTORY

Name

CREATE DIRECTORY -- create an alias for a file system directory path

Synopsis

CREATE DIRECTORY *name* AS '*pathname*'

Description

The CREATE DIRECTORY command creates an alias for a file system directory pathname. You must be a database superuser to use this command.

When the alias is specified as the appropriate parameter to the programs of the UTL_FILE package, the operating system files are created in, or accessed from the directory corresponding to the given alias.

Parameters

name

The directory alias name.

pathname

The fully-qualified directory path represented by the alias name. The CREATE DIRECTORY command does not create the operating system directory. The physical directory must be created independently using the appropriate operating system commands.

Notes

The operating system user id, enterprisedb, must have the appropriate read and/or write privileges on the directory if the UTL_FILE package is to be used to create and/or read files using the directory.

The directory alias is stored in the pg_catalog.edb_dir system catalog table. Note that edb_dir is not a table compatible with Oracle databases.

The directory alias can also be viewed from the Oracle catalog views SYS.ALL_DIRECTORIES and SYS.DBA_DIRECTORIES, which are compatible with Oracle databases.

Use the DROP DIRECTORY command to delete the directory alias. When a directory alias is deleted, the corresponding physical file system directory is not affected. The file system directory must be deleted using the appropriate operating system commands.

In a Linux system, the directory name separator is a forward slash (/).

In a Windows system, the directory name separator can be specified as a forward slash (/) or two consecutive backslashes (\\).

Examples

Create an alias named empdir for directory /tmp/empdir on Linux:

CREATE DIRECTORY empdir AS '/tmp/empdir';

Create an alias named empdir for directory C:\TEMP\EMPDIR on Windows:

CREATE DIRECTORY empdir AS 'C:/TEMP/EMPDIR';

View all of the directory aliases:

SELECT * FROM pg_catalog.edb_dir;

dirname | dirowner | dirpath | diracl

---------+----------+----------------+--------

empdir | 10 | C:/TEMP/EMPDIR |

(1 row)

View the directory aliases using a view compatible with Oracle databases:

SELECT * FROM SYS.ALL_DIRECTORIES;

owner | directory_name | directory_path

--------------+----------------+----------------

ENTERPRISEDB | EMPDIR | C:/TEMP/EMPDIR

(1 row)

See Also

DROP DIRECTORY

CREATE FUNCTION

Name

CREATE FUNCTION -- define a new function

Synopsis

CREATE [OR REPLACE] FUNCTION *name* [(*parameters*)]

RETURN *data_type*

[

> IMMUTABLE | STABLE | VOLATILE | DETERMINISTIC | [NOT] LEAKPROOF
>
> | CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT
>
> | [EXTERNAL] SECURITY INVOKER | [EXTERNAL] SECURITY DEFINER
>
> | AUTHID DEFINER | AUTHID CURRENT_USER
>
> | PARALLEL { UNSAFE | RESTRICTED | SAFE }
>
> | COST *execution_cost*
>
> | ROWS *result_rows* | SET *configuration_parameter* { TO *value* | = *value* | FROM CURRENT } ...] { IS | AS }

[*declarations*]

BEGIN

statements

END [*name*];

Description

CREATE FUNCTION defines a new function. CREATE OR REPLACE FUNCTION will either create a new function, or replace an existing definition.

If a schema name is included, then the function is created in the specified schema. Otherwise it is created in the current schema. The name of the new function must not match any existing function with the same argument types in the same schema. However, functions of different input argument types may share a name (this is called overloading). (Overloading of functions is an Advanced Server feature - overloading of stored functions is not compatible with Oracle databases.)

To update the definition of an existing function, use CREATE OR REPLACE FUNCTION. It is not possible to change the name or argument types of a function this way (if you tried, you would actually be creating a new, distinct function). Also, CREATE OR REPLACE FUNCTION will not let you change the return type of an existing function. To do that, you must drop and recreate the function.

The user that creates the function becomes the owner of the function.

Parameters

name

> *name* is the identifier of the function. If you specify the [OR REPLACE] clause and a function with the same name already exists in the schema, the new function will replace the existing one. If you do not specify [OR REPLACE], the new function will not replace the existing function with the same name in the same schema.

parameters

> *parameters* is a list of formal parameters.

data_type

data_type is the data type of the value returned by the function's RETURN statement.

declarations

> *declarations* are variable, cursor, type, or subprogram declarations. If subprogram declarations are included, they must be declared after all other variable, cursor, and type declarations.

statements

> *statements* are SPL program statements (the BEGIN - END block may contain an EXCEPTION section).

IMMUTABLE

STABLE

VOLATILE

> These attributes inform the query optimizer about the behavior of the function; you can specify only one choice. VOLATILE is the default behavior.
>
> IMMUTABLE indicates that the function cannot modify the database and always reaches the same result when given the same argument values; it does not do database lookups or otherwise use information not directly present in its argument list. If you include this clause, any call of the function with all-constant arguments can be immediately replaced with the function value.
>
> STABLE indicates that the function cannot modify the database, and that within a single table scan, it will consistently return the same result for the same argument values, but that its result could change across SQL statements. This is the appropriate selection for function that depend on database lookups, parameter variables (such as the current time zone), etc.
>
> VOLATILE indicates that the function value can change even within a single table scan, so no optimizations can be made. Please note that any function that has side-effects must be classified volatile, even if its result is quite predictable, to prevent calls from being optimized away.

DETERMINISTIC

> DETERMINISTIC is a synonym for IMMUTABLE. A DETERMINISTIC function cannot modify the database and always reaches the same result when given the same argument values; it does not do database lookups or otherwise use information not directly present in its argument list. If you include this clause, any call of the function with all-constant arguments can be immediately replaced with the function value.

[NOT] LEAKPROOF

> A LEAKPROOF function has no side effects, and reveals no information about the values used to call the function.

CALLED ON NULL INPUT RETURNS NULL ON NULL INPUT STRICT

> CALLED ON NULL INPUT (the default) indicates that the procedure will be called normally when some of its arguments are NULL. It is the author's responsibility to check for NULL values if necessary and respond appropriately.

RETURNS NULL ON NULL INPUT or STRICT indicates that the procedure always returns NULL whenever any of its arguments are NULL. If these clauses are specified, the procedure is not executed when there are NULL arguments; instead a NULL result is assumed automatically.

[EXTERNAL] SECURITY DEFINER

SECURITY DEFINER specifies that the function will execute with the privileges of the user that created it; this is the default. The key word EXTERNAL is allowed for SQL conformance, but is optional.

[EXTERNAL] SECURITY INVOKER

The SECURITY INVOKER clause indicates that the function will execute with the privileges of the user that calls it. The key word EXTERNAL is allowed for SQL conformance, but is optional.

AUTHID DEFINER

AUTHID CURRENT_USER

The AUTHID DEFINER clause is a synonym for [EXTERNAL] SECURITY DEFINER. If the AUTHID clause is omitted or if AUTHID DEFINER is specified, the rights of the function owner are used to determine access privileges to database objects.

The AUTHID CURRENT_USER clause is a synonym for [EXTERNAL] SECURITY INVOKER. If AUTHID CURRENT_USER is specified, the rights of the current user executing the function are used to determine access privileges.

PARALLEL { UNSAFE | RESTRICTED | SAFE }

The PARALLEL clause enables the use of parallel sequential scans (parallel mode). A parallel sequential scan uses multiple workers to scan a relation in parallel during a query in contrast to a serial sequential scan.

When set to UNSAFE, the function cannot be executed in parallel mode. The presence of such a function in a SQL statement forces a serial execution plan. This is the default setting if the PARALLEL clause is omitted.

When set to RESTRICTED, the function can be executed in parallel mode, but the execution is restricted to the parallel group leader. If the qualification for any particular relation has anything that is parallel restricted, that relation won't be chosen for parallelism.

When set to SAFE, the function can be executed in parallel mode with no restriction.

COST *execution_cost*

execution_cost is a positive number giving the estimated execution cost for the function, in units of cpu_operator_cost. If the function returns a set, this is the cost per returned row. Larger values cause the planner to try to avoid evaluating the function more often than necessary.

ROWS *result_rows*

result_rows is a positive number giving the estimated number of rows that the planner should expect the function to return. This is only allowed when the function is declared to return a set. The default assumption is 1000 rows.

SET *configuration_parameter* { TO *value* | = *value* | FROM CURRENT }

The SET clause causes the specified configuration parameter to be set to the specified value when the function is entered, and then restored to its prior value when the function exits. SET FROM CURRENT saves the session's current value of the parameter as the value to be applied when the function is entered.

If a SET clause is attached to a function, then the effects of a SET LOCAL command executed inside the function for the same variable are restricted to the function; the configuration parameter's prior value is restored at function exit. An ordinary SET command (without LOCAL) overrides the SET clause, much as it would do for a previous SET LOCAL command, with the effects of such a command persisting after procedure exit, unless the current transaction is rolled back.

Please Note: The STRICT, LEAKPROOF, PARALLEL, COST, ROWS and SET keywords provide extended functionality for Advanced Server and are not supported by Oracle.

Notes

Advanced Server allows function overloading; that is, the same name can be used for several different functions so long as they have distinct input (IN, IN OUT) argument data types.

Examples

The function emp_comp takes two numbers as input and returns a computed value. The SELECT command illustrates use of the function.

```
CREATE OR REPLACE FUNCTION emp_comp (

p_sal NUMBER,

p_comm NUMBER

) RETURN NUMBER

IS

BEGIN

RETURN (p_sal + NVL(p_comm, 0)) * 24;

END;

SELECT ename "Name", sal "Salary", comm "Commission", emp_comp(sal, comm)

"Total Compensation" FROM emp;
```

Name | Salary | Commission | Total Compensation

--------+---------+------------+--------------------

SMITH | 800.00 | | 19200.00

ALLEN | 1600.00 | 300.00 | 45600.00

WARD | 1250.00 | 500.00 | 42000.00

JONES | 2975.00 | | 71400.00

MARTIN | 1250.00 | 1400.00 | 63600.00

BLAKE | 2850.00 | | 68400.00

CLARK | 2450.00 | | 58800.00

SCOTT | 3000.00 | | 72000.00

KING | 5000.00 | | 120000.00

TURNER | 1500.00 | 0.00 | 36000.00

ADAMS | 1100.00 | | 26400.00

JAMES | 950.00 | | 22800.00

FORD | 3000.00 | | 72000.00

```
MILLER | 1300.00 | | 31200.00

(14 rows)
```

Function sal_range returns a count of the number of employees whose salary falls in the specified range. The following anonymous block calls the function a number of times using the arguments' default values for the first two calls.

```
CREATE OR REPLACE FUNCTION sal_range (

p_sal_min NUMBER DEFAULT 0,

p_sal_max NUMBER DEFAULT 10000

) RETURN INTEGER

IS

v_count INTEGER;

BEGIN

SELECT COUNT(*) INTO v_count FROM emp

WHERE sal BETWEEN p_sal_min AND p_sal_max;

RETURN v_count;

END;

BEGIN

DBMS_OUTPUT.PUT_LINE('Number of employees with a salary: ' ||

sal_range);

DBMS_OUTPUT.PUT_LINE('Number of employees with a salary of at least '

|| '$2000.00: ' || sal_range(2000.00));

DBMS_OUTPUT.PUT_LINE('Number of employees with a salary between '

|| '$2000.00 and $3000.00: ' || sal_range(2000.00, 3000.00));

END;

Number of employees with a salary: 14

Number of employees with a salary of at least $2000.00: 6

Number of employees with a salary between $2000.00 and $3000.00: 5
```

The following example demonstrates using the AUTHID CURRENT_USER clause and STRICT keyword in a function declaration:

```
CREATE OR REPLACE FUNCTION dept_salaries(dept_id int) RETURN NUMBER

STRICT

AUTHID CURRENT_USER

BEGIN
```

```
RETURN QUERY (SELECT sum(salary) FROM emp WHERE deptno = id);

END;
```

Include the STRICT keyword to instruct the server to return NULL if any input parameter passed is NULL; if a NULL value is passed, the function will not execute.

The dept_salaries function executes with the privileges of the role that is calling the function. If the current user does not have sufficient privileges to perform the SELECT statement querying the emp table (to display employee salaries), the function will report an error. To instruct the server to use the privileges associated with the role that defined the function, replace the AUTHID CURRENT_USER clause with the AUTHID DEFINER clause.

Pragmas

PRAGMA RESTRICT_REFERENCE

> Advanced Server accepts but ignores syntax referencing pragma restrict_reference.

See Also DROP FUNCTION

CREATE INDEX

Name

CREATE INDEX -- define a new index

Synopsis

CREATE [UNIQUE] INDEX *name* ON *table*

({ *column* | (*expression*) })

[TABLESPACE *tablespace*]

Description

CREATE INDEX constructs an index, *name*, on the specified table. Indexes are primarily used to enhance database performance (though inappropriate use will result in slower performance).

Note: An index cannot be created on a partitioned table.

The key field(s) for the index are specified as column names, or alternatively as expressions written in parentheses. Multiple fields can be specified to create multicolumn indexes.

An index field can be an expression computed from the values of one or more columns of the table row. This feature can be used to obtain fast access to data based on some transformation of the basic data. For example, an index computed on UPPER(col) would allow the clause WHERE UPPER(col) = 'JIM' to use an index.

Advanced Server provides the B-tree index method. The B-tree index method is an implementation of Lehman-Yao high-concurrency B-trees.

Indexes are not used for IS NULL clauses by default.

All functions and operators used in an index definition must be "immutable", that is, their results must depend only on their arguments and never on any outside influence (such as the contents of another table or the current time). This restriction ensures that the behavior of the index is well-defined. To use a user-defined function in an index expression remember to mark the function immutable when you create it.

Parameters

UNIQUE

Causes the system to check for duplicate values in the table when the index is created (if data already exist) and each time data is added. Attempts to insert or update data which would result in duplicate entries will generate an error.

name

The name of the index to be created. No schema name can be included here; the index is always created in the same schema as its parent table.

table

The name (possibly schema-qualified) of the table to be indexed.

column

The name of a column in the table.

expression

An expression based on one or more columns of the table. The expression usually must be written with surrounding parentheses, as shown in the syntax. However, the parentheses may be omitted if the expression has the form of a function call.

tablespace

The tablespace in which to create the index. If not specified, default_tablespace is used, or the database's default tablespace if default_tablespace is an empty string.

Notes

Up to 32 fields may be specified in a multicolumn index.

Examples

To create a B-tree index on the column, ename, in the table, emp:

CREATE INDEX name_idx ON emp (ename);

To create the same index as above, but have it reside in the index_tblspc tablespace:

CREATE INDEX name_idx ON emp (ename) TABLESPACE index_tblspc;

See Also

DROP INDEX, ALTER INDEX

CREATE MATERIALIZED VIEW

Name

CREATE MATERIALIZED VIEW -- define a new materialized view

Synopsis

CREATE MATERIALIZED VIEW *name* [*build_clause*][*create_mv_refresh*] AS subquery

> Where *build_clause* is:

> BUILD {IMMEDIATE | DEFERRED}
>
> Where *create_mv_refresh* is:
>
> REFRESH [COMPLETE] [ON DEMAND]

Description

CREATE MATERIALIZED VIEW defines a view of a query that is not updated each time the view is referenced in a query. By default, the view is populated when the view is created; you can include the BUILD DEFERRED keywords to delay the population of the view.

A materialized view may be schema-qualified; if you specify a schema name when invoking the CREATE MATERIALIZED VIEW command, the view will be created in the specified schema. The view name must be distinct from the name of any other view, table, sequence, or index in the same schema.

Parameters

name

The name (optionally schema-qualified) of a view to be created.

subquery

A SELECT statement that specifies the contents of the view. Refer to SELECT for more information about valid queries.

build_clause

Include a *build_clause* to specify when the view should be populated. Specify BUILD IMMEDIATE, or BUILD DEFERRED:

- BUILD IMMEDIATE instructs the server to populate the view immediately. This is the default behavior.
- BUILD DEFERRED instructs the server to populate the view at a later time (during a REFRESH operation).

create_mv_refresh

Include the *create_mv_refresh* clause to specify when the contents of a materialized view should be updated. The clause contains the REFRESH keyword followed by COMPLETE and/or ON DEMAND, where:

- COMPLETE instructs the server to discard the current content and reload the materialized view by executing the view's defining query when the materialized view is refreshed.
- ON DEMAND instructs the server to refresh the materialized view on demand by calling the DBMS_MVIEW package or by calling the Postgres REFRESH MATERIALIZED VIEW statement. This is the default behavior.

Notes

Materialized views are read only - the server will not allow an INSERT, UPDATE, or DELETE on a view.

Access to tables referenced in the view is determined by permissions of the view owner; the user of a view must have permissions to call all functions used by the view.

For more information about the Postgres REFRESH MATERIALIZED VIEW command, please see the PostgreSQL Core Documentation available at:

https://www.postgresql.org/docs/10/static/sql-refreshmaterializedview.html

Examples

The following statement creates a materialized view named dept_30:

```
CREATE MATERIALIZED VIEW dept_30 BUILD IMMEDIATE AS SELECT * FROM emp WHERE deptno = 30;
```

The view contains information retrieved from the emp table about any employee that works in department 30.

CREATE PACKAGE

Name

CREATE PACKAGE -- define a new package specification

Synopsis

CREATE [OR REPLACE] PACKAGE *name*

[AUTHID { DEFINER | CURRENT_USER }]

{ IS | AS }

[*declaration*;] [, ...]

[{ PROCEDURE *proc_name*

[(*argname* [IN | IN OUT | OUT] *argtype* [DEFAULT *value*]

[, ...])];

[PRAGMA RESTRICT_REFERENCES(*name*,

{ RNDS | RNPS | TRUST | WNDS | WNPS } [, ...]);]

|

FUNCTION *func_name*

[(*argname* [IN | IN OUT | OUT] *argtype* [DEFAULT *value*]

[, ...])]

RETURN *rettype* [DETERMINISTIC];

[PRAGMA RESTRICT_REFERENCES(*name*,

{ RNDS | RNPS | TRUST | WNDS | WNPS } [, ...]);]

}

] [, ...]

END [*name*]

Description

CREATE PACKAGE defines a new package specification. CREATE OR REPLACE PACKAGE will either create a new package specification, or replace an existing specification.

If a schema name is included, then the package is created in the specified schema. Otherwise it is created in the current schema. The name of the new package must not match any existing package in the same schema unless the intent is to update the definition of an existing package, in which case use CREATE OR REPLACE PACKAGE.

The user that creates the procedure becomes the owner of the package.

Parameters

name

The name (optionally schema-qualified) of the package to create.

DEFINER | CURRENT_USER

Specifies whether the privileges of the package owner (DEFINER) or the privileges of the current user executing a program in the package (CURRENT_USER) are to be used to determine whether or not access is allowed to database objects referenced in the package. DEFINER is the default.

declaration

A public variable, type, cursor, or REF CURSOR declaration.

proc_name

The name of a public procedure.

argname

The name of an argument.

IN | IN OUT | OUT

The argument mode.

argtype

The data type(s) of the program's arguments.

DEFAULT *value*

Default value of an input argument.

func_name

The name of a public function.

rettype

The return data type.

DETERMINISTIC

> DETERMINISTIC is a synonym for IMMUTABLE. A DETERMINISTIC procedure cannot modify the database and always reaches the same result when given the same argument values; it does not do database lookups or otherwise use information not directly present in its argument list. If you include this clause, any call of the procedure with all-constant arguments can be immediately replaced with the procedure value.

RNDS | RNPS | TRUST | WNDS | WNPS

The keywords are accepted for compatibility and ignored.

Examples

The package specification, empinfo, contains three public components - a public variable, a public procedure, and a public function.

```
CREATE OR REPLACE PACKAGE empinfo

IS

emp_name VARCHAR2(10);
```

PROCEDURE get_name (

p_empno NUMBER

);

FUNCTION display_counter

RETURN INTEGER;

END;

See Also

DROP PACKAGE

CREATE PACKAGE BODY

Name

CREATE BODY PACKAGE -- define a new package body

Synopsis

CREATE [OR REPLACE] PACKAGE BODY *name*

{ IS | AS }

[*declaration*;] [, ...]

[{ PROCEDURE *proc_name*

[(*argname* [IN | IN OUT | OUT] *argtype* [DEFAULT *value*]

[, ...])]

[STRICT]

[LEAKPROOF]

[PARALLEL { UNSAFE | RESTRICTED | SAFE }]

[COST *execution_cost*]

[ROWS *result_rows*]

[SET *config_param* { TO *value* | = *value* | FROM CURRENT }]

{ IS | AS }

program_body

END [*proc_name*];

|

FUNCTION *func_name*

[(*argname* [IN | IN OUT | OUT] *argtype* [DEFAULT *value*]

[, ...])]

RETURN *rettype* [DETERMINISTIC]

[STRICT]

[LEAKPROOF]

[PARALLEL { UNSAFE | RESTRICTED | SAFE }]

[COST *execution_cost*]

[ROWS *result_rows*]

[SET *config_param* { TO *value* | = *value* | FROM CURRENT }]

{ IS | AS }

program_body

END [*func_name*];

}

] [, ...]

[BEGIN

statement; [, ...]]

END [*name*]

Description

CREATE PACKAGE BODY defines a new package body. CREATE OR REPLACE PACKAGE BODY will either create a new package body, or replace an existing body.

If a schema name is included, then the package body is created in the specified schema. Otherwise it is created in the current schema. The name of the new package body must match an existing package specification in the same schema. The new package body name must not match any existing package body in the same schema unless the intent is to update the definition of an existing package body, in which case use CREATE OR REPLACE PACKAGE BODY.

Parameters

name

The name (optionally schema-qualified) of the package body to create.

declaration

A private variable, type, cursor, or REF CURSOR declaration.

proc_name

The name of a public or private procedure. If *proc_name* exists in the package specification with an identical signature, then it is public, otherwise it is private.

argname

The name of an argument.

IN | IN OUT | OUT

The argument mode.

argtype

The data type(s) of the program's arguments.

DEFAULT *value*

Default value of an input argument.

STRICT

The STRICT keyword specifies that the function will not be executed if called with a NULL argument; instead the function will return NULL.

LEAKPROOF

The LEAKPROOF keyword specifies that the function will not reveal any information about arguments, other than through a return value.

PARALLEL { UNSAFE | RESTRICTED | SAFE }

> The PARALLEL clause enables the use of parallel sequential scans (parallel mode). A parallel sequential scan uses multiple workers to scan a relation in parallel during a query in contrast to a serial sequential scan.
>
> When set to UNSAFE, the procedure or function cannot be executed in parallel mode. The presence of such a procedure or function forces a serial execution plan. This is the default setting if the PARALLEL clause is omitted.
>
> When set to RESTRICTED, the procedure or function can be executed in parallel mode, but the execution is restricted to the parallel group leader. If the qualification for any particular relation has anything that is parallel restricted, that relation won't be chosen for parallelism.
>
> When set to SAFE, the procedure or function can be executed in parallel mode with no restriction.

execution_cost

> *execution_cost* specifies a positive number giving the estimated execution cost for the function, in units of cpu_operator_cost. If the function returns a set, this is the cost per returned row. The default is 0.0025.

result_rows

> *result_rows* is the estimated number of rows that the query planner should expect the function to return. The default is 1000.

SET

> Use the *SET* clause to specify a parameter value for the duration of the function:
>
> *config_param* specifies the parameter name.
>
> *value* specifies the parameter value.
>
> *FROM CURRENT* guarantees that the parameter value is restored when the function ends.

program_body

The declarations and SPL statements that comprise the body of the function or procedure.

The declarations may include variable, type, REF CURSOR, or subprogram declarations. If subprogram declarations are included, they must be declared after all other variable, type, and REF CURSOR declarations.

func_name

The name of a public or private function. If *func_name* exists in the package specification with an identical signature, then it is public, otherwise it is private.

rettype

The return data type.

DETERMINISTIC

Include DETERMINISTIC to specify that the function will always return the same result when given the same argument values. A DETERMINISTIC function must not modify the database.

Note: The DETERMINISTIC keyword is equivalent to the PostgreSQL IMMUTABLE option.

Note: If DETERMINISTIC is specified for a public function in the package body, it must also be specified for the function declaration in the package specification. For private functions, there is no function declaration in the package specification.

statement

An SPL program statement. Statements in the package initialization section are executed once per session the first time the package is referenced.

Please Note: The STRICT, LEAKPROOF, PARALLEL, COST, ROWS and SET keywords provide extended functionality for Advanced Server and are not supported by Oracle.

Examples

The following is the package body for the empinfo package.

```
CREATE OR REPLACE PACKAGE BODY empinfo

IS

v_counter INTEGER;

PROCEDURE get_name (

p_empno NUMBER

)

IS

BEGIN

SELECT ename INTO emp_name FROM emp WHERE empno = p_empno;

v_counter := v_counter + 1;

END;

FUNCTION display_counter

RETURN INTEGER

IS

BEGIN

RETURN v_counter;

END;
```

```
BEGIN

v_counter := 0;

DBMS_OUTPUT.PUT_LINE('Initialized counter');

END;
```

The following two anonymous blocks execute the procedure and function in the empinfo package and display the public variable.

```
BEGIN

empinfo.get_name(7369);

DBMS_OUTPUT.PUT_LINE('Employee Name : ' || empinfo.emp_name);

DBMS_OUTPUT.PUT_LINE('Number of queries: ' || empinfo.display_counter);

END;

Initialized counter

Employee Name : SMITH

Number of queries: 1

BEGIN

empinfo.get_name(7900);

DBMS_OUTPUT.PUT_LINE('Employee Name : ' || empinfo.emp_name);

DBMS_OUTPUT.PUT_LINE('Number of queries: ' || empinfo.display_counter);

END;

Employee Name : JAMES

Number of queries: 2
```

See Also

CREATE PACKAGE, DROP PACKAGE

CREATE PROCEDURE

Name

CREATE PROCEDURE -- define a new stored procedure

Synopsis

CREATE [OR REPLACE] PROCEDURE *name* [(*parameters*)] [

> IMMUTABLE | STABLE | VOLATILE | DETERMINISTIC | [NOT] LEAKPROOF
>
> | CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT
>
> | [EXTERNAL] SECURITY INVOKER | [EXTERNAL] SECURITY DEFINER

| AUTHID DEFINER | AUTHID CURRENT_USER

| PARALLEL { UNSAFE | RESTRICTED | SAFE }

| COST *execution_cost*

| ROWS *result_rows* | SET *configuration_parameter* { TO *value* | = *value* | FROM CURRENT } ...] { IS | AS }

[*declarations*]

BEGIN

statements

END [*name*];

Description

CREATE PROCEDURE defines a new stored procedure. CREATE OR REPLACE PROCEDURE will either create a new procedure, or replace an existing definition.

If a schema name is included, then the procedure is created in the specified schema. Otherwise it is created in the current schema. The name of the new procedure must not match any existing procedure in the same schema unless the intent is to update the definition of an existing procedure, in which case use CREATE OR REPLACE PROCEDURE.

Parameters

name

name is the identifier of the procedure. If you specify the [OR REPLACE] clause and a procedure with the same name already exists in the schema, the new procedure will replace the existing one. If you do not specify [OR REPLACE], the new procedure will not replace the existing procedure with the same name in the same schema.

parameters

parameters is a list of formal parameters.

declarations

declarations are variable, cursor, type, or subprogram declarations. If subprogram declarations are included, they must be declared after all other variable, cursor, and type declarations.

statements

statements are SPL program statements (the BEGIN - END block may contain an EXCEPTION section).

IMMUTABLE

STABLE

VOLATILE

These attributes inform the query optimizer about the behavior of the procedure; you can specify only one choice. VOLATILE is the default behavior.

IMMUTABLE indicates that the procedure cannot modify the database and always reaches the same result when given the same argument values; it does not do database lookups or otherwise use information not directly present in its argument list. If you include this clause, any call of the procedure with all-constant arguments can be immediately replaced with the procedure value.

STABLE indicates that the procedure cannot modify the database, and that within a single table scan, it will

consistently return the same result for the same argument values, but that its result could change across SQL statements. This is the appropriate selection for procedures that depend on database lookups, parameter variables (such as the current time zone), etc.

VOLATILE indicates that the procedure value can change even within a single table scan, so no optimizations can be made. Please note that any function that has side-effects must be classified volatile, even if its result is quite predictable, to prevent calls from being optimized away.

DETERMINISTIC

DETERMINISTIC is a synonym for IMMUTABLE. A DETERMINISTIC procedure cannot modify the database and always reaches the same result when given the same argument values; it does not do database lookups or otherwise use information not directly present in its argument list. If you include this clause, any call of the procedure with all-constant arguments can be immediately replaced with the procedure value.

[NOT] LEAKPROOF

A LEAKPROOF procedure has no side effects, and reveals no information about the values used to call the procedure.

CALLED ON NULL INPUT RETURNS NULL ON NULL INPUT STRICT

CALLED ON NULL INPUT (the default) indicates that the procedure will be called normally when some of its arguments are NULL. It is the author's responsibility to check for NULL values if necessary and respond appropriately.

RETURNS NULL ON NULL INPUT or STRICT indicates that the procedure always returns NULL whenever any of its arguments are NULL. If these clauses are specified, the procedure is not executed when there are NULL arguments; instead a NULL result is assumed automatically.

[EXTERNAL] SECURITY DEFINER

SECURITY DEFINER specifies that the procedure will execute with the privileges of the user that created it; this is the default. The key word EXTERNAL is allowed for SQL conformance, but is optional.

[EXTERNAL] SECURITY INVOKER

The SECURITY INVOKER clause indicates that the procedure will execute with the privileges of the user that calls it. The key word EXTERNAL is allowed for SQL conformance, but is optional.

AUTHID DEFINER

AUTHID CURRENT_USER

The AUTHID DEFINER clause is a synonym for [EXTERNAL] SECURITY DEFINER. If the AUTHID clause is omitted or if AUTHID DEFINER is specified, the rights of the procedure owner are used to determine access privileges to database objects.

The AUTHID CURRENT_USER clause is a synonym for [EXTERNAL] SECURITY INVOKER. If AUTHID CURRENT_USER is specified, the rights of the current user executing the procedure are used to determine access privileges.

PARALLEL { UNSAFE | RESTRICTED | SAFE }

The PARALLEL clause enables the use of parallel sequential scans (parallel mode). A parallel sequential scan uses multiple workers to scan a relation in parallel during a query in contrast to a serial sequential scan.

When set to UNSAFE, the procedure cannot be executed in parallel mode. The presence of such a procedure forces a serial execution plan. This is the default setting if the PARALLEL clause is omitted.

When set to RESTRICTED, the procedure can be executed in parallel mode, but the execution is restricted to the parallel group leader. If the qualification for any particular relation has anything that is parallel restricted, that relation won't be chosen for parallelism.

> When set to SAFE, the procedure can be executed in parallel mode with no restriction.

COST *execution_cost*

> *execution_cost* is a positive number giving the estimated execution cost for the procedure, in units of cpu_operator_cost. If the procedure returns a set, this is the cost per returned row. Larger values cause the planner to try to avoid evaluating the function more often than necessary.

ROWS *result_rows*

> *result_rows* is a positive number giving the estimated number of rows that the planner should expect the procedure to return. This is only allowed when the procedure is declared to return a set. The default assumption is 1000 rows.

SET *configuration_parameter* { TO *value* | = *value* | FROM CURRENT }

> The SET clause causes the specified configuration parameter to be set to the specified value when the procedure is entered, and then restored to its prior value when the procedure exits. SET FROM CURRENT saves the session's current value of the parameter as the value to be applied when the procedure is entered.
>
> If a SET clause is attached to a procedure, then the effects of a SET LOCAL command executed inside the procedure for the same variable are restricted to the procedure; the configuration parameter's prior value is restored at procedure exit. An ordinary SET command (without LOCAL) overrides the SET clause, much as it would do for a previous SET LOCAL command, with the effects of such a command persisting after procedure exit, unless the current transaction is rolled back.

Please Note: The STRICT, LEAKPROOF, PARALLEL, COST, ROWS and SET keywords provide extended functionality for Advanced Server and are not supported by Oracle.

Examples

The following procedure lists the employees in the emp table:

```
CREATE OR REPLACE PROCEDURE list_emp

IS

v_empno NUMBER(4);

v_ename VARCHAR2(10);

CURSOR emp_cur IS

SELECT empno, ename FROM emp ORDER BY empno;

BEGIN

OPEN emp_cur;

DBMS_OUTPUT.PUT_LINE('EMPNO ENAME');

DBMS_OUTPUT.PUT_LINE('----- -------');

LOOP

FETCH emp_cur INTO v_empno, v_ename;

EXIT WHEN emp_cur%NOTFOUND;

DBMS_OUTPUT.PUT_LINE(v_empno || ' ' || v_ename);

END LOOP;
```

```
CLOSE emp_cur;

END;

EXEC list_emp;

EMPNO ENAME

----- -------

7369 SMITH

7499 ALLEN

7521 WARD

7566 JONES

7654 MARTIN

7698 BLAKE

7782 CLARK

7788 SCOTT

7839 KING

7844 TURNER

7876 ADAMS

7900 JAMES

7902 FORD

7934 MILLER
```

The following procedure uses IN OUT and OUT arguments to return an employee's number, name, and job based upon a search using first, the given employee number, and if that is not found, then using the given name. An anonymous block calls the procedure.

```
CREATE OR REPLACE PROCEDURE emp_job (

p_empno IN OUT emp.empno%TYPE,

p_ename IN OUT emp.ename%TYPE,

p_job OUT emp.job%TYPE

)

IS

v_empno emp.empno%TYPE;

v_ename emp.ename%TYPE;

v_job emp.job%TYPE;

BEGIN

SELECT ename, job INTO v_ename, v_job FROM emp WHERE empno = p_empno;
```

```
p_ename := v_ename;
p_job := v_job;
DBMS_OUTPUT.PUT_LINE('Found employee # ' || p_empno);
EXCEPTION
WHEN NO_DATA_FOUND THEN
BEGIN
SELECT empno, job INTO v_empno, v_job FROM emp
WHERE ename = p_ename;
p_empno := v_empno;
p_job := v_job;
DBMS_OUTPUT.PUT_LINE('Found employee ' || p_ename);
EXCEPTION
WHEN NO_DATA_FOUND THEN
DBMS_OUTPUT.PUT_LINE('Could not find an employee with ' ||
'number, ' || p_empno || ' nor name, ' || p_ename);
p_empno := NULL;
p_ename := NULL;
p_job := NULL;
END;
END;
DECLARE
v_empno emp.empno%TYPE;
v_ename emp.ename%TYPE;
v_job emp.job%TYPE;
BEGIN
v_empno := 0;
v_ename := 'CLARK';
emp_job(v_empno, v_ename, v_job);
DBMS_OUTPUT.PUT_LINE('Employee No: ' || v_empno);
DBMS_OUTPUT.PUT_LINE('Name : ' || v_ename);
DBMS_OUTPUT.PUT_LINE('Job : ' || v_job);
END;
```

Found employee CLARK

Employee No: 7782

Name : CLARK

Job : MANAGER

The following example demonstrates using the AUTHID DEFINER and SET clauses in a procedure declaration. The update_salary procedure conveys the privileges of the role that defined the procedure to the role that is calling the procedure (while the procedure executes):

CREATE OR REPLACE PROCEDURE update_salary(id INT, new_salary NUMBER)

SET SEARCH_PATH = 'public' SET WORK_MEM = '1MB'

AUTHID DEFINER IS

BEGIN

UPDATE emp SET salary = new_salary WHERE emp_id = id;

END;

Include the SET clause to set the procedure's search path to public and the work memory to 1MB. Other procedures, functions and objects will not be affected by these settings.

In this example, the AUTHID DEFINER clause temporarily grants privileges to a role that might otherwise not be allowed to execute the statements within the procedure. To instruct the server to use the privileges associated with the role invoking the procedure, replace the AUTHID DEFINER clause with the AUTHID CURRENT_USER clause.

See Also

DROP PROCEDURE

CREATE PROFILE

Name

CREATE PROFILE – create a new profile

Synopsis

CREATE PROFILE *profile_name *[LIMIT {*parameter value*} ...];

Description

CREATE PROFILE creates a new profile. Include the LIMIT clause and one or more space-delimited *parameter*/*value* pairs to specify the rules enforced by Advanced Server.

Advanced Server creates a default profile named DEFAULT. When you use the CREATE ROLE command to create a new role, the new role is automatically associated with the DEFAULT profile. If you upgrade from a previous version of Advanced Server to Advanced Server 10, the upgrade process will automatically create the roles in the upgraded version to the DEFAULT profile.

You must be a superuser to use CREATE PROFILE.

Include the LIMIT clause and one or more space-delimited *parameter*/*value* pairs to specify the rules enforced by Advanced Server.

Parameters

profile_name

The name of the profile.

parameter

The password attribute that will be monitored by the rule.

value

The value the *parameter* must reach before an action is taken by the server.

Advanced Server supports the *value* shown below for each *parameter*:

FAILED_LOGIN_ATTEMPTS specifies the number of failed login attempts that a user may make before the server locks the user out of their account 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 may make an unlimited number of failed login attempts.

PASSWORD_LOCK_TIME specifies the length of time that must pass before the server unlocks an account that has been locked because of 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 it is manually unlocked by a database superuser.

PASSWORD_LIFE_TIME specifies the number of days that the current password may be used before the user is prompted to provide a new password. Include the PASSWORD_GRACE_TIME clause when using the PASSWORD_LIFE_TIME clause to specify the number of days that will pass after the password expires before connections by the role are rejected. If PASSWORD_GRACE_TIME is not specified, the password will expire on the day specified by the default value of PASSWORD_GRACE_TIME, and the user will not be allowed to execute any command until a new password is provided. 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 does not have an expiration date.

PASSWORD_GRACE_TIME specifies the length of the grace period after a password expires until the user is forced to change their password. When the grace period expires, a user will be allowed to connect, but will not be allowed to 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_REUSE_TIME specifies the number of days a user must wait before re-using a password. The PASSWORD_REUSE_TIME and PASSWORD_REUSE_MAX parameters are intended to be used together.

If you specify a finite value for one of these parameters while the other is UNLIMITED, old passwords can never be reused. If both parameters are set to UNLIMITED there are no restrictions on password reuse. 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_REUSE_TIME specified in the DEFAULT profile.
- UNLIMITED – The password can be re-used without restrictions.

PASSWORD_REUSE_MAX specifies the number of password changes that must occur before a password can be reused. The PASSWORD_REUSE_TIME and PASSWORD_REUSE_MAX parameters are intended to be used together. If you specify a finite value for one of these parameters while the other is UNLIMITED, old passwords can never be reused. If both parameters are set to UNLIMITED there are no restrictions on password reuse. Supported values are:

- An INTEGER value greater than or equal to 0.
- DEFAULT - the value of PASSWORD_REUSE_MAX specified in the DEFAULT profile.
- UNLIMITED – The password can be re-used without restrictions.

PASSWORD_VERIFY_FUNCTION specifies password complexity. Supported values are:

- The name of a PL/SQL function.
- DEFAULT - the value of PASSWORD_VERIFY_FUNCTION specified in the DEFAULT profile.
- NULL

Notes

Use DROP PROFILE command to remove the profile.

Examples

The following command creates a profile named acctg. The profile specifies that if a user has not authenticated with the correct password in five attempts, the account will be locked for one day:

```
CREATE PROFILE acctg LIMIT FAILED_LOGIN_ATTEMPTS 5 PASSWORD_LOCK_TIME 1;
```

The following command creates a profile named sales. The profile specifies that a user must change their password every 90 days:

```
CREATE PROFILE sales LIMIT PASSWORD_LIFE_TIME 90 PASSWORD_GRACE_TIME 3;
```

If the user has not changed their password before the 90 days specified in the profile has passed, they will be issued a warning at login. After a grace period of 3 days, their account will not be allowed to invoke any commands until they change their password.

The following command creates a profile named accts. The profile specifies that a user cannot re-use a password within 180 days of the last use of the password, and must change their password at least 5 times before re-using the password:

```
CREATE PROFILE accts LIMIT PASSWORD_REUSE_TIME 180 PASSWORD_REUSE_MAX 5;
```

The following command creates a profile named resources; the profile calls a user-defined function named password_rules that will verify that the password provided meets their standards for complexity:

```
CREATE PROFILE resources LIMIT PASSWORD_VERIFY_FUNCTION password_rules;
```

CREATE QUEUE

Advanced Server includes extra syntax (not offered by Oracle) with the CREATE QUEUE SQL command. This syntax can be used in association with DBMS_AQADM.

Name

CREATE QUEUE – create a queue.

Synopsis

Use CREATE QUEUE to define a new queue:

> CREATE QUEUE *name* QUEUE TABLE *queue_table_name* [({ *option_name option_value*} [, ...])]

where *option_name* and the corresponding *option_value* can be:

> TYPE [normal_queue | exception_queue]

> RETRIES [INTEGER]

> RETRYDELAY [DOUBLE PRECISION]

> RETENTION [DOUBLE PRECISION]

Description

The CREATE QUEUE command allows a database superuser or any user with the system-defined aq_administrator_role privilege to create a new queue in the current database.

If the name of the queue is schema-qualified, the queue is created in the specified schema. If a schema is not included in the CREATE QUEUE command, the queue is created in the current schema. A queue may only be created in the schema in which the queue table resides. The name of the queue must be unique from the name of any other queue in the same schema.

Use DROP QUEUE to remove a queue.

Parameters

name

The name (optionally schema-qualified) of the queue to be created.

queue_table_name

The name of the queue table with which this queue is associated.

option_name option_value

The name of any options that will be associated with the new queue, and the corresponding value for the option. If the call to CREATE QUEUE includes duplicate option names, the server will return an error. The following values are supported:

TYPE	Specify normal_queue to indicate that the queue is a normal queue, or exception_queue to indicate that the queue is an exception queue. An exception queue will only accept dequeue operations.
RETRIES	An INTEGER value that specifies the maximum number of attempts to remove a message from a queue.
RETRYDELAY	A DOUBLE PRECISION value that specifies the number of seconds after a ROLLBACK that the server will wait before retrying a message.

TYPE	Specify normal_queue to indicate that the queue is a normal queue, or exception_queue to indicate that the queue is an exception queue. An exception queue will only accept dequeue operations.
RETENTION	A DOUBLE PRECISION value that specifies the number of seconds that a message will be saved in the queue table after dequeueing.

Example

The following command creates a queue named work_order that is associated with a queue table named work_order_table:

CREATE QUEUE work_order QUEUE TABLE work_order_table (RETRIES 5, RETRYDELAY 2);

The server will allow 5 attempts to remove a message from the queue, and enforce a retry delay of 2 seconds between attempts.

See Also

ALTER QUEUE, DROP QUEUE

CREATE QUEUE TABLE

Advanced Server includes extra syntax (not offered by Oracle) with the CREATE QUEUE TABLE SQL command. This syntax can be used in association with DBMS_AQADM.

Name

CREATE QUEUE TABLE-- create a new queue table.

Synopsis

Use CREATE QUEUE TABLE to define a new queue table:

```
CREATE QUEUE TABLE name OF type_name [ ( { option_name option_value } [, ... ] ) ]
```

where *option_name* and the corresponding option_value can be:

option_name	*option_value*
SORT_LIST	priority, enq_time
MULTIPLE_CONSUMERS	FALSE, TRUE
MESSAGE_GROUPING	NONE, TRANSACTIONAL
STORAGE_CLAUSE	TABLESPACE *tablespace_name*, PCTFREE integer, PCTUSED integer, INITRANS integer, MAXTRANS integer, STORAGE *storage_option* Where *storage_option* is one or more of the following: MINEXTENTS integer, MAXEXTENTS integer, PCTINCREASE integer, INITIAL *size_clause*, NEXT, FREELISTS integer, OPTIMAL *size_clause*, BUFFER_POOL {KEEP|RECYCLE|DEFAULT}. Please note that only the TABLESPACE option is enforced; all others are accepted for compatibility and ignored. Use the TABLESPACE clause to specify the name of a tablespace in which the table will be created.

Description

CREATE QUEUE TABLE allows a superuser or a user with the aq_administrator_role privilege to create a new queue table.

If the call to CREATE QUEUE TABLE includes a schema name, the queue table is created in the specified schema. If no schema name is provided, the new queue table is created in the current schema.

The name of the queue table must be unique from the name of any other queue table in the same schema.

Parameters

name

The name (optionally schema-qualified) of the new queue table.

type_name

The name of an existing type that describes the payload of each entry in the queue table. For information about defining a type, see CREATE TYPE.

option_name option_value

The name of any options that will be associated with the new queue table, and the corresponding value for the option. If the call to CREATE QUEUE TABLE includes duplicate option names, the server will return an error. The following values are accepted:

SORT_LIST	Use the SORT_LIST option to control the dequeueing order of the queue; specify the names of the column(s) that will be used to sort the queue (in ascending order). The currently accepted values are the following combinations of enq_time and priority: enq_time. priority priority. enq_time priority enq_time Any other value will return an ERROR.
MULTIPLE_CONSUMERS	A BOOLEAN value that indicates if a message can have more than one consumer (TRUE), or are limited to one consumer per message (FALSE).
MESSAGE_GROUPING	Specify none to indicate that each message should be dequeued individually, or transactional to indicate that messages that are added to the queue as a result of one transaction should be dequeued as a group.
STORAGE_CLAUSE	Use STORAGE_CLAUSE to specify table attributes. STORAGE_CLAUSE may be TABLESPACE *tablespace_name*, PCTFREE integer, PCTUSED integer, INITRANS integer, MAXTRANS integer, STORAGE *storage_option* Where *storage_option* is one or more of the following: MINEXTENTS integer, MAXEXTENTS integer, PCTINCREASE integer, INITIAL *size_clause*, NEXT, FREELISTS integer, OPTIMAL *size_clause*, BUFFER_POOL {KEEP|RECYCLE|DEFAULT}. Please note that only the TABLESPACE option is enforced; all others are accepted for compatibility and ignored. Use the TABLESPACE clause to specify the name of a tablespace in which the table will be created.

Example

You must create a user-defined type before creating a queue table; the type describes the columns and data types within the table. The following command creates a type named work_order:

CREATE TYPE work_order AS (name VARCHAR2, project TEXT, completed BOOLEAN);

The following command uses the work_order type to create a queue table named work_order_table:

CREATE QUEUE TABLE work_order_table OF work_order (sort_list (enq_time, priority));

See Also

ALTER QUEUE TABLE, DROP QUEUE TABLE

CREATE ROLE

Name

CREATE ROLE -- define a new database role

Synopsis

CREATE ROLE *name* [IDENTIFIED BY *password* [REPLACE *old_password*]]

Description

CREATE ROLE adds a new role to the Advanced Server database cluster. A role is an entity that can own database objects and have database privileges; a role can be considered a “user”, a “group”, or both depending on how it is used. The newly created role does not have the LOGIN attribute, so it cannot be used to start a session. Use the ALTER ROLE command to give the role LOGIN rights. You must have CREATEROLE privilege or be a database superuser to use the CREATE ROLE command.

If the IDENTIFIED BY clause is specified, the CREATE ROLE command also creates a schema owned by, and with the same name as the newly created role.

Note that roles are defined at the database cluster level, and so are valid in all databases in the cluster.

Parameters

name

The name of the new role.

IDENTIFIED BY *password*

Sets the role’s password. (A password is only of use for roles having the LOGIN attribute, but you can nonetheless define one for roles without it.) If you do not plan to use password authentication you can omit this option.

Notes

Use ALTER ROLE to change the attributes of a role, and DROP ROLE to remove a role. The attributes specified by CREATE ROLE can be modified by later ALTER ROLE commands.

Use GRANT and REVOKE to add and remove members of roles that are being used as groups.

The maximum length limit for role name and password is 63 characters.

Examples

Create a role (and a schema) named, admins, with a password:

CREATE ROLE admins IDENTIFIED BY Rt498zb;

See Also

ALTER ROLE, DROP ROLE, GRANT, REVOKE, SET ROLE

CREATE SCHEMA

Name

CREATE SCHEMA -- define a new schema

Synopsis

CREATE SCHEMA AUTHORIZATION *username schema_element* [...]

Description

This variation of the CREATE SCHEMA command creates a new schema owned by *username* and populated with one or more objects. The creation of the schema and objects occur within a single transaction so either all objects are created or none of them including the schema. (Please note: if you are using an Oracle database, no new schema is created – *username*, and therefore the schema, must pre-exist.)

A schema is essentially a namespace: it contains named objects (tables, views, etc.) whose names may duplicate those of other objects existing in other schemas. Named objects are accessed either by “qualifying” their names with the schema name as a prefix, or by setting a search path that includes the desired schema(s). Unqualified objects are created in the current schema (the one at the front of the search path, which can be determined with the function CURRENT_SCHEMA). (The search path concept and the CURRENT_SCHEMA function are not compatible with Oracle databases.)

CREATE SCHEMA includes subcommands to create objects within the schema. The subcommands are treated essentially the same as separate commands issued after creating the schema. All the created objects will be owned by the specified user.

Parameters

username

The name of the user who will own the new schema. The schema will be named the same as *username*. Only superusers may create schemas owned by users other than themselves. (Please note: In Advanced Server the role, *username*, must already exist, but the schema must not exist. In Oracle, the user (equivalently, the schema) must exist.)

schema_element

An SQL statement defining an object to be created within the schema. CREATE TABLE, CREATE VIEW, and GRANT are accepted as clauses within CREATE SCHEMA. Other kinds of objects may be created in separate commands after the schema is created.

Notes

To create a schema, the invoking user must have the CREATE privilege for the current database. (Of course, superusers bypass this check.)

In Advanced Server, there are other forms of the CREATE SCHEMA command that are not compatible with Oracle databases.

Examples

CREATE SCHEMA AUTHORIZATION enterprisedb

CREATE TABLE empjobs (ename VARCHAR2(10), job VARCHAR2(9))

CREATE VIEW managers AS SELECT ename FROM empjobs WHERE job = 'MANAGER'

GRANT SELECT ON managers TO PUBLIC;

CREATE SEQUENCE

Name

CREATE SEQUENCE -- define a new sequence generator

Synopsis

CREATE SEQUENCE *name* [INCREMENT BY *increment*]

[{ NOMINVALUE | MINVALUE *minvalue* }]

[{ NOMAXVALUE | MAXVALUE *maxvalue* }]

[START WITH *start*] [CACHE *cache* | NOCACHE] [CYCLE]

Description

CREATE SEQUENCE creates a new sequence number generator. This involves creating and initializing a new special single-row table with the name, *name*. The generator will be owned by the user issuing the command.

If a schema name is given then the sequence is created in the specified schema, otherwise it is created in the current schema. The sequence name must be distinct from the name of any other sequence, table, index, or view in the same schema.

After a sequence is created, use the functions NEXTVAL and CURRVAL to operate on the sequence. These functions are documented in Section 2.4.9.

Parameters

name

The name (optionally schema-qualified) of the sequence to be created.

increment

The optional clause INCREMENT BY *increment* specifies the value to add to the current sequence value to create a new value. A positive value will make an ascending sequence, a negative one a descending sequence. The default value is 1.

NOMINVALUE | MINVALUE *minvalue*

The optional clause MINVALUE *minvalue* determines the minimum value a sequence can generate. If this clause is not supplied, then defaults will be used. The defaults are 1 and -2^{63}-1 for ascending and descending sequences, respectively. Note that the key words, NOMINVALUE, may be used to set this behavior to the default.

NOMAXVALUE | MAXVALUE *maxvalue*

The optional clause MAXVALUE *maxvalue* determines the maximum value for the sequence. If this clause is not supplied, then default values will be used. The defaults are 2^{63}-1 and -1 for ascending and descending sequences, respectively. Note that the key words, NOMAXVALUE, may be used to set this behavior to the default.

start

The optional clause START WITH *start* allows the sequence to begin anywhere. The default starting value is *minvalue* for ascending sequences and *maxvalue* for descending ones.

cache

The optional clause CACHE *cache* specifies how many sequence numbers are to be preallocated and stored in memory for faster access. The minimum value is 1 (only one value can be generated at a time, i.e., NOCACHE), and this is also the default.

CYCLE

The CYCLE option allows the sequence to wrap around when the *maxvalue* or *minvalue* has been reached by an ascending or descending sequence respectively. If the limit is reached, the next number generated will be the *minvalue* or *maxvalue*, respectively.

If CYCLE is omitted (the default), any calls to NEXTVAL after the sequence has reached its maximum value will return an error. Note that the key words, NO CYCLE, may be used to obtain the default behavior, however, this term is not compatible with Oracle databases.

Notes

Sequences are based on big integer arithmetic, so the range cannot exceed the range of an eight-byte integer (-9223372036854775808 to 9223372036854775807). On some older platforms, there may be no compiler support for eight-byte integers, in which case sequences use regular INTEGER arithmetic (range -2147483648 to +2147483647).

Unexpected results may be obtained if a *cache* setting greater than one is used for a sequence object that will be used concurrently by multiple sessions. Each session will allocate and cache successive sequence values during one access to the sequence object and increase the sequence object's last value accordingly. Then, the next *cache*-1 uses of NEXTVAL within that session simply return the preallocated values without touching the sequence object. So, any numbers allocated but not used within a session will be lost when that session ends, resulting in "holes" in the sequence.

Furthermore, although multiple sessions are guaranteed to allocate distinct sequence values, the values may be generated out of sequence when all the sessions are considered. For example, with a *cache* setting of 10, session A might reserve values 1..10 and return NEXTVAL=1, then session B might reserve values 11..20 and return NEXTVAL=11 before session A has generated NEXTVAL=2. Thus, with a *cache* setting of one it is safe to assume that NEXTVAL values are generated sequentially; with a *cache* setting greater than one you should only assume that the NEXTVAL values are all distinct, not that they are generated purely sequentially. Also, the last value will reflect the latest value reserved by any session, whether or not it has yet been returned by NEXTVAL.

Examples

Create an ascending sequence called serial, starting at 101:

```
CREATE SEQUENCE serial START WITH 101;
```

Select the next number from this sequence:

```
SELECT serial.NEXTVAL FROM DUAL;

nextval

---------

101

(1 row)
```

Create a sequence called supplier_seq with the NOCACHE option:

CREATE SEQUENCE supplier_seq

MINVALUE 1

START WITH 1

INCREMENT BY 1

NOCACHE;

Select the next number from this sequence:

SELECT supplier_seq.NEXTVAL FROM DUAL;

nextval

1

(1 row)

See Also

ALTER SEQUENCE, DROP SEQUENCE

CREATE SYNONYM

Name

CREATE SYNONYM -- define a new synonym

Synopsis

CREATE [OR REPLACE] [PUBLIC] SYNONYM [*schema*.]*syn_name* FOR *object_schema*.*object_name*[*@dblink_name*];

Description

CREATE SYNONYM defines a synonym for certain types of database objects. Advanced Server supports synonyms for:

- tables
- views
- materialized views
- sequences
- stored procedures
- stored functions
- types
- objects that are accessible through a database link
- other synonyms

Parameters:

syn_name

> *syn_name* is the name of the synonym. A synonym name must be unique within a schema.

schema

> *schema* specifies the name of the schema that the synonym resides in. If you do not specify a schema name, the synonym is created in the first existing schema in your search path.

object_name

> *object_name* specifies the name of the object.

object_schema

> *object_schema* specifies the name of the schema that the referenced object resides in.

dblink_name

> *dblink_name* specifies the name of the database link through which an object is accessed.

Include the REPLACE clause to replace an existing synonym definition with a new synonym definition.

Include the PUBLIC clause to create the synonym in the public schema. The CREATE PUBLIC SYNONYM command, compatible with Oracle databases, creates a synonym that resides in the public schema:

> CREATE [OR REPLACE] PUBLIC SYNONYM *syn_name* FOR *object_schema*.*object_name*;

This just a shorthand way to write:

> CREATE [OR REPLACE] SYNONYM public.*syn_name* FOR *object_schema*.*object_name*;

Notes

Access to the object referenced by the synonym is determined by the permissions of the current user of the synonym; the synonym user must have the appropriate permissions on the underlying database object.

Examples

Create a synonym for the emp table in a schema named, enterprisedb:

CREATE SYNONYM personnel FOR enterprisedb.emp;

See Also

DROP SYNONYM

CREATE TABLE

Name

CREATE TABLE -- define a new table

Synopsis

CREATE [GLOBAL TEMPORARY] TABLE *table_name* (

{ *column_name data_type* [DEFAULT *default_expr*]

[*column_constraint* [...]] | *table_constraint* } [, ...]

)

[ON COMMIT { PRESERVE ROWS | DELETE ROWS }]

[TABLESPACE *tablespace*]

where *column_constraint* is:

[CONSTRAINT *constraint_name*]

{ NOT NULL |

NULL |

UNIQUE [USING INDEX TABLESPACE *tablespace*] |

PRIMARY KEY [USING INDEX TABLESPACE *tablespace*] |

CHECK (*expression*) |

REFERENCES *reftable* [(*refcolumn*)]

[ON DELETE *action*] }

[DEFERRABLE | NOT DEFERRABLE] [INITIALLY DEFERRED |

INITIALLY IMMEDIATE]

and *table_constraint* is:

[CONSTRAINT *constraint_name*]

{ UNIQUE (*column_name* [, ...])

[USING INDEX TABLESPACE *tablespace*] |

PRIMARY KEY (*column_name* [, ...])

[USING INDEX TABLESPACE *tablespace*] |

CHECK (*expression*) |

FOREIGN KEY (*column_name* [, ...])

REFERENCES *reftable* [(*refcolumn* [, ...])]

[ON DELETE *action*] }

[DEFERRABLE | NOT DEFERRABLE]

[INITIALLY DEFERRED | INITIALLY IMMEDIATE]

Description

CREATE TABLE will create a new, initially empty table in the current database. The table will be owned by the user issuing the command.

If a schema name is given (for example, CREATE TABLE *myschema.mytable* ...) then the table is created in the specified schema. Otherwise it is created in the current schema. Temporary tables exist in a special schema, so a schema name may not be given when creating a temporary table. The table name must be distinct from the name of any other table, sequence, index, or view in the same schema.

CREATE TABLE also automatically creates a data type that represents the composite type corresponding to one row of the table. Therefore, tables cannot have the same name as any existing data type in the same schema.

A table cannot have more than 1600 columns. (In practice, the effective limit is lower because of tuple-length constraints).

The optional constraint clauses specify constraints (or tests) that new or updated rows must satisfy for an insert or

update operation to succeed. A constraint is an SQL object that helps define the set of valid values in the table in various ways.

There are two ways to define constraints: table constraints and column constraints. A column constraint is defined as part of a column definition. A table constraint definition is not tied to a particular column, and it can encompass more than one column. Every column constraint can also be written as a table constraint; a column constraint is only a notational convenience if the constraint only affects one column.

Parameters

GLOBAL TEMPORARY

If specified, the table is created as a temporary table. Temporary tables are automatically dropped at the end of a session, or optionally at the end of the current transaction (see ON COMMIT below). Existing permanent tables with the same name are not visible to the current session while the temporary table exists, unless they are referenced with schema-qualified names. In addition, temporary tables are not visible outside the session in which it was created. (This aspect of global temporary tables is not compatible with Oracle databases.) Any indexes created on a temporary table are automatically temporary as well.

table_name

The name (optionally schema-qualified) of the table to be created.

column_name

The name of a column to be created in the new table.

data_type

The data type of the column. This may include array specifiers. For more information on the data types included with Advanced Server, refer to Section 2.2.

DEFAULT *default_expr*

The DEFAULT clause assigns a default data value for the column whose column definition it appears within. The value is any variable-free expression (subqueries and cross-references to other columns in the current table are not allowed). The data type of the default expression must match the data type of the column.

The default expression will be used in any insert operation that does not specify a value for the column. If there is no default for a column, then the default is null.

CONSTRAINT *constraint_name*

An optional name for a column or table constraint. If not specified, the system generates a name.

NOT NULL

The column is not allowed to contain null values.

NULL

The column is allowed to contain null values. This is the default.

This clause is only available for compatibility with non-standard SQL databases. Its use is discouraged in new applications.

UNIQUE - column constraint UNIQUE (*column_name* [, ...]) - table constraint

The UNIQUE constraint specifies that a group of one or more distinct columns of a table may contain only unique values. The behavior of the unique table constraint is the same as that for column constraints, with the additional capability to span multiple columns.

For the purpose of a unique constraint, null values are not considered equal.

Each unique table constraint must name a set of columns that is different from the set of columns named by any other unique or primary key constraint defined for the table. (Otherwise it would just be the same constraint listed twice.)

PRIMARY KEY - column constraint PRIMARY KEY (*column_name* [, ...]) - table constraint

The primary key constraint specifies that a column or columns of a table may contain only unique (non-duplicate), non-null values. Technically, PRIMARY KEY is merely a combination of UNIQUE and NOT NULL, but identifying a set of columns as primary key also provides metadata about the design of the schema, as a primary key implies that other tables may rely on this set of columns as a unique identifier for rows.

Only one primary key can be specified for a table, whether as a column constraint or a table constraint.

The primary key constraint should name a set of columns that is different from other sets of columns named by any unique constraint defined for the same table.

CHECK (*expression*)

The CHECK clause specifies an expression producing a Boolean result which new or updated rows must satisfy for an insert or update operation to succeed. Expressions evaluating to TRUE or “unknown” succeed. Should any row of an insert or update operation produce a FALSE result an error exception is raised and the insert or update does not alter the database. A check constraint specified as a column constraint should reference that column’s value only, while an expression appearing in a table constraint may reference multiple columns.

Currently, CHECK expressions cannot contain subqueries nor refer to variables other than columns of the current row.

REFERENCES reftable [(*refcolumn*)] [ON DELETE *action*] - column constraint FOREIGN KEY (*column* [, ...]) REFERENCES *reftable* [(*refcolumn* [, ...])] [ON DELETE *action*] - table constraint

These clauses specify a foreign key constraint, which requires that a group of one or more columns of the new table must only contain values that match values in the referenced column(s) of some row of the referenced table. If *refcolumn* is omitted, the primary key of the *reftable* is used. The referenced columns must be the columns of a unique or primary key constraint in the referenced table.

In addition, when the data in the referenced columns is changed, certain actions are performed on the data in this table’s columns. The ON DELETE clause specifies the action to perform when a referenced row in the referenced table is being deleted. Referential actions cannot be deferred even if the constraint is deferrable. Here are the following possible actions for each clause:

CASCADE

> Delete any rows referencing the deleted row, or update the value of the referencing column to the new value of the referenced column, respectively.

SET NULL

> Set the referencing column(s) to NULL.

If the referenced column(s) are changed frequently, it may be wise to add an index to the foreign key column so that referential actions associated with the foreign key column can be performed more efficiently.

DEFERRABLE NOT DEFERRABLE

This controls whether the constraint can be deferred. A constraint that is not deferrable will be checked immediately after every command. Checking of constraints that are deferrable may be postponed until the end of the transaction (using the SET CONSTRAINTS command). NOT DEFERRABLE is the default. Only foreign key constraints currently accept this clause. All other constraint types are not deferrable.

INITIALLY IMMEDIATE INITIALLY DEFERRED

If a constraint is deferrable, this clause specifies the default time to check the constraint. If the constraint is INITIALLY IMMEDIATE, it is checked after each statement. This is the default. If the constraint is INITIALLY

DEFERRED, it is checked only at the end of the transaction. The constraint check time can be altered with the SET CONSTRAINTS command.

ON COMMIT

The behavior of temporary tables at the end of a transaction block can be controlled using ON COMMIT. The two options are:

PRESERVE ROWS

> No special action is taken at the ends of transactions. This is the default behavior. (Note that this aspect is not compatible with Oracle databases. The Oracle default is DELETE ROWS.)

DELETE ROWS

> All rows in the temporary table will be deleted at the end of each transaction block. Essentially, an automatic TRUNCATE is done at each commit.

TABLESPACE *tablespace*

The *tablespace* is the name of the tablespace in which the new table is to be created. If not specified, default tablespace is used, or the database's default tablespace if default_tablespace is an empty string.

USING INDEX TABLESPACE *tablespace*

This clause allows selection of the tablespace in which the index associated with a UNIQUE or PRIMARY KEY constraint will be created. If not specified, default tablespace is used, or the database's default tablespace if default_tablespace is an empty string.

Notes

Advanced Server automatically creates an index for each unique constraint and primary key constraint to enforce the uniqueness. Thus, it is not necessary to create an explicit index for primary key columns. (See CREATE INDEX for more information.)

Examples

Create table dept and table emp:

```
CREATE TABLE dept (

deptno NUMBER(2) NOT NULL CONSTRAINT dept_pk PRIMARY KEY,

dname VARCHAR2(14),

loc VARCHAR2(13)

);

CREATE TABLE emp (

empno NUMBER(4) NOT NULL CONSTRAINT emp_pk PRIMARY KEY,

ename VARCHAR2(10),

job VARCHAR2(9),

mgr NUMBER(4),

hiredate DATE,

sal NUMBER(7,2),
```

```
comm NUMBER(7,2),
deptno NUMBER(2) CONSTRAINT emp_ref_dept_fk
REFERENCES dept(deptno)
);
```

Define a unique table constraint for the table dept. Unique table constraints can be defined on one or more columns of the table.

```
CREATE TABLE dept (
deptno NUMBER(2) NOT NULL CONSTRAINT dept_pk PRIMARY KEY,
dname VARCHAR2(14) CONSTRAINT dept_dname_uq UNIQUE,
loc VARCHAR2(13)
);
```

Define a check column constraint:

```
CREATE TABLE emp (
empno NUMBER(4) NOT NULL CONSTRAINT emp_pk PRIMARY KEY,
ename VARCHAR2(10),
job VARCHAR2(9),
mgr NUMBER(4),
hiredate DATE,
sal NUMBER(7,2) CONSTRAINT emp_sal_ck CHECK (sal > 0),
comm NUMBER(7,2),
deptno NUMBER(2) CONSTRAINT emp_ref_dept_fk
REFERENCES dept(deptno)
);
```

Define a check table constraint:

```
CREATE TABLE emp (
empno NUMBER(4) NOT NULL CONSTRAINT emp_pk PRIMARY KEY,
ename VARCHAR2(10),
job VARCHAR2(9),
mgr NUMBER(4),
hiredate DATE,
sal NUMBER(7,2),
comm NUMBER(7,2),
```

```
deptno NUMBER(2) CONSTRAINT emp_ref_dept_fk

REFERENCES dept(deptno),

CONSTRAINT new_emp_ck CHECK (ename IS NOT NULL AND empno > 7000)

);
```

Define a primary key table constraint for the table jobhist. Primary key table constraints can be defined on one or more columns of the table.

```
CREATE TABLE jobhist (

empno NUMBER(4) NOT NULL,

startdate DATE NOT NULL,

enddate DATE,

job VARCHAR2(9),

sal NUMBER(7,2),

comm NUMBER(7,2),

deptno NUMBER(2),

chgdesc VARCHAR2(80),

CONSTRAINT jobhist_pk PRIMARY KEY (empno, startdate)

);
```

This assigns a literal constant default value for the column, job and makes the default value of hiredate be the date at which the row is inserted.

```
CREATE TABLE emp (

empno NUMBER(4) NOT NULL CONSTRAINT emp_pk PRIMARY KEY,

ename VARCHAR2(10),

job VARCHAR2(9) DEFAULT 'SALESMAN',

mgr NUMBER(4),

hiredate DATE DEFAULT SYSDATE,

sal NUMBER(7,2),

comm NUMBER(7,2),

deptno NUMBER(2) CONSTRAINT emp_ref_dept_fk

REFERENCES dept(deptno)

);
```

Create table dept in tablespace diskvol1:

```
CREATE TABLE dept (

deptno NUMBER(2) NOT NULL CONSTRAINT dept_pk PRIMARY KEY,
```

dname VARCHAR2(14),

loc VARCHAR2(13)

) TABLESPACE diskvol1;

See Also

ALTER TABLE, DROP TABLE

CREATE TABLE AS

Name

CREATE TABLE AS -- define a new table from the results of a query

Synopsis

CREATE [GLOBAL TEMPORARY] TABLE *table_name*

[(*column_name* [, ...])]

[ON COMMIT { PRESERVE ROWS | DELETE ROWS }]

[TABLESPACE *tablespace*]

AS *query*

Description

CREATE TABLE AS creates a table and fills it with data computed by a SELECT command. The table columns have the names and data types associated with the output columns of the SELECT (except that you can override the column names by giving an explicit list of new column names).

CREATE TABLE AS bears some resemblance to creating a view, but it is really quite different: it creates a new table and evaluates the query just once to fill the new table initially. The new table will not track subsequent changes to the source tables of the query. In contrast, a view re-evaluates its defining SELECT statement whenever it is queried.

Parameters

GLOBAL TEMPORARY

> If specified, the table is created as a temporary table. Refer to CREATE TABLE for details.

table_name

The name (optionally schema-qualified) of the table to be created.

column_name

The name of a column in the new table. If column names are not provided, they are taken from the output column names of the query.

query

> A query statement (a SELECT command). Refer to SELECT for a description of the allowed syntax.

CREATE TRIGGER

Name

CREATE TRIGGER -- define a new trigger

Synopsis

CREATE [OR REPLACE] TRIGGER name

{ BEFORE | AFTER | INSTEAD OF }

{ INSERT | UPDATE | DELETE }

[OR { INSERT | UPDATE | DELETE }] [, ...]

ON table

[REFERENCING { OLD AS *old* | NEW AS *new* } ...]

[FOR EACH ROW]

[WHEN *condition*]

[DECLARE

declaration; [, ...]]

BEGIN

statement; [, ...]

[EXCEPTION

{ WHEN *exception* [OR *exception*] [...] THEN

statement; [, ...] } [, ...]

]

END

Description

CREATE TRIGGER defines a new trigger. CREATE OR REPLACE TRIGGER will either create a new trigger, or replace an existing definition.

If you are using the CREATE TRIGGER keywords to create a new trigger, the name of the new trigger must not match any existing trigger defined on the same table. New triggers will be created in the same schema as the table on which the triggering event is defined.

If you are updating the definition of an existing trigger, use the CREATE OR REPLACE TRIGGER keywords.

When you use syntax that is compatible with Oracle to create a trigger, the trigger runs as a SECURITY DEFINER function.

Parameters

name

The name of the trigger to create.

BEFORE | AFTER

Determines whether the trigger is fired before or after the triggering event.

INSERT | UPDATE | DELETE

Defines the triggering event.

table

The name of the table on which the triggering event occurs.

condition

> *condition* is a Boolean expression that determines if the trigger will actually be executed; if *condition* evaluates to TRUE, the trigger will fire.
>
> If the trigger definition includes the FOR EACH ROW keywords, the WHEN clause can refer to columns of the old and/or new row values by writing OLD.*column_name* or NEW.*column_name* respectively. INSERT triggers cannot refer to OLD and DELETE triggers cannot refer to NEW.
>
> If the trigger includes the INSTEAD OF keywords, it may not include a WHEN clause. A WHEN clause cannot contain subqueries.

REFERENCING { OLD AS *old* | NEW AS *new* } ...

REFERENCING clause to reference old rows and new rows, but restricted in that *old* may only be replaced by an identifier named old or any equivalent that is saved in all lowercase (for example, REFERENCING OLD AS old, REFERENCING OLD AS OLD, or REFERENCING OLD AS "old"). Also, *new* may only be replaced by an identifier named new or any equivalent that is saved in all lowercase (for example, REFERENCING NEW AS new, REFERENCING NEW AS NEW, or REFERENCING NEW AS "new").

Either one, or both phrases OLD AS *old* and NEW AS *new* may be specified in the REFERENCING clause (for example, REFERENCING NEW AS New OLD AS Old).

This clause is not compatible with Oracle databases in that identifiers other than old or new may not be used.

FOR EACH ROW

Determines whether the trigger should be fired once for every row affected by the triggering event, or just once per SQL statement. If specified, the trigger is fired once for every affected row (row-level trigger), otherwise the trigger is a statement-level trigger.

declaration

A variable, type, REF CURSOR, or subprogram declaration. If subprogram declarations are included, they must be declared after all other variable, type, and REF CURSOR declarations.

statement

An SPL program statement. Note that a DECLARE - BEGIN - END block is considered an SPL statement unto itself. Thus, the trigger body may contain nested blocks.

exception

An exception condition name such as NO_DATA_FOUND, OTHERS, etc.

Examples

The following is a statement-level trigger that fires after the triggering statement (insert, update, or delete on table emp) is executed.

```
CREATE OR REPLACE TRIGGER user_audit_trig
```

```
AFTER INSERT OR UPDATE OR DELETE ON emp
DECLARE
v_action VARCHAR2(24);
BEGIN
IF INSERTING THEN
v_action := ' added employee(s) on ';
ELSIF UPDATING THEN
v_action := ' updated employee(s) on ';
ELSIF DELETING THEN
v_action := ' deleted employee(s) on ';
END IF;
DBMS_OUTPUT.PUT_LINE('User ' || USER || v_action ||
TO_CHAR(SYSDATE,'YYYY-MM-DD'));
END;
```

The following is a row-level trigger that fires before each row is either inserted, updated, or deleted on table emp.

```
CREATE OR REPLACE TRIGGER emp_sal_trig
BEFORE DELETE OR INSERT OR UPDATE ON emp
FOR EACH ROW
DECLARE
sal_diff NUMBER;
BEGIN
IF INSERTING THEN
DBMS_OUTPUT.PUT_LINE('Inserting employee ' || :NEW.empno);
DBMS_OUTPUT.PUT_LINE('..New salary: ' || :NEW.sal);
END IF;
IF UPDATING THEN
sal_diff := :NEW.sal - :OLD.sal;
DBMS_OUTPUT.PUT_LINE('Updating employee ' || :OLD.empno);
DBMS_OUTPUT.PUT_LINE('..Old salary: ' || :OLD.sal);
DBMS_OUTPUT.PUT_LINE('..New salary: ' || :NEW.sal);
DBMS_OUTPUT.PUT_LINE('..Raise : ' || sal_diff);
END IF;
```

```
IF DELETING THEN
DBMS_OUTPUT.PUT_LINE('Deleting employee ' || :OLD.empno);
DBMS_OUTPUT.PUT_LINE('..Old salary: ' || :OLD.sal);
END IF;
END;
```

See Also

DROP TRIGGER

CREATE TYPE

Name

CREATE TYPE -- define a new user-defined type, which can be an object type, a collection type (a nested table type or a varray type), or a composite type.

Synopsis

Object Type

```
CREATE [ OR REPLACE ] TYPE name
[ AUTHID { DEFINER | CURRENT_USER } ]
{ IS | AS } OBJECT
( { attribute { datatype | objtype | collecttype } }
[, ...]
[ method_spec ] [, ...]
) [ [ NOT ] { FINAL | INSTANTIABLE } ] ...
```

where *method_spec* is:

```
[ [ NOT ] { FINAL | INSTANTIABLE } ] ...
[ OVERRIDING ]
subprogram_spec
```

and *subprogram_spec* is:

```
{ MEMBER | STATIC }
{ PROCEDURE proc_name
[ ( [ SELF [ IN | IN OUT ] name ]
[, argname [ IN | IN OUT | OUT ] argtype
[ DEFAULT value ]
] ...)
```

]

|

FUNCTION *func_name*

[([SELF [IN | IN OUT] name]

[, *argname* [IN | IN OUT | OUT] *argtype*

[DEFAULT *value*]

] ...)

]

RETURN *rettype*

}

Nested Table Type

CREATE [OR REPLACE] TYPE *name* { IS | AS } TABLE OF

{ *datatype* | *objtype* | *collecttype* }

Varray Type

CREATE [OR REPLACE] TYPE *name* { IS | AS }

{ VARRAY | VARYING ARRAY } (*maxsize*) OF { *datatype* | *objtype* }

Composite Type

CREATE [OR REPLACE] TYPE *name* { IS | AS }

([*attribute datatype*][, ...]

)

Description

CREATE TYPE defines a new, user-defined data type. The types that can be created are an object type, a nested table type, a varray type, or a composite type. Nested table and varray types belong to the category of types known as *collections*.

Composite types are not compatible with Oracle databases. However, composite types can be accessed by SPL programs as with other types described in this section.

Note: For packages only, a composite type can be included in a user-defined record type declared with the TYPE IS RECORD statement within the package specification or package body. Such nested structure is not permitted in other SPL programs such as functions, procedures, triggers, etc.

In the CREATE TYPE command, if a schema name is included, then the type is created in the specified schema, otherwise it is created in the current schema. The name of the new type must not match any existing type in the same schema unless the intent is to update the definition of an existing type, in which case use CREATE OR REPLACE TYPE.

Note: The OR REPLACE option cannot be currently used to add, delete, or modify the attributes of an existing object type. Use the DROP TYPE command to first delete the existing object type. The OR REPLACE option can be used to add, delete, or modify the methods in an existing object type.

Note: The PostgreSQL form of the ALTER TYPE ALTER ATTRIBUTE command can be used to change the data

type of an attribute in an existing object type. However, the ALTER TYPE command cannot add or delete attributes in the object type.

The user that creates the type becomes the owner of the type.

Parameters

name

The name (optionally schema-qualified) of the type to create.

DEFINER | CURRENT_USER

Specifies whether the privileges of the object type owner (DEFINER) or the privileges of the current user executing a method in the object type (CURRENT_USER) are to be used to determine whether or not access is allowed to database objects referenced in the object type. DEFINER is the default.

attribute

The name of an attribute in the object type or composite type.

datatype

The data type that defines an attribute of the object type or composite type, or the elements of the collection type that is being created.

objtype

The name of an object type that defines an attribute of the object type or the elements of the collection type that is being created.

collecttype

The name of a collection type that defines an attribute of the object type or the elements of the collection type that is being created.

FINAL NOT FINAL

For an object type, specifies whether or not a subtype can be derived from the object type. FINAL (subtype cannot be derived from the object type) is the default.

For *method_spec*, specifies whether or not the method may be overridden in a subtype. NOT FINAL (method may be overridden in a subtype) is the default.

INSTANTIABLE NOT INSTANTIABLE

For an object type, specifies whether or not an object instance can be created of this object type. INSTANTIABLE (an instance of this object type can be created) is the default. If NOT INSTANTIABLE is specified, then NOT FINAL must be specified as well. If *method_spec* for any method in the object type contains the NOT INSTANTIABLE qualifier, then the object type, itself, must be defined with NOT INSTANTIABLE and NOT FINAL following the closing parenthesis of the object type specification.

For *method_spec*, specifies whether or not the object type definition provides an implementation for the method. INSTANTIABLE (the CREATE TYPE BODY command for the object type provides the implementation of the method) is the default. If NOT INSTANTIABLE is specified, then the CREATE TYPE BODY command for the object type must not contain the implementation of the method.

OVERRIDING

If OVERRIDING is specified, *method_spec* overrides an identically named method with the same number of identically named method arguments with the same data types, in the same order, and the same return type (if the method is a function) as defined in a supertype.

MEMBER STATIC

Specify MEMBER if the subprogram operates on an object instance. Specify STATIC if the subprogram operates independently of any particular object instance.

proc_name

The name of the procedure to create.

SELF [IN | IN OUT] *name*

For a member method there is an implicit, built-in parameter named SELF whose data type is that of the object type being defined. SELF refers to the object instance that is currently invoking the method. SELF can be explicitly declared as an IN or IN OUT parameter in the parameter list. If explicitly declared, SELF must be the first parameter in the parameter list. If SELF is not explicitly declared, its parameter mode defaults to IN OUT for member procedures and IN for member functions.

argname

The name of an argument. The argument is referenced by this name within the method body.

argtype

The data type(s) of the method's arguments. The argument types may be a base data type or a user-defined type such as a nested table or an object type. A length must not be specified for any base type - for example, specify VARCHAR2, not VARCHAR2(10).

DEFAULT *value*

Supplies a default value for an input argument if one is not supplied in the method call. DEFAULT may not be specified for arguments with modes IN OUT or OUT.

func_name

The name of the function to create.

rettype

The return data type, which may be any of the types listed for *argtype*. As for *argtype*, a length must not be specified for *rettype*.

maxsize

The maximum number of elements permitted in the varray.

Examples

Creating an Object Type

Create object type addr_obj_typ.

```
CREATE OR REPLACE TYPE addr_obj_typ AS OBJECT (

street VARCHAR2(30),

city VARCHAR2(20),

state CHAR(2),

zip NUMBER(5)

);
```

Create object type emp_obj_typ that includes a member method display_emp.

```
CREATE OR REPLACE TYPE emp_obj_typ AS OBJECT (
empno NUMBER(4),
ename VARCHAR2(20),
addr ADDR_OBJ_TYP,
MEMBER PROCEDURE display_emp (SELF IN OUT emp_obj_typ)
);
```

Create object type dept_obj_typ that includes a static method get_dname.

```
CREATE OR REPLACE TYPE dept_obj_typ AS OBJECT (
deptno NUMBER(2),
STATIC FUNCTION get_dname (p_deptno IN NUMBER) RETURN VARCHAR2,
MEMBER PROCEDURE display_dept
);
```

Creating a Collection Type

Create a nested table type, budget_tbl_typ, of data type, NUMBER(8,2).

```
CREATE OR REPLACE TYPE budget_tbl_typ IS TABLE OF NUMBER(8,2);
```

Creating and Using a Composite Type

The following example shows the usage of a composite type accessed from an anonymous block.

The composite type is created by the following:

```
CREATE OR REPLACE TYPE emphist_typ AS (
empno NUMBER(4),
ename VARCHAR2(10),
hiredate DATE,
job VARCHAR2(9),
sal NUMBER(7,2)
);
```

The following is the anonymous block that accesses the composite type:

```
DECLARE
v_emphist EMPHIST_TYP;
BEGIN
v_emphist.empno := 9001;
v_emphist.ename := 'SMITH';
```

```
v_emphist.hiredate := '01-AUG-17';

v_emphist.job := 'SALESMAN';

v_emphist.sal := 8000.00;

DBMS_OUTPUT.PUT_LINE(' EMPNO: ' || v_emphist.empno);

DBMS_OUTPUT.PUT_LINE(' ENAME: ' || v_emphist.ename);

DBMS_OUTPUT.PUT_LINE('HIREDATE: ' || v_emphist.hiredate);

DBMS_OUTPUT.PUT_LINE(' JOB: ' || v_emphist.job);

DBMS_OUTPUT.PUT_LINE(' SAL: ' || v_emphist.sal);

END;

EMPNO: 9001

ENAME: SMITH

HIREDATE: 01-AUG-17 00:00:00

JOB: SALESMAN

SAL: 8000.00
```

The following example shows the usage of a composite type accessed from a user-defined record type, declared within a package body.

The composite type is created by the following:

```
CREATE OR REPLACE TYPE salhist_typ AS (

startdate DATE,

job VARCHAR2(9),

sal NUMBER(7,2)

);
```

The package specification is defined by the following:

```
CREATE OR REPLACE PACKAGE emp_salhist

IS

PROCEDURE fetch_emp (

p_empno IN NUMBER

);

END;
```

The package body is defined by the following:

```
CREATE OR REPLACE PACKAGE BODY emp_salhist

IS
```

```
TYPE emprec_typ IS RECORD (
empno NUMBER(4),
ename VARCHAR(10),
salhist SALHIST_TYP
);
TYPE emp_arr_typ IS TABLE OF emprec_typ INDEX BY BINARY_INTEGER;
emp_arr emp_arr_typ;
PROCEDURE fetch_emp (
p_empno IN NUMBER
)
IS
CURSOR emp_cur IS SELECT e.empno, e.ename, h.startdate, h.job, h.sal
FROM emp e, jobhist h
WHERE e.empno = p_empno
AND e.empno = h.empno;
i INTEGER := 0;
BEGIN
DBMS_OUTPUT.PUT_LINE('EMPNO ENAME STARTDATE JOB ' ||
'SAL ');
DBMS_OUTPUT.PUT_LINE('----- ------- --------- --------- ' ||
'---------');
FOR r_emp IN emp_cur LOOP
i := i + 1;
emp_arr(i) := (r_emp.empno, r_emp.ename,
(r_emp.startdate, r_emp.job, r_emp.sal));
END LOOP;
FOR i IN 1 .. emp_arr.COUNT LOOP
DBMS_OUTPUT.PUT_LINE(emp_arr(i).empno || ' ' ||
RPAD(emp_arr(i).ename,8) || ' ' ||
TO_CHAR(emp_arr(i).salhist.startdate,'DD-MON-YY') || ' ' ||
RPAD(emp_arr(i).salhist.job,10) || ' ' ||
TO_CHAR(emp_arr(i).salhist.sal,'99,999.99'));
```

END LOOP;

END;

END;

Note that in the declaration of the TYPE emprec_typ IS RECORD data structure in the package body, the salhist field is defined with the SALHIST_TYP composite type as created by the CREATE TYPE salhist_typ statement.

The associative array definition TYPE emp_arr_typ IS TABLE OF emprec_typ references the record type data structure emprec_typ that includes the field salhist that is defined with the SALHIST_TYP composite type.

Invocation of the package procedure that loads the array from a join of the emp and jobhist tables, then displays the array content is shown by the following:

EXEC emp_salhist.fetch_emp(7788);

EMPNO ENAME STARTDATE JOB SAL

----- ------- --------- --------- ---------

7788 SCOTT 19-APR-87 CLERK 1,000.00

7788 SCOTT 13-APR-88 CLERK 1,040.00

7788 SCOTT 05-MAY-90 ANALYST 3,000.00

EDB-SPL Procedure successfully completed

See Also

CREATE TYPE BODY, DROP TYPE

CREATE TYPE BODY

Name

CREATE TYPE BODY -- define a new object type body

Synopsis

CREATE [OR REPLACE] TYPE BODY *name*

{ IS | AS }

method_spec [...]

END

where *method_spec* is:

subprogram_spec

and *subprogram_spec* is:

{ MEMBER | STATIC }

{ PROCEDURE *proc_name*

[([SELF [IN | IN OUT] *name*]

[, *argname* [IN | IN OUT | OUT] *argtype*

[DEFAULT *value*]

] ...)

]

{ IS | AS }

program_body

END;

|

FUNCTION *func_name*

[([SELF [IN | IN OUT] name]

[, *argname* [IN | IN OUT | OUT] *argtype*

[DEFAULT *value*]

] ...)

]

RETURN *rettype*

{ IS |AS }

program_body

END;

}

Description

CREATE TYPE BODY defines a new object type body. CREATE OR REPLACE TYPE BODY will either create a new object type body, or replace an existing body.

If a schema name is included, then the object type body is created in the specified schema. Otherwise it is created in the current schema. The name of the new object type body must match an existing object type specification in the same schema. The new object type body name must not match any existing object type body in the same schema unless the intent is to update the definition of an existing object type body, in which case use CREATE OR REPLACE TYPE BODY.

Parameters

name

The name (optionally schema-qualified) of the object type for which a body is to be created.

MEMBER STATIC

Specify MEMBER if the subprogram operates on an object instance. Specify STATIC if the subprogram operates independently of any particular object instance.

proc_name

The name of the procedure to create.

SELF [IN | IN OUT] *name*

For a member method there is an implicit, built-in parameter named SELF whose data type is that of the object type being defined. SELF refers to the object instance that is currently invoking the method. SELF can be explicitly declared as an IN or IN OUT parameter in the parameter list. If explicitly declared, SELF must be the first parameter in the parameter list. If SELF is not explicitly declared, its parameter mode defaults to IN OUT for member procedures and IN for member functions.

argname

The name of an argument. The argument is referenced by this name within the method body.

argtype

The data type(s) of the method's arguments. The argument types may be a base data type or a user-defined type such as a nested table or an object type. A length must not be specified for any base type - for example, specify VARCHAR2, not VARCHAR2(10).

DEFAULT *value*

Supplies a default value for an input argument if one is not supplied in the method call. DEFAULT may not be specified for arguments with modes IN OUT or OUT.

program_body

The declarations and SPL statements that comprise the body of the function or procedure.

func_name

The name of the function to create.

rettype

The return data type, which may be any of the types listed for *argtype*. As for *argtype*, a length must not be specified for *rettype*.

Examples

Create the object type body for object type emp_obj_typ given in the example for the CREATE TYPE command.

```
CREATE OR REPLACE TYPE BODY emp_obj_typ AS

MEMBER PROCEDURE display_emp (SELF IN OUT emp_obj_typ)

IS

BEGIN

   DBMS_OUTPUT.PUT_LINE('Employee No  : ' || empno);

   DBMS_OUTPUT.PUT_LINE('Name          : ' || ename);

   DBMS_OUTPUT.PUT_LINE('Street        : ' || addr.street);

   DBMS_OUTPUT.PUT_LINE('City/State/Zip: ' || addr.city || ', ' ||

      addr.state || ' ' || LPAD(addr.zip,5,'0'));

END;

END;
```

Create the object type body for object type dept_obj_typ given in the example for the CREATE TYPE command.

```
CREATE OR REPLACE TYPE BODY dept_obj_typ AS
STATIC FUNCTION get_dname (p_deptno IN NUMBER) RETURN VARCHAR2
IS
v_dname VARCHAR2(14);
BEGIN
CASE p_deptno
WHEN 10 THEN v_dname := 'ACCOUNING';
WHEN 20 THEN v_dname := 'RESEARCH';
WHEN 30 THEN v_dname := 'SALES';
WHEN 40 THEN v_dname := 'OPERATIONS';
ELSE v_dname := 'UNKNOWN';
END CASE;
RETURN v_dname;
END;
MEMBER PROCEDURE display_dept
IS
BEGIN
DBMS_OUTPUT.PUT_LINE('Dept No : ' || SELF.deptno);
DBMS_OUTPUT.PUT_LINE('Dept Name : ' ||
dept_obj_typ.get_dname(SELF.deptno));
END;
END;
```

See Also

CREATE TYPE, DROP TYPE

CREATE USER

Name

CREATE USER -- define a new database user account

Synopsis

CREATE USER *name* [IDENTIFIED BY *password*]

Description

CREATE USER adds a new user to an Advanced Server database cluster. You must be a database superuser to use this command.

When the CREATE USER command is given, a schema will also be created with the same name as the new user and owned by the new user. Objects with unqualified names created by this user will be created in this schema.

Parameters

name

The name of the user.

password

The user's password. The password can be changed later using ALTER USER.

Notes

The maximum length allowed for the user name and password is 63 characters.

Examples

Create a user named, john.

CREATE USER john IDENTIFIED BY abc;

See Also

DROP USER

CREATE USER|ROLE… PROFILE MANAGEMENT CLAUSES

Name

CREATE USER|ROLE

Synopsis

CREATE USER|ROLE *name* [[WITH] *option* […]]

where option can be the following compatible clauses:

PROFILE *profile_name* | ACCOUNT {LOCK|UNLOCK} | PASSWORD EXPIRE [AT '*timestamp*']

or option can be the following non-compatible clauses:

| LOCK TIME '*timestamp*'

For information about the administrative clauses of the CREATE USER or CREATE ROLE command that are supported by Advanced Server, please see the PostgreSQL core documentation available at:

https://www.postgresql.org/docs/10/static/sql-commands.html

Description

CREATE ROLE|USER… PROFILE adds a new role with an associated profile to an Advanced Server database cluster.

Roles created with the CREATE USER command are (by default) login roles. Roles created with the CREATE ROLE command are (by default) not login roles. To create a login account with the CREATE ROLE command, you must include the LOGIN keyword.

Only a database superuser can use the CREATE USER|ROLE clauses that enforce profile management; these

clauses enforce the following behaviors:

> Include the PROFILE clause and a *profile_name* to associate a pre-defined profile with a role, or to change which pre-defined profile is associated with a user.
>
> Include the ACCOUNT clause and the LOCK or UNLOCK keyword to specify that the user account should be placed in a locked or unlocked state.
>
> Include the LOCK TIME '*timestamp*' clause and a date/time value to lock the role at the specified time, and unlock the role at the time indicated by the PASSWORD_LOCK_TIME parameter of the profile assigned to this role. If LOCK TIME is used with the ACCOUNT LOCK clause, the role can only be unlocked by a database superuser with the ACCOUNT UNLOCK clause.
>
> Include the PASSWORD EXPIRE clause with the optional AT '*timestamp*' keywords to specify a date/time when the password associated with the role will expire. If you omit the AT '*timestamp*' keywords, the password will expire immediately.

Each login role may only have one profile. To discover the profile that is currently associated with a login role, query the profile column of the DBA_USERS view.

Parameters

name

The name of the role.

profile_name

The name of the profile associated with the role.

timestamp

> The date and time at which the clause will be enforced. When specifying a value for *timestamp*, enclose the value in single-quotes.

Examples

The following example uses CREATE USER to create a login role named john who is associated with the acctg_profile profile:

CREATE USER john PROFILE acctg_profile IDENTIFIED BY “1safepwd”;

john can log in to the server, using the password 1safepwd.

The following example uses CREATE ROLE to create a login role named john who is associated with the acctg_profile profile:

CREATE ROLE john PROFILE acctg_profile LOGIN PASSWORD “1safepwd”;

john can log in to the server, using the password 1safepwd.

CREATE VIEW

Name

CREATE VIEW -- define a new view

Synopsis

CREATE [OR REPLACE] VIEW *name* [(*column_name* [, ...])]

AS *query*

Description

CREATE VIEW defines a view of a query. The view is not physically materialized. Instead, the query is run every time the view is referenced in a query.

CREATE OR REPLACE VIEW is similar, but if a view of the same name already exists, it is replaced.

If a schema name is given (for example, CREATE VIEW myschema.myview ...) then the view is created in the specified schema. Otherwise it is created in the current schema. The view name must be distinct from the name of any other view, table, sequence, or index in the same schema.

Parameters

name

The name (optionally schema-qualified) of a view to be created.

column_name

An optional list of names to be used for columns of the view. If not given, the column names are deduced from the query.

query

A query (that is, a SELECT statement) which will provide the columns and rows of the view.

Refer to SELECT for more information about valid queries.

Notes

Currently, views are read only - the system will not allow an insert, update, or delete on a view. You can get the effect of an updatable view by creating rules that rewrite inserts, etc. on the view into appropriate actions on other tables.

Access to tables referenced in the view is determined by permissions of the view owner. However, functions called in the view are treated the same as if they had been called directly from the query using the view. Therefore the user of a view must have permissions to call all functions used by the view.

Examples

Create a view consisting of all employees in department 30:

CREATE VIEW dept_30 AS SELECT * FROM emp WHERE deptno = 30;

See Also

DROP VIEW

DELETE

Name

DELETE -- delete rows of a table

Synopsis

DELETE [*optimizer_hint*] FROM *table*[@*dblink*]

[WHERE *condition*]

[RETURNING *return_expression* [, ...]

{ INTO { *record* | *variable* [, ...] }

| BULK COLLECT INTO *collection* [, ...] }]

Description

DELETE deletes rows that satisfy the WHERE clause from the specified table. If the WHERE clause is absent, the effect is to delete all rows in the table. The result is a valid, but empty table.

Note: The TRUNCATE command provides a faster mechanism to remove all rows from a table.

The RETURNING INTO { *record* | *variable* [, ...] } clause may only be specified if the DELETE command is used within an SPL program. In addition the result set of the DELETE command must not include more than one row, otherwise an exception is thrown. If the result set is empty, then the contents of the target record or variables are set to null.

The RETURNING BULK COLLECT INTO *collection* [, ...] clause may only be specified if the DELETE command is used within an SPL program. If more than one *collection* is specified as the target of the BULK COLLECT INTO clause, then each *collection* must consist of a single, scalar field – i.e., *collection* must not be a record. The result set of the DELETE command may contain none, one, or more rows. *return_expression* evaluated for each row of the result set, becomes an element in *collection* starting with the first element. Any existing rows in *collection* are deleted. If the result set is empty, then *collection* will be empty.

You must have the DELETE privilege on the table to delete from it, as well as the SELECT privilege for any table whose values are read in the condition.

Parameters

optimizer_hint

Comment-embedded hints to the optimizer for selection of an execution plan.

table

The name (optionally schema-qualified) of an existing table.

dblink

> Database link name identifying a remote database. See the CREATE DATABASE LINK command for information on database links.

condition

A value expression that returns a value of type BOOLEAN that determines the rows which are to be deleted.

return_expression

An expression that may include one or more columns from *table*. If a column name from *table* is specified in *return_expression*, the value substituted for the column when *return_expression* is evaluated is the value from the deleted row.

record

A record whose field the evaluated *return_expression* is to be assigned. The first *return_expression* is assigned to the first field in *record*, the second *return_expression* is assigned to the second field in *record*, etc. The number of fields in *record* must exactly match the number of expressions and the fields must be type-compatible with their assigned expressions.

variable

A variable to which the evaluated *return_expression* is to be assigned. If more than one *return_expression* and *variable* are specified, the *first return_expression* is assigned to the first *variable*, the second *return_expression* is assigned to the second *variable*, etc. The number of variables specified following the INTO keyword must exactly match the number of expressions following the RETURNING keyword and the variables must be type-compatible with their assigned expressions.

collection

A collection in which an element is created from the evaluated *return_expression*. There can be either a single collection which may be a collection of a single field or a collection of a record type, or there may be more than one collection in which case each collection must consist of a single field. The number of return expressions must match in number and order the number of fields in all specified collections. Each corresponding *return_expression* and *collection* field must be type-compatible.

Examples

Delete all rows for employee 7900 from the jobhist table:

```
DELETE FROM jobhist WHERE empno = 7900;
```

Clear the table jobhist:

```
DELETE FROM jobhist;
```

See Also

TRUNCATE

DROP DATABASE LINK

Name

DROP DATABASE LINK -- remove a database link

Synopsis

```
DROP [ PUBLIC ] DATABASE LINK name
```

Description

DROP DATABASE LINK drops existing database links. To execute this command you must be a superuser or the owner of the database link.

Parameters

name

The name of a database link to be removed.

PUBLIC

Indicates that *name* is a public database link.

Examples

Remove the public database link named, oralink:

```
DROP PUBLIC DATABASE LINK oralink;
```

Remove the private database link named, edblink:

DROP DATABASE LINK edblink;

See Also

CREATE DATABASE LINK

DROP DIRECTORY

Name

DROP DIRECTORY -- remove a directory alias for a file system directory path

Synopsis

DROP DIRECTORY *name*

Description

DROP DIRECTORY drops an existing alias for a file system directory path that was created with the CREATE DIRECTORY command. To execute this command you must be a superuser.

When a directory alias is deleted, the corresponding physical file system directory is not affected. The file system directory must be deleted using the appropriate operating system commands.

Parameters

name

The name of a directory alias to be removed.

Examples

Remove the directory alias named empdir:

DROP DIRECTORY empdir;

See Also

CREATE DIRECTORY

DROP FUNCTION

Name

DROP FUNCTION -- remove a function

Synopsis

DROP FUNCTION *name*

[([[*argmode*] [*argname*] *argtype*] [, ...])]

Description

DROP FUNCTION removes the definition of an existing function. To execute this command you must be a superuser or the owner of the function. All input (IN, IN OUT) argument data types to the function must be specified if there is at least one input argument. (This requirement is not compatible with Oracle databases. In Oracle, only the function name is specified. Advanced Server allows overloading of function names, so the

function signature given by the input argument data types is required in the Advanced Server DROP FUNCTION command.)

Parameters

name

The name (optionally schema-qualified) of an existing function.

argmode

The mode of an argument: IN, IN OUT, or OUT. If omitted, the default is IN. Note that DROP FUNCTION does not actually pay any attention to OUT arguments, since only the input arguments are needed to determine the function's identity. So it is sufficient to list only the IN and IN OUT arguments. (Specification of *argmode* is not compatible with Oracle databases and applies only to Advanced Server.)

argname

The name of an argument. Note that DROP FUNCTION does not actually pay any attention to argument names, since only the argument data types are needed to determine the function's identity. (Specification of *argname* is not compatible with Oracle databases and applies only to Advanced Server.)

argtype

The data type of an argument of the function. (Specification of *argtype* is not compatible with Oracle databases and applies only to Advanced Server.)

Examples

The following command removes the emp_comp function.

```
DROP FUNCTION emp_comp(NUMBER, NUMBER);
```

See Also

CREATE FUNCTION

DROP INDEX

Name

DROP INDEX -- remove an index

Synopsis

DROP INDEX *name*

Description

DROP INDEX drops an existing index from the database system. To execute this command you must be a superuser or the owner of the index. If any objects depend on the index, an error will be given and the index will not be dropped.

Parameters

name

The name (optionally schema-qualified) of an index to remove.

Examples

This command will remove the index, name_idx:

DROP INDEX name_idx;

See Also

ALTER INDEX, CREATE INDEX

DROP PACKAGE

Name

DROP PACKAGE -- remove a package

Synopsis

DROP PACKAGE [BODY] *name*

Description

DROP PACKAGE drops an existing package. To execute this command you must be a superuser or the owner of the package. If BODY is specified, only the package body is removed – the package specification is not dropped. If BODY is omitted, both the package specification and body are removed.

Parameters

name

The name (optionally schema-qualified) of a package to remove.

Examples

This command will remove the emp_admin package:

DROP PACKAGE emp_admin;

See Also

CREATE PACKAGE, CREATE PACKAGE BODY

DROP PROCEDURE

Name

DROP PROCEDURE -- remove a procedure

Synopsis

DROP PROCEDURE *name*

Description

DROP PROCEDURE removes the definition of an existing procedure. To execute this command you must be a superuser or the owner of the procedure.

Parameters

name

The name (optionally schema-qualified) of an existing procedure.

Examples

The following command removes the select_emp procedure.

DROP PROCEDURE select_emp;

See Also

CREATE PROCEDURE

DROP PROFILE

Name

DROP PROFILE – drop a user-defined profile

Synopsis

DROP PROFILE [IF EXISTS] *profile_name* [CASCADE | RESTRICT];

Description

Include the IF EXISTS clause to instruct the server to not throw an error if the specified profile does not exist. The server will issue a notice if the profile does not exist.

Include the optional CASCADE clause to reassign any users that are currently associated with the profile to the default profile, and then drop the profile. Include the optional RESTRICT clause to instruct the server to not drop any profile that is associated with a role. This is the default behavior.

Parameters

profile_name

The name of the profile being dropped.

Example

The following example drops a profile named acctg_profile:

DROP PROFILE acctg_profile CASCADE;

The command first re-associates any roles associated with the acctg_profile profile with the default profile, and then drops the acctg_profile profile.

The following example drops a profile named acctg_profile:

DROP PROFILE acctg_profile RESTRICT;

The RESTRICT clause in the command instructs the server to not drop acctg_profile if there are any roles associated with the profile.

DROP QUEUE

Advanced Server includes extra syntax (not offered by Oracle) with the DROP QUEUE SQL command. This syntax can be used in association with DBMS_AQADM.

Name

DROP QUEUE -- drop an existing queue.

Synopsis

Use DROP QUEUE to drop an existing queue:

```
DROP QUEUE [IF EXISTS] name
```

Description

DROP QUEUE allows a superuser or a user with the aq_administrator_role privilege to drop an existing queue.

Parameters

name

The name (possibly schema-qualified) of the queue that is being dropped.

IF EXISTS

Include the IF EXISTS clause to instruct the server to not return an error if the queue does not exist. The server will issue a notice.

Examples

The following example drops a queue named work_order:

```
DROP QUEUE work_order;
```

See Also

CREATE QUEUE, ALTER QUEUE

DROP QUEUE TABLE

Advanced Server includes extra syntax (not offered by Oracle) with the DROP QUEUE TABLE SQL command. This syntax can be used in association with DBMS_AQADM.

Name

DROP QUEUE TABLE-- drop a queue table.

Synopsis

Use DROP QUEUE TABLE to delete a queue table:

```
DROP QUEUE TABLE [ IF EXISTS ] name [, ...] [CASCADE | RESTRICT]
```

Description

DROP QUEUE TABLE allows a superuser or a user with the aq_administrator_role privilege to delete a queue table.

Parameters

name

The name (possibly schema-qualified) of the queue table that will be deleted.

IF EXISTS

Include the IF EXISTS clause to instruct the server to not return an error if the queue table does not exist. The server will issue a notice.

CASCADE

Include the CASCADE keyword to automatically delete any objects that depend on the queue table.

RESTRICT

Include the RESTRICT keyword to instruct the server to refuse to delete the queue table if any objects depend on it. This is the default.

Example

The following example deletes a queue table named work_order_table and any objects that depend on it:

DROP QUEUE TABLE work_order_table CASCADE;

See Also

CREATE QUEUE TABLE, ALTER QUEUE TABLE

DROP SYNONYM

Name

DROP SYNONYM -- remove a synonym

Synopsis

DROP [PUBLIC] SYNONYM [*schema*.]*syn_name*

Description

DROP SYNONYM deletes existing synonyms. To execute this command you must be a superuser or the owner of the synonym, and have USAGE privileges on the schema in which the synonym resides.

Parameters:

syn_name

> *syn_name* is the name of the synonym. A synonym name must be unique within a schema.

schema

> *schema* specifies the name of the schema that the synonym resides in.

Like any other object that can be schema-qualified, you may have two synonyms with the same name in your search path. To disambiguate the name of the synonym that you are dropping, include a schema name. Unless a synonym is schema qualified in the DROP SYNONYM command, Advanced Server deletes the first instance of the synonym it finds in your search path.

You can optionally include the PUBLIC clause to drop a synonym that resides in the public schema. The DROP PUBLIC SYNONYM command, compatible with Oracle databases, drops a synonym that resides in the public schema:

> DROP PUBLIC SYNONYM *syn_name*;

The following example drops the synonym, personnel:

DROP SYNONYM personnel;

DROP ROLE

Name

DROP ROLE -- remove a database role

Synopsis

DROP ROLE *name* [CASCADE]

Description

DROP ROLE removes the specified role. To drop a superuser role, you must be a superuser yourself; to drop non-superuser roles, you must have CREATEROLE privilege.

A role cannot be removed if it is still referenced in any database of the cluster; an error will be raised if so. Before dropping the role, you must drop all the objects it owns (or reassign their ownership) and revoke any privileges the role has been granted.

It is not necessary to remove role memberships involving the role; DROP ROLE automatically revokes any memberships of the target role in other roles, and of other roles in the target role. The other roles are not dropped nor otherwise affected.

Alternatively, if the only objects owned by the role belong within a schema that is owned by the role and has the same name as the role, the CASCADE option can be specified. In this case the issuer of the DROP ROLE *name* CASCADE command must be a superuser and the named role, the schema, and all objects within the schema will be deleted.

Parameters

name

The name of the role to remove.

CASCADE

If specified, also drops the schema owned by, and with the same name as the role (and all objects owned by the role belonging to the schema) as long as no other dependencies on the role or the schema exist.

Examples

To drop a role:

DROP ROLE admins;

See Also

CREATE ROLE, SET ROLE, GRANT, REVOKE

DROP SEQUENCE

Name

DROP SEQUENCE -- remove a sequence

Synopsis

DROP SEQUENCE *name* [, ...]

Description

DROP SEQUENCE removes sequence number generators. To execute this command you must be a superuser or the owner of the sequence.

Parameters

name

The name (optionally schema-qualified) of a sequence.

Examples

To remove the sequence, serial:

DROP SEQUENCE serial;

See Also

ALTER SEQUENCE, CREATE SEQUENCE

DROP TABLE

Name

DROP TABLE -- remove a table

Synopsis

DROP TABLE *name* [CASCADE | RESTRICT | CASCADE CONSTRAINTS]

Description

DROP TABLE removes tables from the database. Only its owner may destroy a table. To empty a table of rows, without destroying the table, use DELETE. DROP TABLE always removes any indexes, rules, triggers, and constraints that exist for the target table.

Parameters

name

The name (optionally schema-qualified) of the table to drop.

Include the RESTRICT keyword to specify that the server should refuse to drop the table if any objects depend on it. This is the default behavior; the DROP TABLE command will report an error if any objects depend on the table.

Include the CASCADE clause to drop any objects that depend on the table.

Include the CASCADE CONSTRAINTS clause to specify that Advanced Server should drop any dependent constraints (excluding other object types) on the specified table.

Examples

The following command drops a table named emp that has no dependencies:

DROP TABLE emp;

The outcome of a DROP TABLE command will vary depending on whether the table has any dependencies - you can control the outcome by specifying a *drop behavior.* For example, if you create two tables, orders and items, where the items table is dependent on the orders table:

CREATE TABLE orders (order_id int PRIMARY KEY, order_date date, …);

CREATE TABLE items (order_id REFERENCES orders, quantity int, …);

Advanced Server will perform one of the following actions when dropping the orders table, depending on the drop behavior that you specify:

- If you specify DROP TABLE orders RESTRICT, Advanced Server will report an error.
- If you specify DROP TABLE orders CASCADE, Advanced Server will drop the orders table *and* the items table.
- If you specify DROP TABLE orders CASCADE CONSTRAINTS, Advanced Server will drop the orders table and remove the foreign key specification from the items table, but not drop the items table.

See Also

ALTER TABLE, CREATE TABLE

DROP TABLESPACE

Name

DROP TABLESPACE -- remove a tablespace

Synopsis

DROP TABLESPACE *tablespacename*

Description

DROP TABLESPACE removes a tablespace from the system.

A tablespace can only be dropped by its owner or a superuser. The tablespace must be empty of all database objects before it can be dropped. It is possible that objects in other databases may still reside in the tablespace even if no objects in the current database are using the tablespace.

Parameters

tablespacename

The name of a tablespace.

Examples

To remove tablespace employee_space from the system:

DROP TABLESPACE employee_space;

See Also

ALTER TABLESPACE

DROP TRIGGER

Name

DROP TRIGGER -- remove a trigger

Synopsis

DROP TRIGGER *name*

Description

DROP TRIGGER removes a trigger from its associated table. The command must be run by a superuser or the owner of the table on which the trigger is defined.

Parameters

name

The name of a trigger to remove.

Examples

Remove trigger emp_sal_trig:

DROP TRIGGER emp_sal_trig;

See Also

CREATE TRIGGER

DROP TYPE

Name

DROP TYPE -- remove a type definition

Synopsis

DROP TYPE [BODY] *name*

Description

DROP TYPE removes the type definition. To execute this command you must be a superuser or the owner of the type.

The optional BODY qualifier applies only to object type definitions, not to collection types nor to composite types. If BODY is specified, only the object type body is removed – the object type specification is not dropped. If BODY is omitted, both the object type specification and body are removed.

The type will not be deleted if there are other database objects dependent upon the named type.

Parameters

name

The name of a type definition to remove.

Examples

Drop object type addr_obj_typ.

DROP TYPE addr_obj_typ;

Drop nested table type budget_tbl_typ.

DROP TYPE budget_tbl_typ;

See Also

CREATE TYPE, CREATE TYPE BODY

DROP USER

Name

DROP USER -- remove a database user account

Synopsis

DROP USER *name* [CASCADE]

Description

DROP USER removes the specified user. To drop a superuser, you must be a superuser yourself; to drop non-superusers, you must have CREATEROLE privilege.

A user cannot be removed if it is still referenced in any database of the cluster; an error will be raised if so. Before dropping the user, you must drop all the objects it owns (or reassign their ownership) and revoke any privileges the user has been granted.

However, it is not necessary to remove role memberships involving the user; DROP USER automatically revokes any memberships of the target user in other roles, and of other roles in the target user. The other roles are not dropped nor otherwise affected.

Alternatively, if the only objects owned by the user belong within a schema that is owned by the user and has the same name as the user, the CASCADE option can be specified. In this case the issuer of the DROP USER *name* CASCADE command must be a superuser and the named user, the schema, and all objects within the schema will be deleted.

Parameters

name

The name of the user to remove.

CASCADE

If specified, also drops the schema owned by, and with the same name as the user (and all objects owned by the user belonging to the schema) as long as no other dependencies on the user or the schema exist.

Examples

To drop a user account who owns no objects nor has been granted any privileges on other objects:

DROP USER john;

To drop user account, john, who has not been granted any privileges on any objects, and does not own any objects outside of a schema named, john, that is owned by user, john:

DROP USER john CASCADE;

See Also

CREATE USER, ALTER USER

DROP VIEW

Name

DROP VIEW -- remove a view

Synopsis

DROP VIEW *name*

Description

DROP VIEW drops an existing view. To execute this command you must be a database superuser or the owner of the view. The named view will not be deleted if other objects are dependent upon this view (such as a view of a view).

The form of the DROP VIEW command compatible with Oracle does not support a CASCADE clause; to drop a view and it's dependencies, use the PostgreSQL-compatible form of the DROP VIEW command. For more information, see the PostgreSQL core documentation at:

https://www.postgresql.org/docs/10/static/sql-dropview.html

Parameters

name

The name (optionally schema-qualified) of the view to remove.

Examples

This command will remove the view called dept_30:

DROP VIEW dept_30;

See Also

CREATE VIEW

EXEC

Name

EXEC

Synopsis

EXEC *function_name* ['('[*argument_list*]')']

Description

EXECUTE .

Parameters

procedure_name

procedure_name is the (optionally schema-qualified) function name.

argument_list

> *argument_list* specifies a comma-separated list of arguments required by the function. Note that each member of *argument_list* corresponds to a formal argument expected by the function. Each formal argument may be an IN parameter, an OUT parameter, or an INOUT parameter.

Examples

The EXEC statement may take one of several forms, depending on the arguments required by the function:

EXEC update_balance; EXEC update_balance(); EXEC update_balance(1,2,3);

GRANT

Name

GRANT -- define access privileges

Synopsis

GRANT { { SELECT | INSERT | UPDATE | DELETE | REFERENCES }

[,...] | ALL [PRIVILEGES] }

ON *tablename*

TO { *username* | *groupname* | PUBLIC } [, ...]

[WITH GRANT OPTION]

GRANT { { INSERT | UPDATE | REFERENCES } (*column* [, ...]) }

[, ...]

ON *tablename*

TO { *username* | *groupname* | PUBLIC } [, ...]

[WITH GRANT OPTION]

GRANT { SELECT | ALL [PRIVILEGES] }

ON *sequencename*

TO { *username* | *groupname* | PUBLIC } [, ...]

[WITH GRANT OPTION]

GRANT { EXECUTE | ALL [PRIVILEGES] }

ON FUNCTION *progname*

([[*argmode*] [*argname*] *argtype*] [, ...])

TO { *username* | *groupname* | PUBLIC } [, ...]

[WITH GRANT OPTION]

GRANT { EXECUTE | ALL [PRIVILEGES] }

ON PROCEDURE *progname*

[([[*argmode*] [*argname*] *argtype*] [, ...])]

TO { *username* | *groupname* | PUBLIC } [, ...]

[WITH GRANT OPTION]

GRANT { EXECUTE | ALL [PRIVILEGES] }

ON PACKAGE *packagename*

TO { *username* | *groupname* | PUBLIC } [, ...]

[WITH GRANT OPTION]

GRANT *role* [, ...]

TO { *username* | *groupname* | PUBLIC } [, ...]

[WITH ADMIN OPTION]

GRANT { CONNECT | RESOURCE | DBA } [, ...]

TO { *username* | *groupname* } [, ...]

[WITH ADMIN OPTION]

GRANT CREATE [PUBLIC] DATABASE LINK

TO { *username* | *groupname* }

GRANT DROP PUBLIC DATABASE LINK

TO { *username* | *groupname* }

GRANT EXEMPT ACCESS POLICY

TO { *username* | *groupname* }

Description

The GRANT command has three basic variants: one that grants privileges on a database object (table, view, sequence, or program), one that grants membership in a role, and one that grants system privileges. These variants are similar in many ways, but they are different enough to be described separately.

In Advanced Server, the concept of users and groups has been unified into a single type of entity called a *role*. In this context, a *user* is a role that has the LOGIN attribute – the role may be used to create a session and connect to an application. A *group* is a role that does not have the LOGIN attribute – the role may not be used to create a session or connect to an application.

A role may be a member of one or more other roles, so the traditional concept of users belonging to groups is still valid. However, with the generalization of users and groups, users may “belong” to users, groups may “belong” to groups, and groups may “belong” to users, forming a general multi-level hierarchy of roles. User names and group names share the same namespace therefore it is not necessary to distinguish whether a grantee is a user or a group in the GRANT command.

GRANT on Database Objects

This variant of the GRANT command gives specific privileges on a database object to a role. These privileges are added to those already granted, if any.

The key word PUBLIC indicates that the privileges are to be granted to all roles, including those that may be created later. PUBLIC may be thought of as an implicitly defined group that always includes all roles. Any particular role will have the sum of privileges granted directly to it, privileges granted to any role it is presently a member of, and privileges granted to PUBLIC.

If the WITH GRANT OPTION is specified, the recipient of the privilege may in turn grant it to others. Without a grant option, the recipient cannot do that. Grant options cannot be granted to PUBLIC.

There is no need to grant privileges to the owner of an object (usually the user that created it), as the owner has all privileges by default. (The owner could, however, choose to revoke some of his own privileges for safety.) The right to drop an object or to alter its definition in any way is not described by a grantable privilege; it is inherent in the owner, and cannot be granted or revoked. The owner implicitly has all grant options for the object as well.

Depending on the type of object, the initial default privileges may include granting some privileges to PUBLIC. The default is no public access for tables and EXECUTE privilege for functions, procedures, and packages. The object owner may of course revoke these privileges. (For maximum security, issue the REVOKE in the same transaction that creates the object; then there is no window in which another user may use the object.)

The possible privileges are:

SELECT

> Allows SELECT from any column of the specified table, view, or sequence. For sequences, this privilege also allows the use of the currval function.

INSERT

> Allows INSERT of a new row into the specified table.

UPDATE

Allows UPDATE of a column of the specified table. SELECT ... FOR UPDATE also requires this privilege (besides the SELECT privilege).

DELETE

Allows DELETE of a row from the specified table.

REFERENCES

To create a foreign key constraint, it is necessary to have this privilege on both the referencing and referenced tables.

EXECUTE

Allows the use of the specified package, procedure, or function. When applied to a package, allows the use of all of the package’s public procedures, public functions, public variables, records, cursors and other public objects and object types. This is the only type of privilege that is applicable to functions, procedures, and packages.

The Advanced Server syntax for granting the EXECUTE privilege is not fully compatible with Oracle databases. Advanced Server requires qualification of the program name by one of the keywords, FUNCTION, PROCEDURE, or PACKAGE whereas these keywords must be omitted in Oracle. For functions, Advanced Server requires all input (IN, IN OUT) argument data types after the function name (including an empty parenthesis if there are no function arguments). For procedures, all input argument data types must be specified if the procedure has one or more input arguments. In Oracle, function and procedure signatures must be omitted. This is due to the fact that all programs share the same namespace in Oracle, whereas functions, procedures, and packages have their own individual namespace in Advanced Server to allow program name overloading to a certain extent.

ALL PRIVILEGES

Grant all of the available privileges at once.

The privileges required by other commands are listed on the reference page of the respective command.

GRANT on Roles

This variant of the GRANT command grants membership in a role to one or more other roles. Membership in a role is significant because it conveys the privileges granted to a role to each of its members.

If the WITH ADMIN OPTION is specified, the member may in turn grant membership in the role to others, and revoke membership in the role as well. Without the admin option, ordinary users cannot do that.

Database superusers can grant or revoke membership in any role to anyone. Roles having the CREATEROLE privilege can grant or revoke membership in any role that is not a superuser.

There are three pre-defined roles that have the following meanings:

CONNECT

Granting the CONNECT role is equivalent to giving the grantee the LOGIN privilege. The grantor must have the CREATEROLE privilege.

RESOURCE

Granting the RESOURCE role is equivalent to granting the CREATE and USAGE privileges on a schema that has the same name as the grantee. This schema must exist before the grant is given. The grantor must have the privilege to grant CREATE or USAGE privileges on this schema to the grantee.

DBA

Granting the DBA role is equivalent to making the grantee a superuser. The grantor must be a superuser.

Notes

The REVOKE command is used to revoke access privileges.

When a non-owner of an object attempts to GRANT privileges on the object, the command will fail outright if the user has no privileges whatsoever on the object. As long as a privilege is available, the command will proceed, but it will grant only those privileges for which the user has grant options. The GRANT ALL PRIVILEGES forms will issue a warning message if no grant options are held, while the other forms will issue a warning if grant options for any of the privileges specifically named in the command are not held. (In principle these statements apply to the object owner as well, but since the owner is always treated as holding all grant options, the cases can never occur.)

It should be noted that database superusers can access all objects regardless of object privilege settings. This is comparable to the rights of root in a Unix system. As with root, it’s unwise to operate as a superuser except when absolutely necessary.

If a superuser chooses to issue a GRANT or REVOKE command, the command is performed as though it were issued by the owner of the affected object. In particular, privileges granted via such a command will appear to have been granted by the object owner. (For role membership, the membership appears to have been granted by the containing role itself.)

GRANT and REVOKE can also be done by a role that is not the owner of the affected object, but is a member of the role that owns the object, or is a member of a role that holds privileges WITH GRANT OPTION on the object. In this case the privileges will be recorded as having been granted by the role that actually owns the object or holds the privileges WITH GRANT OPTION.

> For example, if table t1 is owned by role g1, of which role u1 is a member, then u1 can grant privileges on t1 to u2, but those privileges will appear to have been granted directly by g1. Any other member of role g1 could

revoke them later.

If the role executing GRANT holds the required privileges indirectly via more than one role membership path, it is unspecified which containing role will be recorded as having done the grant. In such cases it is best practice to use SET ROLE to become the specific role you want to do the GRANT as.

Currently, Advanced Server does not support granting or revoking privileges for individual columns of a table. One possible workaround is to create a view having just the desired columns and then grant privileges to that view.

Examples

Grant insert privilege to all users on table emp:

GRANT INSERT ON emp TO PUBLIC;

Grant all available privileges to user mary on view salesemp:

GRANT ALL PRIVILEGES ON salesemp TO mary;

Note that while the above will indeed grant all privileges if executed by a superuser or the owner of emp, when executed by someone else it will only grant those permissions for which the someone else has grant options.

Grant membership in role admins to user joe:

GRANT admins TO joe;

Grant CONNECT privilege to user joe:

GRANT CONNECT TO joe;

See Also

REVOKE, SET ROLE

GRANT on System Privileges

This variant of the GRANT command gives a role the ability to perform certain *system* operations within a database. System privileges relate to the ability to create or delete certain database objects that are not necessarily within the confines of one schema. Only database superusers can grant system privileges.

CREATE [PUBLIC] DATABASE LINK

The CREATE [PUBLIC] DATABASE LINK privilege allows the specified role to create a database link. Include the PUBLIC keyword to allow the role to create public database links; omit the PUBLIC keyword to allow the specified role to create private database links.

DROP PUBLIC DATABASE LINK

The DROP PUBLIC DATABASE LINK privilege allows a role to drop a public database link. System privileges are not required to drop a private database link. A private database link may be dropped by the link owner or a database superuser.

EXEMPT ACCESS POLICY

The EXEMPT ACCESS POLICY privilege allows a role to execute a SQL command without invoking any policy function that may be associated with the target database object. That is, the role is exempt from all security policies in the database.

The EXEMPT ACCESS POLICY privilege is not inheritable by membership to a role that has the EXEMPT ACCESS POLICY privilege. For example, the following sequence of GRANT commands does not result in user

joe obtaining the EXEMPT ACCESS POLICY privilege even though joe is granted membership to the enterprisedb role, which has been granted the EXEMPT ACCESS POLICY privilege:

GRANT EXEMPT ACCESS POLICY TO enterprisedb;

GRANT enterprisedb TO joe;

The rolpolicyexempt column of the system catalog table pg_authid is set to true if a role has the EXEMPT ACCESS POLICY privilege.

Examples

Grant CREATE PUBLIC DATABASE LINK privilege to user joe:

GRANT CREATE PUBLIC DATABASE LINK TO joe;

Grant DROP PUBLIC DATABASE LINK privilege to user joe:

GRANT DROP PUBLIC DATABASE LINK TO joe;

Grant the EXEMPT ACCESS POLICY privilege to user joe:

GRANT EXEMPT ACCESS POLICY TO joe;

Using the ALTER ROLE Command to Assign System Privileges

The Advanced Server ALTER ROLE command also supports syntax that you can use to assign:

- the privilege required to create a public or private database link.
- the privilege required to drop a public database link.
- the EXEMPT ACCESS POLICY privilege.

The ALTER ROLE syntax is functionally equivalent to the respective commands compatible with Oracle databases.

See Also

REVOKE, ALTER ROLE

INSERT

Name

INSERT -- create new rows in a table

Synopsis

INSERT INTO *table*[@*dblink*] [(*column* [, ...])]

{ VALUES ({ *expression* | DEFAULT } [, ...])

[RETURNING *return_expression* [, ...]

{ INTO { *record* | *variable* [, ...] }

| BULK COLLECT INTO *collection* [, ...] }]

| *query* }

Description

INSERT allows you to insert new rows into a table. You can insert a single row at a time or several rows as a result of a query.

The columns in the target list may be listed in any order. Each column not present in the target list will be inserted using a default value, either its declared default value or null.

If the expression for each column is not of the correct data type, automatic type conversion will be attempted.

The RETURNING INTO { *record* | *variable* [, ...] } clause may only be specified when the INSERT command is used within an SPL program and only when the VALUES clause is used.

The RETURNING BULK COLLECT INTO *collection* [, ...] clause may only be specified if the INSERT command is used within an SPL program. If more than one *collection* is specified as the target of the BULK COLLECT INTO clause, then each *collection* must consist of a single, scalar field – i.e., *collection* must not be a record. *return_expression* evaluated for each inserted row, becomes an element in *collection* starting with the first element. Any existing rows in *collection* are deleted. If the result set is empty, then *collection* will be empty.

You must have INSERT privilege to a table in order to insert into it. If you use the *query* clause to insert rows from a query, you also need to have SELECT privilege on any table used in the query.

Parameters

table

The name (optionally schema-qualified) of an existing table.

dblink

> Database link name identifying a remote database. See the CREATE DATABASE LINK command for information on database links.

column

The name of a column in *table*.

expression

An expression or value to assign to *column*.

DEFAULT

This column will be filled with its default value.

query

> A query (SELECT statement) that supplies the rows to be inserted. Refer to the SELECT command for a description of the syntax.

return_expression

An expression that may include one or more columns from *table*. If a column name from *table* is specified in *return_expression*, the value substituted for the column when *return_expression* is evaluated is determined as follows:

> If the column specified in *return_expression* is assigned a value in the INSERT command, then the assigned value is used in the evaluation of *return_expression*.
>
> If the column specified in *return_expression* is not assigned a value in the INSERT command and there is no default value for the column in the table's column definition, then null is used in the evaluation of *return_expression*.
>
> If the column specified in *return_expression* is not assigned a value in the INSERT command and there is a default value for the column in the table's column definition, then the default value is used in the evaluation of

return_expression.

record

A record whose field the evaluated *return_expression* is to be assigned. The first *return_expression* is assigned to the first field in *record*, the second *return_expression* is assigned to the second field in *record*, etc. The number of fields in *record* must exactly match the number of expressions and the fields must be type-compatible with their assigned expressions.

variable

A variable to which the evaluated *return_expression* is to be assigned. If more than one *return_expression* and *variable* are specified, the first *return_expression* is assigned to the first *variable*, the second *return_expression* is assigned to the second *variable*, etc. The number of variables specified following the INTO keyword must exactly match the number of expressions following the RETURNING keyword and the variables must be type-compatible with their assigned expressions.

collection

A collection in which an element is created from the evaluated *return_expression*. There can be either a single collection which may be a collection of a single field or a collection of a record type, or there may be more than one collection in which case each collection must consist of a single field. The number of return expressions must match in number and order the number of fields in all specified collections. Each corresponding *return_expression* and *collection* field must be type-compatible.

Examples

Insert a single row into table emp:

INSERT INTO emp VALUES (8021,'JOHN','SALESMAN',7698,'22-FEB-07',1250,500,30);

In this second example, the column, comm, is omitted and therefore it will have the default value of null:

INSERT INTO emp (empno, ename, job, mgr, hiredate, sal, deptno)

VALUES (8022,'PETERS','CLERK',7698,'03-DEC-06',950,30);

The third example uses the DEFAULT clause for the hiredate and comm columns rather than specifying a value:

INSERT INTO emp VALUES (8023,'FORD','ANALYST',7566,NULL,3000,NULL,20);

This example creates a table for the department names and then inserts into the table by selecting from the dname column of the dept table:

CREATE TABLE deptnames (

deptname VARCHAR2(14)

);

INSERT INTO deptnames SELECT dname FROM dept;

LOCK

Name

LOCK -- lock a table

Synopsis

LOCK TABLE *name* [, ...] IN *lockmode* MODE [NOWAIT]

where *lockmode* is one of:

ROW SHARE | ROW EXCLUSIVE | SHARE | SHARE ROW EXCLUSIVE | EXCLUSIVE

Description

LOCK TABLE obtains a table-level lock, waiting if necessary for any conflicting locks to be released. If NOWAIT is specified, LOCK TABLE does not wait to acquire the desired lock: if it cannot be acquired immediately, the command is aborted and an error is emitted. Once obtained, the lock is held for the remainder of the current transaction. (There is no UNLOCK TABLE command; locks are always released at transaction end.)

When acquiring locks automatically for commands that reference tables, Advanced Server always uses the least restrictive lock mode possible. LOCK TABLE provides for cases when you might need more restrictive locking. For example, suppose an application runs a transaction at the isolation level read committed and needs to ensure that data in a table remains stable for the duration of the transaction. To achieve this you could obtain SHARE lock mode over the table before querying. This will prevent concurrent data changes and ensure subsequent reads of the table see a stable view of committed data, because SHARE lock mode conflicts with the ROW EXCLUSIVE lock acquired by writers, and your LOCK TABLE name IN SHARE MODE statement will wait until any concurrent holders of ROW EXCLUSIVE mode locks commit or roll back. Thus, once you obtain the lock, there are no uncommitted writes outstanding; furthermore none can begin until you release the lock.

To achieve a similar effect when running a transaction at the isolation level serializable, you have to execute the LOCK TABLE statement before executing any data modification statement. A serializable transaction’s view of data will be frozen when its first data modification statement begins. A later LOCK TABLE will still prevent concurrent writes - but it won’t ensure that what the transaction reads corresponds to the latest committed values.

If a transaction of this sort is going to change the data in the table, then it should use SHARE ROW EXCLUSIVE lock mode instead of SHARE mode.

This ensures that only one transaction of this type runs at a time. Without this, a deadlock is possible: two transactions might both acquire SHARE mode, and then be unable to also acquire ROW EXCLUSIVE mode to actually perform their updates. (Note that a transaction’s own locks never conflict, so a transaction can acquire ROW EXCLUSIVE mode when it holds SHARE mode - but not if anyone else holds SHARE mode.) To avoid deadlocks, make sure all transactions acquire locks on the same objects in the same order, and if multiple lock modes are involved for a single object, then transactions should always acquire the most restrictive mode first.

Parameters

name

The name (optionally schema-qualified) of an existing table to lock.

The command LOCK TABLE a, b; is equivalent to LOCK TABLE a; LOCK TABLE b. The tables are locked one-by-one in the order specified in the LOCK TABLE command.

lockmode

The lock mode specifies which locks this lock conflicts with.

If no lock mode is specified, then the server uses the most restrictive mode, ACCESS EXCLUSIVE. (ACCESS EXCLUSIVE is not compatible with Oracle databases. In Advanced Server, this configuration mode ensures that no other transaction can access the locked table in any manner.)

NOWAIT

Specifies that LOCK TABLE should not wait for any conflicting locks to be released: if the specified lock cannot be immediately acquired without waiting, the transaction is aborted.

Notes

All forms of LOCK require UPDATE and/or DELETE privileges.

LOCK TABLE is useful only inside a transaction block since the lock is dropped as soon as the transaction ends. A LOCK TABLE command appearing outside any transaction block forms a self-contained transaction, so the lock will be dropped as soon as it is obtained.

LOCK TABLE only deals with table-level locks, and so the mode names involving ROW are all misnomers. These mode names should generally be read as indicating the intention of the user to acquire row-level locks within the locked table. Also, ROW EXCLUSIVE mode is a sharable table lock. Keep in mind that all the lock modes have identical semantics so far as LOCK TABLE is concerned, differing only in the rules about which modes conflict with which.

REVOKE

Name

REVOKE -- remove access privileges

Synopsis

REVOKE { { SELECT | INSERT | UPDATE | DELETE | REFERENCES }

[,...] | ALL [PRIVILEGES] }

ON *tablename*

FROM { *username* | *groupname* | PUBLIC } [, ...]

[CASCADE | RESTRICT]

REVOKE { SELECT | ALL [PRIVILEGES] }

ON *sequencename*

FROM { *username* | *groupname* | PUBLIC } [, ...]

[CASCADE | RESTRICT]

REVOKE { EXECUTE | ALL [PRIVILEGES] }

ON FUNCTION *progname*

([[*argmode*] [*argname*] *argtype*] [, ...])

FROM { *username* | *groupname* | PUBLIC } [, ...]

[CASCADE | RESTRICT]

REVOKE { EXECUTE | ALL [PRIVILEGES] }

ON PROCEDURE *progname*

[([[*argmode*] [*argname*] *argtype*] [, ...])]

FROM { *username* | *groupname* | PUBLIC } [, ...]

[CASCADE | RESTRICT]

REVOKE { EXECUTE | ALL [PRIVILEGES] }

ON PACKAGE *packagename*

FROM { *username* | *groupname* | PUBLIC } [, ...]

[CASCADE | RESTRICT]

REVOKE *role* [, ...] FROM { *username* | *groupname* | PUBLIC }

[, ...]

[CASCADE | RESTRICT]

REVOKE { CONNECT | RESOURCE | DBA } [, ...]

FROM { *username* | *groupname* } [, ...]

REVOKE CREATE [PUBLIC] DATABASE LINK

FROM { *username* | *groupname* }

REVOKE DROP PUBLIC DATABASE LINK

FROM { *username* | *groupname* }

REVOKE EXEMPT ACCESS POLICY

FROM { *username* | *groupname* }

Description

The REVOKE command revokes previously granted privileges from one or more roles. The key word PUBLIC refers to the implicitly defined group of all roles.

See the description of the GRANT command for the meaning of the privilege types.

Note that any particular role will have the sum of privileges granted directly to it, privileges granted to any role it is presently a member of, and privileges granted to PUBLIC. Thus, for example, revoking SELECT privilege from PUBLIC does not necessarily mean that all roles have lost SELECT privilege on the object: those who have it granted directly or via another role will still have it.

If the privilege had been granted with the grant option, the grant option for the privilege is revoked as well as the privilege, itself.

If a user holds a privilege with grant option and has granted it to other users then the privileges held by those other users are called dependent privileges. If the privilege or the grant option held by the first user is being revoked and dependent privileges exist, those dependent privileges are also revoked if CASCADE is specified, else the revoke action will fail. This recursive revocation only affects privileges that were granted through a chain of users that is traceable to the user that is the subject of this REVOKE command. Thus, the affected users may effectively keep the privilege if it was also granted through other users.

Note: CASCADE is not an option compatible with Oracle databases. By default Oracle always cascades dependent privileges, but Advanced Server requires the CASCADE keyword to be explicitly given, otherwise the REVOKE command will fail.

When revoking membership in a role, GRANT OPTION is instead called ADMIN OPTION, but the behavior is similar.

Notes

A user can only revoke privileges that were granted directly by that user. If, for example, user A has granted a privilege with grant option to user B, and user B has in turned granted it to user C, then user A cannot revoke the privilege directly from C. Instead, user A could revoke the grant option from user B and use the CASCADE option so that the privilege is in turn revoked from user C. For another example, if both A and B have granted the same privilege to C, A can revoke his own grant but not B’s grant, so C will still effectively have the privilege.

When a non-owner of an object attempts to REVOKE privileges on the object, the command will fail outright if the user has no privileges whatsoever on the object. As long as some privilege is available, the command will proceed, but it will revoke only those privileges for which the user has grant options. The REVOKE ALL PRIVILEGES forms will issue a warning message if no grant options are held, while the other forms will issue a warning if grant options for any of the privileges specifically named in the command are not held. (In principle these statements apply to the object owner as well, but since the owner is always treated as holding all grant options, the cases can never occur.)

If a superuser chooses to issue a GRANT or REVOKE command, the command is performed as though it were issued by the owner of the affected object. Since all privileges ultimately come from the object owner (possibly indirectly via chains of grant options), it is possible for a superuser to revoke all privileges, but this may require use of CASCADE as stated above.

REVOKE can also be done by a role that is not the owner of the affected object, but is a member of the role that owns the object, or is a member of a role that holds privileges WITH GRANT OPTION on the object. In this case the command is performed as though it were issued by the containing role that actually owns the object or holds the privileges WITH GRANT OPTION. For example, if table t1 is owned by role g1, of which role u1 is a member, then u1 can revoke privileges on t1 that are recorded as being granted by g1. This would include grants made by u1 as well as by other members of role g1.

If the role executing REVOKE holds privileges indirectly via more than one role membership path, it is unspecified which containing role will be used to perform the command. In such cases it is best practice to use SET ROLE to become the specific role you want to do the REVOKE as. Failure to do so may lead to revoking privileges other than the ones you intended, or not revoking anything at all.

Please Note: The Advanced Server ALTER ROLE command also supports syntax that revokes the system privileges required to create a public or private database link, or exemptions from fine-grained access control policies (DBMS_RLS). The ALTER ROLE syntax is functionally equivalent to the respective REVOKE command, compatible with Oracle databases.

Examples

Revoke insert privilege for the public on table emp:

REVOKE INSERT ON emp FROM PUBLIC;

Revoke all privileges from user mary on view salesemp:

REVOKE ALL PRIVILEGES ON salesemp FROM mary;

Note that this actually means “revoke all privileges that I granted”.

Revoke membership in role admins from user joe:

REVOKE admins FROM joe;

Revoke CONNECT privilege from user joe:

REVOKE CONNECT FROM joe;

Revoke CREATE DATABASE LINK privilege from user joe:

REVOKE CREATE DATABASE LINK FROM joe;

Revoke the EXEMPT ACCESS POLICY privilege from user joe:

REVOKE EXEMPT ACCESS POLICY FROM joe;

See Also

GRANT, SET ROLE

ROLLBACK

Name

ROLLBACK -- abort the current transaction

Synopsis

ROLLBACK [WORK]

Description

ROLLBACK rolls back the current transaction and causes all the updates made by the transaction to be discarded.

Parameters

WORK

Optional key word - has no effect.

Notes

Use COMMIT to successfully terminate a transaction.

Issuing ROLLBACK when not inside a transaction does no harm.

Examples

To abort all changes:

ROLLBACK;

See Also

COMMIT, ROLLBACK TO SAVEPOINT, SAVEPOINT

ROLLBACK TO SAVEPOINT

Name

ROLLBACK TO SAVEPOINT -- roll back to a savepoint

Synopsis

ROLLBACK [WORK] TO [SAVEPOINT] *savepoint_name*

Description

Roll back all commands that were executed after the savepoint was established. The savepoint remains valid and can be rolled back to again later, if needed.

ROLLBACK TO SAVEPOINT implicitly destroys all savepoints that were established after the named savepoint.

Parameters

savepoint_name

The savepoint to which to roll back.

Notes

Specifying a savepoint name that has not been established is an error.

ROLLBACK TO SAVEPOINT is not supported within SPL programs.

Examples

To undo the effects of the commands executed savepoint depts was established:

```
\set AUTOCOMMIT off

INSERT INTO dept VALUES (50, 'HR', 'NEW YORK');

SAVEPOINT depts;

INSERT INTO emp (empno, ename, deptno) VALUES (9001, 'JONES', 50);

INSERT INTO emp (empno, ename, deptno) VALUES (9002, 'ALICE', 50);

ROLLBACK TO SAVEPOINT depts;
```

See Also

COMMIT, ROLLBACK, SAVEPOINT

SAVEPOINT

Name

SAVEPOINT -- define a new savepoint within the current transaction

Synopsis

SAVEPOINT *savepoint_name*

Description

SAVEPOINT establishes a new savepoint within the current transaction.

A savepoint is a special mark inside a transaction that allows all commands that are executed after it was established to be rolled back, restoring the transaction state to what it was at the time of the savepoint.

Parameters

savepoint_name

The name to be given to the savepoint.

Notes

Use ROLLBACK TO SAVEPOINT to roll back to a savepoint.

Savepoints can only be established when inside a transaction block. There can be multiple savepoints defined within a transaction.

When another savepoint is established with the same name as a previous savepoint, the old savepoint is kept, though only the more recent one will be used when rolling back.

SAVEPOINT is not supported within SPL programs.

Examples

To establish a savepoint and later undo the effects of all commands executed after it was established:

```
\set AUTOCOMMIT off

INSERT INTO dept VALUES (50, 'HR', 'NEW YORK');

SAVEPOINT depts;

INSERT INTO emp (empno, ename, deptno) VALUES (9001, 'JONES', 50);

INSERT INTO emp (empno, ename, deptno) VALUES (9002, 'ALICE', 50);

SAVEPOINT emps;

INSERT INTO jobhist VALUES (9001,'17-SEP-07',NULL,'CLERK',800,NULL,50,'New Hire');

INSERT INTO jobhist VALUES (9002,'20-SEP-07',NULL,'CLERK',700,NULL,50,'New Hire');

ROLLBACK TO depts;

COMMIT;
```

The above transaction will commit a row into the dept table, but the inserts into the emp and jobhist tables are rolled back.

See Also

COMMIT, ROLLBACK, ROLLBACK TO SAVEPOINT

SELECT

Name

SELECT -- retrieve rows from a table or view

Synopsis

```
SELECT [ optimizer_hint ] [ ALL | DISTINCT ]

* | expression [ AS output_name ] [, ...]

FROM from_item [, ...]

[ WHERE condition ]

[ [ START WITH start_expression ]

CONNECT BY { PRIOR parent_expr = child_expr |

child_expr = PRIOR parent_expr }

[ ORDER SIBLINGS BY expression [ ASC | DESC ] [, ...] ] ]

[ GROUP BY { expression | ROLLUP ( expr_list ) |

CUBE ( expr_list ) | GROUPING SETS ( expr_list ) } [, ...]

[ LEVEL ] ]
```

[HAVING *condition* [, ...]]

[{ UNION [ALL] | INTERSECT | MINUS } *select*]

[ORDER BY *expression* [ASC | DESC] [, ...]]

[FOR UPDATE [WAIT *n*|NOWAIT|SKIP LOCKED]]

where *from_item* can be one of:

table_name[@*dblink*] [*alias*]

(*select*) *alias*

from_item [NATURAL] *join_type from_item*

[ON *join_condition* | USING (*join_column* [, ...])]

Description

SELECT retrieves rows from one or more tables. The general processing of SELECT is as follows:

1. All elements in the FROM list are computed. (Each element in the FROM list is a real or virtual table.) If more than one element is specified in the FROM list, they are cross-joined together. (See FROM clause, below.)

2. If the WHERE clause is specified, all rows that do not satisfy the condition are eliminated from the output. (See WHERE clause, below.)

3. If the GROUP BY clause is specified, the output is divided into groups of rows that match on one or more values. If the HAVING clause is present, it eliminates groups that do not satisfy the given condition. (See GROUP BY clause and HAVING clause below.)

4. Using the operators UNION, INTERSECT, and MINUS, the output of more than one SELECT statement can be combined to form a single result set. The UNION operator returns all rows that are in one or both of the result sets. The INTERSECT operator returns all rows that are strictly in both result sets. The MINUS operator returns the rows that are in the first result set but not in the second. In all three cases, duplicate rows are eliminated. In the case of the UNION operator, if ALL is specified then duplicates are not eliminated. (See UNION clause, INTERSECT clause, and MINUS clause below.)

5. The actual output rows are computed using the SELECT output expressions for each selected row. (See SELECT list below.)

6. The CONNECT BY clause is used to select data that has a hierarchical relationship. Such data has a parent-child relationship between rows. (See CONNECT BY clause.)

7. If the ORDER BY clause is specified, the returned rows are sorted in the specified order. If ORDER BY is not given, the rows are returned in whatever order the system finds fastest to produce. (See ORDER BY clause below.)

8. DISTINCT eliminates duplicate rows from the result. ALL (the default) will return all candidate rows, including duplicates. (See DISTINCT clause below.)

9. The FOR UPDATE clause causes the SELECT statement to lock the selected rows against concurrent updates. (See FOR UPDATE clause below.)

You must have SELECT privilege on a table to read its values. The use of FOR UPDATE requires UPDATE privilege as well.

Parameters

optimizer_hint

Comment-embedded hints to the optimizer for selection of an execution plan. See Section 3.4 for information

about optimizer hints.

FROM Clause

The FROM clause specifies one or more source tables for a SELECT statement. The syntax is:

FROM *source* [, ...]

Where *source* can be one of following elements:

table_name[@*dblink*]

> The name (optionally schema-qualified) of an existing table or view. *dblink* is a database link name identifying a remote database. See the CREATE DATABASE LINK command for information on database links.

alias

A substitute name for the FROM item containing the alias. An alias is used for brevity or to eliminate ambiguity for self-joins (where the same table is scanned multiple times). When an alias is provided, it completely hides the actual name of the table or function; for example given FROM foo AS f, the remainder of the SELECT must refer to this FROM item as f not foo.

select

A sub-SELECT can appear in the FROM clause. This acts as though its output were created as a temporary table for the duration of this single SELECT command. Note that the sub-SELECT must be surrounded by parentheses, and an alias must be provided for it.

join_type

One of the following:

> [INNNER] JOIN
>
> LEFT [OUTER] JOIN
>
> RIGHT [OUTER] JOIN
>
> FULL [OUTER] JOIN
>
> CROSS JOIN

For the INNER and OUTER join types, a join condition must be specified, namely exactly one of NATURAL, ON *join_condition*, or USING (join_column [, ...]). See below for the meaning. For CROSS JOIN, none of these clauses may appear.

A JOIN clause combines two FROM items. Use parentheses if necessary to determine the order of nesting. In the absence of parentheses, JOINs nest left-to-right. In any case JOIN binds more tightly than the commas separating FROM items.

CROSS JOIN and INNER JOIN produce a simple Cartesian product, the same result as you get from listing the two items at the top level of FROM, but restricted by the join condition (if any). CROSS JOIN is equivalent to INNER JOIN ON (TRUE), that is, no rows are removed by qualification. These join types are just a notational convenience, since they do nothing you couldn’t do with plain FROM and WHERE.

LEFT OUTER JOIN returns all rows in the qualified Cartesian product (i.e., all combined rows that pass its join condition), plus one copy of each row in the left-hand table for which there was no right-hand row that passed the join condition. This left-hand row is extended to the full width of the joined table by inserting null values for the right-hand columns. Note that only the JOIN clause’s own condition is considered while deciding which rows have matches. Outer conditions are applied afterwards.

Conversely, RIGHT OUTER JOIN returns all the joined rows, plus one row for each unmatched right-hand row (extended with nulls on the left). This is just a notational convenience, since you could convert it to a LEFT OUTER JOIN by switching the left and right inputs.

FULL OUTER JOIN returns all the joined rows, plus one row for each unmatched left-hand row (extended with nulls on the right), plus one row for each unmatched right-hand row (extended with nulls on the left).

ON *join_condition*

join_condition is an expression resulting in a value of type BOOLEAN (similar to a WHERE clause) that specifies which rows in a join are considered to match.

USING (*join_column* [, ...])

A clause of the form USING (a, b, ...) is shorthand for ON left_table.a = right_table.a AND left_table.b = right_table.b .... Also, USING implies that only one of each pair of equivalent columns will be included in the join output, not both.

NATURAL

NATURAL is shorthand for a USING list that mentions all columns in the two tables that have the same names.

If multiple sources are specified, the result is the Cartesian product (cross join) of all the sources. Usually qualification conditions are added to restrict the returned rows to a small subset of the Cartesian product.

Example

The following example selects all of the entries from the dept table:

SELECT * FROM dept;

deptno | dname | loc

-------+-------------+-----------

10 | ACCOUNTING | NEW YORK

20 | RESEARCH | DALLAS

30 | SALES | CHICAGO

40 | OPERATIONS | BOSTON

(4 rows)

WHERE Clause

The optional WHERE clause has the form:

WHERE *condition*

where *condition* is any expression that evaluates to a result of type BOOLEAN. Any row that does not satisfy this condition will be eliminated from the output. A row satisfies the condition if it returns TRUE when the actual row values are substituted for any variable references.

Example

The following example joins the contents of the emp and dept tables, WHERE the value of the deptno column in the emp table is equal to the value of the deptno column in the deptno table:

SELECT d.deptno, d.dname, e.empno, e.ename, e.mgr, e.hiredate

```
FROM emp e, dept d

WHERE d.deptno = e.deptno;

deptno | dname | empno | ename | mgr | hiredate

--------+------------+-------+--------+------+-------------------

10 | ACCOUNTING | 7934 | MILLER | 7782 | 23-JAN-82 00:00:00

10 | ACCOUNTING | 7782 | CLARK | 7839 | 09-JUN-81 00:00:00

10 | ACCOUNTING | 7839 | KING | | 17-NOV-81 00:00:00

20 | RESEARCH | 7788 | SCOTT | 7566 | 19-APR-87 00:00:00

20 | RESEARCH | 7566 | JONES | 7839 | 02-APR-81 00:00:00

20 | RESEARCH | 7369 | SMITH | 7902 | 17-DEC-80 00:00:00

20 | RESEARCH | 7876 | ADAMS | 7788 | 23-MAY-87 00:00:00

20 | RESEARCH | 7902 | FORD | 7566 | 03-DEC-81 00:00:00

30 | SALES | 7521 | WARD | 7698 | 22-FEB-81 00:00:00

30 | SALES | 7844 | TURNER | 7698 | 08-SEP-81 00:00:00

30 | SALES | 7499 | ALLEN | 7698 | 20-FEB-81 00:00:00

30 | SALES | 7698 | BLAKE | 7839 | 01-MAY-81 00:00:00

30 | SALES | 7654 | MARTIN | 7698 | 28-SEP-81 00:00:00

30 | SALES | 7900 | JAMES | 7698 | 03-DEC-81 00:00:00

(14 rows)
```

GROUP BY Clause

The optional GROUP BY clause has the form:

GROUP BY { *expression* | ROLLUP (*expr_list*) |

CUBE (*expr_list*) | GROUPING SETS (*expr_list*) } [, ...]

GROUP BY will condense into a single row all selected rows that share the same values for the grouped expressions. *expression* can be an input column name, or the name or ordinal number of an output column (SELECT list item), or an arbitrary expression formed from input-column values. In case of ambiguity, a GROUP BY name will be interpreted as an input-column name rather than an output column name.

ROLLUP, CUBE, and GROUPING SETS are extensions to the GROUP BY clause for supporting multidimensional analysis. See Section 2.3.71.3 for information on using these extensions.

Aggregate functions, if any are used, are computed across all rows making up each group, producing a separate value for each group (whereas without GROUP BY, an aggregate produces a single value computed across all the selected rows). When GROUP BY is present, it is not valid for the SELECT list expressions to refer to ungrouped columns except within aggregate functions, since there would be more than one possible value to return for an ungrouped column.

Example

The following example computes the sum of the sal column in the emp table, grouping the results by department number:

SELECT deptno, SUM(sal) AS total

FROM emp

GROUP BY deptno;

deptno | total

--------+----------

10 | 8750.00

20 | 10875.00

30 | 9400.00

(3 rows)

HAVING Clause

The optional HAVING clause has the form:

HAVING *condition*

where *condition* is the same as specified for the WHERE clause.

HAVING eliminates group rows that do not satisfy the specified condition. HAVING is different from WHERE; WHERE filters individual rows before the application of GROUP BY, while HAVING filters group rows created by GROUP BY. Each column referenced in condition must unambiguously reference a grouping column, unless the reference appears within an aggregate function.

Example

To sum the column, sal of all employees, group the results by department number and show those group totals that are less than 10000:

SELECT deptno, SUM(sal) AS total

FROM emp

GROUP BY deptno

HAVING SUM(sal) \< 10000;

deptno | total

--------+---------

10 | 8750.00

30 | 9400.00

(2 rows)

SELECT List

The SELECT list (between the key words SELECT and FROM) specifies expressions that form the output rows of

the SELECT statement. The expressions can (and usually do) refer to columns computed in the FROM clause. Using the clause AS *output_name*, another name can be specified for an output column. This name is primarily used to label the column for display. It can also be used to refer to the column's value in ORDER BY and GROUP BY clauses, but not in the WHERE or HAVING clauses; there you must write out the expression instead.

Instead of an expression, * can be written in the output list as a shorthand for all the columns of the selected rows.

Example

The SELECT list in the following example specifies that the result set should include the empno column, the ename column, the mgr column and the hiredate column:

SELECT empno, ename, mgr, hiredate FROM emp;

empno | ename | mgr | hiredate

-------+--------+------+--------------------

7934 | MILLER | 7782 | 23-JAN-82 00:00:00

7782 | CLARK | 7839 | 09-JUN-81 00:00:00

7839 | KING | | 17-NOV-81 00:00:00

7788 | SCOTT | 7566 | 19-APR-87 00:00:00

7566 | JONES | 7839 | 02-APR-81 00:00:00

7369 | SMITH | 7902 | 17-DEC-80 00:00:00

7876 | ADAMS | 7788 | 23-MAY-87 00:00:00

7902 | FORD | 7566 | 03-DEC-81 00:00:00

7521 | WARD | 7698 | 22-FEB-81 00:00:00

7844 | TURNER | 7698 | 08-SEP-81 00:00:00

7499 | ALLEN | 7698 | 20-FEB-81 00:00:00

7698 | BLAKE | 7839 | 01-MAY-81 00:00:00

7654 | MARTIN | 7698 | 28-SEP-81 00:00:00

7900 | JAMES | 7698 | 03-DEC-81 00:00:00

(14 rows)

UNION Clause

The UNION clause has the form:

select_statement UNION [ALL] *select_statement*

select_statement is any SELECT statement without an ORDER BY or FOR UPDATE clause. (ORDER BY can be attached to a sub-expression if it is enclosed in parentheses. Without parentheses, these clauses will be taken to apply to the result of the UNION, not to its right-hand input expression.)

The UNION operator computes the set union of the rows returned by the involved SELECT statements. A row is in the set union of two result sets if it appears in at least one of the result sets. The two SELECT statements that represent the direct operands of the UNION must produce the same number of columns, and corresponding

columns must be of compatible data types.

The result of UNION does not contain any duplicate rows unless the ALL option is specified. ALL prevents elimination of duplicates.

Multiple UNION operators in the same SELECT statement are evaluated left to right, unless otherwise indicated by parentheses.

Currently, FOR UPDATE may not be specified either for a UNION result or for any input of a UNION.

INTERSECT Clause

The INTERSECT clause has the form:

select_statement INTERSECT *select_statement*

select_statement is any SELECT statement without an ORDER BY or FOR UPDATE clause.

The INTERSECT operator computes the set intersection of the rows returned by the involved SELECT statements. A row is in the intersection of two result sets if it appears in both result sets.

The result of INTERSECT does not contain any duplicate rows.

Multiple INTERSECT operators in the same SELECT statement are evaluated left to right, unless parentheses dictate otherwise. INTERSECT binds more tightly than UNION. That is, A UNION B INTERSECT C will be read as A UNION (B INTERSECT C).

MINUS Clause

The MINUS clause has this general form:

select_statement MINUS *select_statement*

select_statement is any SELECT statement without an ORDER BY or FOR UPDATE clause.

The MINUS operator computes the set of rows that are in the result of the left SELECT statement but not in the result of the right one.

The result of MINUS does not contain any duplicate rows.

Multiple MINUS operators in the same SELECT statement are evaluated left to right, unless parentheses dictate otherwise. MINUS binds at the same level as UNION.

CONNECT BY Clause

The CONNECT BY clause determines the parent-child relationship of rows when performing a hierarchical query. It has the general form:

CONNECT BY { PRIOR *parent_expr* = *child_expr* |

child_expr = PRIOR *parent_expr* }

parent_expr is evaluated on a candidate parent row. If *parent_expr* = *child_expr* results in TRUE for a row returned by the FROM clause, then this row is considered a child of the parent.

The following optional clauses may be specified in conjunction with the CONNECT BY clause:

START WITH *start_expression*

The rows returned by the FROM clause on which *start_expression* evaluates to TRUE become the root nodes of the hierarchy.

ORDER SIBLINGS BY *expression* [ASC | DESC] [, ...]

Sibling rows of the hierarchy are ordered by *expression* in the result set.

Note: Advanced Server does not support the use of AND (or other operators) in the CONNECT BY clause.

ORDER BY Clause

The optional ORDER BY clause has the form:

ORDER BY *expression* [ASC | DESC] [, ...]

expression can be the name or ordinal number of an output column (SELECT list item), or it can be an arbitrary expression formed from input-column values.

The ORDER BY clause causes the result rows to be sorted according to the specified expressions. If two rows are equal according to the leftmost expression, they are compared according to the next expression and so on. If they are equal according to all specified expressions, they are returned in an implementation-dependent order.

The ordinal number refers to the ordinal (left-to-right) position of the result column. This feature makes it possible to define an ordering on the basis of a column that does not have a unique name. This is never absolutely necessary because it is always possible to assign a name to a result column using the AS clause.

It is also possible to use arbitrary expressions in the ORDER BY clause, including columns that do not appear in the SELECT result list. Thus the following statement is valid:

SELECT ename FROM emp ORDER BY empno;

A limitation of this feature is that an ORDER BY clause applying to the result of a UNION, INTERSECT, or MINUS clause may only specify an output column name or number, not an expression.

If an ORDER BY expression is a simple name that matches both a result column name and an input column name, ORDER BY will interpret it as the result column name. This is the opposite of the choice that GROUP BY will make in the same situation. This inconsistency is made to be compatible with the SQL standard.

Optionally one may add the key word ASC (ascending) or DESC (descending) after any expression in the ORDER BY clause. If not specified, ASC is assumed by default.

The null value sorts higher than any other value. In other words, with ascending sort order, null values sort at the end, and with descending sort order, null values sort at the beginning.

Character-string data is sorted according to the locale-specific collation order that was established when the database cluster was initialized.

Examples

The following two examples are identical ways of sorting the individual results according to the contents of the second column (dname):

SELECT * FROM dept ORDER BY dname;

deptno | dname | loc

--------+------------+----------

10 | ACCOUNTING | NEW YORK

40 | OPERATIONS | BOSTON

20 | RESEARCH | DALLAS

30 | SALES | CHICAGO

(4 rows)

SELECT * FROM dept ORDER BY 2;

deptno | dname | loc

--------+------------+----------

10 | ACCOUNTING | NEW YORK

40 | OPERATIONS | BOSTON

20 | RESEARCH | DALLAS

30 | SALES | CHICAGO

(4 rows)

DISTINCT Clause

If a SELECT statement specifies DISTINCT, all duplicate rows are removed from the result set (one row is kept from each group of duplicates). The ALL keyword specifies the opposite: all rows are kept; that is the default.

FOR UPDATE Clause

The FOR UPDATE clause takes the form:

FOR UPDATE [WAIT *n*|NOWAIT|SKIP LOCKED]

FOR UPDATE causes the rows retrieved by the SELECT statement to be locked as though for update. This prevents a row from being modified or deleted by other transactions until the current transaction ends; any transaction that attempts to UPDATE, DELETE, or SELECT FOR UPDATE a selected row will be blocked until the current transaction ends. If an UPDATE, DELETE, or SELECT FOR UPDATE from another transaction has already locked a selected row or rows, SELECT FOR UPDATE will wait for the first transaction to complete, and will then lock and return the updated row (or no row, if the row was deleted).

FOR UPDATE cannot be used in contexts where returned rows cannot be clearly identified with individual table rows (for example, with aggregation).

Use FOR UPDATE options to specify locking preferences:

- Include the WAIT *n* keywords to specify the number of seconds (or fractional seconds) that the SELECT statement will wait for a row locked by another session. Use a decimal form to specify fractional seconds; for example, WAIT 1.5 instructs the server to wait one and a half seconds. Specify up to 4 digits to the right of the decimal.
- Include the NOWAIT keyword to report an error immediately if a row cannot be locked by the current session.
- Include SKIP LOCKED to instruct the server to lock rows if possible, and skip rows that are already locked by another session.

SET CONSTRAINTS

Name

SET CONSTRAINTS -- set constraint checking modes for the current transaction

Synopsis

SET CONSTRAINTS { ALL | *name* [, ...] } { DEFERRED | IMMEDIATE }

Description

SET CONSTRAINTS sets the behavior of constraint checking within the current transaction. IMMEDIATE constraints are checked at the end of each statement. DEFERRED constraints are not checked until transaction commit. Each constraint has its own IMMEDIATE or DEFERRED mode.

Upon creation, a constraint is given one of three characteristics: DEFERRABLE INITIALLY DEFERRED, DEFERRABLE INITIALLY IMMEDIATE, or NOT DEFERRABLE. The third class is always IMMEDIATE and is not affected by the SET CONSTRAINTS command. The first two classes start every transaction in the indicated mode, but their behavior can be changed within a transaction by SET CONSTRAINTS.

SET CONSTRAINTS with a list of constraint names changes the mode of just those constraints (which must all be deferrable). If there are multiple constraints matching any given name, all are affected. SET CONSTRAINTS ALL changes the mode of all deferrable constraints.

When SET CONSTRAINTS changes the mode of a constraint from DEFERRED to IMMEDIATE, the new mode takes effect retroactively: any outstanding data modifications that would have been checked at the end of the transaction are instead checked during the execution of the SET CONSTRAINTS command. If any such constraint is violated, the SET CONSTRAINTS fails (and does not change the constraint mode). Thus, SET CONSTRAINTS can be used to force checking of constraints to occur at a specific point in a transaction.

Currently, only foreign key constraints are affected by this setting. Check and unique constraints are always effectively not deferrable.

Notes

This command only alters the behavior of constraints within the current transaction. Thus, if you execute this command outside of a transaction block it will not appear to have any effect.

SET ROLE

Name

SET ROLE -- set the current user identifier of the current session

Synopsis

SET ROLE { *rolename* | NONE }

Description

This command sets the current user identifier of the current SQL session context to be *rolename*. After SET ROLE, permissions checking for SQL commands is carried out as though the named role were the one that had logged in originally.

The specified *rolename* must be a role that the current session user is a member of. (If the session user is a superuser, any role can be selected.)

NONE resets the current user identifier to be the current session user identifier. These forms may be executed by any user.

Notes

Using this command, it is possible to either add privileges or restrict one’s privileges. If the session user role has the INHERITS attribute, then it automatically has all the privileges of every role that it could SET ROLE to; in this case SET ROLE effectively drops all the privileges assigned directly to the session user and to the other roles it is

a member of, leaving only the privileges available to the named role. On the other hand, if the session user role has the NOINHERITS attribute, SET ROLE drops the privileges assigned directly to the session user and instead acquires the privileges available to the named role. In particular, when a superuser chooses to SET ROLE to a non-superuser role, she loses her superuser privileges.

Examples

User mary takes on the identity of role admins:

SET ROLE admins;

User mary reverts back to her own identity:

SET ROLE NONE;

SET TRANSACTION

Name

SET TRANSACTION -- set the characteristics of the current transaction

Synopsis

SET TRANSACTION *transaction_mode*

where *transaction_mode* is one of:

ISOLATION LEVEL { SERIALIZABLE | READ COMMITTED }

READ WRITE | READ ONLY

Description

The SET TRANSACTION command sets the characteristics of the current transaction. It has no effect on any subsequent transactions. The available transaction characteristics are the transaction isolation level and the transaction access mode (read/write or read-only). The isolation level of a transaction determines what data the transaction can see when other transactions are running concurrently:

READ COMMITTED

A statement can only see rows committed before it began. This is the default.

SERIALIZABLE

All statements of the current transaction can only see rows committed before the first query or data-modification statement was executed in this transaction.

The transaction isolation level cannot be changed after the first query or data-modification statement (SELECT, INSERT, DELETE, UPDATE, or FETCH) of a transaction has been executed. The transaction access mode determines whether the transaction is read/write or read-only. Read/write is the default.

When a transaction is read-only, the following SQL commands are disallowed: INSERT, UPDATE, and DELETE if the table they would write to is not a temporary table; all CREATE, ALTER, and DROP commands; COMMENT, GRANT, REVOKE, TRUNCATE; and EXECUTE if the command it would execute is among those listed. This is a high-level notion of read-only that does not prevent all writes to disk.

TRUNCATE

Name

TRUNCATE -- empty a table

Synopsis

TRUNCATE TABLE *name* [DROP STORAGE]

Description

TRUNCATE quickly removes all rows from a table. It has the same effect as an unqualified DELETE but since it does not actually scan the table, it is faster. This is most useful on large tables.

The DROP STORAGE clause is accepted for compatibility, but is ignored.

Parameters

name

The name (optionally schema-qualified) of the table to be truncated.

Notes

TRUNCATE cannot be used if there are foreign-key references to the table from other tables. Checking validity in such cases would require table scans, and the whole point is not to do one.

TRUNCATE will not run any user-defined ON DELETE triggers that might exist for the table.

Examples

Truncate the table bigtable:

TRUNCATE TABLE bigtable;

See Also

DROP VIEW, DELETE

UPDATE

Name

UPDATE -- update rows of a table

Synopsis

UPDATE [*optimizer_hint*] *table*[*@dblink*]

SET *column* = { *expression* | DEFAULT } [, ...]

[WHERE *condition*]

[RETURNING *return_expression* [, ...]

{ INTO { *record* | *variable* [, ...] }

| BULK COLLECT INTO *collection* [, ...] }]

Description

UPDATE changes the values of the specified columns in all rows that satisfy the condition. Only the columns to be modified need be mentioned in the SET clause; columns not explicitly modified retain their previous values.

The RETURNING INTO { *record* | *variable* [, ...] } clause may only be specified within an SPL program. In addition the result set of the UPDATE command must not return more than one row, otherwise an exception is thrown. If the result set is empty, then the contents of the target record or variables are set to null.

The RETURNING BULK COLLECT INTO *collection* [, ...] clause may only be specified if the UPDATE command is used within an SPL program. If more than one *collection* is specified as the target of the BULK COLLECT INTO clause, then each *collection* must consist of a single, scalar field – i.e., *collection* must not be a record. The result set of the UPDATE command may contain none, one, or more rows. *return_expression* evaluated for each row of the result set, becomes an element in *collection* starting with the first element. Any existing rows in *collection* are deleted. If the result set is empty, then *collection* will be empty.

You must have the UPDATE privilege on the table to update it, as well as the SELECT privilege to any table whose values are read in expression or *condition*.

Parameters

optimizer_hint

Comment-embedded hints to the optimizer for selection of an execution plan.

table

The name (optionally schema-qualified) of the table to update.

dblink

> Database link name identifying a remote database. See the CREATE DATABASE LINK command for information on database links.

column

The name of a column in table.

expression

An expression to assign to the column. The expression may use the old values of this and other columns in the table.

DEFAULT

Set the column to its default value (which will be null if no specific default expression has been assigned to it).

condition

An expression that returns a value of type BOOLEAN. Only rows for which this expression returns true will be updated.

return_expression

An expression that may include one or more columns from table. If a column name from table is specified in *return_expression*, the value substituted for the column when *return_expression* is evaluated is determined as follows:

> If the column specified in *return_expression* is assigned a value in the UPDATE command, then the assigned value is used in the evaluation of *return_expression*.
>
> If the column specified in *return_expression* is not assigned a value in the UPDATE command, then the column's current value in the affected row is used in the evaluation of *return_expression*.

record

A record whose field the evaluated *return_expression* is to be assigned. The first *return_expression* is assigned to the first field in *record*, the second *return_expression* is assigned to the second field in *record*, etc. The number of fields in *record* must exactly match the number of expressions and the fields must be type-compatible with their assigned expressions.

variable

A variable to which the evaluated *return_expression* is to be assigned. If more than one *return_expression* and *variable* are specified, the first *return_expression* is assigned to the first *variable*, the second *return_expression* is assigned to the second *variable*, etc. The number of variables specified following the INTO keyword must exactly match the number of expressions following the RETURNING keyword and the variables must be type-compatible with their assigned expressions.

collection

A collection in which an element is created from the evaluated *return_expression*. There can be either a single collection which may be a collection of a single field or a collection of a record type, or there may be more than one collection in which case each collection must consist of a single field. The number of return expressions must match in number and order the number of fields in all specified collections. Each corresponding *return_expression* and *collection* field must be type-compatible.

Examples

Change the location to AUSTIN for department 20 in the dept table:

```
UPDATE dept SET loc = 'AUSTIN' WHERE deptno = 20;
```

For all employees with job = SALESMAN in the emp table, update the salary by 10% and increase the commission by 500.

```
UPDATE emp SET sal = sal * 1.1, comm = comm + 500 WHERE job = 'SALESMAN';
```

Functions and Operators

Advanced Server provides a large number of functions and operators for the built-in data types.

Logical Operators

The usual logical operators are available: AND, OR, NOT

SQL uses a three-valued Boolean logic where the null value represents "unknown". Observe the following truth tables:

Table - AND/OR Truth Table

a	b	a AND b	a OR b
True	True	True	True
True	False	False	True
True	Null	Null	True
False	False	False	False
False	Null	False	Null
Null	Null	Null	Null

Table - NOT Truth Table

a	NOT a
True	False
False	True
Null	Null

The operators AND and OR are commutative, that is, you can switch the left and right operand without affecting the result.

Comparison Operators

The usual comparison operators are shown in the following table.

Table - Comparison Operators

Operator	Description
\<	Less than
>	Greater than
\<=	Less than or equal to
>=	Greater than or equal to
\=	Equal
\<>	Not equal
!=	Not equal

Comparison operators are available for all data types where this makes sense. All comparison operators are binary operators that return values of type BOOLEAN; expressions like 1 \< 2 \< 3 are not valid (because there is no \< operator to compare a Boolean value with 3).

In addition to the comparison operators, the special BETWEEN construct is available.

a BETWEEN *x* AND *y*

is equivalent to

a >= *x* AND *a* \<= *y*

Similarly,

a NOT BETWEEN *x* AND *y*

is equivalent to

a \< *x* OR *a* > *y*

There is no difference between the two respective forms apart from the CPU cycles required to rewrite the first one into the second one internally.

To check whether a value is or is not null, use the constructs

expression IS NULL

expression IS NOT NULL

Do not write *expression* = NULL because NULL is not "equal to" NULL. (The null value represents an unknown value, and it is not known whether two unknown values are equal.) This behavior conforms to the SQL standard.

Some applications may expect that *expression* = NULL returns true if *expression* evaluates to the null value. It is highly recommended that these applications be modified to comply with the SQL standard.

Mathematical Functions and Operators

Mathematical operators are provided for many Advanced Server types. For types without common mathematical conventions for all possible permutations (e.g., date/time types) the actual behavior is described in subsequent sections.

The following table shows the available mathematical operators.

Table - Mathematical Operators

Operator	Description	Example	Result
+	Addition	2 + 3	5
-	Subtraction	2 – 3	-1
*	Multiplication	2 * 3	6
/	Division (See the following note.)	4 / 2	2
**	Exponentiation Operator	2 ** 3	8

Note: If the db_dialect configuration parameter in the postgresql.conf file is set to redwood, then division of a pair of INTEGER data types does not result in a truncated value. Any fractional result is retained as shown by the following example:

```
edb=# SET db_dialect TO redwood;

SET

edb=# SHOW db_dialect;

db_dialect

------------

redwood

(1 row)

edb=# SELECT CAST('10' AS INTEGER) / CAST('3' AS INTEGER) FROM dual;

?column?

--------------------

3.3333333333333333

(1 row)
```

This behavior is compatible with Oracle databases where there is no native INTEGER data type, and any INTEGER data type specification is internally converted to NUMBER(38), which results in retaining any fractional result.

If the db_dialect configuration parameter is set to postgres, then division of a pair of INTEGER data types results in a truncated value as shown by the following example:

```
edb=# SET db_dialect TO postgres;

SET
```

```
edb=# SHOW db_dialect;

db_dialect

------------

postgres

(1 row)

edb=# SELECT CAST('10' AS INTEGER) / CAST('3' AS INTEGER) FROM dual;

?column?

----------

3

(1 row)
```

This behavior is compatible with PostgreSQL databases where division involving any pair of INTEGER, SMALLINT, or BIGINT data types results in truncation of the result. The same truncated result is returned by Advanced Server when db_dialect is set to postgres as shown in the previous example.

Note however, that even when db_dialect is set to redwood, only division with a pair of INTEGER data types results in no truncation of the result. Division that includes only SMALLINT or BIGINT data types, with or without an INTEGER data type, does result in truncation in the PostgreSQL fashion without retaining the fractional portion as shown by the following where INTEGER and SMALLINT are involved in the division:

```
edb=# SHOW db_dialect;

db_dialect

------------

redwood

(1 row)

edb=# SELECT CAST('10' AS INTEGER) / CAST('3' AS SMALLINT) FROM dual;

?column?

----------

3

(1 row)
```

The following table shows the available mathematical functions. Many of these functions are provided in multiple forms with different argument types. Except where noted, any given form of a function returns the same data type as its argument. The functions working with DOUBLE PRECISION data are mostly implemented on top of the host system's C library; accuracy and behavior in boundary cases may therefore vary depending on the host system.

Table - Mathematical Functions

Function	Return Type	Description	Example	Result
ABS(*x*)	Same as *x*	Absolute value	ABS(-17.4)	17.4

CEIL(DOUBLE PRECISION or NUMBER)	Same as input	Smallest integer not less than argument	CEIL(-42.8)	-42
EXP(DOUBLE PRECISION or NUMBER)	Same as input	Exponential	EXP(1.0)	2.7182818284590452
FLOOR(DOUBLE PRECISION or NUMBER)	Same as input	Largest integer not greater than argument	FLOOR(-42.8)	43
LN(DOUBLE PRECISION or NUMBER)	Same as input	Natural logarithm	LN(2.0)	0.6931471805599453
LOG(*b* NUMBER, *x* NUMBER)	NUMBER	Logarithm to base *b*	LOG(2.0, 64.0)	6.0000000000000000
MOD(*y*, *x*)	Same as argument types	Remainder of *y*/*x*	MOD(9, 4)	1
NVL(*x*, *y*)	Same as argument types; where both arguments are of the same data type	If *x* is null, then NVL returns *y*	NVL(9, 0)	9
POWER(*a* DOUBLE PRECISION, *b* DOUBLE PRECISION)	DOUBLE PRECISION	*a* raised to the power of *b*	POWER(9.0, 3.0)	729.0000000000000000
POWER(*a* NUMBER, *b* NUMBER)	NUMBER	*a* raised to the power of *b*	POWER(9.0, 3.0)	729.0000000000000000
ROUND(DOUBLE PRECISION or NUMBER)	Same as input	Round to nearest integer	ROUND(42.4)	42
ROUND(*v* NUMBER, *s* INTEGER)	NUMBER	Round to *s* decimal places	ROUND(42.4382, 2)	42.44
SIGN(DOUBLE PRECISION or NUMBER)	Same as input	Sign of the argument (-1, 0, +1)	SIGN(-8.4)	-1
SQRT(DOUBLE PRECISION or NUMBER)	Same as input	Square root	SQRT(2.0)	1.414213562373095
TRUNC(DOUBLE PRECISION or NUMBER)	Same as input	Truncate toward zero	TRUNC(42.8)	42
TRUNC(*v* NUMBER, *s* INTEGER)	NUMBER	Truncate to s decimal places	TRUNC(42.4382, 2)	42.43
WIDTH_BUCKET(*op* NUMBER, *b1* NUMBER, *b2* NUMBER, *count* INTEGER)	INTEGER	Return the bucket to which *op* would be assigned in an equidepth histogram with *count* buckets, in the range *b1* to *b2*	WIDTH_BUCKET(5.35, 0.024, 10.06, 5)	3

The following table shows the available trigonometric functions. All trigonometric functions take arguments and return values of type DOUBLE PRECISION.

Table - Trigonometric Functions

Function	Description
ACOS(*x*)	Inverse cosine
ASIN(*x*)	Inverse sine
ATAN(*x*)	Inverse tangent
ATAN2(*x*, *y*)	Inverse tangent of *x*/*y*
COS(*x*)	Cosine
SIN(*x*)	Sine
TAN(*x*)	Tangent

String Functions and Operators

This section describes functions and operators for examining and manipulating string values. Strings in this context include values of the types CHAR, VARCHAR2, and CLOB. Unless otherwise noted, all of the functions listed below work on all of these types, but be wary of potential effects of automatic padding when using the CHAR type. Generally, the functions described here also work on data of non-string types by converting that data to a string representation first.

Table - SQL String Functions and Operators

Function	Return Type	Description	Example	Result
string || *string*	CLOB	String concatenation	'Enterprise' || 'DB'	EnterpriseDB
CONCAT(*string*, *string*)	CLOB	String concatenation	'a' || 'b'	ab
HEXTORAW(*varchar2*)	RAW	Converts a VARCHAR2 value to a RAW value	HEXTORAW('303132')	'012'
RAWTOHEX(*raw*)	VARCHAR2	Converts a RAW value to a HEXADECIMAL value	RAWTOHEX('012')	'303132'
INSTR(*string*, *set*, [*start* [, *occurrence*]])	INTEGER	Finds the location of a set of characters in a string, starting at position *start* in the string, *string*, and looking for the first, second, third and so on occurrences of the set. Returns 0 if the set is not found.	INSTR('PETER PIPER PICKED a PECK of PICKLED PEPPERS','PI',1,3)	30
INSTRB(*string*, *set*)	INTEGER	Returns the position of the *set* within the *string*. Returns 0 if *set* is not found.	INSTRB('PETER PIPER PICKED a PECK of PICKLED PEPPERS', 'PICK')	13
INSTRB(*string*, *set*, *start*)	INTEGER	Returns the position of the *set* within the *string*, beginning at *start*. Returns 0 if *set* is not found.	INSTRB('PETER PIPER PICKED a PECK of PICKLED PEPPERS','PICK', 14)	30
INSTRB(*string*, *set*, *start*, *occurrence*)	INTEGER	Returns the position of the specified *occurrence* of *set* within the *string*, beginning at *start*. Returns 0 if *set* is not found.	INSTRB('PETER PIPER PICKED a PECK of PICKLED PEPPERS','PICK', 1, 2)	30
LOWER(*string*)	CLOB	Convert *string* to lower case	LOWER('TOM')	tom

SUBSTR(*string*, *start* [, *count*])	CLOB	Extract substring starting from *start* and going for *count* characters. If *count* is not specified, the string is clipped from the start till the end.	SUBSTR('This is a test',6,2)	is
SUBSTRB(*string*, *start* [, *count*])	CLOB	Same as SUBSTR except *start* and *count* are in number of bytes.	SUBSTRB('abc',3) (assuming a double-byte character set)	c
SUBSTR2(*string*, *start* [, *count*])	CLOB	Alias for SUBSTR.	SUBSTR2('This is a test',6,2)	is
SUBSTR2(*string*, *start* [, *count*])	CLOB	Alias for SUBSTRB.	SUBSTR2('abc',3) (assuming a double-byte character set)	c
SUBSTR4(*string*, *start* [, *count*])	CLOB	Alias for SUBSTR.	SUBSTR4('This is a test',6,2)	is
SUBSTR4(*string*, *start* [, *count*])	CLOB	Alias for SUBSTRB.	SUBSTR4('abc',3) (assuming a double-byte character set)	c
SUBSTRC(*string*, *start* [, *count*])	CLOB	Alias for SUBSTR.	SUBSTRC('This is a test',6,2)	is
SUBSTRC(*string*, *start* [, *count*])	CLOB	Alias for SUBSTRB.	SUBSTRC('abc',3) (assuming a double-byte character set)	c
TRIM([LEADING | TRAILING | BOTH] [*characters*] FROM *string*)	CLOB	Remove the longest string containing only the characters (a space by default) from the start/end/both ends of the string.	TRIM(BOTH 'x' FROM 'xTomxx')	Tom
LTRIM(*string* [, *set*])	CLOB	Removes all the characters specified in *set* from the left of a given *string*. If *set* is not specified, a blank space is used as default.	LTRIM('abcdefghi', 'abc')	defghi
RTRIM(*string* [, *set*])	CLOB	Removes all the characters specified in *set* from the right of a given *string*. If *set* is not specified, a blank space is used as default.	RTRIM('abcdefghi', 'ghi')	abcdef
UPPER(*string*)	CLOB	Convert *string* to upper case	UPPER('tom')	TOM

Additional string manipulation functions are available and are listed in the following table. Some of them are used internally to implement the SQL-standard string functions listed in Table 2-17.

Table - Other String Functions

Function	Return Type	Description	Example	Result
ASCII(*string*)	INTEGER	ASCII code of the first byte of the argument	ASCII('x')	120
CHR(INTEGER)	CLOB	Character with the given ASCII code	CHR(65)	A
DECODE(*expr*, *expr1a*, *expr1b* [, *expr2a*, *expr2b*]... [, *default*])	Same as argument types of *expr1b*, *expr2b*,..., *default*	Finds first match of *expr* with *expr1a*, *expr2a*, etc. When match found, returns corresponding parameter pair, *expr1b*, *expr2b*, etc. If no match found, returns *default*. If no match found and *default* not specified, returns null.	DECODE(3, 1,'One', 2,'Two', 3,'Three', 'Not found')	Three

INITCAP(*string*)	CLOB	Convert the first letter of each word to uppercase and the rest to lowercase. Words are sequences of alphanumeric characters separated by non-alphanumeric characters.	INITCAP('hi THOMAS')	Hi Thomas
LENGTH	INTEGER	Returns the number of characters in a string value.	LENGTH('Côte d''Azur')	11
LENGTHC	INTEGER	This function is identical in functionality to LENGTH; the function name is supported for compatibility.	LENGTHC('Côte d''Azur')	11
LENGTH2	INTEGER	This function is identical in functionality to LENGTH; the function name is supported for compatibility.	LENGTH2('Côte d''Azur')	11
LENGTH4	INTEGER	This function is identical in functionality to LENGTH; the function name is supported for compatibility.	LENGTH4('Côte d''Azur')	11
LENGTHB	INTEGER	Returns the number of bytes required to hold the given value.	LENGTHB('Côte d''Azur')	12
LPAD(*string*, *length* INTEGER [, *fill*])	CLOB	Fill up *string* to size, *length* by prepending the characters, *fill* (a space by default). If *string* is already longer than *length* then it is truncated (on the right).	LPAD('hi', 5, 'xy')	xyxhi
REPLACE(*string*, *search_string* [, *replace_string*]	CLOB	Replaces one value in a string with another. If you do not specify a value for *replace_string*, the *search_string* value when found, is removed.	REPLACE('GEORGE', 'GE', 'EG')	EGOREG
RPAD(*string*, *length* INTEGER [, *fill*])	CLOB	Fill up *string* to size, *length* by appending the characters, *fill* (a space by default). If *string* is already longer than *length* then it is truncated.	RPAD('hi', 5, 'xy')	hixyx
TRANSLATE(*string*, *from*, *to*)	CLOB	Any character in *string* that matches a character in the *from* set is replaced by the corresponding character in the *to* set.	TRANSLATE('12345', '14', 'ax')	a23x5

Pattern Matching String Functions

Advanced Server offers support for the REGEXP_COUNT, REGEXP_INSTR and REGEXP_SUBSTR functions. These functions search a string for a pattern specified by a regular expression, and return information about occurrences of the pattern within the string. The pattern should be a POSIX-style regular expression; for more information about forming a POSIX-style regular expression, please refer to the core documentation at:

https://www.postgresql.org/docs/10/static/functions-matching.html

REGEXP_COUNT

REGEXP_COUNT searches a string for a regular expression, and returns a count of the times that the regular expression occurs. The signature is:

INTEGER REGEXP_COUNT (*srcstr* TEXT, *pattern* TEXT, *position* DEFAULT 1 *modifier* DEFAULT NULL)

Parameters

srcstr

> *srcstr* specifies the string to search.

pattern

> *pattern* specifies the regular expression for which REGEXP_COUNT will search.

position

> *position* is an integer value that indicates the position in the source string at which REGEXP_COUNT will begin searching. The default value is 1.

modifier

> *modifier* specifies values that control the pattern matching behavior. The default value is NULL. For a complete list of the modifiers supported by Advanced Server, see the PostgreSQL core documentation available at:
>
> https://www.postgresql.org/docs/10/static/functions-matching.html

Example

In the following simple example, REGEXP_COUNT returns a count of the number of times the letter i is used in the character string 'reinitializing':

```
edb=# SELECT REGEXP_COUNT('reinitializing', 'i', 1) FROM DUAL;

regexp_count

--------------

5

(1 row)
```

In the first example, the command instructs REGEXP_COUNT begins counting in the first position; if we modify the command to start the count on the 6th position:

```
edb=# SELECT REGEXP_COUNT('reinitializing', 'i', 6) FROM DUAL;

regexp_count

--------------

3

(1 row)
```

REGEXP_COUNT returns 3; the count now excludes any occurrences of the letter i that occur before the 6th position.

REGEXP_INSTR

REGEXP_INSTR searches a string for a POSIX-style regular expression. This function returns the position within the string where the match was located. The signature is:

INTEGER REGEXP_INSTR (*srcstr* TEXT, *pattern* TEXT, *position* INT DEFAULT 1, *occurrence* INT DEFAULT 1, *returnparam* INT DEFAULT 0, *modifier* TEXT DEFAULT NULL, *subexpression* INT DEFAULT 0,)

Parameters:

srcstr

> *srcstr* specifies the string to search.

pattern

> *pattern* specifies the regular expression for which REGEXP_INSTR will search.

position

> *position* specifies an integer value that indicates the start position in a source string. The default value is 1.

occurrence

> *occurrence* specifies which match is returned if more than one occurrence of the pattern occurs in the string that is searched. The default value is 1.

returnparam

> *returnparam* is an integer value that specifies the location within the string that REGEXP_INSTR should return. The default value is 0. Specify:
>
> 0 to return the location within the string of the first character that matches *pattern*.
>
> A value greater than 0 to return the position of the first character following the end of the *pattern*.

modifier

> *modifier* specifies values that control the pattern matching behavior. The default value is NULL. For a complete list of the modifiers supported by Advanced Server, see the PostgreSQL core documentation available at:
>
> https://www.postgresql.org/docs/10/static/functions-matching.html

subexpression

> subexpression is an integer value that identifies the portion of the *pattern* that will be returned by REGEXP_INSTR. The default value of *subexpression* is 0.
>
> If you specify a value for *subexpression*, you must include one (or more) set of parentheses in the *pattern* that isolate a portion of the value being searched for. The value specified by *subexpression* indicates which set of parentheses should be returned; for example, if *subexpression* is 2, REGEXP_INSTR will return the position of the second set of parentheses.

Example

In the following simple example, REGEXP_INSTR searches a string that contains the a phone number for the first occurrence of a pattern that contains three consecutive digits:

```
edb=# SELECT REGEXP_INSTR('800-555-1212', '[0-9][0-9][0-9]', 1, 1) FROM DUAL;

regexp_instr

--------------

1

(1 row)
```

The command instructs REGEXP_INSTR to return the position of the first occurrence. If we modify the command to return the start of the second occurrence of three consecutive digits:

```
edb=# SELECT REGEXP_INSTR('800-555-1212', '[0-9][0-9][0-9]', 1, 2) FROM DUAL;

regexp_instr
```

5

(1 row)

REGEXP_INSTR returns 5; the second occurrence of three consecutive digits begins in the 5th position.

REGEXP_SUBSTR

The REGEXP_SUBSTR function searches a string for a pattern specified by a POSIX compliant regular expression. REGEXP_SUBSTR returns the string that matches the pattern specified in the call to the function. The signature of the function is:

TEXT REGEXP_SUBSTR (*srcstr* TEXT, *pattern* TEXT, *position* INT DEFAULT 1, *occurrence* INT DEFAULT 1, *modifier* TEXT DEFAULT NULL, *subexpression* INT DEFAULT 0)

Parameters:

srcstr

> *srcstr* specifies the string to search.

pattern

> *pattern* specifies the regular expression for which REGEXP_SUBSTR will search.

position

> *position* specifies an integer value that indicates the start position in a source string. The default value is 1.

occurrence

> *occurrence* specifies which match is returned if more than one occurrence of the pattern occurs in the string that is searched. The default value is 1.

modifier

> *modifier* specifies values that control the pattern matching behavior. The default value is NULL. For a complete list of the modifiers supported by Advanced Server, see the PostgreSQL core documentation available at:
>
> https://www.postgresql.org/docs/10/static/functions-matching.html

subexpression

> subexpression is an integer value that identifies the portion of the *pattern* that will be returned by REGEXP_SUBSTR. The default value of *subexpression* is 0.
>
> If you specify a value for *subexpression*, you must include one (or more) set of parentheses in the *pattern* that isolate a portion of the value being searched for. The value specified by *subexpression* indicates which set of parentheses should be returned; for example, if *subexpression* is 2, REGEXP_SUBSTR will return the value contained within the second set of parentheses.

Example

In the following simple example, REGEXP_SUBSTR searches a string that contains a phone number for the first set of three consecutive digits:

```
edb=# SELECT REGEXP_SUBSTR('800-555-1212', '[0-9][0-9][0-9]', 1, 1) FROM DUAL;
```

```
regexp_substr

---------------

800

(1 row)
```

It locates the first occurrence of three digits and returns the string (800); if we modify the command to check for the second occurrence of three consecutive digits:

```
edb=# SELECT REGEXP_SUBSTR('800-555-1212', '[0-9][0-9][0-9]', 1, 2) FROM DUAL;

regexp_substr

---------------

555

(1 row)
```

REGEXP_SUBSTR returns 555, the contents of the second substring.

Pattern Matching Using the LIKE Operator

Advanced Server provides pattern matching using the traditional SQL LIKE operator. The syntax for the LIKE operator is as follows.

string LIKE *pattern* [ESCAPE *escape-character*]

string NOT LIKE *pattern* [ESCAPE *escape-character*]

Every *pattern* defines a set of strings. The LIKE expression returns true if *string* is contained in the set of strings represented by *pattern*. As expected, the NOT LIKE expression returns false if LIKE returns true, and vice versa. An equivalent expression is NOT (*string* LIKE *pattern*).

If *pattern* does not contain percent signs or underscore, then the pattern only represents the string itself; in that case LIKE acts like the equals operator. An underscore (_) in *pattern* stands for (matches) any single character; a percent sign (%) matches any string of zero or more characters.

Some examples:

'abc' LIKE 'abc' true

'abc' LIKE 'a%' true

'abc' LIKE '_b_' true

'abc' LIKE 'c' false

LIKE pattern matches always cover the entire string. To match a pattern anywhere within a string, the pattern must therefore start and end with a percent sign.

To match a literal underscore or percent sign without matching other characters, the respective character in *pattern* must be preceded by the escape character. The default escape character is the backslash but a different one may be selected by using the ESCAPE clause. To match the escape character itself, write two escape characters.

Note that the backslash already has a special meaning in string literals, so to write a pattern constant that contains a backslash you must write two backslashes in an SQL statement. Thus, writing a pattern that actually matches a

literal backslash means writing four backslashes in the statement. You can avoid this by selecting a different escape character with ESCAPE; then a backslash is not special to LIKE anymore. (But it is still special to the string literal parser, so you still need two of them.)

It's also possible to select no escape character by writing ESCAPE ''. This effectively disables the escape mechanism, which makes it impossible to turn off the special meaning of underscore and percent signs in the pattern.

Data Type Formatting Functions

The Advanced Server formatting functions described in the following table provide a powerful set of tools for converting various data types (date/time, integer, floating point, numeric) to formatted strings and for converting from formatted strings to specific data types. These functions all follow a common calling convention: the first argument is the value to be formatted and the second argument is a string template that defines the output or input format.

Table - Formatting Functions

Function	Return Type	Description	Example	Result
TO_CHAR(DATE [, *format*])	VARCHAR2	Convert a date/time to a string with output, *format*. If omitted default format is DD-MON-YY.	TO_CHAR(SYSDATE, 'MM/DD/YYYY HH12:MI:SS AM')	07/25/2007 09:43:02 AM
TO_CHAR(TIMESTAMP [, *format*])	VARCHAR2	Convert a timestamp to a string with output, *format*. If omitted default format is DD-MON-YY.	TO_CHAR(CURRENT_TIMESTAMP, 'MM/DD/YYYY HH12:MI:SS AM')	08/13/2015 08:55:22 PM
TO_CHAR(INTEGER [, *format*])	VARCHAR2	Convert an integer to a string with output, *format*	TO_CHAR(2412, '999,999S')	2,412+
TO_CHAR(NUMBER [, *format*])	VARCHAR2	Convert a decimal number to a string with output, *format*	TO_CHAR(10125.35, '999,999.99')	10,125.35
TO_CHAR(DOUBLE PRECISION, *format*)	VARCHAR2	Convert a floating-point number to a string with output, *format*	TO_CHAR(CAST(123.5282 AS REAL), '999.99')	123.53
TO_DATE(*string* [, *format*])	DATE	Convert a date formatted string to a DATE data type	TO_DATE('2007-07-04 13:39:10', 'YYYY-MM-DD HH24:MI:SS')	04-JUL-07 13:39:10
TO_NUMBER(*string* [, *format*])	NUMBER	Convert a number formatted string to a NUMBER data type	TO_NUMBER('2,412-', '999,999S')	-2412

TO_TIMESTAMP(*string*, *format*)	TIMESTAMP	Convert a timestamp formatted string to a TIMESTAMP data type	TO_TIMESTAMP('05 Dec 2000 08:30:25 pm', 'DD Mon YYYY hh12:mi:ss pm')	05-DEC-00 20:30:25

In an output template string (for TO_CHAR), there are certain patterns that are recognized and replaced with appropriately-formatted data from the value to be formatted. Any text that is not a template pattern is simply copied verbatim. Similarly, in an input template string (for anything but TO_CHAR), template patterns identify the parts of the input data string to be looked at and the values to be found there.

The following table shows the template patterns available for formatting date values using the TO_CHAR and TO_DATE functions.

Table - Template Date/Time Format Patterns

Pattern	Description
HH	Hour of day (01-12)
HH12	Hour of day (01-12)
HH24	Hour of day (00-23)
MI	Minute (00-59)
SS	Second (00-59)
SSSSS	Seconds past midnight (0-86399)
FF*n*	Fractional seconds where *n* is an optional integer from 1 to 9 for the number of digits to return. If omitted, the default is 6.
AM or A.M. or PM or P.M.	Meridian indicator (uppercase)
am or a.m. or pm or p.m.	Meridian indicator (lowercase)
Y,YYY	Year (4 and more digits) with comma
YEAR	Year (spelled out)
SYEAR	Year (spelled out) (BC dates prefixed by a minus sign)
YYYY	Year (4 and more digits)
SYYYY	Year (4 and more digits) (BC dates prefixed by a minus sign)
YYY	Last 3 digits of year
YY	Last 2 digits of year
Y	Last digit of year
IYYY	ISO year (4 and more digits)
IYY	Last 3 digits of ISO year
IY	Last 2 digits of ISO year
I	Last 1 digit of ISO year
BC or B.C. or AD or A.D.	Era indicator (uppercase)
bc or b.c. or ad or a.d.	Era indicator (lowercase)
MONTH	Full uppercase month name

Month	Full mixed-case month name
month	Full lowercase month name
MON	Abbreviated uppercase month name (3 chars in English, localized lengths vary)
Mon	Abbreviated mixed-case month name (3 chars in English, localized lengths vary)
mon	Abbreviated lowercase month name (3 chars in English, localized lengths vary)
MM	Month number (01-12)
DAY	Full uppercase day name
Day	Full mixed-case day name
day	Full lowercase day name
DY	Abbreviated uppercase day name (3 chars in English, localized lengths vary)
Dy	Abbreviated mixed-case day name (3 chars in English, localized lengths vary)
dy	Abbreviated lowercase day name (3 chars in English, localized lengths vary)
DDD	Day of year (001-366)
DD	Day of month (01-31)
D	Day of week (1-7; Sunday is 1)
W	Week of month (1-5) (The first week starts on the first day of the month)
WW	Week number of year (1-53) (The first week starts on the first day of the year)
IW	ISO week number of year; the first Thursday of the new year is in week 1
CC	Century (2 digits); the 21st century starts on 2001-01-01
SCC	Same as CC except BC dates are prefixed by a minus sign
J	Julian Day (days since January 1, 4712 BC)
Q	Quarter
RM	Month in Roman numerals (I-XII; I=January) (uppercase)
rm	Month in Roman numerals (i-xii; i=January) (lowercase)
RR	First 2 digits of the year when given only the last 2 digits of the year. Result is based upon an algorithm using the current year and the given 2-digit year. The first 2 digits of the given 2-digit year will be the same as the first 2 digits of the current year with the following exceptions: If the given 2-digit year is < 50 and the last 2 digits of the current year is >= 50, then the first 2 digits for the given year is 1 greater than the first 2 digits of the current year. If the given 2-digit year is >= 50 and the last 2 digits of the current year is < 50, then the first 2 digits for the given year is 1 less than the first 2 digits of the current year.
RRRR	Only affects TO_DATE function. Allows specification of 2-digit or 4-digit year. If 2-digit year given, then returns first 2 digits of year like RR format. If 4-digit year given, returns the given 4-digit year.

Certain modifiers may be applied to any template pattern to alter its behavior. For example, FMMonth is the Month pattern with the FM modifier. The following table shows the modifier patterns for date/time formatting.

Table - Template Pattern Modifiers for Date/Time Formatting

Modifier	Description	Example
FM prefix	Fill mode (suppress padding blanks and zeros)	FMMonth
TH suffix	Uppercase ordinal number suffix	DDTH
th suffix	Lowercase ordinal number suffix	DDth
FX prefix	Fixed format global option (see usage notes)	FX Month DD Day
SP suffix	Spell mode	DDSP

Usage notes for date/time formatting:

- FM suppresses leading zeroes and trailing blanks that would otherwise be added to make the output of a pattern fixed-width.
- TO_TIMESTAMP and TO_DATE skip multiple blank spaces in the input string if the FX option is not used. FX must be specified as the first item in the template. For example TO_TIMESTAMP('2000 JUN', 'YYYY MON') is correct, but TO_TIMESTAMP('2000 JUN', 'FXYYYY MON') returns an error, because TO_TIMESTAMP expects one space only.
- Ordinary text is allowed in TO_CHAR templates and will be output literally.
- In conversions from string to timestamp or date, the CC field is ignored if there is a YYY, YYYY or Y,YYY field. If CC is used with YY or Y then the year is computed as (CC-1)*100+YY.

The following table shows the template patterns available for formatting numeric values.

Table - Template Patterns for Numeric Formatting

Pattern	Description
9	Value with the specified number of digits
0	Value with leading zeroes
. (period)	Decimal point
, (comma)	Group (thousand) separator
$	Dollar sign
PR	Negative value in angle brackets
S	Sign anchored to number (uses locale)
L	Currency symbol (uses locale)
D	Decimal point (uses locale)
G	Group separator (uses locale)
MI	Minus sign specified in right-most position (if number \< 0)
RN or rn	Roman numeral (input between 1 and 3999)
V	Shift specified number of digits (see notes)

Usage notes for numeric formatting:

- 9 results in a value with the same number of digits as there are 9s. If a digit is not available it outputs a space.
- TH does not convert values less than zero and does not convert fractional numbers.

V effectively multiplies the input values by 10*^{n}*, where *n* is the number of digits following V. TO_CHAR does not support the use of V combined with a decimal point. (E.g., 99.9V99 is not allowed.)

The following table shows some examples of the use of the TO_CHAR and TO_DATE functions.

Table - TO_CHAR Examples

Expression	Result
TO_CHAR(CURRENT_TIMESTAMP, 'Day, DD HH12:MI:SS')	'Tuesday , 06 05:39:18'
TO_CHAR(CURRENT_TIMESTAMP, 'FMDay, FMDD HH12:MI:SS')	'Tuesday, 6 05:39:18'
TO_CHAR(-0.1, '99.99')	' -.10'
TO_CHAR(-0.1, 'FM9.99')	'-.1'
TO_CHAR(0.1, '0.9')	' 0.1'
TO_CHAR(12, '9990999.9')	' 0012.0'
TO_CHAR(12, 'FM9990999.9')	'0012.'
TO_CHAR(485, '999')	' 485'
TO_CHAR(-485, '999')	'-485'

TO_CHAR(1485, '9,999')	' 1,485'
TO_CHAR(1485, '9G999')	' 1,485'
TO_CHAR(148.5, '999.999')	' 148.500'
TO_CHAR(148.5, 'FM999.999')	'148.5'
TO_CHAR(148.5, 'FM999.990')	'148.500'
TO_CHAR(148.5, '999D999')	' 148.500'
TO_CHAR(3148.5, '9G999D999')	' 3,148.500'
TO_CHAR(-485, '999S')	'485-'
TO_CHAR(-485, '999MI')	'485-'
TO_CHAR(485, '999MI')	'485 '
TO_CHAR(485, 'FM999MI')	'485'
TO_CHAR(-485, '999PR')	'\<485>'
TO_CHAR(485, 'L999')	'$ 485'
TO_CHAR(485, 'RN')	' CDLXXXV'
TO_CHAR(485, 'FMRN')	'CDLXXXV'
TO_CHAR(5.2, 'FMRN')	'V'
TO_CHAR(12, '99V999')	' 12000'
TO_CHAR(12.4, '99V999')	' 12400'
TO_CHAR(12.45, '99V9')	' 125'

IMMUTABLE TO_CHAR(TIMESTAMP, format) Function

There are certain cases of the TO_CHAR function that can result in usage of an IMMUTABLE form of the function. Basically, a function is IMMUTABLE if the function does not modify the database, and the function returns the same, consistent value dependent upon only its input parameters. That is, the settings of configuration parameters, the locale, the content of the database, etc. do not affect the results returned by the function.

For more information about function volatility categories VOLATILE, STABLE, and IMMUTABLE, please see the PostgreSQL Core documentation at:

https://www.postgresql.org/docs/10/static/xfunc-volatility.html

A particular advantage of an IMMUTABLE function is that it can be used in the CREATE INDEX command to create an index based on that function.

In order for the TO_CHAR function to use the IMMUTABLE form the following conditions must be satisfied:

- The first parameter of the TO_CHAR function must be of data type TIMESTAMP.
- The format specified in the second parameter of the TO_CHAR function must not affect the return value of the function based on factors such as language, locale, etc. For example a format of 'YYYY-MM-DD HH24:MI:SS' can be used for an IMMUTABLE form of the function since, regardless of locale settings, the result of the function is the date and time expressed solely in numeric form. However, a format of 'DD-MON-YYYY' cannot be used for an IMMUTABLE form of the function because the 3-character abbreviation of the month may return different results depending upon the locale setting.

Format patterns that result in a non-immutable function include any variations of spelled out or abbreviated months (MONTH, MON), days (DAY, DY), median indicators (AM, PM), or era indicators (BC, AD).

For the following example, a table with a TIMESTAMP column is created.

```
CREATE TABLE ts_tbl (ts_col TIMESTAMP);
```

The following shows the successful creation of an index with the IMMUTABLE form of the TO_CHAR function.

```
edb=# CREATE INDEX ts_idx ON ts_tbl (TO_CHAR(ts_col,'YYYY-MM-DD HH24:MI:SS'));
```

CREATE INDEX

edb=# \dS ts_idx

Index "public.ts_idx"

Column | Type | Definition

---------+-------------------+--

to_char | character varying | to_char(ts_col, 'YYYY-MM-DD HH24:MI:SS'::character varying)

btree, for table "public.ts_tbl"

The following results in an error because the format specified in the TO_CHAR function prevents the use of the IMMUTABLE form since the 3-character month abbreviation, MON, may result in different return values based on the locale setting.

edb=# CREATE INDEX ts_idx_2 ON ts_tbl (TO_CHAR(ts_col, 'DD-MON-YYYY'));

ERROR: functions in index expression must be marked IMMUTABLE

Date/Time Functions and Operators

Table 2-25 shows the available functions for date/time value processing, with details appearing in the following subsections. The following table illustrates the behaviors of the basic arithmetic operators (+, -). For formatting functions, refer to Section 2.4.7. You should be familiar with the background information on date/time data types from Section 2.2.4.

Table - Date/Time Operators

Operator	Example	Result
+	DATE '2001-09-28' + 7	05-OCT-01 00:00:00
+	TIMESTAMP '2001-09-28 13:30:00' + 3	01-OCT-01 13:30:00
-	DATE '2001-10-01' – 7	24-SEP-01 00:00:00
-	TIMESTAMP '2001-09-28 13:30:00' - 3	25-SEP-01 13:30:00
-	TIMESTAMP '2001-09-29 03:00:00' - TIMESTAMP '2001-09-27 12:00:00'	@ 1 day 15 hours

In the date/time functions of the following table the use of the DATE and TIMESTAMP data types are interchangeable.

Table - Date/Time Functions

Function	Return Type	Description	Example	Result
ADD_MONTHS(DATE, NUMBER)	DATE	Add months to a date; see Section 2.4.8.1	ADD_MONTHS('28-FEB-97', 3.8)	31-MAY-97 00:00:00
CURRENT_DATE	DATE	Current date; see Section 2.4.8.8	CURRENT_DATE	04-JUL-07

CURRENT_TIMESTAMP	TIMESTAMP	Returns the current date and time; see Section 2.4.8.8	CURRENT_TIMESTAMP	04-JUL-07 15:33:23.48
EXTRACT(*field* FROM TIMESTAMP)	DOUBLE PRECISION	Get subfield; see Section 2.4.8.2	EXTRACT(hour FROM TIMESTAMP '2001-02-16 20:38:40')	20
LAST_DAY(DATE)	DATE	Returns the last day of the month represented by the given date. If the given date contains a time portion, it is carried forward to the result unchanged.	LAST_DAY('14-APR-98')	30-APR-98 00:00:00
LOCALTIMESTAMP [(*precision*)]	TIMESTAMP	Current date and time (start of current transaction); see Section 2.4.8.8	LOCALTIMESTAMP	04-JUL-07 15:33:23.48
MONTHS_BETWEEN(DATE, DATE)	NUMBER	Number of months between two dates; see Section 2.4.8.3	MONTHS_BETWEEN('28-FEB-07', '30-NOV-06')	3
NEXT_DAY(DATE, *dayofweek*)	DATE	Date falling on *dayofweek* following specified date; see Section 2.4.8.4	NEXT_DAY('16-APR-07','FRI')	20-APR-07 00:00:00
NEW_TIME(DATE, VARCHAR, VARCHAR)	DATE	Converts a date and time to an alternate time zone	NEW_TIME(TO_DATE '2005/05/29 01:45', 'AST', 'PST')	2005/05/29 21:45:00
NUMTODSINTERVAL(NUMBER, INTERVAL)	INTERVAL	Converts a number to a specified day or second interval; see Section 2.4.8.9.	SELECT numtodsinterval(100, ‘hour’);	4 days 04:0(

NUMTOYMINTERVAL(NUMBER, INTERVAL)	INTERVAL	Converts a number to a specified year or month interval; see Section 2.4.8.10.	SELECT numtoyminterval(100, ‘month’);	8 years 4 m
ROUND(DATE [, *format*])	DATE	Date rounded according to *format*; see Section 2.4.8.6	ROUND(TO_DATE('29-MAY-05'),'MON')	01-JUN-05 00:00:00
SYS_EXTRACT_UTC(TIMESTAMP WITH TIME ZONE)	TIMESTAMP	Returns Coordinated Universal Time	SYS_EXTRACT_UTC(CAST('24-MAR-11 12:30:00PM -04:00' AS TIMESTAMP WITH TIME ZONE))	24-MAR-11 16:30:00
SYSDATE	DATE	Returns current date and time	SYSDATE	01-AUG-12 11:12:34
SYSTIMESTAMP()	TIMESTAMP	Returns current date and time	SYSTIMESTAMP	01-AUG-12 11:11:23.66 -07:00
TRUNC(DATE [*format*])	DATE	Truncate according to format; see Section 2.4.8.7	TRUNC(TO_DATE('29-MAY-05'), 'MON')	01-MAY-05 00:00:00

ADD_MONTHS

The ADD_MONTHS functions adds (or subtracts if the second parameter is negative) the specified number of months to the given date. The resulting day of the month is the same as the day of the month of the given date except when the day is the last day of the month in which case the resulting date always falls on the last day of the month.

Any fractional portion of the number of months parameter is truncated before performing the calculation.

If the given date contains a time portion, it is carried forward to the result unchanged.

The following are examples of the ADD_MONTHS function.

```
SELECT ADD_MONTHS('13-JUN-07',4) FROM DUAL;

add_months

--------------------

13-OCT-07 00:00:00

(1 row)

SELECT ADD_MONTHS('31-DEC-06',2) FROM DUAL;

add_months

--------------------
```

28-FEB-07 00:00:00

(1 row)

SELECT ADD_MONTHS('31-MAY-04',-3) FROM DUAL;

add_months

29-FEB-04 00:00:00

(1 row)

EXTRACT

The EXTRACT function retrieves subfields such as year or hour from date/time values. The EXTRACT function returns values of type DOUBLE PRECISION. The following are valid field names:

YEAR

The year field

SELECT EXTRACT(YEAR FROM TIMESTAMP '2001-02-16 20:38:40') FROM DUAL;

date_part

2001

(1 row)

MONTH

The number of the month within the year (1 - 12)

SELECT EXTRACT(MONTH FROM TIMESTAMP '2001-02-16 20:38:40') FROM DUAL;

date_part

2

(1 row)

DAY

The day (of the month) field (1 - 31)

SELECT EXTRACT(DAY FROM TIMESTAMP '2001-02-16 20:38:40') FROM DUAL;

date_part

16

(1 row)

HOUR

The hour field (0 - 23)

SELECT EXTRACT(HOUR FROM TIMESTAMP '2001-02-16 20:38:40') FROM DUAL;

date_part

20

(1 row)

MINUTE

The minutes field (0 - 59)

SELECT EXTRACT(MINUTE FROM TIMESTAMP '2001-02-16 20:38:40') FROM DUAL;

date_part

38

(1 row)

SECOND

The seconds field, including fractional parts (0 - 59)

SELECT EXTRACT(SECOND FROM TIMESTAMP '2001-02-16 20:38:40') FROM DUAL;

date_part

40

(1 row)

MONTHS_BETWEEN

The MONTHS_BETWEEN function returns the number of months between two dates. The result is a numeric value which is positive if the first date is greater than the second date or negative if the first date is less than the second date.

The result is always a whole number of months if the day of the month of both date parameters is the same, or both date parameters fall on the last day of their respective months.

The following are some examples of the MONTHS_BETWEEN function.

SELECT MONTHS_BETWEEN('15-DEC-06','15-OCT-06') FROM DUAL;

months_between

2

(1 row)

SELECT MONTHS_BETWEEN('15-OCT-06','15-DEC-06') FROM DUAL;

months_between

-2

(1 row)

SELECT MONTHS_BETWEEN('31-JUL-00','01-JUL-00') FROM DUAL;

months_between

0.967741935

(1 row)

SELECT MONTHS_BETWEEN('01-JAN-07','01-JAN-06') FROM DUAL;

months_between

12

(1 row)

NEXT_DAY

The NEXT_DAY function returns the first occurrence of the given weekday strictly greater than the given date. At least the first three letters of the weekday must be specified - e.g., SAT. If the given date contains a time portion, it is carried forward to the result unchanged.

The following are examples of the NEXT_DAY function.

SELECT NEXT_DAY(TO_DATE('13-AUG-07','DD-MON-YY'),'SUNDAY') FROM DUAL;

next_day

19-AUG-07 00:00:00

(1 row)

SELECT NEXT_DAY(TO_DATE('13-AUG-07','DD-MON-YY'),'MON') FROM DUAL;

next_day

20-AUG-07 00:00:00

(1 row)

NEW_TIME

The NEW_TIME function converts a date and time from one time zone to another. NEW_TIME returns a value of type DATE. The syntax is:

NEW_TIME(*DATE*, *time_zone1*, *time_zone2*)

time_zone1 and *time_zone2* must be string values from the Time Zone column of the following table:

Table - Time Zones

Time Zone	Offset from UTC	Description
AST	UTC+4	Atlantic Standard Time
ADT	UTC+3	Atlantic Daylight Time
BST	UTC+11	Bering Standard Time
BDT	UTC+10	Bering Daylight Time
CST	UTC+6	Central Standard Time
CDT	UTC+5	Central Daylight Time
EST	UTC+5	Eastern Standard Time
EDT	UTC+4	Eastern Daylight Time
GMT	UTC	Greenwich Mean Time
HST	UTC+10	Alaska-Hawaii Standard Time
HDT	UTC+9	Alaska-Hawaii Daylight Time
MST	UTC+7	Mountain Standard Time
MDT	UTC+6	Mountain Daylight Time
NST	UTC+3:30	Newfoundland Standard Time
PST	UTC+8	Pacific Standard Time
PDT	UTC+7	Pacific Daylight Time
YST	UTC+9	Yukon Standard Time
YDT	UTC+8	Yukon Daylight Time

Following is an example of the NEW_TIME function.

SELECT NEW_TIME(TO_DATE('08-13-07 10:35:15','MM-DD-YY HH24:MI:SS'),'AST', 'PST') "Pacific Standard Time" FROM DUAL;

Pacific Standard Time

13-AUG-07 06:35:15

(1 row)

ROUND

The ROUND function returns a date rounded according to a specified template pattern. If the template pattern is omitted, the date is rounded to the nearest day. The following table shows the template patterns for the ROUND function.

Table - Template Date Patterns for the ROUND Function

Pattern	Description

CC, SCC	Returns January 1, *cc*01 where *cc* is first 2 digits of the given year if last 2 digits \<= 50, or 1 greater than the first 2 digits of the given year if last 2 digits > 50; (for AD years)
SYYY, YYYY, YEAR, SYEAR, YYY, YY, Y	Returns January 1, *yyyy* where *yyyy* is rounded to the nearest year; rounds down on June 30, rounds up on July 1
IYYY, IYY, IY, I	Rounds to the beginning of the ISO year which is determined by rounding down if the month and day is on or before June 30th, or by rounding up if the month and day is July 1st or later
Q	Returns the first day of the quarter determined by rounding down if the month and day is on or before the 15th of the second month of the quarter, or by rounding up if the month and day is on the 16th of the second month or later of the quarter
MONTH, MON, MM, RM	Returns the first day of the specified month if the day of the month is on or prior to the 15th; returns the first day of the following month if the day of the month is on the 16th or later
WW	Round to the nearest date that corresponds to the same day of the week as the first day of the year
IW	Round to the nearest date that corresponds to the same day of the week as the first day of the ISO year
W	Round to the nearest date that corresponds to the same day of the week as the first day of the month
DDD, DD, J	Rounds to the start of the nearest day; 11:59:59 AM or earlier rounds to the start of the same day; 12:00:00 PM or later rounds to the start of the next day
DAY, DY, D	Rounds to the nearest Sunday
HH, HH12, HH24	Round to the nearest hour
MI	Round to the nearest minute

Following are examples of usage of the ROUND function.

The following examples round to the nearest hundred years.

```
SELECT TO_CHAR(ROUND(TO_DATE('1950','YYYY'),'CC'),'DD-MON-YYYY') "Century" FROM DUAL;

Century

-------------

01-JAN-1901

(1 row)

SELECT TO_CHAR(ROUND(TO_DATE('1951','YYYY'),'CC'),'DD-MON-YYYY') "Century" FROM DUAL;

Century

-------------

01-JAN-2001

(1 row)
```

The following examples round to the nearest year.

```
SELECT TO_CHAR(ROUND(TO_DATE('30-JUN-1999','DD-MON-YYYY'),'Y'),'DD-MON-YYYY') "Year" FROM
DUAL;

Year

-------------
```

```
01-JAN-1999

(1 row)

SELECT TO_CHAR(ROUND(TO_DATE('01-JUL-1999','DD-MON-YYYY'),'Y'),'DD-MON-YYYY') "Year" FROM
DUAL;

Year

-------------

01-JAN-2000

(1 row)
```

The following examples round to the nearest ISO year. The first example rounds to 2004 and the ISO year for 2004 begins on December 29th of 2003. The second example rounds to 2005 and the ISO year for 2005 begins on January 3rd of that same year.

(An ISO year begins on the first Monday from which a 7 day span, Monday thru Sunday, contains at least 4 days of the new year. Thus, it is possible for the beginning of an ISO year to start in December of the prior year.)

```
SELECT TO_CHAR(ROUND(TO_DATE('30-JUN-2004','DD-MON-YYYY'),'IYYY'),'DD-MON-YYYY') "ISO Year"
FROM DUAL;

ISO Year

-------------

29-DEC-2003

(1 row)

SELECT TO_CHAR(ROUND(TO_DATE('01-JUL-2004','DD-MON-YYYY'),'IYYY'),'DD-MON-YYYY') "ISO Year"
FROM DUAL;

ISO Year

-------------

03-JAN-2005

(1 row)
```

The following examples round to the nearest quarter.

```
SELECT ROUND(TO_DATE('15-FEB-07','DD-MON-YY'),'Q') "Quarter" FROM DUAL;

Quarter

--------------------

01-JAN-07 00:00:00

(1 row)

SELECT ROUND(TO_DATE('16-FEB-07','DD-MON-YY'),'Q') "Quarter" FROM DUAL;

Quarter

--------------------
```

01-APR-07 00:00:00

(1 row)

The following examples round to the nearest month.

```
SELECT ROUND(TO_DATE('15-DEC-07','DD-MON-YY'),'MONTH') "Month" FROM DUAL;

Month

---------------------

01-DEC-07 00:00:00

(1 row)

SELECT ROUND(TO_DATE('16-DEC-07','DD-MON-YY'),'MONTH') "Month" FROM DUAL;

Month

---------------------

01-JAN-08 00:00:00

(1 row)
```

The following examples round to the nearest week. The first day of 2007 lands on a Monday so in the first example, January 18th is closest to the Monday that lands on January 15th. In the second example, January 19th is closer to the Monday that falls on January 22nd.

```
SELECT ROUND(TO_DATE('18-JAN-07','DD-MON-YY'),'WW') "Week" FROM DUAL;

Week

---------------------

15-JAN-07 00:00:00

(1 row)

SELECT ROUND(TO_DATE('19-JAN-07','DD-MON-YY'),'WW') "Week" FROM DUAL;

Week

---------------------

22-JAN-07 00:00:00

(1 row)
```

The following examples round to the nearest ISO week. An ISO week begins on a Monday. In the first example, January 1, 2004 is closest to the Monday that lands on December 29, 2003. In the second example, January 2, 2004 is closer to the Monday that lands on January 5, 2004.

```
SELECT ROUND(TO_DATE('01-JAN-04','DD-MON-YY'),'IW') "ISO Week" FROM DUAL;

ISO Week

---------------------

29-DEC-03 00:00:00
```

(1 row)

SELECT ROUND(TO_DATE('02-JAN-04','DD-MON-YY'),'IW') "ISO Week" FROM DUAL;

ISO Week

05-JAN-04 00:00:00

(1 row)

The following examples round to the nearest week where a week is considered to start on the same day as the first day of the month.

SELECT ROUND(TO_DATE('05-MAR-07','DD-MON-YY'),'W') "Week" FROM DUAL;

Week

08-MAR-07 00:00:00

(1 row)

SELECT ROUND(TO_DATE('04-MAR-07','DD-MON-YY'),'W') "Week" FROM DUAL;

Week

01-MAR-07 00:00:00

(1 row)

The following examples round to the nearest day.

SELECT ROUND(TO_DATE('04-AUG-07 11:59:59 AM','DD-MON-YY HH:MI:SS AM'),'J') "Day" FROM DUAL;

Day

04-AUG-07 00:00:00

(1 row)

SELECT ROUND(TO_DATE('04-AUG-07 12:00:00 PM','DD-MON-YY HH:MI:SS AM'),'J') "Day" FROM DUAL;

Day

05-AUG-07 00:00:00

(1 row)

The following examples round to the start of the nearest day of the week (Sunday).

SELECT ROUND(TO_DATE('08-AUG-07','DD-MON-YY'),'DAY') "Day of Week" FROM DUAL;

Day of Week

```
--------------------

05-AUG-07 00:00:00

(1 row)

SELECT ROUND(TO_DATE('09-AUG-07','DD-MON-YY'),'DAY') "Day of Week" FROM DUAL;

Day of Week

--------------------

12-AUG-07 00:00:00

(1 row)
```

The following examples round to the nearest hour.

```
SELECT TO_CHAR(ROUND(TO_DATE('09-AUG-07 08:29','DD-MON-YY HH:MI'),'HH'),'DD-MON-YY
HH24:MI:SS') "Hour" FROM DUAL;

Hour

--------------------

09-AUG-07 08:00:00

(1 row)

SELECT TO_CHAR(ROUND(TO_DATE('09-AUG-07 08:30','DD-MON-YY HH:MI'),'HH'),'DD-MON-YY
HH24:MI:SS') "Hour" FROM DUAL;

Hour

--------------------

09-AUG-07 09:00:00

(1 row)
```

The following examples round to the nearest minute.

```
SELECT TO_CHAR(ROUND(TO_DATE('09-AUG-07 08:30:29','DD-MON-YY HH:MI:SS'),'MI'),'DD-MON-YY
HH24:MI:SS') "Minute" FROM DUAL;

Minute

--------------------

09-AUG-07 08:30:00

(1 row)

SELECT TO_CHAR(ROUND(TO_DATE('09-AUG-07 08:30:30','DD-MON-YY HH:MI:SS'),'MI'),'DD-MON-YY
HH24:MI:SS') "Minute" FROM DUAL;

Minute

--------------------

09-AUG-07 08:31:00

(1 row)
```

TRUNC

The TRUNC function returns a date truncated according to a specified template pattern. If the template pattern is omitted, the date is truncated to the nearest day. The following table shows the template patterns for the TRUNC function.

Table - Template Date Patterns for the TRUNC Function

Pattern	Description
CC, SCC	Returns January 1, *cc*01 where *cc* is first 2 digits of the given year
SYYY, YYYY, YEAR, SYEAR, YYY, YY, Y	Returns January 1, *yyyy* where *yyyy* is the given year
IYYY, IYY, IY, I	Returns the start date of the ISO year containing the given date
Q	Returns the first day of the quarter containing the given date
MONTH, MON, MM, RM	Returns the first day of the specified month
WW	Returns the largest date just prior to, or the same as the given date that corresponds to the same day of the week as the first day of the year
IW	Returns the start of the ISO week containing the given date
W	Returns the largest date just prior to, or the same as the given date that corresponds to the same day of the week as the first day of the month
DDD, DD, J	Returns the start of the day for the given date
DAY, DY, D	Returns the start of the week (Sunday) containing the given date
HH, HH12, HH24	Returns the start of the hour
MI	Returns the start of the minute

Following are examples of usage of the TRUNC function.

The following example truncates down to the hundred years unit.

```
SELECT TO_CHAR(TRUNC(TO_DATE('1951','YYYY'),'CC'),'DD-MON-YYYY') "Century" FROM DUAL;

Century

-------------

01-JAN-1901

(1 row)
```

The following example truncates down to the year.

```
SELECT TO_CHAR(TRUNC(TO_DATE('01-JUL-1999','DD-MON-YYYY'),'Y'),'DD-MON-YYYY') "Year" FROM
DUAL;

Year

-------------

01-JAN-1999

(1 row)
```

The following example truncates down to the beginning of the ISO year.

```
SELECT TO_CHAR(TRUNC(TO_DATE('01-JUL-2004','DD-MON-YYYY'),'IYYY'),'DD-MON-YYYY') "ISO Year"
FROM DUAL;

ISO Year
```

29-DEC-2003

(1 row)

The following example truncates down to the start date of the quarter.

```
SELECT TRUNC(TO_DATE('16-FEB-07','DD-MON-YY'),'Q') "Quarter" FROM DUAL;

Quarter

---

01-JAN-07 00:00:00

(1 row)
```

The following example truncates to the start of the month.

```
SELECT TRUNC(TO_DATE('16-DEC-07','DD-MON-YY'),'MONTH') "Month" FROM DUAL;

Month

---

01-DEC-07 00:00:00

(1 row)
```

The following example truncates down to the start of the week determined by the first day of the year. The first day of 2007 lands on a Monday so the Monday just prior to January 19th is January 15th.

```
SELECT TRUNC(TO_DATE('19-JAN-07','DD-MON-YY'),'WW') "Week" FROM DUAL;

Week

---

15-JAN-07 00:00:00

(1 row)
```

The following example truncates to the start of an ISO week. An ISO week begins on a Monday. January 2, 2004 falls in the ISO week that starts on Monday, December 29, 2003.

```
SELECT TRUNC(TO_DATE('02-JAN-04','DD-MON-YY'),'IW') "ISO Week" FROM DUAL;

ISO Week

---

29-DEC-03 00:00:00

(1 row)
```

The following example truncates to the start of the week where a week is considered to start on the same day as the first day of the month.

```
SELECT TRUNC(TO_DATE('21-MAR-07','DD-MON-YY'),'W') "Week" FROM DUAL;

Week
```

```
--------------------

15-MAR-07 00:00:00

(1 row)
```

The following example truncates to the start of the day.

```
SELECT TRUNC(TO_DATE('04-AUG-07 12:00:00 PM','DD-MON-YY HH:MI:SS AM'),'J') "Day" FROM DUAL;

Day

--------------------

04-AUG-07 00:00:00

(1 row)
```

The following example truncates to the start of the week (Sunday).

```
SELECT TRUNC(TO_DATE('09-AUG-07','DD-MON-YY'),'DAY') "Day of Week" FROM DUAL;

Day of Week

--------------------

05-AUG-07 00:00:00

(1 row)
```

The following example truncates to the start of the hour.

```
SELECT TO_CHAR(TRUNC(TO_DATE('09-AUG-07 08:30','DD-MON-YY HH:MI'),'HH'),'DD-MON-YY
HH24:MI:SS') "Hour" FROM DUAL;

Hour

--------------------

09-AUG-07 08:00:00

(1 row)
```

The following example truncates to the minute.

```
SELECT TO_CHAR(TRUNC(TO_DATE('09-AUG-07 08:30:30','DD-MON-YY HH:MI:SS'),'MI'),'DD-MON-YY
HH24:MI:SS') "Minute" FROM DUAL;

Minute

--------------------

09-AUG-07 08:30:00

(1 row)
```

CURRENT DATE/TIME

Advanced Server provides a number of functions that return values related to the current date and time. These functions all return values based on the start time of the current transaction.

- CURRENT_DATE
- CURRENT_TIMESTAMP
- LOCALTIMESTAMP
- LOCALTIMESTAMP(precision)

CURRENT_DATE returns the current date and time based on the start time of the current transaction. The value of CURRENT_DATE will not change if called multiple times within a transaction.

SELECT CURRENT_DATE FROM DUAL;

date

06-AUG-07

CURRENT_TIMESTAMP returns the current date and time. When called from a single SQL statement, it will return the same value for each occurrence within the statement. If called from multiple statements within a transaction, may return different values for each occurrence. If called from a function, may return a different value than the value returned by current_timestamp in the caller.

SELECT CURRENT_TIMESTAMP, CURRENT_TIMESTAMP FROM DUAL;

current_timestamp | current_timestamp

----------+----------

02-SEP-13 17:52:29.261473 +05:00 | 02-SEP-13 17:52:29.261474 +05:00

LOCALTIMESTAMP can optionally be given a precision parameter which causes the result to be rounded to that many fractional digits in the seconds field. Without a precision parameter, the result is given to the full available precision.

SELECT LOCALTIMESTAMP FROM DUAL;

timestamp

06-AUG-07 16:11:35.973

(1 row)

SELECT LOCALTIMESTAMP(2) FROM DUAL;

timestamp

06-AUG-07 16:11:44.58

(1 row)

Since these functions return the start time of the current transaction, their values do not change during the transaction. This is considered a feature: the intent is to allow a single transaction to have a consistent notion of the “current” time, so that multiple modifications within the same transaction bear the same time stamp. Other database systems may advance these values more frequently.

NUMTODSINTERVAL

The NUMTODSINTERVAL function converts a numeric value to a time interval that includes day through second

interval units. When calling the function, specify the smallest fractional interval type to be included in the result set. The valid interval types are DAY, HOUR, MINUTE, and SECOND.

The following example converts a numeric value to a time interval that includes days and hours:

SELECT numtodsinterval(100, ‘hour’);

numtodsinterval

---

4 days 04:00:00

(1 row)

The following example converts a numeric value to a time interval that includes minutes and seconds:

SELECT numtodsinterval(100, ‘second’);

numtodsinterval

---

1 min 40 secs

(1 row)

NUMTOYMINTERVAL

The NUMTOYMINTERVAL function converts a numeric value to a time interval that includes year through month interval units. When calling the function, specify the smallest fractional interval type to be included in the result set. The valid interval types are YEAR and MONTH.

The following example converts a numeric value to a time interval that includes years and months:

SELECT numtoyminterval(100, ‘month’);

numtoyminterval

---

8 years 4 mons

(1 row)

The following example converts a numeric value to a time interval that includes years only:

SELECT numtoyminterval(100, ‘year’);

numtoyminterval

---

100 years

(1 row)

Sequence Manipulation Functions

This section describes Advanced Server's functions for operating on sequence objects. Sequence objects (also called sequence generators or just sequences) are special single-row tables created with the CREATE SEQUENCE command. A sequence object is usually used to generate unique identifiers for rows of a table. The sequence functions, listed below, provide simple, multiuser-safe methods for obtaining successive sequence values from sequence objects.

sequence.NEXTVAL

sequence.CURRVAL

sequence is the identifier assigned to the sequence in the CREATE SEQUENCE command. The following describes the usage of these functions.

NEXTVAL

Advance the sequence object to its next value and return that value. This is done atomically: even if multiple sessions execute NEXTVAL concurrently, each will safely receive a distinct sequence value.

CURRVAL

Return the value most recently obtained by NEXTVAL for this sequence in the current session. (An error is reported if NEXTVAL has never been called for this sequence in this session.) Notice that because this is returning a session-local value, it gives a predictable answer whether or not other sessions have executed NEXTVAL since the current session did.

If a sequence object has been created with default parameters, NEXTVAL calls on it will return successive values beginning with 1. Other behaviors can be obtained by using special parameters in the CREATE SEQUENCE command.

Important: To avoid blocking of concurrent transactions that obtain numbers from the same sequence, a NEXTVAL operation is never rolled back; that is, once a value has been fetched it is considered used, even if the transaction that did the NEXTVAL later aborts. This means that aborted transactions may leave unused "holes" in the sequence of assigned values.

Conditional Expressions

The following section describes the SQL-compliant conditional expressions available in Advanced Server.

CASE

The SQL CASE expression is a generic conditional expression, similar to if/else statements in other languages:

CASE WHEN *condition* THEN *result*

[WHEN ...]

[ELSE *result*]

END

CASE clauses can be used wherever an expression is valid. *condition* is an expression that returns a BOOLEAN result. If the result is true then the value of the CASE expression is the *result* that follows the condition. If the result is false any subsequent WHEN clauses are searched in the same manner. If no WHEN *condition* is true then the value of the CASE expression is the *result* in the ELSE clause. If the ELSE clause is omitted and no condition matches, the result is null.

An example:

SELECT * FROM test;

a

1

2

3

(3 rows)

SELECT a,

CASE WHEN a=1 THEN 'one'

WHEN a=2 THEN 'two'

ELSE 'other'

END

FROM test;

a | case

---+-------

1 | one

2 | two

3 | other

(3 rows)

The data types of all the *result* expressions must be convertible to a single output type.

The following “simple” CASE expression is a specialized variant of the general form above:

CASE *expression*

WHEN *value* THEN *result*

[WHEN ...]

[ELSE *result*]

END

The *expression* is computed and compared to all the *value* specifications in the WHEN clauses until one is found that is equal. If no match is found, the *result* in the ELSE clause (or a null value) is returned.

The example above can be written using the simple CASE syntax:

SELECT a,

CASE a WHEN 1 THEN 'one'

WHEN 2 THEN 'two'

ELSE 'other'

END

FROM test;

a | case

---+-------

1 | one

2 | two

3 | other

(3 rows)

A CASE expression does not evaluate any subexpressions that are not needed to determine the result. For example, this is a possible way of avoiding a division-by-zero failure:

SELECT ... WHERE CASE WHEN x \<> 0 THEN y/x > 1.5 ELSE false END;

COALESCE

The COALESCE function returns the first of its arguments that is not null. Null is returned only if all arguments are null.

COALESCE(*value* [, *value2*] ...)

It is often used to substitute a default value for null values when data is retrieved for display or further computation. For example:

SELECT COALESCE(description, short_description, '(none)') ...

Like a CASE expression, COALESCE will not evaluate arguments that are not needed to determine the result; that is, arguments to the right of the first non-null argument are not evaluated. This SQL-standard function provides capabilities similar to NVL and IFNULL, which are used in some other database systems.

NULLIF

The NULLIF function returns a null value if *value1* and *value2* are equal; otherwise it returns *value1*.

NULLIF(*value1*, *value2*)

This can be used to perform the inverse operation of the COALESCE example given above:

SELECT NULLIF(value1, '(none)') ...

If *value1* is (none), return a null, otherwise return *value1*.

NVL

The NVL function returns the first of its arguments that is not null. NVL evaluates the first expression; if that expression evaluates to NULL, NVL returns the second expression.

NVL(*expr1*, *expr2*)

The return type is the same as the argument types; all arguments must have the same data type (or be coercible to a common type). NVL returns NULL if all arguments are NULL.

The following example computes a bonus for non-commissioned employees, If an employee is a commissioned employee, this expression returns the employees commission; if the employee is not a commissioned employee (that is, his commission is NULL), this expression returns a bonus that is 10% of his salary.

```
bonus = NVL(emp.commission, emp.salary * .10)
```

NVL2

NVL2 evaluates an expression, and returns either the second or third expression, depending on the value of the first expression. If the first expression is not NULL, NVL2 returns the value in expr2; if the first expression is NULL, NVL2 returns the value in expr3.

```
NVL2(expr1, expr2, expr3)
```

The return type is the same as the argument types; all arguments must have the same data type (or be coercible to a common type).

The following example computes a bonus for commissioned employees - if a given employee is a commissioned employee, this expression returns an amount equal to 110% of his commission; if the employee is not a commissioned employee (that is, his commission is NULL), this expression returns 0.

```
bonus = NVL2(emp.commission, emp.commission * 1.1, 0)
```

GREATEST and LEAST

The GREATEST and LEAST functions select the largest or smallest value from a list of any number of expressions.

GREATEST(*value* [, *value2*] ...)

LEAST(*value* [, *value2*] ...)

The expressions must all be convertible to a common data type, which will be the type of the result. Null values in the list are ignored. The result will be null only if all the expressions evaluate to null.

Note that GREATEST and LEAST are not in the SQL standard, but are a common extension.

Aggregate Functions

Aggregate functions compute a single result value from a set of input values. The built-in aggregate functions are listed in the following tables.

Table - General-Purpose Aggregate Functions

Function	Argument Type	Return Type	Description
AVG(*expression*)	INTEGER, REAL, DOUBLE PRECISION, NUMBER	NUMBER for any integer type, DOUBLE PRECISION for a floating-point argument, otherwise the same as the argument data type	The average (arithmetic mean) of all input values
COUNT(*)		BIGINT	Number of input rows

COUNT(*expression*)	Any	BIGINT	Number of input rows for which the value of expression is not null
MAX(*expression*)	Any numeric, string, date/time, or bytea type	Same as argument type	Maximum value of expression across all input values
MIN(*expression*)	Any numeric, string, date/time, or bytea type	Same as argument type	Minimum value of expression across all input values
SUM(*expression*)	INTEGER, REAL, DOUBLE PRECISION, NUMBER	BIGINT for SMALLINT or INTEGER arguments, NUMBER for BIGINT arguments, DOUBLE PRECISION for floating-point arguments, otherwise the same as the argument data type	Sum of expression across all input values

It should be noted that except for COUNT, these functions return a null value when no rows are selected. In particular, SUM of no rows returns null, not zero as one might expect. The COALESCE function may be used to substitute zero for null when necessary.

The following table shows the aggregate functions typically used in statistical analysis. (These are separated out merely to avoid cluttering the listing of more-commonly-used aggregates.) Where the description mentions *N*, it means the number of input rows for which all the input expressions are non-null. In all cases, null is returned if the computation is meaningless, for example when *N* is zero.

Table - Aggregate Functions for Statistics

Function	Argument Type	Return Type	Description
CORR(*Y*, *X*)	DOUBLE PRECISION	DOUBLE PRECISION	Correlation coefficient
COVAR_POP(*Y*, *X*)	DOUBLE PRECISION	DOUBLE PRECISION	Population covariance
COVAR_SAMP(*Y*, *X*)	DOUBLE PRECISION	DOUBLE PRECISION	Sample covariance
REGR_AVGX(*Y*, *X*)	DOUBLE PRECISION	DOUBLE PRECISION	Average of the independent variable (sum(*X*) / *N*)
REGR_AVGY(*Y*, *X*)	DOUBLE PRECISION	DOUBLE PRECISION	Average of the dependent variable (sum(*Y*) / *N*)
REGR_COUNT(*Y*, *X*)	DOUBLE PRECISION	DOUBLE PRECISION	Number of input rows in which both expressions are nonnull
REGR_INTERCEPT(*Y*, *X*)	DOUBLE PRECISION	DOUBLE PRECISION	y-intercept of the least-squares-fit linear equation determined by the (*X*, *Y*) pairs
REGR_R2(*Y*, *X*)	DOUBLE PRECISION	DOUBLE PRECISION	Square of the correlation coefficient
REGR_SLOPE(*Y*, *X*)	DOUBLE PRECISION	DOUBLE PRECISION	Slope of the least-squares-fit linear equation determined by the (*X*, *Y*) pairs
REGR_SXX(*Y*, *X*)	DOUBLE PRECISION	DOUBLE PRECISION	Sum (X^2) – sum (X)^2 / *N* (“sum of squares” of the independent variable)

REGR_SXY(*Y*, *X*)	DOUBLE PRECISION	DOUBLE PRECISION	Sum (*X***Y*) – sum (*X*) * sum (*Y*) / *N* (“sum of products” of independent times dependent variable)
REGR_SYY(*Y*, *X*)	DOUBLE PRECISION	DOUBLE PRECISION	Sum (Y^2) – sum (Y)^2 / *N* (“sum of squares” of the dependent variable)
STDDEV(*expression*)	INTEGER, REAL, DOUBLE PRECISION, NUMBER	DOUBLE PRECISION for floating-point arguments, otherwise NUMBER	Historic alias for STDDEV_SAMP
STDDEV_POP(*expression*)	INTEGER, REAL, DOUBLE PRECISION, NUMBER	DOUBLE PRECISION for floating-point arguments, otherwise NUMBER	Population standard deviation of the input values
STDDEV_SAMP(*expression*)	INTEGER, REAL, DOUBLE PRECISION, NUMBER	DOUBLE PRECISION for floating-point arguments, otherwise NUMBER	Sample standard deviation of the input values
VARIANCE(*expression*)	INTEGER, REAL, DOUBLE PRECISION, NUMBER	DOUBLE PRECISION for floating-point arguments, otherwise NUMBER	Historical alias for VAR_SAMP
VAR_POP(*expression*)	INTEGER, REAL, DOUBLE PRECISION, NUMBER	DOUBLE PRECISION for floating-point arguments, otherwise NUMBER	Population variance of the input values (square of the population standard deviation)
VAR_SAMP(*expression*)	INTEGER, REAL, DOUBLE PRECISION, NUMBER	DOUBLE PRECISION for floating-point arguments, otherwise NUMBER	Sample variance of the input values (square of the sample standard deviation)

Subquery Expressions

This section describes the SQL-compliant subquery expressions available in Advanced Server. All of the expression forms documented in this section return Boolean (true/false) results.

EXISTS

The argument of EXISTS is an arbitrary SELECT statement, or subquery. The subquery is evaluated to determine whether it returns any rows. If it returns at least one row, the result of EXISTS is TRUE; if the subquery returns no rows, the result of EXISTS is FALSE.

EXISTS(*subquery*)

The subquery can refer to variables from the surrounding query, which will act as constants during any one evaluation of the subquery.

The subquery will generally only be executed far enough to determine whether at least one row is returned, not all the way to completion. It is unwise to write a subquery that has any side effects (such as calling sequence functions); whether the side effects occur or not may be difficult to predict.

Since the result depends only on whether any rows are returned, and not on the contents of those rows, the output list of the subquery is normally uninteresting. A common coding convention is to write all EXISTS tests in the form EXISTS(SELECT 1 WHERE ...). There are exceptions to this rule however, such as subqueries that use INTERSECT.

This simple example is like an inner join on deptno, but it produces at most one output row for each dept row, even though there are multiple matching emp rows:

SELECT dname FROM dept WHERE EXISTS (SELECT 1 FROM emp WHERE emp.deptno = dept.deptno);

dname

ACCOUNTING

RESEARCH

SALES

(3 rows)

IN

The right-hand side is a parenthesized subquery, which must return exactly one column. The left-hand expression is evaluated and compared to each row of the subquery result. The result of IN is TRUE if any equal subquery row is found. The result is FALSE if no equal row is found (including the special case where the subquery returns no rows).

expression IN (*subquery*)

Note that if the left-hand expression yields null, or if there are no equal right-hand values and at least one right-hand row yields null, the result of the IN construct will be null, not false. This is in accordance with SQL's normal rules for Boolean combinations of null values.

As with EXISTS, it's unwise to assume that the subquery will be evaluated completely.

NOT IN

The right-hand side is a parenthesized subquery, which must return exactly one column. The left-hand expression is evaluated and compared to each row of the subquery result. The result of NOT IN is TRUE if only unequal subquery rows are found (including the special case where the subquery returns no rows). The result is FALSE if any equal row is found.

expression NOT IN (*subquery*)

Note that if the left-hand expression yields null, or if there are no equal right-hand values and at least one right-hand row yields null, the result of the NOT IN construct will be null, not true. This is in accordance with SQL's normal rules for Boolean combinations of null values.

As with EXISTS, it's unwise to assume that the subquery will be evaluated completely.

ANY/SOME

The right-hand side is a parenthesized subquery, which must return exactly one column. The left-hand expression is evaluated and compared to each row of the subquery result using the given operator, which must yield a Boolean result. The result of ANY is TRUE if any true result is obtained. The result is FALSE if no true result is found (including the special case where the subquery returns no rows).

expression operator ANY (*subquery*)

expression operator SOME (*subquery*)

SOME is a synonym for ANY. IN is equivalent to = ANY.

Note that if there are no successes and at least one right-hand row yields null for the operator's result, the result of the ANY construct will be null, not false. This is in accordance with SQL's normal rules for Boolean combinations of null values.

As with EXISTS, it's unwise to assume that the subquery will be evaluated completely.

ALL

The right-hand side is a parenthesized subquery, which must return exactly one column. The left-hand expression is evaluated and compared to each row of the subquery result using the given operator, which must yield a Boolean result. The result of ALL is TRUE if all rows yield true (including the special case where the subquery returns no rows). The result is FALSE if any false result is found. The result is null if the comparison does not return false for any row, and it returns null for at least one row.

expression operator ALL (*subquery*)

NOT IN is equivalent to \<> ALL. As with EXISTS, it's unwise to assume that the subquery will be evaluated completely.

3.1 Oracle Catalog Views

The Oracle Catalog Views provide information about database objects in a manner compatible with the Oracle data dictionary views.

ALL_ALL_TABLES

The ALL_ALL_TABLES view provides information about the tables accessible by the current user.

Name	Type	Description
owner	TEXT	User name of the table's owner.
schema_name	TEXT	Name of the schema in which the table belongs.
table_name	TEXT	The name of the table.
tablespace_name	TEXT	Name of the tablespace in which the table resides if other than the default tablespace.
status	CHARACTER VARYING (5)	Included for compatibility only; always set to VALID.
temporary	TEXT	Y if the table is temporary; N if the table is permanent.

ALL_CONS_COLUMNS

The ALL_CONS_COLUMNS view provides information about the columns specified in constraints placed on tables accessible by the current user.

Name	Type	Description
owner	TEXT	User name of the constraint's owner.

schema_name	TEXT	Name of the schema in which the constraint belongs.
constraint_name	TEXT	The name of the constraint.
table_name	TEXT	The name of the table to which the constraint belongs.
column_name	TEXT	The name of the column referenced in the constraint.
position	SMALLINT	The position of the column within the object definition.
constraint_def	TEXT	The definition of the constraint.

ALL_CONSTRAINTS

The ALL_CONSTRAINTS view provides information about the constraints placed on tables accessible by the current user.

Name	Type	Description
owner	TEXT	User name of the constraint's owner.
schema_name	TEXT	Name of the schema in which the constraint belongs.
constraint_name	TEXT	The name of the constraint.
constraint_type	TEXT	The constraint type. Possible values are: C – check constraint F – foreign key constraint P – primary key constraint U – unique key constraint R – referential integrity constraint V – constraint on a view O – with read-only, on a view
table_name	TEXT	Name of the table to which the constraint belongs.
search_condition	TEXT	Search condition that applies to a check constraint.
r_owner	TEXT	Owner of a table referenced by a referential constraint.
r_constraint_name	TEXT	Name of the constraint definition for a referenced table.
delete_rule	TEXT	The delete rule for a referential constraint. Possible values are: C – cascade R – restrict N – no action
deferrable	BOOLEAN	Specified if the constraint is deferrable (T or F).
deferred	BOOLEAN	Specifies if the constraint has been deferred (T or F).
index_owner	TEXT	User name of the index owner.
index_name	TEXT	The name of the index.
constraint_def	TEXT	The definition of the constraint.

ALL_DB_LINKS

The ALL_DB_LINKS view provides information about the database links accessible by the current user.

Name	Type	Description
owner	TEXT	User name of the database link's owner.
db_link	TEXT	The name of the database link.
type	CHARACTER VARYING	Type of remote server. Value will be either REDWOOD or EDB
username	TEXT	User name of the user logging in.
host	TEXT	Name or IP address of the remote server.

ALL_DIRECTORIES

The ALL_DIRECTORIES view provides information about all directories created with the CREATE DIRECTORY command.

Name	Type	Description
owner	CHARACTER VARYING(30)	User name of the directory's owner.
directory_name	CHARACTER VARYING(30)	The alias name assigned to the directory.
directory_path	CHARACTER VARYING(4000)	The path to the directory.

ALL_IND_COLUMNS

The ALL_IND_COLUMNS view provides information about columns included in indexes on the tables accessible by the current user.

Name	Type	Description
index_owner	TEXT	User name of the index's owner.
schema_name	TEXT	Name of the schema in which the index belongs.
index_name	TEXT	The name of the index.
table_owner	TEXT	User name of the table owner.
table_name	TEXT	The name of the table to which the index belongs.
column_name	TEXT	The name of the column.
column_position	SMALLINT	The position of the column within the index.
column_length	SMALLINT	The length of the column (in bytes).
char_length	NUMERIC	The length of the column (in characters).
descend	CHARACTER(1)	Always set to Y (descending); included for compatibility only.

ALL_INDEXES

The ALL_INDEXES view provides information about the indexes on tables that may be accessed by the current user.

Name	Type	Description
owner	TEXT	User name of the index's owner.

schema_name	TEXT	Name of the schema in which the index belongs.
index_name	TEXT	The name of the index.
index_type	TEXT	The index type is always BTREE. Included for compatibility only.
table_owner	TEXT	User name of the owner of the indexed table.
table_name	TEXT	The name of the indexed table.
table_type	TEXT	Included for compatibility only. Always set to TABLE.
uniqueness	TEXT	Indicates if the index is UNIQUE or NONUNIQUE.
compression	CHARACTER(1)	Always set to N (not compressed). Included for compatibility only.
tablespace_name	TEXT	Name of the tablespace in which the table resides if other than the default tablespace.
logging	TEXT	Always set to LOGGING. Included for compatibility only.
status	TEXT	Included for compatibility only; always set to VALID.
partitioned	CHARACTER(3)	Indicates that the index is partitioned. Currently, always set to NO.
temporary	CHARACTER(1)	Indicates that an index is on a temporary table. Always set to N; included for compatibility only.
secondary	CHARACTER(1)	Included for compatibility only. Always set to N.
join_index	CHARACTER(3)	Included for compatibility only. Always set to NO.
dropped	CHARACTER(3)	Included for compatibility only. Always set to NO.

ALL_JOBS

The ALL_JOBS view provides information about all jobs that reside in the database.

Name	Type	Description
job	INTEGER	The identifier of the job (Job ID).
log_user	TEXT	The name of the user that submitted the job.
priv_user	TEXT	Same as log_user. Included for compatibility only.
schema_user	TEXT	The name of the schema used to parse the job.
last_date	TIMESTAMP WITH TIME ZONE	The last date that this job executed successfully.
last_sec	TEXT	Same as last_date.
this_date	TIMESTAMP WITH TIME ZONE	The date that the job began executing.
this_sec	TEXT	Same as this_date
next_date	TIMESTAMP WITH TIME ZONE	The next date that this job will be executed.
next_sec	TEXT	Same as next_date.
total_time	INTERVAL	The execution time of this job (in seconds).
broken	TEXT	If Y, no attempt will be made to run this job. If N, this job will attempt to execute.
interval	TEXT	Determines how often the job will repeat.
failures	BIGINT	The number of times that the job has failed to complete since it's last successful execution.
what	TEXT	The job definition (PL/SQL code block) that runs when the job executes.
nls_env	CHARACTER VARYING(4000)	Always NULL. Provided for compatibility only.

misc_env	BYTEA	Always NULL. Provided for compatibility only.
instance	NUMERIC	Always 0. Provided for compatibility only.

ALL_OBJECTS

The ALL_OBJECTS view provides information about all objects that reside in the database.

Name	Type	Description
owner	TEXT	User name of the object's owner.
schema_name	TEXT	Name of the schema in which the object belongs.
object_name	TEXT	Name of the object.
object_type	TEXT	Type of the object – possible values are: INDEX, FUNCTION, PACKAGE, PACKAGE BODY, PROCEDURE, SEQUENCE, SYNONYM, TABLE, TRIGGER, and VIEW.
status	CHARACTER VARYING	Whether or not the state of the object is valid. Currently, Included for compatibility only; always set to VALID.
temporary	TEXT	Y if a temporary object; N if this is a permanent object.

ALL_PART_KEY_COLUMNS

The ALL_PART_KEY_COLUMNS view provides information about the key columns of the partitioned tables that reside in the database.

Name	Type	Description
owner	TEXT	The owner of the table.
schema_name	TEXT	The name of the schema in which the table resides.
name	TEXT	The name of the table in which the column resides.
object_type	CHARACTER(5)	For compatibility only; always TABLE.
column_name	TEXT	The name of the column on which the key is defined.
column_position	INTEGER	1 for the first column; 2 for the second column, etc.

ALL_PART_TABLES

The ALL_PART_TABLES view provides information about all of the partitioned tables that reside in the database.

Name	Type	Description
owner	TEXT	The owner of the partitioned table.
schema_name	TEXT	The name of the schema in which the table resides.
table_name	TEXT	The name of the table.
partitioning_type	TEXT	The partitioning type used to define table partitions.
subpartitioning_type	TEXT	The subpartitioning type used to define table subpartitions.
partition_count	BIGINT	The number of partitions in the table.
def_subpartition_count	INTEGER	The number of subpartitions in the table.
partitioning_key_count	INTEGER	The number of partitioning keys specified.
subpartitioning_key_count	INTEGER	The number of subpartitioning keys specified.

Name	Type	Description
status	CHARACTER VARYING(8)	Provided for compatibility only. Always VALID.
def_tablespace_name	CHARACTER VARYING(30)	Provided for compatibility only. Always NULL.
def_pct_free	NUMERIC	Provided for compatibility only. Always NULL.
def_pct_used	NUMERIC	Provided for compatibility only. Always NULL.
def_ini_trans	NUMERIC	Provided for compatibility only. Always NULL.
def_max_trans	NUMERIC	Provided for compatibility only. Always NULL.
def_initial_extent	CHARACTER VARYING(40)	Provided for compatibility only. Always NULL.
def_next_extent	CHARACTER VARYING(40)	Provided for compatibility only. Always NULL.
def_min_extents	CHARACTER VARYING(40)	Provided for compatibility only. Always NULL.
def_max_extents	CHARACTER VARYING(40)	Provided for compatibility only. Always NULL.
def_pct_increase	CHARACTER VARYING(40)	Provided for compatibility only. Always NULL.
def_freelists	NUMERIC	Provided for compatibility only. Always NULL.
def_freelist_groups	NUMERIC	Provided for compatibility only. Always NULL.
def_logging	CHARACTER VARYING(7)	Provided for compatibility only. Always YES.
def_compression	CHARACTER VARYING(8)	Provided for compatibility only. Always NONE
def_buffer_pool	CHARACTER VARYING(7)	Provided for compatibility only. Always DEFAULT
ref_ptn_constraint_name	CHARACTER VARYING(30)	Provided for compatibility only. Always NULL
interval	CHARACTER VARYING(1000)	Provided for compatibility only. Always NULL

ALL_POLICIES

The ALL_POLICIES view provides information on all policies in the database. This view is accessible only to superusers.

Name	Type	Description
object_owner	TEXT	Name of the owner of the object.
schema_name	TEXT	Name of the schema in which the object belongs.
object_name	TEXT	Name of the object on which the policy applies.
policy_group	TEXT	Included for compatibility only; always set to an empty string.
policy_name	TEXT	Name of the policy.
pf_owner	TEXT	Name of the schema containing the policy function, or the schema containing the package that contains the policy function.
package	TEXT	Name of the package containing the policy function (if the function belongs to a package).
function	TEXT	Name of the policy function.
sel	TEXT	Whether or not the policy applies to SELECT commands. Possible values are YES or NO.
ins	TEXT	Whether or not the policy applies to INSERT commands. Possible values are YES or NO.
upd	TEXT	Whether or not the policy applies to UPDATE commands. Possible values are YES or NO.

del	TEXT	Whether or not the policy applies to DELETE commands. Possible values are YES or NO.
idx	TEXT	Whether or not the policy applies to index maintenance. Possible values are YES or NO.
chk_option	TEXT	Whether or not the check option is in force for INSERT and UPDATE commands. Possible values are YES or NO.
enable	TEXT	Whether or not the policy is enabled on the object. Possible values are YES or NO.
static_policy	TEXT	Included for compatibility only; always set to NO.
policy_type	TEXT	Included for compatibility only; always set to UNKNOWN.
long_predicate	TEXT	Included for compatibility only; always set to YES.

ALL_QUEUES

The ALL_QUEUES view provides information about any currently defined queues.

Name	Type	Description
owner	TEXT	User name of the queue owner.
name	TEXT	The name of the queue.
queue_table	TEXT	The name of the queue table in which the queue resides.
qid	OID	The system-assigned object ID of the queue.
queue_type	CHARACTER VARYING	The queue type; may be EXCEPTION_QUEUE, NON_PERSISTENT_QUEUE, or NORMAL_QUEUE.
max_retries	NUMERIC	The maximum number of dequeue attempts.
retrydelay	NUMERIC	The maximum time allowed between retries.
enqueue_enabled	CHARACTER VARYING	YES if the queue allows enqueuing; NO if the queue does not.
dequeue_enabled	CHARACTER VARYING	YES if the queue allows dequeuing; NO if the queue does not.
retention	CHARACTER VARYING	The number of seconds that a processed message is retained in the queue.
user_comment	CHARACTER VARYING	A user-specified comment.
network_name	CHARACTER VARYING	The name of the network on which the queue resides.

sharded	CHARACTER VARYING	YES if the queue resides on a sharded network; NO if the queue does not.

ALL_QUEUE_TABLES

The ALL_QUEUE_TABLES view provides information about all of the queue tables in the database.

Name	Type	Description
owner	TEXT	Role name of the owner of the queue table.
queue_table	TEXT	The user-specified name of the queue table.
type	CHARACTER VARYING	The type of data stored in the queue table.
object_type	TEXT	The user-defined payload type.
sort_order	CHARACTER VARYING	The order in which the queue table is sorted.
recipients	CHARACTER VARYING	Always SINGLE.
message_grouping	CHARACTER VARYING	Always NONE.
compatible	CHARACTER VARYING	The release number of the Advanced Server release with which this queue table is compatible.
primary_instance	NUMERIC	Always 0.
secondary_instance	NUMERIC	Always 0.
owner_instance	NUMERIC	The instance number of the instance that owns the queue table.
user_comment	CHARACTER VARYING	The user comment provided when the table was created.
secure	CHARACTER VARYING	YES indicates that the queue table is secure; NO indicates that it is not.

ALL_SEQUENCES

The ALL_SEQUENCES view provides information about all user-defined sequences on which the user has select, or update privileges.

Name	Type	Description
sequence_owner	TEXT	User name of the sequence's owner.
schema_name	TEXT	Name of the schema in which the sequence resides.
sequence_name	TEXT	Name of the sequence.
min_value	NUMERIC	The lowest value that the server will assign to the sequence.
max_value	NUMERIC	The highest value that the server will assign to the sequence.
increment_by	NUMERIC	The value added to the current sequence number to create the next sequent number.
cycle_flag	CHARACTER VARYING	Specifies if the sequence should wrap when it reaches min_value or max_value.
order_flag	CHARACTER VARYING	Will always return Y.
cache_size	NUMERIC	The number of pre-allocated sequence numbers stored in memory.
last_number	NUMERIC	The value of the last sequence number saved to disk.

ALL_SOURCE

The ALL_SOURCE view provides a source code listing of the following program types: functions, procedures, triggers, package specifications, and package bodies.

Name	Type	Description
owner	TEXT	User name of the program’s owner.
schema_name	TEXT	Name of the schema in which the program belongs.
name	TEXT	Name of the program.
type	TEXT	Type of program – possible values are: FUNCTION, PACKAGE, PACKAGE BODY, PROCEDURE, and TRIGGER.
line	INTEGER	Source code line number relative to a given program.
text	TEXT	Line of source code text.

ALL_SUBPART_KEY_COLUMNS

The ALL_SUBPART_KEY_COLUMNS view provides information about the key columns of those partitioned tables which are subpartitioned that reside in the database.

Name	Type	Description
owner	TEXT	The owner of the table.
schema_name	TEXT	The name of the schema in which the table resides.
name	TEXT	The name of the table in which the column resides.
object_type	CHARACTER(5)	For compatibility only; always TABLE.
column_name	TEXT	The name of the column on which the key is defined.
column_position	INTEGER	1 for the first column; 2 for the second column, etc.

ALL_SYNONYMS

The ALL_SYNONYMS view provides information on all synonyms that may be referenced by the current user.

Name	Type	Description
owner	TEXT	User name of the synonym's owner.
schema_name	TEXT	The name of the schema in which the synonym resides.
synonym_name	TEXT	Name of the synonym.
table_owner	TEXT	User name of the object's owner.
table_schema_name	TEXT	The name of the schema in which the table resides.
table_name	TEXT	The name of the object that the synonym refers to.
db_link	TEXT	The name of any associated database link.

ALL_TAB_COLUMNS

The ALL_TAB_COLUMNS view provides information on all columns in all user-defined tables and views.

Name	Type	Description
owner	CHARACTER VARYING	User name of the owner of the table or view in which the column resides.
schema_name	CHARACTER VARYING	Name of the schema in which the table or view resides.
table_name	CHARACTER VARYING	Name of the table or view.
column_name	CHARACTER VARYING	Name of the column.
data_type	CHARACTER VARYING	Data type of the column.
data_length	NUMERIC	Length of text columns.
data_precision	NUMERIC	Precision (number of digits) for NUMBER columns.
data_scale	NUMERIC	Scale of NUMBER columns.
nullable	CHARACTER(1)	Whether or not the column is nullable. Possible values are: Y – column is nullable; N – column does not allow null.
column_id	NUMERIC	Relative position of the column within the table or view.
data_default	CHARACTER VARYING	Default value assigned to the column.

ALL_TAB_PARTITIONS

The ALL_TAB_PARTITIONS view provides information about all of the partitions that reside in the database.

Name	Type	Description
table_owner	TEXT	The owner of the table in which the partition resides.
schema_name	TEXT	The name of the schema in which the table resides.
table_name	TEXT	The name of the table.
composite	TEXT	YES if the table is subpartitioned; NO if the table is not subpartitioned.
partition_name	TEXT	The name of the partition.
subpartition_count	BIGINT	The number of subpartitions in the partition.

Name	Type	Description
high_value	TEXT	The high partitioning value specified in the CREATE TABLE statement.
high_value_length	INTEGER	The length of the partitioning value.
partition_position	INTEGER	1 for the first partition; 2 for the second partition, etc.
tablespace_name	TEXT	The name of the tablespace in which the partition resides.
pct_free	NUMERIC	Included for compatibility only; always 0
pct_used	NUMERIC	Included for compatibility only; always 0
ini_trans	NUMERIC	Included for compatibility only; always 0
max_trans	NUMERIC	Included for compatibility only; always 0
initial_extent	NUMERIC	Included for compatibility only; always NULL
next_extent	NUMERIC	Included for compatibility only; always NULL
min_extent	NUMERIC	Included for compatibility only; always 0
max_extent	NUMERIC	Included for compatibility only; always 0
pct_increase	NUMERIC	Included for compatibility only; always 0
freelists	NUMERIC	Included for compatibility only; always NULL
freelist_groups	NUMERIC	Included for compatibility only; always NULL
logging	CHARACTER VARYING(7)	Included for compatibility only; always YES
compression	CHARACTER VARYING(8)	Included for compatibility only; always NONE
num_rows	NUMERIC	Same as pg_class.reltuples.
blocks	INTEGER	Same as pg_class.relpages.
empty_blocks	NUMERIC	Included for compatibility only; always NULL
avg_space	NUMERIC	Included for compatibility only; always NULL
chain_cnt	NUMERIC	Included for compatibility only; always NULL
avg_row_len	NUMERIC	Included for compatibility only; always NULL
sample_size	NUMERIC	Included for compatibility only; always NULL
last_analyzed	TIMESTAMP WITHOUT TIME ZONE	Included for compatibility only; always NULL
buffer_pool	CHARACTER VARYING(7)	Included for compatibility only; always NULL
global_stats	CHARACTER VARYING(3)	Included for compatibility only; always YES
user_stats	CHARACTER VARYING(3)	Included for compatibility only; always NO
backing_table	REGCLASS	Name of the partition backing table.

ALL_TAB_SUBPARTITIONS

The ALL_TAB_SUBPARTITIONS view provides information about all of the subpartitions that reside in the database.

Name	Type	Description
table_owner	TEXT	The owner of the table in which the subpartition resides.
schema_name	TEXT	The name of the schema in which the table resides.
table_name	TEXT	The name of the table.
partition_name	TEXT	The name of the partition.
subpartition_name	TEXT	The name of the subpartition.
high_value	TEXT	The high subpartitioning value specified in the CREATE TABLE statement.
high_value_length	INTEGER	The length of the subpartitioning value.

Name	Type	Description
subpartition_position	INTEGER	1 for the first subpartition; 2 for the second subpartition, etc.
tablespace_name	TEXT	The name of the tablespace in which the subpartition resides.
pct_free	NUMERIC	Included for compatibility only; always 0
pct_used	NUMERIC	Included for compatibility only; always 0
ini_trans	NUMERIC	Included for compatibility only; always 0
max_trans	NUMERIC	Included for compatibility only; always 0
initial_extent	NUMERIC	Included for compatibility only; always NULL
next_extent	NUMERIC	Included for compatibility only; always NULL
min_extent	NUMERIC	Included for compatibility only; always 0
max_extent	NUMERIC	Included for compatibility only; always 0
pct_increase	NUMERIC	Included for compatibility only; always 0
freelists	NUMERIC	Included for compatibility only; always NULL
freelist_groups	NUMERIC	Included for compatibility only; always NULL
logging	CHARACTER VARYING(7)	Included for compatibility only; always YES
compression	CHARACTER VARYING(8)	Included for compatibility only; always NONE
num_rows	NUMERIC	Same as pg_class.reltuples.
blocks	INTEGER	Same as pg_class.relpages.
empty_blocks	NUMERIC	Included for compatibility only; always NULL
avg_space	NUMERIC	Included for compatibility only; always NULL
chain_cnt	NUMERIC	Included for compatibility only; always NULL
avg_row_len	NUMERIC	Included for compatibility only; always NULL
sample_size	NUMERIC	Included for compatibility only; always NULL
last_analyzed	TIMESTAMP WITHOUT TIME ZONE	Included for compatibility only; always NULL
buffer_pool	CHARACTER VARYING(7)	Included for compatibility only; always NULL
global_stats	CHARACTER VARYING(3)	Included for compatibility only; always YES
user_stats	CHARACTER VARYING(3)	Included for compatibility only; always NO
backing_table	REGCLASS	Name of the subpartition backing table.

ALL_TABLES

The ALL_TABLES view provides information on all user-defined tables.

Name	Type	Description
owner	TEXT	User name of the table’s owner.
schema_name	TEXT	Name of the schema in which the table belongs.
table_name	TEXT	Name of the table.
tablespace_name	TEXT	Name of the tablespace in which the table resides if other than the default tablespace.
status	CHARACTER VARYING(5)	Whether or not the state of the table is valid. Currently, Included for compatibility only; always set to VALID.
temporary	CHARACTER(1)	Y if this is a temporary table; N if this is not a temporary table.

ALL_TRIGGERS

The ALL_TRIGGERS view provides information about the triggers on tables that may be accessed by the current user.

Name	Type	Description
owner	TEXT	User name of the trigger's owner.
schema_name	TEXT	The name of the schema in which the trigger resides.
trigger_name	TEXT	The name of the trigger.
trigger_type	TEXT	The type of the trigger. Possible values are: BEFORE ROW BEFORE STATEMENT AFTER ROW AFTER STATEMENT
triggering_event	TEXT	The event that fires the trigger.
table_owner	TEXT	The user name of the owner of the table on which the trigger is defined.
base_object_type	TEXT	Included for compatibility only. Value will always be TABLE.
table_name	TEXT	The name of the table on which the trigger is defined.
referencing_name	TEXT	Included for compatibility only. Value will always be REFERENCING NEW AS NEW OLD AS OLD.
status	TEXT	Status indicates if the trigger is enabled (VALID) or disabled (NOTVALID).
description	TEXT	Included for compatibility only.
trigger_body	TEXT	The body of the trigger.
action_statement	TEXT	The SQL command that executes when the trigger fires.

ALL_TYPES

The ALL_TYPES view provides information about the object types available to the current user.

Name	Type	Description
owner	TEXT	The owner of the object type.
schema_name	TEXT	The name of the schema in which the type is defined.
type_name	TEXT	The name of the type.
type_oid	OID	The object identifier (OID) of the type.
typecode	TEXT	The typecode of the type. Possible values are: OBJECT COLLECTION OTHER
attributes	INTEGER	The number of attributes in the type.

ALL_USERS

The ALL_USERS view provides information on all user names.

Name	Type	Description
username	TEXT	Name of the user.
user_id	OID	Numeric user id assigned to the user.
created	TIMESTAMP WITHOUT TIME ZONE	Included for compatibility only; always NULL.

ALL_VIEW_COLUMNS

The ALL_VIEW_COLUMNS view provides information on all columns in all user-defined views.

Name	Type	Description
owner	CHARACTER VARYING	User name of the view's owner.
schema_name	CHARACTER VARYING	Name of the schema in which the view belongs.
view_name	CHARACTER VARYING	Name of the view.
column_name	CHARACTER VARYING	Name of the column.
data_type	CHARACTER VARYING	Data type of the column.
data_length	NUMERIC	Length of text columns.
data_precision	NUMERIC	Precision (number of digits) for NUMBER columns.
data_scale	NUMERIC	Scale of NUMBER columns.
nullable	CHARACTER(1)	Whether or not the column is nullable – possible values are: Y – column is nullable; N – column does not allow null.
column_id	NUMERIC	Relative position of the column within the view.
data_default	CHARACTER VARYING	Default value assigned to the column.

ALL_VIEWS

The ALL_VIEWS view provides information about all user-defined views.

Name	Type	Description
owner	TEXT	User name of the view's owner.
schema_name	TEXT	Name of the schema in which the view belongs.
view_name	TEXT	Name of the view.
text	TEXT	The SELECT statement that defines the view.

DBA_ALL_TABLES

The DBA_ALL_TABLES view provides information about all tables in the database.

Name	Type	Description
owner	TEXT	User name of the table's owner.

schema_name	TEXT	Name of the schema in which the table belongs.
table_name	TEXT	Name of the table.
tablespace_name	TEXT	Name of the tablespace in which the table resides if other than the default tablespace.
status	CHARACTER VARYING(5)	Included for compatibility only; always set to VALID.
temporary	TEXT	Y if the table is temporary; N if the table is permanent.

DBA_CONS_COLUMNS

The DBA_CONS_COLUMNS view provides information about all columns that are included in constraints that are specified in on all tables in the database.

Name	Type	Description
owner	TEXT	User name of the constraint's owner.
schema_name	TEXT	Name of the schema in which the constraint belongs.
constraint_name	TEXT	The name of the constraint.
table_name	TEXT	The name of the table to which the constraint belongs.
column_name	TEXT	The name of the column referenced in the constraint.
position	SMALLINT	The position of the column within the object definition.
constraint_def	TEXT	The definition of the constraint.

DBA_CONSTRAINTS

The DBA_CONSTRAINTS view provides information about all constraints on tables in the database.

Name	Type	Description
owner	TEXT	User name of the constraint's owner.
schema_name	TEXT	Name of the schema in which the constraint belongs.
constraint_name	TEXT	The name of the constraint.
constraint_type	TEXT	The constraint type. Possible values are: C – check constraint F – foreign key constraint P – primary key constraint U – unique key constraint R – referential integrity constraint V – constraint on a view O – with read-only, on a view
table_name	TEXT	Name of the table to which the constraint belongs.
search_condition	TEXT	Search condition that applies to a check constraint.
r_owner	TEXT	Owner of a table referenced by a referential constraint.

r_constraint_name	TEXT	Name of the constraint definition for a referenced table.
delete_rule	TEXT	The delete rule for a referential constraint. Possible values are: C – cascade R - restrict N – no action
deferrable	BOOLEAN	Specified if the constraint is deferrable (T or F).
deferred	BOOLEAN	Specifies if the constraint has been deferred (T or F).
index_owner	TEXT	User name of the index owner.
index_name	TEXT	The name of the index.
constraint_def	TEXT	The definition of the constraint.

DBA_DB_LINKS

The DBA_DB_LINKS view provides information about all database links in the database.

Name	Type	Description
owner	TEXT	User name of the database link's owner.
db_link	TEXT	The name of the database link.
type	CHARACTER VARYING	Type of remote server. Value will be either REDWOOD or EDB
username	TEXT	User name of the user logging in.
host	TEXT	Name or IP address of the remote server.

DBA_DIRECTORIES

The DBA_DIRECTORIES view provides information about all directories created with the CREATE DIRECTORY command.

Name	Type	Description
owner	CHARACTER VARYING(30)	User name of the directory's owner.
directory_name	CHARACTER VARYING(30)	The alias name assigned to the directory.
directory_path	CHARACTER VARYING(4000)	The path to the directory.

DBA_IND_COLUMNS

The DBA_IND_COLUMNS view provides information about all columns included in indexes, on all tables in the database.

Name	Type	Description
index_owner	TEXT	User name of the index's owner.
schema_name	TEXT	Name of the schema in which the index belongs.
index_name	TEXT	Name of the index.
table_owner	TEXT	User name of the table's owner.

table_name	TEXT	Name of the table in which the index belongs.
column_name	TEXT	Name of column or attribute of object column.
column_position	SMALLINT	The position of the column in the index.
column_length	SMALLINT	The length of the column (in bytes).
char_length	NUMERIC	The length of the column (in characters).
descend	CHARACTER(1)	Always set to Y (descending); included for compatibility only.

DBA_INDEXES

The DBA_INDEXES view provides information about all indexes in the database.

Name	Type	Description
owner	TEXT	User name of the index's owner.
schema_name	TEXT	Name of the schema in which the index resides.
index_name	TEXT	The name of the index.
index_type	TEXT	The index type is always BTREE. Included for compatibility only.
table_owner	TEXT	User name of the owner of the indexed table.
table_name	TEXT	The name of the indexed table.
table_type	TEXT	Included for compatibility only. Always set to TABLE.
uniqueness	TEXT	Indicates if the index is UNIQUE or NONUNIQUE.
compression	CHARACTER(1)	Always set to N (not compressed). Included for compatibility only.
tablespace_name	TEXT	Name of the tablespace in which the table resides if other than the default tablespace.
logging	TEXT	Included for compatibility only. Always set to LOGGING.
status	TEXT	Whether or not the state of the object is valid. (VALID or INVALID).
partitioned	CHARACTER(3)	Indicates that the index is partitioned. Always set to NO.
temporary	CHARACTER(1)	Indicates that an index is on a temporary table. Always set to N.
secondary	CHARACTER(1)	Included for compatibility only. Always set to N.
join_index	CHARACTER(3)	Included for compatibility only. Always set to NO.
dropped	CHARACTER(3)	Included for compatibility only. Always set to NO.

DBA_JOBS

The DBA_JOBS view provides information about all jobs in the database.

Name	Type	Description
job	INTEGER	The identifier of the job (Job ID).
log_user	TEXT	The name of the user that submitted the job.
priv_user	TEXT	Same as log_user. Included for compatibility only.
schema_user	TEXT	The name of the schema used to parse the job.
last_date	TIMESTAMP WITH TIME ZONE	The last date that this job executed successfully.
last_sec	TEXT	Same as last_date.
this_date	TIMESTAMP WITH TIME ZONE	The date that the job began executing.
this_sec	TEXT	Same as this_date

next_date	TIMESTAMP WITH TIME ZONE	The next date that this job will be executed.
next_sec	TEXT	Same as next_date.
total_time	INTERVAL	The execution time of this job (in seconds).
broken	TEXT	If Y, no attempt will be made to run this job. If N, this job will attempt to execute.
interval	TEXT	Determines how often the job will repeat.
failures	BIGINT	The number of times that the job has failed to complete since it's last successful execution.
what	TEXT	The job definition (PL/SQL code block) that runs when the job executes.
nls_env	CHARACTER VARYING(4000)	Always NULL. Provided for compatibility only.
misc_env	BYTEA	Always NULL. Provided for compatibility only.
instance	NUMERIC	Always 0. Provided for compatibility only.

DBA_OBJECTS

The DBA_OBJECTS view provides information about all objects in the database.

Name	Type	Description
owner	TEXT	User name of the object's owner.
schema_name	TEXT	Name of the schema in which the object belongs.
object_name	TEXT	Name of the object.
object_type	TEXT	Type of the object – possible values are: INDEX, FUNCTION, PACKAGE, PACKAGE BODY, PROCEDURE, SEQUENCE, SYNONYM, TABLE, TRIGGER, and VIEW.
status	CHARACTER VARYING	Included for compatibility only; always set to VALID.
temporary	TEXT	Y if the table is temporary; N if the table is permanent.

DBA_PART_KEY_COLUMNS

The DBA_PART_KEY_COLUMNS view provides information about the key columns of the partitioned tables that reside in the database.

Name	Type	Description
owner	TEXT	The owner of the table.
schema_name	TEXT	The name of the schema in which the table resides.
name	TEXT	The name of the table in which the column resides.
object_type	CHARACTER(5)	For compatibility only; always TABLE.
column_name	TEXT	The name of the column on which the key is defined.
column_position	INTEGER	1 for the first column; 2 for the second column, etc.

DBA_PART_TABLES

The DBA_PART_TABLES view provides information about all of the partitioned tables in the database.

Name	Type	Description
owner	TEXT	The owner of the partitioned table.
schema_name	TEXT	The schema in which the table resides.
table_name	TEXT	The name of the table.
partitioning_type	TEXT	The type used to define table partitions.
subpartitioning_type	TEXT	The subpartitioning type used to define table subpartitions.
partition_count	BIGINT	The number of partitions in the table.
def_subpartition_count	INTEGER	The number of subpartitions in the table.
partitioning_key_count	INTEGER	The number of partitioning keys specified.
subpartitioning_key_count	INTEGER	The number of subpartitioning keys specified.
status	CHARACTER VARYING(8)	Provided for compatibility only. Always VALID.
def_tablespace_name	CHARACTER VARYING(30)	Provided for compatibility only. Always NULL.
def_pct_free	NUMERIC	Provided for compatibility only. Always NULL.
def_pct_used	NUMERIC	Provided for compatibility only. Always NULL.
def_ini_trans	NUMERIC	Provided for compatibility only. Always NULL.
def_max_trans	NUMERIC	Provided for compatibility only. Always NULL.
def_initial_extent	CHARACTER VARYING(40)	Provided for compatibility only. Always NULL.
def_next_extent	CHARACTER VARYING(40)	Provided for compatibility only. Always NULL.
def_min_extents	CHARACTER VARYING(40)	Provided for compatibility only. Always NULL.
def_max_extents	CHARACTER VARYING(40)	Provided for compatibility only. Always NULL.
def_pct_increase	CHARACTER VARYING(40)	Provided for compatibility only. Always NULL.
def_freelists	NUMERIC	Provided for compatibility only. Always NULL.
def_freelist_groups	NUMERIC	Provided for compatibility only. Always NULL.
def_logging	CHARACTER VARYING(7)	Provided for compatibility only. Always YES.
def_compression	CHARACTER VARYING(8)	Provided for compatibility only. Always NONE
def_buffer_pool	CHARACTER VARYING(7)	Provided for compatibility only. Always DEFAULT
ref_ptn_constraint_name	CHARACTER VARYING(30)	Provided for compatibility only. Always NULL
interval	CHARACTER VARYING(1000)	Provided for compatibility only. Always NULL

DBA_POLICIES

The DBA_POLICIES view provides information on all policies in the database. This view is accessible only to superusers.

Name	Type	Description
object_owner	TEXT	Name of the owner of the object.
schema_name	TEXT	The name of the schema in which the object resides.
object_name	TEXT	Name of the object to which the policy applies.

policy_group	TEXT	Name of the policy group. Included for compatibility only; always set to an empty string.
policy_name	TEXT	Name of the policy.
pf_owner	TEXT	Name of the schema containing the policy function, or the schema containing the package that contains the policy function.
package	TEXT	Name of the package containing the policy function (if the function belongs to a package).
function	TEXT	Name of the policy function.
sel	TEXT	Whether or not the policy applies to SELECT commands. Possible values are YES or NO.
ins	TEXT	Whether or not the policy applies to INSERT commands. Possible values are YES or NO.
upd	TEXT	Whether or not the policy applies to UPDATE commands. Possible values are YES or NO.
del	TEXT	Whether or not the policy applies to DELETE commands. Possible values are YES or NO.
idx	TEXT	Whether or not the policy applies to index maintenance. Possible values are YES or NO.
chk_option	TEXT	Whether or not the check option is in force for INSERT and UPDATE commands. Possible values are YES or NO.
Enable	TEXT	Whether or not the policy is enabled on the object. Possible values are YES or NO.
static_policy	TEXT	Included for compatibility only; always set to NO.
policy_type	TEXT	Included for compatibility only; always set to UNKNOWN.
long_predicate	TEXT	Included for compatibility only; always set to YES.

DBA_PROFILES

The DBA_PROFILES view provides information about existing profiles. The table includes a row for each profile/resource combination.

Name	Type	Description
profile	CHARACTER VARYING(128)	The name of the profile.
resource_name	CHARACTER VARYING(32)	The name of the resource associated with the profile.
resource_type	CHARACTER VARYING(8)	The type of resource governed by the profile; currently PASSWORD for all supported resources.
limit	CHARACTER VARYING(128)	The limit values of the resource.
common	CHARACTER VARYING(3)	YES for a user-created profile; NO for a system-defined profile.

DBA_QUEUES

The DBA_QUEUES view provides information about any currently defined queues.

Name	Type	Description
owner	TEXT	User name of the queue owner.
name	TEXT	The name of the queue.
queue_table	TEXT	The name of the queue table in which the queue resides.
qid	OID	The system-assigned object ID of the queue.

queue_type	CHARACTER VARYING	The queue type; may be EXCEPTION_QUEUE, NON_PERSISTENT_QUEUE, or NORMAL_QUEUE.
max_retries	NUMERIC	The maximum number of dequeue attempts.
retrydelay	NUMERIC	The maximum time allowed between retries.
enqueue_enabled	CHARACTER VARYING	YES if the queue allows enqueuing; NO if the queue does not.
dequeue_enabled	CHARACTER VARYING	YES if the queue allows dequeuing; NO if the queue does not.
retention	CHARACTER VARYING	The number of seconds that a processed message is retained in the queue.
user_comment	CHARACTER VARYING	A user-specified comment.
network_name	CHARACTER VARYING	The name of the network on which the queue resides.
sharded	CHARACTER VARYING	YES if the queue resides on a sharded network; NO if the queue does not.

DBA_QUEUE_TABLES

The DBA_QUEUE_TABLES view provides information about all of the queue tables in the database.

Name	Type	Description
owner	TEXT	Role name of the owner of the queue table.
queue_table	TEXT	The user-specified name of the queue table.
type	CHARACTER VARYING	The type of data stored in the queue table.
object_type	TEXT	The user-defined payload type.
sort_order	CHARACTER VARYING	The order in which the queue table is sorted.

recipients	CHARACTER VARYING	Always SINGLE.
message_grouping	CHARACTER VARYING	Always NONE.
compatible	CHARACTER VARYING	The release number of the Advanced Server release with which this queue table is compatible.
primary_instance	NUMERIC	Always 0.
secondary_instance	NUMERIC	Always 0.
owner_instance	NUMERIC	The instance number of the instance that owns the queue table.
user_comment	CHARACTER VARYING	The user comment provided when the table was created.
secure	CHARACTER VARYING	YES indicates that the queue table is secure; NO indicates that it is not.

DBA_ROLE_PRIVS

The DBA_ROLE_PRIVS view provides information on all roles that have been granted to users. A row is created for each role to which a user has been granted.

Name	Type	Description
grantee	TEXT	User name to whom the role was granted.
granted_role	TEXT	Name of the role granted to the grantee.
admin_option	TEXT	YES if the role was granted with the admin option, NO otherwise.
default_role	TEXT	YES if the role is enabled when the grantee creates a session.

DBA_ROLES

The DBA_ROLES view provides information on all roles with the NOLOGIN attribute (groups).

Name	Type	Description
role	TEXT	Name of a role having the NOLOGIN attribute – i.e., a group.
password_required	TEXT	Included for compatibility only; always N.

DBA_SEQUENCES

The DBA_SEQUENCES view provides information about all user-defined sequences.

Name	Type	Description
sequence_owner	TEXT	User name of the sequence's owner.
schema_name	TEXT	The name of the schema in which the sequence resides.
sequence_name	TEXT	Name of the sequence.
min_value	NUMERIC	The lowest value that the server will assign to the sequence.
max_value	NUMERIC	The highest value that the server will assign to the sequence.
increment_by	NUMERIC	The value added to the current sequence number to create the next sequent number.
cycle_flag	CHARACTER VARYING	Specifies if the sequence should wrap when it reaches min_value or max_value.
order_flag	CHARACTER VARYING	Will always return Y.
cache_size	NUMERIC	The number of pre-allocated sequence numbers stored in memory.
last_number	NUMERIC	The value of the last sequence number saved to disk.

DBA_SOURCE

The DBA_SOURCE view provides the source code listing of all objects in the database.

Name	Type	Description
owner	TEXT	User name of the program’s owner.
schema_name	TEXT	Name of the schema in which the program belongs.
name	TEXT	Name of the program.
type	TEXT	Type of program – possible values are: FUNCTION, PACKAGE, PACKAGE BODY, PROCEDURE, and TRIGGER.
line	INTEGER	Source code line number relative to a given program.
text	TEXT	Line of source code text.

DBA_SUBPART_KEY_COLUMNS

The DBA_SUBPART_KEY_COLUMNS view provides information about the key columns of those partitioned tables which are subpartitioned that reside in the database.

Name	Type	Description
owner	TEXT	The owner of the table.
schema_name	TEXT	The name of the schema in which the table resides.
name	TEXT	The name of the table in which the column resides.
object_type	CHARACTER(5)	For compatibility only; always TABLE.
column_name	TEXT	The name of the column on which the key is defined.
column_position	INTEGER	1 for the first column; 2 for the second column, etc.

DBA_SYNONYMS

The DBA_SYNONYM view provides information about all synonyms in the database.

Name	Type	Description

owner	TEXT	User name of the synonym's owner.
schema_name	TEXT	Name of the schema in which the synonym belongs.
synonym_name	TEXT	Name of the synonym.
table_owner	TEXT	User name of the table's owner on which the synonym is defined.
table_schema_name	TEXT	The name of the schema in which the table resides.
table_name	TEXT	Name of the table on which the synonym is defined.
db_link	TEXT	Name of any associated database link.

DBA_TAB_COLUMNS

The DBA_TAB_COLUMNS view provides information about all columns in the database.

Name	Type	Description
owner	CHARACTER VARYING	User name of the owner of the table or view in which the column resides.
schema_name	CHARACTER VARYING	Name of the schema in which the table or view resides.
table_name	CHARACTER VARYING	Name of the table or view in which the column resides.
column_name	CHARACTER VARYING	Name of the column.
data_type	CHARACTER VARYING	Data type of the column.
data_length	NUMERIC	Length of text columns.
data_precision	NUMERIC	Precision (number of digits) for NUMBER columns.
data_scale	NUMERIC	Scale of NUMBER columns.
nullable	CHARACTER(1)	Whether or not the column is nullable – possible values are: Y – column is nullable; N – column does not allow null.
column_id	NUMERIC	Relative position of the column within the table or view.
data_default	CHARACTER VARYING	Default value assigned to the column.

DBA_TAB_PARTITIONS

The DBA_TAB_PARTITIONS view provides information about all of the partitions that reside in the database.

Name	Type	Description
table_owner	TEXT	The owner of the table in which the partition resides.
schema_name	TEXT	The name of the schema in which the table resides.
table_name	TEXT	The name of the table.
composite	TEXT	YES if the table is subpartitioned; NO if the table is not subpartitioned.
partition_name	TEXT	The name of the partition.
subpartition_count	BIGINT	The number of subpartitions in the partition.
high_value	TEXT	The high partitioning value specified in the CREATE TABLE statement.
high_value_length	INTEGER	The length of the partitioning value.
partition_position	INTEGER	1 for the first partition; 2 for the second partition, etc.

Name	Type	Description
tablespace_name	TEXT	The name of the tablespace in which the partition resides.
pct_free	NUMERIC	Included for compatibility only; always 0
pct_used	NUMERIC	Included for compatibility only; always 0
ini_trans	NUMERIC	Included for compatibility only; always 0
max_trans	NUMERIC	Included for compatibility only; always 0
initial_extent	NUMERIC	Included for compatibility only; always NULL
next_extent	NUMERIC	Included for compatibility only; always NULL
min_extent	NUMERIC	Included for compatibility only; always 0
max_extent	NUMERIC	Included for compatibility only; always 0
pct_increase	NUMERIC	Included for compatibility only; always 0
freelists	NUMERIC	Included for compatibility only; always NULL
freelist_groups	NUMERIC	Included for compatibility only; always NULL
logging	CHARACTER VARYING(7)	Included for compatibility only; always YES
compression	CHARACTER VARYING(8)	Included for compatibility only; always NONE
num_rows	NUMERIC	Same as pg_class.reltuples.
blocks	INTEGER	Same as pg_class.relpages.
empty_blocks	NUMERIC	Included for compatibility only; always NULL
avg_space	NUMERIC	Included for compatibility only; always NULL
chain_cnt	NUMERIC	Included for compatibility only; always NULL
avg_row_len	NUMERIC	Included for compatibility only; always NULL
sample_size	NUMERIC	Included for compatibility only; always NULL
last_analyzed	TIMESTAMP WITHOUT TIME ZONE	Included for compatibility only; always NULL
buffer_pool	CHARACTER VARYING(7)	Included for compatibility only; always NULL
global_stats	CHARACTER VARYING(3)	Included for compatibility only; always YES
user_stats	CHARACTER VARYING(3)	Included for compatibility only; always NO
backing_table	REGCLASS	Name of the partition backing table.

DBA_TAB_SUBPARTITIONS

The DBA_TAB_SUBPARTITIONS view provides information about all of the subpartitions that reside in the database.

Name	Type	Description
table_owner	TEXT	The owner of the table in which the subpartition resides.
schema_name	TEXT	The name of the schema in which the table resides.
table_name	TEXT	The name of the table.
partition_name	TEXT	The name of the partition.
subpartition_name	TEXT	The name of the subpartition.
high_value	TEXT	The high subpartitioning value specified in the CREATE TABLE statement.
high_value_length	INTEGER	The length of the subpartitioning value.
subpartition_position	INTEGER	1 for the first subpartition; 2 for the second subpartition, etc.
tablespace_name	TEXT	The name of the tablespace in which the subpartition resides.
pct_free	NUMERIC	Included for compatibility only; always 0
pct_used	NUMERIC	Included for compatibility only; always 0

Name	Type	Description
ini_trans	NUMERIC	Included for compatibility only; always 0
max_trans	NUMERIC	Included for compatibility only; always 0
initial_extent	NUMERIC	Included for compatibility only; always NULL
next_extent	NUMERIC	Included for compatibility only; always NULL
min_extent	NUMERIC	Included for compatibility only; always 0
max_extent	NUMERIC	Included for compatibility only; always 0
pct_increase	NUMERIC	Included for compatibility only; always 0
freelists	NUMERIC	Included for compatibility only; always NULL
freelist_groups	NUMERIC	Included for compatibility only; always NULL
logging	CHARACTER VARYING(7)	Included for compatibility only; always YES
compression	CHARACTER VARYING(8)	Included for compatibility only; always NONE
num_rows	NUMERIC	Same as pg_class.reltuples.
blocks	INTEGER	Same as pg_class.relpages.
empty_blocks	NUMERIC	Included for compatibility only; always NULL
avg_space	NUMERIC	Included for compatibility only; always NULL
chain_cnt	NUMERIC	Included for compatibility only; always NULL
avg_row_len	NUMERIC	Included for compatibility only; always NULL
sample_size	NUMERIC	Included for compatibility only; always NULL
last_analyzed	TIMESTAMP WITHOUT TIME ZONE	Included for compatibility only; always NULL
buffer_pool	CHARACTER VARYING(7)	Included for compatibility only; always NULL
global_stats	CHARACTER VARYING(3)	Included for compatibility only; always YES
user_stats	CHARACTER VARYING(3)	Included for compatibility only; always NO
backing_table	REGCLASS	Name of the subpartition backing table.

DBA_TABLES

The DBA_TABLES view provides information about all tables in the database.

Name	Type	Description
owner	TEXT	User name of the table’s owner.
schema_name	TEXT	Name of the schema in which the table belongs.
table_name	TEXT	Name of the table.
tablespace_name	TEXT	Name of the tablespace in which the table resides if other than the default tablespace.
status	CHARACTER VARYING(5)	Included for compatibility only; always set to VALID.
temporary	CHARACTER(1)	Y if the table is temporary; N if the table is permanent.

DBA_TRIGGERS

The DBA_TRIGGERS view provides information about all triggers in the database.

Name	Type	Description
owner	TEXT	User name of the trigger’s owner.
schema_name	TEXT	The name of the schema in which the trigger resides.

trigger_name	TEXT	The name of the trigger.
trigger_type	TEXT	The type of the trigger. Possible values are: BEFORE ROW BEFORE STATEMENT AFTER ROW AFTER STATEMENT
triggering_event	TEXT	The event that fires the trigger.
table_owner	TEXT	The user name of the owner of the table on which the trigger is defined.
base_object_type	TEXT	Included for compatibility only. Value will always be TABLE.
table_name	TEXT	The name of the table on which the trigger is defined.
referencing_names	TEXT	Included for compatibility only. Value will always be REFERENCING NEW AS NEW OLD AS OLD.
status	TEXT	Status indicates if the trigger is enabled (VALID) or disabled (NOTVALID).
description	TEXT	Included for compatibility only.
trigger_body	TEXT	The body of the trigger.
action_statement	TEXT	The SQL command that executes when the trigger fires.

DBA_TYPES

The DBA_TYPES view provides information about all object types in the database.

Name	Type	Description
owner	TEXT	The owner of the object type.
schema_name	TEXT	The name of the schema in which the type is defined.
type_name	TEXT	The name of the type.
type_oid	OID	The object identifier (OID) of the type.
typecode	TEXT	The typecode of the type. Possible values are: OBJECT COLLECTION OTHER
attributes	INTEGER	The number of attributes in the type.

DBA_USERS

The DBA_USERS view provides information about all users of the database.

Name	Type	Description
username	TEXT	User name of the user.
user_id	OID	ID number of the user.

password	CHARACTER VARYING(30)	The password (encrypted) of the user.
account_status	CHARACTER VARYING(32)	The current status of the account. Possible values are: OPEN EXPIRED EXPIRED(GRACE) EXPIRED & LOCKED EXPIRED & LOCKED(TIMED) EXPIRED(GRACE) & LOCKED EXPIRED(GRACE) & LOCKED(TIMED) LOCKED LOCKED(TIMED) Use the edb_get_role_status(*role_id*) function to get the current status of the account.
lock_date	TIMESTAMP WITHOUT TIME ZONE	If the account status is LOCKED, lock_date displays the date and time the account was locked.
expiry_date	TIMESTAMP WITHOUT TIME ZONE	The expiration date of the password. Use the edb_get_password_expiry_date(*role_id*) function to get the current password expiration date.
default_tablespace	TEXT	The default tablespace associated with the account.
temporary_tablespace	CHARACTER VARYING(30)	Included for compatibility only. The value will always be '' (an empty string).
created	TIMESTAMP WITHOUT TIME ZONE	Included for compatibility only. The value is always NULL.
profile	CHARACTER VARYING(30)	The profile associated with the user.
initial_rsrc_consumer_group	CHARACTER VARYING(30)	Included for compatibility only. The value is always NULL.
external_name	CHARACTER VARYING(4000)	Included for compatibility only. The value is always NULL.

DBA_VIEW_COLUMNS

The DBA_VIEW_COLUMNS view provides information on all columns in the database.

Name	Type	Description
owner	CHARACTER VARYING	User name of the view's owner.
schema_name	CHARACTER VARYING	Name of the schema in which the view belongs.
view_name	CHARACTER VARYING	Name of the view.
column_name	CHARACTER VARYING	Name of the column.
data_type	CHARACTER VARYING	Data type of the column.
data_length	NUMERIC	Length of text columns.
data_precision	NUMERIC	Precision (number of digits) for NUMBER columns.
data_scale	NUMERIC	Scale of NUMBER columns.
nullable	CHARACTER(1)	Whether or not the column is nullable – possible values are: Y – column is nullable; N – column does not allow null.

column_id	NUMERIC	Relative position of the column within the view.
data_default	CHARACTER VARYING	Default value assigned to the column.

DBA_VIEWS

The DBA_VIEWS view provides information about all views in the database.

Name	Type	Description
owner	TEXT	User name of the view's owner.
schema_name	TEXT	Name of the schema in which the view belongs.
view_name	TEXT	Name of the view.
text	TEXT	The text of the SELECT statement that defines the view.

USER_ALL_TABLES

The USER_ALL_TABLES view provides information about all tables owned by the current user.

Name	Type	Description
schema_name	TEXT	Name of the schema in which the table belongs.
table_name	TEXT	Name of the table.
tablespace_name	TEXT	Name of the tablespace in which the table resides if other than the default tablespace.
status	CHARACTER VARYING(5)	Included for compatibility only; always set to VALID..
temporary	TEXT	Y if the table is temporary; N if the table is permanent.

USER_CONS_COLUMNS

The USER_CONS_COLUMNS view provides information about all columns that are included in constraints in tables that are owned by the current user.

Name	Type	Description
owner	TEXT	User name of the constraint's owner.
schema_name	TEXT	Name of the schema in which the constraint belongs.
constraint_name	TEXT	The name of the constraint.
table_name	TEXT	The name of the table to which the constraint belongs.
column_name	TEXT	The name of the column referenced in the constraint.
position	SMALLINT	The position of the column within the object definition.
constraint_def	TEXT	The definition of the constraint.

USER_CONSTRAINTS

The USER_CONSTRAINTS view provides information about all constraints placed on tables that are owned by the current user.

Name	Type	Description
owner	TEXT	The name of the owner of the constraint.
schema_name	TEXT	Name of the schema in which the constraint belongs.
constraint_name	TEXT	The name of the constraint.
constraint_type	TEXT	The constraint type. Possible values are: C – check constraint F – foreign key constraint P – primary key constraint U – unique key constraint R – referential integrity constraint V – constraint on a view O – with read-only, on a view
table_name	TEXT	Name of the table to which the constraint belongs.
search_condition	TEXT	Search condition that applies to a check constraint.
r_owner	TEXT	Owner of a table referenced by a referential constraint.
r_constraint_name	TEXT	Name of the constraint definition for a referenced table.
delete_rule	TEXT	The delete rule for a referential constraint. Possible values are: C – cascade R – restrict N – no action
deferrable	BOOLEAN	Specified if the constraint is deferrable (T or F).
deferred	BOOLEAN	Specifies if the constraint has been deferred (T or F).
index_owner	TEXT	User name of the index owner.
index_name	TEXT	The name of the index.
constraint_def	TEXT	The definition of the constraint.

USER_DB_LINKS

The USER_DB_LINKS view provides information about all database links that are owned by the current user.

Name	Type	Description
db_link	TEXT	The name of the database link.
type	CHARACTER VARYING	Type of remote server. Value will be either REDWOOD or EDB
username	TEXT	User name of the user logging in.
password	TEXT	Password used to authenticate on the remote server.
host	TEXT	Name or IP address of the remote server.

USER_IND_COLUMNS

The USER_IND_COLUMNS view provides information about all columns referred to in indexes on tables that are owned by the current user.

Name	Type	Description
schema_name	TEXT	Name of the schema in which the index belongs.
index_name	TEXT	The name of the index.
table_name	TEXT	The name of the table to which the index belongs.
column_name	TEXT	The name of the column.
column_position	SMALLINT	The position of the column within the index.
column_length	SMALLINT	The length of the column (in bytes).
char_length	NUMERIC	The length of the column (in characters).
descend	CHARACTER(1)	Always set to Y (descending); included for compatibility only.

USER_INDEXES

The USER_INDEXES view provides information about all indexes on tables that are owned by the current user.

Name	Type	Description
schema_name	TEXT	Name of the schema in which the index belongs.
index_name	TEXT	The name of the index.
index_type	TEXT	Included for compatibility only. The index type is always BTREE.
table_owner	TEXT	User name of the owner of the indexed table.
table_name	TEXT	The name of the indexed table.
table_type	TEXT	Included for compatibility only. Always set to TABLE.
uniqueness	TEXT	Indicates if the index is UNIQUE or NONUNIQUE.
compression	CHARACTER(1)	Included for compatibility only. Always set to N (not compressed).
tablespace_name	TEXT	Name of the tablespace in which the table resides if other than the default tablespace.
logging	TEXT	Included for compatibility only. Always set to LOGGING.
status	TEXT	Whether or not the state of the object is valid. (VALID or INVALID).
partitioned	CHARACTER(3)	Included for compatibility only. Always set to NO.
temporary	CHARACTER(1)	Included for compatibility only. Always set to N.
secondary	CHARACTER(1)	Included for compatibility only. Always set to N.
join_index	CHARACTER(3)	Included for compatibility only. Always set to NO.
dropped	CHARACTER(3)	Included for compatibility only. Always set to NO.

USER_JOBS

The USER_JOBS view provides information about all jobs owned by the current user.

Name	Type	Description
job	INTEGER	The identifier of the job (Job ID).
log_user	TEXT	The name of the user that submitted the job.
priv_user	TEXT	Same as log_user. Included for compatibility only.
schema_user	TEXT	The name of the schema used to parse the job.
last_date	TIMESTAMP WITH TIME ZONE	The last date that this job executed successfully.

last_sec	TEXT	Same as last_date.
this_date	TIMESTAMP WITH TIME ZONE	The date that the job began executing.
this_sec	TEXT	Same as this_date.
next_date	TIMESTAMP WITH TIME ZONE	The next date that this job will be executed.
next_sec	TEXT	Same as next_date.
total_time	INTERVAL	The execution time of this job (in seconds).
broken	TEXT	If Y, no attempt will be made to run this job. If N, this job will attempt to execute.
interval	TEXT	Determines how often the job will repeat.
failures	BIGINT	The number of times that the job has failed to complete since it's last successful execution.
what	TEXT	The job definition (PL/SQL code block) that runs when the job executes.
nls_env	CHARACTER VARYING(4000)	Always NULL. Provided for compatibility only.
misc_env	BYTEA	Always NULL. Provided for compatibility only.
instance	NUMERIC	Always 0. Provided for compatibility only.

USER_OBJECTS

The USER_OBJECTS view provides information about all objects that are owned by the current user.

Name	Type	Description
schema_name	TEXT	Name of the schema in which the object belongs.
object_name	TEXT	Name of the object.
object_type	TEXT	Type of the object – possible values are: INDEX, FUNCTION, PACKAGE, PACKAGE BODY, PROCEDURE, SEQUENCE, SYNONYM, TABLE, TRIGGER, and VIEW.
status	CHARACTER VARYING	Included for compatibility only; always set to VALID.
temporary	TEXT	Y if the object is temporary; N if the object is not temporary.

USER_PART_KEY_COLUMNS

The USER_PART_KEY_COLUMNS view provides information about the key columns of the partitioned tables that reside in the database.

Name	Type	Description
schema_name	TEXT	The name of the schema in which the table resides.
name	TEXT	The name of the table in which the column resides.
object_type	CHARACTER(5)	For compatibility only; always TABLE.
column_name	TEXT	The name of the column on which the key is defined.
column_position	INTEGER	1 for the first column; 2 for the second column, etc.

USER_PART_TABLES

The USER_PART_TABLES view provides information about all of the partitioned tables in the database that are owned by the current user.

Name	Type	Description
schema_name	TEXT	The name of the schema in which the table resides.
table_name	TEXT	The name of the table.
partitioning_type	TEXT	The partitioning type used to define table partitions.
subpartitioning_type	TEXT	The subpartitioning type used to define table subpartitions.
partition_count	BIGINT	The number of partitions in the table.
def_subpartition_count	INTEGER	The number of subpartitions in the table.
partitioning_key_count	INTEGER	The number of partitioning keys specified.
subpartitioning_key_count	INTEGER	The number of subpartitioning keys specified.
status	CHARACTER VARYING(8)	Provided for compatibility only. Always VALID.
def_tablespace_name	CHARACTER VARYING(30)	Provided for compatibility only. Always NULL.
def_pct_free	NUMERIC	Provided for compatibility only. Always NULL.
def_pct_used	NUMERIC	Provided for compatibility only. Always NULL.
def_ini_trans	NUMERIC	Provided for compatibility only. Always NULL.
def_max_trans	NUMERIC	Provided for compatibility only. Always NULL.
def_initial_extent	CHARACTER VARYING(40)	Provided for compatibility only. Always NULL.
def_min_extents	CHARACTER VARYING(40)	Provided for compatibility only. Always NULL.
def_max_extents	CHARACTER VARYING(40)	Provided for compatibility only. Always NULL.
def_pct_increase	CHARACTER VARYING(40)	Provided for compatibility only. Always NULL.
def_freelists	NUMERIC	Provided for compatibility only. Always NULL.
def_freelist_groups	NUMERIC	Provided for compatibility only. Always NULL.
def_logging	CHARACTER VARYING(7)	Provided for compatibility only. Always YES.
def_compression	CHARACTER VARYING(8)	Provided for compatibility only. Always NONE
def_buffer_pool	CHARACTER VARYING(7)	Provided for compatibility only. Always DEFAULT
ref_ptn_constraint_name	CHARACTER VARYING(30)	Provided for compatibility only. Always NULL
interval	CHARACTER VARYING(1000)	Provided for compatibility only. Always NULL

USER_POLICIES

The USER_POLICIES view provides information on policies where the schema containing the object on which the policy applies has the same name as the current session user. This view is accessible only to superusers.

Name	Type	Description
schema_name	TEXT	The name of the schema in which the object resides.
object_name	TEXT	Name of the object on which the policy applies.
policy_group	TEXT	Name of the policy group. Included for compatibility only; always set to an empty string.

policy_name	TEXT	Name of the policy.
pf_owner	TEXT	Name of the schema containing the policy function, or the schema containing the package that contains the policy function.
package	TEXT	Name of the package containing the policy function (if the function belongs to a package).
function	TEXT	Name of the policy function.
sel	TEXT	Whether or not the policy applies to SELECT commands. Possible values are YES or NO.
ins	TEXT	Whether or not the policy applies to INSERT commands. Possible values are YES or NO.
upd	TEXT	Whether or not the policy applies to UPDATE commands. Possible values are YES or NO.
del	TEXT	Whether or not the policy applies to DELETE commands. Possible values are YES or NO.
idx	TEXT	Whether or not the policy applies to index maintenance. Possible values are YES or NO.
chk_option	TEXT	Whether or not the check option is in force for INSERT and UPDATE commands. Possible values are YES or NO.
enable	TEXT	Whether or not the policy is enabled on the object. Possible values are YES or NO.
static_policy	TEXT	Whether or not the policy is static. Included for compatibility only; always set to NO.
policy_type	TEXT	Policy type. Included for compatibility only; always set to UNKNOWN.
long_predicate	TEXT	Included for compatibility only; always set to YES.

USER_QUEUES

The USER_QUEUES view provides information about any queue on which the current user has usage privileges.

Name	Type	Description
name	TEXT	The name of the queue.
queue_table	TEXT	The name of the queue table in which the queue resides.
qid	OID	The system-assigned object ID of the queue.
queue_type	CHARACTER VARYING	The queue type; may be EXCEPTION_QUEUE, NON_PERSISTENT_QUEUE, or NORMAL_QUEUE.
max_retries	NUMERIC	The maximum number of dequeue attempts.
retrydelay	NUMERIC	The maximum time allowed between retries.
enqueue_enabled	CHARACTER VARYING	YES if the queue allows enqueuing; NO if the queue does not.
dequeue_enabled	CHARACTER VARYING	YES if the queue allows dequeuing; NO if the queue does not.
retention	CHARACTER VARYING	The number of seconds that a processed message is retained in the queue.

user_comment	CHARACTER VARYING	A user-specified comment.
network_name	CHARACTER VARYING	The name of the network on which the queue resides.
sharded	CHARACTER VARYING	YES if the queue resides on a sharded network; NO if the queue does not.

USER_QUEUE_TABLES

The USER_QUEUE_TABLES view provides information about all of the queue tables accessible by the current user.

Name	Type	Description
queue_table	TEXT	The user-specified name of the queue table.
type	CHARACTER VARYING	The type of data stored in the queue table.
object_type	TEXT	The user-defined payload type.
sort_order	CHARACTER VARYING	The order in which the queue table is sorted.
recipients	CHARACTER VARYING	Always SINGLE.
message_grouping	CHARACTER VARYING	Always NONE.
compatible	CHARACTER VARYING	The release number of the Advanced Server release with which this queue table is compatible.
primary_instance	NUMERIC	Always 0.
secondary_instance	NUMERIC	Always 0.
owner_instance	NUMERIC	The instance number of the instance that owns the queue table.

user_comment	CHARACTER VARYING	The user comment provided when the table was created.
secure	CHARACTER VARYING	YES indicates that the queue table is secure; NO indicates that it is not.

USER_ROLE_PRIVS

The USER_ROLE_PRIVS view provides information about the privileges that have been granted to the current user. A row is created for each role to which a user has been granted.

Name	Type	Description
username	TEXT	The name of the user to which the role was granted.
granted_role	TEXT	Name of the role granted to the grantee.
admin_option	TEXT	YES if the role was granted with the admin option, NO otherwise.
default_role	TEXT	YES if the role is enabled when the grantee creates a session.
os_granted	CHARACTER VARYING(3)	Included for compatibility only; always NO.

USER_SEQUENCES

The USER_SEQUENCES view provides information about all user-defined sequences that belong to the current user.

Name	Type	Description
schema_name	TEXT	The name of the schema in which the sequence resides.
sequence_name	TEXT	Name of the sequence.
min_value	NUMERIC	The lowest value that the server will assign to the sequence.
max_value	NUMERIC	The highest value that the server will assign to the sequence.
increment_by	NUMERIC	The value added to the current sequence number to create the next sequent number.
cycle_flag	CHARACTER VARYING	Specifies if the sequence should wrap when it reaches min_value or max_value.
order_flag	CHARACTER VARYING	Included for compatibility only; always Y.
cache_size	NUMERIC	The number of pre-allocated sequence numbers in memory.
last_number	NUMERIC	The value of the last sequence number saved to disk.

USER_SOURCE

The USER_SOURCE view provides information about all programs owned by the current user.

Name	Type	Description
schema_name	TEXT	Name of the schema in which the program belongs.

name	TEXT	Name of the program.
type	TEXT	Type of program – possible values are: FUNCTION, PACKAGE, PACKAGE BODY, PROCEDURE, and TRIGGER.
line	INTEGER	Source code line number relative to a given program.
text	TEXT	Line of source code text.

USER_SUBPART_KEY_COLUMNS

The USER_SUBPART_KEY_COLUMNS view provides information about the key columns of those partitioned tables which are subpartitioned that belong to the current user.

Name	Type	Description
schema_name	TEXT	The name of the schema in which the table resides.
name	TEXT	The name of the table in which the column resides.
object_type	CHARACTER(5)	For compatibility only; always TABLE.
column_name	TEXT	The name of the column on which the key is defined.
column_position	INTEGER	1 for the first column; 2 for the second column, etc.

USER_SYNONYMS

The USER_SYNONYMS view provides information about all synonyms owned by the current user.

Name	Type	Description
schema_name	TEXT	The name of the schema in which the synonym resides.
synonym_name	TEXT	Name of the synonym.
table_owner	TEXT	User name of the table’s owner on which the synonym is defined.
table_schema_name	TEXT	The name of the schema in which the table resides.
table_name	TEXT	Name of the table on which the synonym is defined.
db_link	TEXT	Name of any associated database link.

USER_TAB_COLUMNS

The USER_TAB_COLUMNS view displays information about all columns in tables and views owned by the current user.

Name	Type	Description
schema_name	CHARACTER VARYING	Name of the schema in which the table or view resides.
table_name	CHARACTER VARYING	Name of the table or view in which the column resides.
column_name	CHARACTER VARYING	Name of the column.
data_type	CHARACTER VARYING	Data type of the column.
data_length	NUMERIC	Length of text columns.
data_precision	NUMERIC	Precision (number of digits) for NUMBER columns.

data_scale	NUMERIC	Scale of NUMBER columns.
nullable	CHARACTER(1)	Whether or not the column is nullable – possible values are: Y Y – column is nullable; N – column does not allow null.
column_id	NUMERIC	Relative position of the column within the table.
data_default	CHARACTER VARYING	Default value assigned to the column.

USER_TAB_PARTITIONS

The USER_TAB_PARTITIONS view provides information about all of the partitions that are owned by the current user.

Name	Type	Description
schema_name	TEXT	The name of the schema in which the table resides.
table_name	TEXT	The name of the table.
composite	TEXT	YES if the table is subpartitioned; NO if the table is not subpartitioned.
partition_name	TEXT	The name of the partition.
subpartition_count	BIGINT	The number of subpartitions in the partition.
high_value	TEXT	The high partitioning value specified in the CREATE TABLE statement.
high_value_length	INTEGER	The length of the partitioning value.
partition_position	INTEGER	1 for the first partition; 2 for the second partition, etc.
tablespace_name	TEXT	The name of the tablespace in which the partition resides.
pct_free	NUMERIC	Included for compatibility only; always 0
pct_used	NUMERIC	Included for compatibility only; always 0
ini_trans	NUMERIC	Included for compatibility only; always 0
max_trans	NUMERIC	Included for compatibility only; always 0
initial_extent	NUMERIC	Included for compatibility only; always NULL
next_extent	NUMERIC	Included for compatibility only; always NULL
min_extent	NUMERIC	Included for compatibility only; always 0
max_extent	NUMERIC	Included for compatibility only; always 0
pct_increase	NUMERIC	Included for compatibility only; always 0
freelists	NUMERIC	Included for compatibility only; always NULL
freelist_groups	NUMERIC	Included for compatibility only; always NULL
logging	CHARACTER VARYING(7)	Included for compatibility only; always YES
compression	CHARACTER VARYING(8)	Included for compatibility only; always NONE
num_rows	NUMERIC	Same as pg_class.reltuples.
blocks	INTEGER	Same as pg_class.relpages.
empty_blocks	NUMERIC	Included for compatibility only; always NULL
avg_space	NUMERIC	Included for compatibility only; always NULL
chain_cnt	NUMERIC	Included for compatibility only; always NULL
avg_row_len	NUMERIC	Included for compatibility only; always NULL
sample_size	NUMERIC	Included for compatibility only; always NULL
last_analyzed	TIMESTAMP WITHOUT TIME ZONE	Included for compatibility only; always NULL
buffer_pool	CHARACTER VARYING(7)	Included for compatibility only; always NULL
global_stats	CHARACTER VARYING(3)	Included for compatibility only; always YES

Name	Type	Description
user_stats	CHARACTER VARYING(3)	Included for compatibility only; always NO
backing_table	REGCLASS	Name of the partition backing table.

USER_TAB_SUBPARTITIONS

The USER_TAB_SUBPARTITIONS view provides information about all of the subpartitions owned by the current user.

Name	Type	Description
schema_name	TEXT	The name of the schema in which the table resides.
table_name	TEXT	The name of the table.
partition_name	TEXT	The name of the partition.
subpartition_name	TEXT	The name of the subpartition.
high_value	TEXT	The high subpartitioning value specified in the CREATE TABLE statement.
high_value_length	INTEGER	The length of the subpartitioning value.
subpartition_position	INTEGER	1 for the first subpartition; 2 for the second subpartition, etc.
tablespace_name	TEXT	The name of the tablespace in which the subpartition resides.
pct_free	NUMERIC	Included for compatibility only; always 0
pct_used	NUMERIC	Included for compatibility only; always 0
ini_trans	NUMERIC	Included for compatibility only; always 0
max_trans	NUMERIC	Included for compatibility only; always 0
initial_extent	NUMERIC	Included for compatibility only; always NULL
next_extent	NUMERIC	Included for compatibility only; always NULL
min_extent	NUMERIC	Included for compatibility only; always 0
max_extent	NUMERIC	Included for compatibility only; always 0
pct_increase	NUMERIC	Included for compatibility only; always 0
freelists	NUMERIC	Included for compatibility only; always NULL
freelist_groups	NUMERIC	Included for compatibility only; always NULL
logging	CHARACTER VARYING(7)	Included for compatibility only; always YES
compression	CHARACTER VARYING(8)	Included for compatibility only; always NONE
num_rows	NUMERIC	Same as pg_class.reltuples.
blocks	INTEGER	Same as pg_class.relpages.
empty_blocks	NUMERIC	Included for compatibility only; always NULL
avg_space	NUMERIC	Included for compatibility only; always NULL
chain_cnt	NUMERIC	Included for compatibility only; always NULL
avg_row_len	NUMERIC	Included for compatibility only; always NULL
sample_size	NUMERIC	Included for compatibility only; always NULL
last_analyzed	TIMESTAMP WITHOUT TIME ZONE	Included for compatibility only; always NULL
buffer_pool	CHARACTER VARYING(7)	Included for compatibility only; always NULL
global_stats	CHARACTER VARYING(3)	Included for compatibility only; always YES
user_stats	CHARACTER VARYING(3)	Included for compatibility only; always NO
backing_table	REGCLASS	Name of the partition backing table.

USER_TABLES

The USER_TABLES view displays information about all tables owned by the current user.

Name	Type	Description
schema_name	TEXT	Name of the schema in which the table belongs.
table_name	TEXT	Name of the table.
tablespace_name	TEXT	Name of the tablespace in which the table resides if other than the default tablespace.
status	CHARACTER VARYING(5)	Included for compatibility only; always set to VALID..
temporary	CHARACTER(1)	Y if the table is temporary; N if the table is not temporary.

USER_TRIGGERS

The USER_TRIGGERS view displays information about all triggers on tables owned by the current user.

Name	Type	Description
schema_name	TEXT	The name of the schema in which the trigger resides.
trigger_name	TEXT	The name of the trigger.
trigger_type	TEXT	The type of the trigger. Possible values are: BEFORE ROW BEFORE STATEMENT AFTER ROW AFTER STATEMENT
triggering_event	TEXT	The event that fires the trigger.
table_owner	TEXT	The user name of the owner of the table on which the trigger is defined.
base_object_type	TEXT	Included for compatibility only. Value will always be TABLE.
table_name	TEXT	The name of the table on which the trigger is defined.
referencing_names	TEXT	Included for compatibility only. Value will always be REFERENCING NEW AS NEW OLD AS OLD.
status	TEXT	Status indicates if the trigger is enabled (VALID) or disabled (NOTVALID).
description	TEXT	Included for compatibility only.
trigger_body	TEXT	The body of the trigger.
action_statement	TEXT	The SQL command that executes when the trigger fires.

USER_TYPES

The USER_TYPES view provides information about all object types owned by the current user.

Name	Type	Description
schema_name	TEXT	The name of the schema in which the type is defined.
type_name	TEXT	The name of the type.

type_oid	OID	The object identifier (OID) of the type.
typecode	TEXT	The typecode of the type. Possible values are: OBJECT COLLECTION OTHER
attributes	INTEGER	The number of attributes in the type.

USER_USERS

The USER_USERS view provides information about the current user.

Name	Type	Description
username	TEXT	User name of the user.
user_id	OID	ID number of the user.
account_status	CHARACTER VARYING(32)	The current status of the account. Possible values are: OPEN EXPIRED EXPIRED(GRACE) EXPIRED & LOCKED EXPIRED & LOCKED(TIMED) EXPIRED(GRACE) & LOCKED EXPIRED(GRACE) & LOCKED(TIMED) LOCKED LOCKED(TIMED) Use the edb_get_role_status(*role_id*) function to get the current status of the account.
lock_date	TIMESTAMP WITHOUT TIME ZONE	If the account status is LOCKED, lock_date displays the date and time the account was locked.
expiry_date	TIMESTAMP WITHOUT TIME ZONE	The expiration date of the account.
default_tablespace	TEXT	The default tablespace associated with the account.
temporary_tablespace	CHARACTER VARYING(30)	Included for compatibility only. The value will always be '' (an empty string).
created	TIMESTAMP WITHOUT TIME ZONE	Included for compatibility only. The value will always be NULL.
initial_rsrc_consumer_group	CHARACTER VARYING(30)	Included for compatibility only. The value will always be NULL.
external_name	CHARACTER VARYING(4000)	Included for compatibility only; always set to NULL.

USER_VIEW_COLUMNS

The USER_VIEW_COLUMNS view provides information about all columns in views owned by the current user.

Name	Type	Description
schema_name	CHARACTER VARYING	Name of the schema in which the view belongs.
view_name	CHARACTER VARYING	Name of the view.
column_name	CHARACTER VARYING	Name of the column.
data_type	CHARACTER VARYING	Data type of the column.
data_length	NUMERIC	Length of text columns.
data_precision	NUMERIC	Precision (number of digits) for NUMBER columns.
data_scale	NUMERIC	Scale of NUMBER columns.
nullable	CHARACTER(1)	Whether or not the column is nullable – possible values are: Y – column is nullable; N – column does not allow null.
column_id	NUMERIC	Relative position of the column within the view.
data_default	CHARACTER VARYING	Default value assigned to the column.

USER_VIEWS

The USER_VIEWS view provides information about all views owned by the current user.

Name	Type	Description
schema_name	TEXT	Name of the schema in which the view resides.
view_name	TEXT	Name of the view.
text	TEXT	The SELECT statement that defines the view.

V$VERSION

The V$VERSION view provides information about product compatibility.

Name	Type	Description
banner	TEXT	Displays product compatibility information.

PRODUCT_COMPONENT_VERSION

The PRODUCT_COMPONENT_VERSION view provides version information about product version compatibility.

Name	Type	Description
product	CHARACTER VARYING(74)	The name of the product.
version	CHARACTER VARYING(74	The version number of the product.
status	CHARACTER VARYING(74)	Included for compatibility; always Available.

3.2 System Catalog Tables

The following system catalog tables contain definitions of database objects. The layout of the system tables is subject to change; if you are writing an application that depends on information stored in the system tables, it would be prudent to use an existing catalog view, or create a catalog view to isolate the application from changes to the system table.

dual

dual is a single-row, single-column table that is provided for compatibility with Oracle databases only.

Column	Type	Modifiers	Description
dummy	VARCHAR2(1)		Provided for compatibility only.

edb_dir

The edb_dir table contains one row for each alias that points to a directory created with the create directory command. A directory is an alias for a pathname that allows a user limited access to the host file system.

You can use a directory to fence a user into a specific directory tree within the file system. For example, the UTL_FILE package offers functions that permit a user to read and write files and directories in the host file system, but only allows access to paths that the database administrator has granted access to via a create directory command.

Column	Type	Modifiers	Description
dirname	"name"	not null	The name of the alias.
dirowner	oid	not null	The OID of the user that owns the alias.
dirpath	text		The directory name to which access is granted.
diracl	aclitem[]		The access control list that determines which users may access the alias.

edb_password_history

The edb_password_history table contains one row for each password change. The table is shared across all databases within a cluster.

Column	Type	References	Description
passhistroleid	oid	pg_authid.oid	The ID of a role.
passhistpassword	text		Role password in md5 encrypted form.
passhistpasswordsetat	timestamptz		The time the password was set.

edb_policy

The edb_policy table contains one row for each policy.

Column	Type	Modifiers	Description
policyname	name	not null	The policy name.

Column	Type	Modifiers	Description
policygroup	oid	not null	Currently unused.
policyobject	oid	not null	The OID of the table secured by this policy (the object_schema plus the object_name).
policykind	char	not null	The kind of object secured by this policy: 'r' for a table 'v' for a view = for a synonym Currently always 'r'.
policyproc	oid	not null	The OID of the policy function (function_schema plus policy_function).
policyinsert	boolean	not null	True if the policy is enforced by INSERT statements.
policyselect	boolean	not null	True if the policy is enforced by SELECT statements.
policydelete	boolean	not null	True if the policy is enforced by DELETE statements.
policyupdate	boolean	not null	True if the policy is enforced by UPDATE statements.
policyindex	boolean	not null	Currently unused.
policyenabled	boolean	not null	True if the policy is enabled.
policyupdatecheck	boolean	not null	True if rows updated by an UPDATE statement must satisfy the policy.
policystatic	boolean	not null	Currently unused.
policytype	integer	not null	Currently unused.
policyopts	integer	not null	Currently unused.
policyseccols	int2vector	not null	The column numbers for columns listed in sec_relevant_cols.

edb_profile

The edb_profile table stores information about the available profiles. edb_profiles is shared across all databases within a cluster.

Column	Type	References	Description
oid	oid		Row identifier (hidden attribute; must be explicitly selected).
prfname	name		The name of the profile.
prffailedloginattempts	integer		The number of failed login attempts allowed by the profile. -1 indicates that the value from the default profile should be used. -2 indicates no limit on failed login attempts.
prfpasswordlocktime	integer		The password lock time associated with the profile (in seconds). -1 indicates that the value from the default profile should be used. -2 indicates that the account should be locked permanently.
prfpasswordlifetime	integer		The password life time associated with the profile (in seconds). -1 indicates that the value from the default profile should be used. -2 indicates that the password never expires.

Column	Type	References	Description
prfpasswordgracetime	integer		The password grace time associated with the profile (in seconds). -1 indicates that the value from the default profile should be used. -2 indicates that the password never expires.
prfpasswordreusetime	integer		The number of seconds that a user must wait before reusing a password. -1 indicates that the value from the default profile should be used. -2 indicates that the old passwords can never be reused.
prfpasswordreusemax	integer		The number of password changes that have to occur before a password can be reused. -1 indicates that the value from the default profile should be used. -2 indicates that the old passwords can never be reused.
prfpasswordverifyfuncdb	oid	pg_database.oid	The OID of the database in which the password verify function exists.
prfpasswordverifyfunc	oid	pg_proc.oid	The OID of the password verify function associated with the profile.

edb_variable

The edb_variable table contains one row for each package level variable (each variable declared within a package).

Column	Type	Modifiers	Description
varname	"name"	not null	The name of the variable.
varpackage	oid	not null	The OID of the pg_namespace row that stores the package.
vartype	oid	not null	The OID of the pg_type row that defines the type of the variable.
varaccess	"char"	not null	+ if the variable is visible outside of the package. - if the variable is only visible within the package. Note: Public variables are declared within the package header; private variables are declared within the package body.
varsrc	text		Contains the source of the variable declaration, including any default value expressions for the variable.
varseq	smallint	not null	The order in which the variable was declared in the package.

pg_synonym

The pg_synonym table contains one row for each synonym created with the CREATE SYNONYM command or CREATE PUBLIC SYNONYM command.

Column	Type	Modifiers	Description
synname	"name"	not null	The name of the synonym.
synnamespace	oid	not null	Replaces synowner. Contains the OID of the pg_namespace row where the synonym is stored
synowner	oid	not null	The OID of the user that owns the synonym.
synobjschema	"name"	not null	The schema in which the referenced object is defined.
synobjname	"name"	not null	The name of the referenced object.

Column	Type	Modifiers	Description
synlink	text		The (optional) name of the database link in which the referenced object is defined.

product_component_version

The product_component_version table contains information about feature compatibility; an application can query this table at installation or run time to verify that features used by the application are available with this deployment.

Column	Type	Description
product	character varying (74)	The name of the product.
version	character varying (74)	The version number of the product.
status	character varying (74)	The status of the release.

3.3 Acknowledgements

The PostgreSQL 8.3, 8.4, 9.0, 9.1, 9.2, 9.3, 9.4, 9.5, 9.6, and 10 Documentation provided the baseline for the portions of this guide that are common to PostgreSQL, and is hereby acknowledged:

Portions of this EnterpriseDB Software and Documentation may utilize the following copyrighted material, the use of which is hereby acknowledged.

PostgreSQL Documentation, Database Management System

PostgreSQL is Copyright © 1996-2017 by the PostgreSQL Global Development Group and is distributed under the terms of the license of the University of California below.

Postgres95 is Copyright © 1994-5 by the Regents of the University of California.

Permission to use, copy, modify, and distribute this software and its documentation for any purpose, without fee, and without a written agreement is hereby granted, provided that the above copyright notice and this paragraph and the following two paragraphs appear in all copies.

IN NO EVENT SHALL THE UNIVERSITY OF CALIFORNIA BE LIABLE TO ANY PARTY FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES, INCLUDING LOST PROFITS, ARISING OUT OF THE USE OF THIS SOFTWARE AND ITS DOCUMENTATION, EVEN IF THE UNIVERSITY OF CALIFORNIA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

THE UNIVERSITY OF CALIFORNIA SPECIFICALLY DISCLAIMS ANY WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE SOFTWARE PROVIDED HEREUNDER IS ON AN "AS-IS" BASIS, AND THE UNIVERSITY OF CALIFORNIA HAS NO OBLIGATIONS TO PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS.

3.4 Introduction

Database Compatibility for Oracle means that an application runs in an Oracle environment as well as in the EDB Postgres Advanced Server (Advanced Server) environment with minimal or no changes to the application code.

This guide provides reference material about the compatibility features offered by Advanced Server:

- SQL Language syntax support
- Compatible Data Types
- SQL Commands
- Catalog Views
- System Catalog Tables

Developing an application that is compatible with Oracle databases in the Advanced Server requires special attention to which features are used in the construction of the application. For example, developing a compatible application means selecting:

- Data types to define the application's database tables that are compatible with Oracle databases
- SQL statements that are compatible with Oracle SQL
- System and built-in functions for use in SQL statements and procedural logic that are compatible with Oracle databases
- Stored Procedure Language (SPL) to create database server-side application logic for stored procedures, functions, triggers, and packages
- System catalog views that are compatible with Oracle's data dictionary

For detailed information about Advanced Server's compatibility features and extended functionality, please see the complete library of Advanced Server documentation, available at:

http://www.enterprisedb.com/products-services-training/products/documentation

What's New

The following database compatibility for Oracle features have been added to Advanced Server 9.6 to create Advanced Server 10:

- Advanced Server now supports usage of a composite type (created by the CREATE TYPE AS command) referenced by a field as its data type within a user-defined record type (declared with the TYPE IS RECORD statement). This record type containing a composite type can only be declared in a package specification or a package body. A composite type is not compatible with Oracle databases. However, composite types can generally be used within all SPL programs (functions, procedures, triggers, packages, etc.) as long as the composite type is not part of a record type (with the exception of packages). For more information on composite types, see Section 2.3.36.

Typographical Conventions Used in this Guide

Certain typographical conventions are used in this manual to clarify the meaning and usage of various commands, statements, programs, examples, etc. This section provides a summary of these conventions.

In the following descriptions a *term* refers to any word or group of words which may be language keywords, user-supplied values, literals, etc. A term's exact meaning depends upon the context in which it is used.

- *Italic font* introduces a new term, typically, in the sentence that defines it for the first time.
- Fixed-width (mono-spaced) font is used for terms that must be given literally such as SQL commands, specific table and column names used in the examples, programming language keywords, etc. For example, SELECT * FROM emp;
- *Italic fixed-width font* is used for terms for which the user must substitute values in actual usage. For example, DELETE FROM *table_name*;

- A vertical pipe | denotes a choice between the terms on either side of the pipe. A vertical pipe is used to separate two or more alternative terms within square brackets (optional choices) or braces (one mandatory choice).

- Square brackets [ ] denote that one or none of the enclosed term(s) may be substituted. For example, [a | b], means choose one of “a” or “b” or neither of the two.

- Braces {} denote that exactly one of the enclosed alternatives must be specified. For example, { a | b }, means exactly one of “a” or “b” must be specified.

- Ellipses ... denote that the proceeding term may be repeated. For example, [a | b] ... means that you may have the sequence, “b a a b a”.

4 Overview

EnterpriseDB has enhanced ECPG (the PostgreSQL pre-compiler) to create ECPGPlus. ECPGPlus is a Pro*C-compatible version of the PostgreSQL C pre-compiler. ECPGPlus translates a program that combines C code and embedded SQL statements into an equivalent C program. As it performs the translation, ECPGPlus verifies that the syntax of each SQL construct is correct.

The following diagram charts the path of a program containing embedded SQL statements as it is compiled into an executable:

To produce an executable from a C program that contains embedded SQL statements, pass the program (my_program.pgc in the diagram above) to the ECPGPlus pre-compiler. ECPGPlus translates each SQL statement in my_program.pgc into C code that calls the ecpglib API, and produces a C program (my_program.c). Then, pass the C program to a C compiler; the C compiler generates an object file (my_program.o). Finally, pass the object file (my_program.o), as well as the ecpglib library file, and any other required libraries to the linker, which in turn produces the executable (my_program).

While the ECPGPlus preprocessor validates the *syntax* of each SQL statement, it cannot validate the *semantics*. For example, the preprocessor will confirm that an INSERT statement is syntactically correct, but it cannot confirm that the table mentioned in the INSERT statement actually exists.

Behind the Scenes

A client application contains a mix of C code and SQL code comprised of the following elements:

- C preprocessor directives
- C declarations (variables, types, functions, ...)
- C definitions (variables, types, functions, ...)
- SQL preprocessor directives
- SQL statements

For example:

```
1 #include \<stdio.h>
2 EXEC SQL INCLUDE sqlca;
3
4 extern void printInt(char *label, int val);
5 extern void printStr(char *label, char *val);
6 extern void printFloat(char *label, float val);
7
8 void displayCustomer(int custNumber)
9 {
10 EXEC SQL BEGIN DECLARE SECTION;
11 VARCHAR custName[50];
12 float custBalance;
13 int custID = custNumber;
14 EXEC SQL END DECLARE SECTION;
15
16 EXEC SQL SELECT name, balance
17 INTO :custName, :custBalance
18 FROM customer
19 WHERE id = :custID;
20
21 printInt("ID", custID);
22 printStr("Name", custName);
23 printFloat("Balance", custBalance);
24 }
```

In the above code fragment:

- Line 1 specifies a directive to the C preprocessor.

C preprocessor directives may be interpreted or ignored; the option is controlled by a command line option (-C PROC) entered when you invoke ECPGPlus. In either case, ECPGPlus copies each C preprocessor directive to the output file (4) without change; any C preprocessor directive found in the source file will appear in the output file.

- Line 2 specifies a directive to the SQL preprocessor.

SQL preprocessor directives are interpreted by the ECPGPlus preprocessor, and are not copied to the output file.

- Lines 4 through 6 contain C declarations.

C declarations are copied to the output file without change, except that each VARCHAR declaration is translated into an equivalent struct declaration.

- Lines 10 through 14 contain an embedded-SQL declaration section.

C variables that you refer to within SQL code are known as *host variables*. If you invoke the ECPGPlus preprocessor in Pro*C mode (-C PROC), you may refer to *any* C variable within a SQL statement; otherwise you must declare each host variable within a BEGIN/END DECLARATION SECTION pair.

- Lines 16 through 19 contain a SQL statement.

SQL statements are translated into calls to the ECPGPlus run-time library.

- Lines 21 through 23 contain C code.

C code is copied to the output file without change.

Any SQL statement must be prefixed with EXEC SQL and extends to the next (unquoted) semicolon. For example:

```
printf(“Updating employee salaries\n”);

EXEC SQL UPDATE emp SET sal = sal * 1.25;

EXEC SQL COMMIT;

printf(“Employee salaries updated\n”);
```

When the preprocessor encounters the code fragment shown above, it passes the C code (the first line and the last line) to the output file without translation and converts each EXEC SQL statement into a call to an ecpglib function. The result would appear similar to the following:

```
printf("Updating employee salaries\n");

{

ECPGdo( __LINE__, 0, 1, NULL, 0, ECPGst_normal,

"update emp set sal = sal * 1.25",

ECPGt_EOIT, ECPGt_EORT);

}

{

ECPGtrans(__LINE__, NULL, "commit");

}

printf(“Employee salaries updated\n”);
```

Installation and Configuration

ECPGPlus is installed by the Advanced Server installation wizard as part of the Database Server component. By default, on a Linux installation, the executable is located in:

```
/opt/edb/10AS/bin
```

On Windows, the executable is located in:

```
C:\Program Files\edb\10AS\bin
```

When invoking the ECPGPlus compiler, the executable must be in your search path (%PATH% on Windows, $PATH on Linux). Use the following commands to set the search path (substituting the name of the directory that holds the ECPGPlus executable for *ecpgPlus*):

On Windows, use the command:

```
SET PATH=ecpgPlus;%PATH%
```

On Linux, use the command:

```
PATH=ecpgPlus:$PATH
```

Constructing a Makefile

A makefile contains a set of instructions that tell the make utility how to transform a program written in C (that contains embedded SQL) into a C program. To try the examples in this guide, you will need:

- a C compiler (and linker)
- the make utility
- ECPGPlus preprocessor and library
- a makefile that contains instructions for ECPGPlus

The following code is an example of a makefile for the samples included in this guide. To use the sample code, save it in a file named makefile in the directory that contains the source code file.

```
INCLUDES = -I$(shell pg_config --includedir) LIBPATH = -L $(shell pg_config --libdir) CFLAGS += $(INCLUDES) -g LDFLAGS += -g LDLIBS += $(LIBPATH) -lecpg -lpq

.SUFFIXES: .pgc,.pc

.pgc.c: ecpg -c $(INCLUDES) $?

.pc.c: ecpg -C PROC -c $(INCLUDES) $?
```

The first two lines use the pg_config program to locate the necessary header files and library directories:

```
INCLUDES = -I$(shell pg_config --includedir) LIBPATH = -L $(shell pg_config --libdir)
```

The pg_config program is shipped with Advanced Server.

make knows that it should use the CFLAGS variable when running the C compiler and LDFLAGS and LDLIBS when invoking the linker. ECPG programs must be linked against the ECPG run-time library (-lecpg) and the libpq library (-lpq)

CFLAGS += $(INCLUDES) -g LDFLAGS += -g LDLIBS += $(LIBPATH) -lecpg -lpq

The sample makefile instructs make how to translate a .pgc or a .pc file into a C program. Two lines in the makefile specify the mode in which the source file will be compiled. The first compile option is:

```
.pgc.c: ecpg -c $(INCLUDES) $?
```

The first option tells make how to transform a file that ends in .pgc (presumably, an ECPG source file) into a file that ends in .c (a C program), using community ECPG (without the ECPGPlus enhancements). It invokes the ECPG pre-compiler with the -c flag (instructing the compiler to convert SQL code into C), using the value of the INCLUDES variable and the name of the .pgc file.

```
.pc.c: ecpg -C PROC -c $(INCLUDES) $?
```

The second option tells make how to transform a file that ends in .pg (an ECPG source file) into a file that ends in .c (a C program), using the ECPGPlus extensions. It invokes the ECPG pre-compiler with the -c flag (instructing the compiler to convert SQL code into C), as well as the -C PROC flag (instructing the compiler to use ECPGPlus in Pro*C-compatibility mode), using the value of the INCLUDES variable and the name of the .pgc file.

When you run make, pass the name of the ECPG source code file you wish to compile. For example, to compile an ECPG source code file named customer_list.pgc, use the command:

make customer_list

The make utility consults the makefile (located in the current directory), discovers that the makefile contains a rule that will compile customer_list.pgc into a C program (customer_list.c), and then uses the rules built into make to compile customer_list.c into an executable program.

ECPGPlus Command Line Options

In the sample makefile shown above, make includes the -C option when invoking ECPGPlus to specify that ECPGPlus should be invoked in Pro*C compatible mode.

If you include the -C PROC keywords on the command line, in addition to the ECPG syntax, you may use Pro*C command line syntax; for example:

```
$ ecpg -C PROC INCLUDE=/opt/edb/as10/headers acct_update.c
```

To display a complete list of the other ECPGPlus options available, navigate to the ECPGPlus installation directory, and enter:

```
./ecpg --help
```

The command line options are:

Option	Description
-c	Automatically generate C code from embedded SQL code.
-C *mode*	Use the -C option to specify a compatibility mode: INFORMIX INFORMIX_SE PROC
-D *symbol*	Define a preprocessor *symbol.*
-h	Parse a header file, this option includes option '-c'.
-i	Parse system, include files as well.
-I directory	Search *directory* for include files.
-o *outfile*	Write the result to *outfile.*

Option	Description
-r *option*	Specify run-time behavior; *option* can be: no_indicator - Do not use indicators, but instead use special values to represent NULL values. prepare - Prepare all statements before using them. questionmarks - Allow use of a question mark as a placeholder. usebulk - Enable bulk processing for INSERT, UPDATE and DELETE statements that operate on host variable arrays.
--regression	Run in regression testing mode.
-t	Turn on autocommit of transactions.
-l	Disable #line directives.
--help	Display the help options.
--version	Output version information.

Please Note: If you do not specify an output file name when invoking ECPGPlus, the output file name is created by stripping off the .pgc file name extension, and appending .c to the file name.

4.1 Using Embedded SQL

Each of the following sections leads with a code sample, followed by an explanation of each section within the code sample.

Example - A Simple Query

The first code sample demonstrates how to execute a SELECT statement (which returns a single row), storing the results in a group of host variables. After declaring host variables, it connects to the edb sample database using a hard-coded role name and the associated password, and queries the emp table. The query returns the values into the declared host variables; after checking the value of the NULL indicator variable, it prints a simple result set onscreen and closes the connection.

```
/************************************************************

* print_emp.pgc

*

*/

#include \<stdio.h>

int main(void)

{

EXEC SQL BEGIN DECLARE SECTION;
```

```
int v_empno;

char v_ename[40];

double v_sal;

double v_comm;

short v_comm_ind;

EXEC SQL END DECLARE SECTION;

EXEC SQL WHENEVER SQLERROR sqlprint;

EXEC SQL CONNECT TO edb USER 'alice' IDENTIFIED BY '1safepwd';

EXEC SQL

SELECT

empno, ename, sal, comm

INTO

:v_empno, :v_ename, :v_sal, :v_comm INDICATOR:v_comm_ind

FROM

emp

WHERE

empno = 7369;

if (v_comm_ind)

printf("empno(%d), ename(%s), sal(%.2f) comm(NULL)\n",

v_empno, v_ename, v_sal);

else

printf("empno(%d), ename(%s), sal(%.2f) comm(%.2f)\n",

v_empno, v_ename, v_sal, v_comm);

EXEC SQL DISCONNECT;

}

/************************************************************
```

The code sample begins by including the prototypes and type definitions for the C stdio library, and then declares the main function:

```
#include \<stdio.h>

int main(void)

{
```

Next, the application declares a set of host variables used to interact with the database server:

```
EXEC SQL BEGIN DECLARE SECTION;

int v_empno;

char v_ename[40];

double v_sal;

double v_comm;

short v_comm_ind;

EXEC SQL END DECLARE SECTION;
```

Please note that if you plan to pre-compile the code in PROC mode, you may omit the BEGIN DECLARE…END DECLARE section. For more information about declaring host variables, refer to Section 3.1.2, *Declaring Host Variables*.

The data type associated with each variable within the declaration section is a C data type. Data passed between the server and the client application must share a compatible data type; for more information about data types, see Section 7.2, *Supported C Data Types*.

The next statement instructs the server how to handle an error:

```
EXEC SQL WHENEVER SQLERROR sqlprint;
```

If the client application encounters an error in the SQL code, the server will print an error message to stderr (standard error), using the sqlprint() function supplied with ecpglib. The next EXEC SQL statement establishes a connection with Advanced Server:

```
EXEC SQL CONNECT TO edb USER 'alice' IDENTIFIED BY '1safepwd';
```

In our example, the client application connects to the edb database, using a role named alice with a password of 1safepwd.

The code then performs a query against the emp table:

```
EXEC SQL

SELECT

empno, ename, sal, comm

INTO

:v_empno, :v_ename, :v_sal, :v_comm INDICATOR :v_comm_ind

FROM

emp

WHERE

empno = 7369;
```

The query returns information about employee number 7369.

The SELECT statement uses an INTO clause to assign the retrieved values (from the empno, ename, sal and comm columns) into the :v_empno, :v_ename, :v_sal and :v_comm host variables (and the :v_comm_ind null indicator). The first value retrieved is assigned to the first variable listed in the INTO clause, the second value is assigned to the second variable, and so on.

The comm column contains the commission values earned by an employee, and could potentially contain a NULL

value. The statement includes the INDICATOR keyword, and a host variable to hold a null indicator.

The code checks the null indicator, and displays the appropriate on-screen results:

```
if (v_comm_ind)

printf("empno(%d), ename(%s), sal(%.2f) comm(NULL)\n",

v_empno, v_ename, v_sal);

else

printf("empno(%d), ename(%s), sal(%.2f) comm(%.2f)\n",

v_empno, v_ename, v_sal, v_comm);
```

If the null indicator is 0 (that is, false), the comm column contains a meaningful value, and the printf function displays the commission. If the null indicator contains a non-zero value, comm is NULL, and printf displays a value of NULL. Please note that a host variable (other than a null indicator) contains no meaningful value if you fetch a NULL into that host variable; you must use null indicators to identify any value which may be NULL.

The final statement in the code sample closes the connection to the server:

```
EXEC SQL DISCONNECT;

}
```

Using Indicator Variables

The previous example included an *indicator variable* that identifies any row in which the value of the comm column (when returned by the server) was NULL. An indicator variable is an extra host variable that denotes if the content of the preceding variable is NULL or truncated. The indicator variable is populated when the contents of a row are stored. An indicator variable may contain the following values:

Indicator Value	Denotes
If an indicator variable is less than 0.	The value returned by the server was NULL.
If an indicator variable is equal to 0.	The value returned by the server was not NULL, and was not truncated.
If an indicator variable is greater than 0.	The value returned by the server was truncated when stored in the host variable.

When including an indicator variable in an INTO clause, you are not required to include the optional INDICATOR keyword.

You may omit an indicator variable if you are certain that a query will never return a NULL value into the corresponding host variable. If you omit an indicator variable and a query returns a NULL value, ecpglib will raise a run-time error.

Declaring Host Variables

You can use a *host variable* in a SQL statement at any point that a value may appear within that statement. A host variable is a C variable that you can use to pass data values from the client application to the server, and return data from the server to the client application. A host variable can be:

- an array
- a typedef

- a pointer
- a struct
- any scalar C data type

The code fragments that follow demonstrate using host variables in code compiled in PROC mode, and in non-PROC mode. The SQL statement adds a row to the dept table, inserting the values returned by the variables v_deptno, v_dname and v_loc into the deptno column, the dname column and the loc column, respectively.

If you are compiling in PROC mode, you may omit the EXEC SQL BEGIN DECLARE SECTION and EXEC SQL END DECLARE SECTION directives. PROC mode permits you to use C function parameters as host variables:

void addDept(int v_deptno, char v_dname, char v_loc) { EXEC SQL INSERT INTO dept VALUES(:v_deptno, :v_dname, :v_loc); }

If you are not compiling in PROC mode, you must wrap embedded variable declarations with the EXEC SQL BEGIN DECLARE SECTION and the EXEC SQL END DECLARE SECTION directives, as shown below:

void addDept(int v_deptno, char v_dname, char v_loc) { EXEC SQL BEGIN DECLARE SECTION; int v_deptno_copy = v_deptno; char v_dname_copy[14+1] = v_dname; char v_loc_copy[13+1] = v_loc; EXEC SQL END DECLARE SECTION;

EXEC SQL INSERT INTO dept VALUES(:v_deptno, :v_dname, :v_loc); }

You can also include the INTO clause in a SELECT statement to use the host variables to retrieve information:

```
EXEC SQL SELECT deptno, dname, loc INTO :v_deptno, :v_dname, v_loc FROM dept;
```

Each column returned by the SELECT statement must have a type-compatible target variable in the INTO clause. This is a simple example that retrieves a single row; to retrieve more than one row, you must define a cursor, as demonstrated in the next example.

Example - Using a Cursor to Process a Result Set

The code sample that follows demonstrates using a cursor to process a result set. There are four basic steps involved in creating and using a cursor:

1. Use the DECLARE CURSOR statement to define a cursor.

2. Use the OPEN CURSOR statement to open the cursor.

3. Use the FETCH statement to retrieve data from a cursor.

4. Use the CLOSE CURSOR statement to close the cursor.

After declaring host variables, our example connects to the edb database using a user-supplied role name and password, and queries the emp table. The query returns the values into a cursor named employees. The code sample then opens the cursor, and loops through the result set a row at a time, printing the result set. When the sample detects the end of the result set, it closes the connection.

/***

* print_emps.pgc

*

*/

#include \<stdio.h>

int main(int argc, char *argv[])

```
{
EXEC SQL BEGIN DECLARE SECTION;
char *username = argv[1];
char *password = argv[2];
int v_empno;
char v_ename[40];
double v_sal;
double v_comm;
short v_comm_ind;
EXEC SQL END DECLARE SECTION;
EXEC SQL WHENEVER SQLERROR sqlprint;
EXEC SQL CONNECT TO edb USER :username IDENTIFIED BY :password;
EXEC SQL DECLARE employees CURSOR FOR
SELECT
empno, ename, sal, comm
FROM
emp;
EXEC SQL OPEN employees;
EXEC SQL WHENEVER NOT FOUND DO break;
for (;;)
{
EXEC SQL FETCH NEXT FROM employees
INTO
:v_empno, :v_ename, :v_sal, :v_comm INDICATOR :v_comm_ind;
if (v_comm_ind)
printf("empno(%d), ename(%s), sal(%.2f) comm(NULL)\n",
v_empno, v_ename, v_sal);
else
printf("empno(%d), ename(%s), sal(%.2f) comm(%.2f)\n",
v_empno, v_ename, v_sal, v_comm);
}
EXEC SQL CLOSE employees;
```

```
EXEC SQL DISCONNECT;

}

/************************************************************
```

The code sample begins by including the prototypes and type definitions for the C stdio library, and then declares the main function:

```
#include \<stdio.h>

int main(int argc, char *argv[])

{
```

Next, the application declares a set of host variables used to interact with the database server:

```
EXEC SQL BEGIN DECLARE SECTION;

char *username = argv[1];

char *password = argv[2];

int v_empno;

char v_ename[40];

double v_sal;

double v_comm;

short v_comm_ind;

EXEC SQL END DECLARE SECTION;
```

argv[] is an array that contains the command line arguments entered when the user runs the client application. argv[1] contains the first command line argument (in this case, a username), and argv[2] contains the second command line argument (a password); please note that we have omitted the error-checking code you would normally include a real-world application. The declaration initializes the values of username and password, setting them to the values entered when the user invoked the client application.

You may be thinking that you could refer to argv[1] and argv[2] in a SQL statement (instead of creating a separate copy of each variable); that will not work. All host variables must be declared within a BEGIN/END DECLARE SECTION (unless you are compiling in PROC mode). Since argv is a function *parameter* (not an automatic variable), it cannot be declared within a BEGIN/END DECLARE SECTION. If you are compiling in PROC mode, you can refer to *any* C variable within a SQL statement.

The next statement instructs the server to respond to an SQL error by printing the text of the error message returned by ECPGPlus or the database server:

```
EXEC SQL WHENEVER SQLERROR sqlprint;
```

Then, the client application establishes a connection with Advanced Server:

```
EXEC SQL CONNECT TO edb USER :username IDENTIFIED BY :password;
```

The CONNECT statement creates a connection to the edb database, using the values found in the :username and :password host variables to authenticate the application to the server when connecting.

The next statement declares a cursor named employees:

```
EXEC SQL DECLARE employees CURSOR FOR
```

```
SELECT

empno, ename, sal, comm

FROM

emp;
```

employees will contain the result set of a SELECT statement on the emp table. The query returns employee information from the following columns: empno, ename, sal and comm. Notice that when you declare a cursor, you do not include an INTO clause - instead, you specify the target variables (or descriptors) when you FETCH from the cursor.

Before fetching rows from the cursor, the client application must OPEN the cursor:

```
EXEC SQL OPEN employees;
```

In the subsequent FETCH section, the client application will loop through the contents of the cursor; the client application includes a WHENEVER statement that instructs the server to break (that is, terminate the loop) when it reaches the end of the cursor:

```
EXEC SQL WHENEVER NOT FOUND DO break;
```

The client application then uses a FETCH statement to retrieve each row from the cursor INTO the previously declared host variables:

```
for (;;)

{

EXEC SQL FETCH NEXT FROM employees

INTO

:v_empno, :v_ename, :v_sal, :v_comm INDICATOR :v_comm_ind;
```

The FETCH statement uses an INTO clause to assign the retrieved values into the :v_empno, :v_ename, :v_sal and :v_comm host variables (and the :v_comm_ind null indicator). The first value in the cursor is assigned to the first variable listed in the INTO clause, the second value is assigned to the second variable, and so on.

The FETCH statement also includes the INDICATOR keyword and a host variable to hold a null indicator. If the comm column for the retrieved record contains a NULL value, v_comm_ind is set to a non-zero value, indicating that the column is NULL.

The code then checks the null indicator, and displays the appropriate on-screen results:

```
if (v_comm_ind)

printf("empno(%d), ename(%s), sal(%.2f) comm(NULL)\n",

v_empno, v_ename, v_sal);

else

printf("empno(%d), ename(%s), sal(%.2f) comm(%.2f)\n",

v_empno, v_ename, v_sal, v_comm);

}
```

If the null indicator is 0 (that is, false), v_comm contains a meaningful value, and the printf function displays the commission. If the null indicator contains a non-zero value, comm is NULL, and printf displays the string 'NULL'. Please note that a host variable (other than a null indicator) contains no meaningful value if you fetch a NULL into

that host variable; you must use null indicators for any value which may be NULL.

The final statements in the code sample close the cursor (employees), and the connection to the server:

EXEC SQL CLOSE employees; EXEC SQL DISCONNECT;

4.2 Using Descriptors

Dynamic SQL allows a client application to execute SQL statements that are composed at runtime. This is useful when you don't know the content or form a statement will take when you are writing a client application. ECPGPlus does *not* allow you to use a host variable in place of an identifier (such as a table name, column name or index name); instead, you should use dynamic SQL statements to build a string that includes the information, and then execute that string. The string is passed between the client and the server in the form of a *descriptor*. A descriptor is a data structure that contains both the data and the information about the shape of the data.

A client application must use a GET DESCRIPTOR statement to retrieve information from a descriptor. The following steps describe the basic flow of a client application using dynamic SQL:

1. Use an ALLOCATE DESCRIPTOR statement to allocate a descriptor for the result set (select list).
2. Use an ALLOCATE DESCRIPTOR statement to allocate a descriptor for the input parameters (bind variables).
3. Obtain, assemble or compute the text of an SQL statement.
4. Use a PREPARE statement to parse and syntax-check the SQL statement.
5. Use a DESCRIBE statement to describe the select list into the select-list descriptor.
6. Use a DESCRIBE statement to describe the input parameters into the bind-variables descriptor.
7. Prompt the user (if required) for a value for each input parameter. Use a SET DESCRIPTOR statement to assign the values into a descriptor.
8. Use a DECLARE CURSOR statement to define a cursor for the statement.
9. Use an OPEN CURSOR statement to open a cursor for the statement.
10. Use a FETCH statement to fetch each row from the cursor, storing each row in select-list descriptor.
11. Use a GET DESCRIPTOR command to interrogate the select-list descriptor to find the value of each column in the current row.
12. Use a CLOSE CURSOR statement to close the cursor and free any cursor resources.

A descriptor may contain the attributes listed in the table below:

Field	Type	Attribute Description
CARDINALITY	integer	The number of rows in the result set.
DATA	N/A	The data value.
DATETIME_INTERVAL_CODE	integer	If TYPE is 9: 1 - DATE 2 - TIME 3 - TIMESTAMP 4 - TIME WITH TIMEZONE 5 - TIMESTAMP WITH TIMEZONE
DATETIME_INTERVAL_PRECISION	integer	Unused.
INDICATOR	integer	Indicates a NULL or truncated value.

Field	Type	Attribute Description
KEY_MEMBER	integer	Unused (returns FALSE).
LENGTH	integer	The data length (as stored on server).
NAME	string	The name of the column in which the data resides.
NULLABLE	integer	Unused (returns TRUE).
OCTET_LENGTH	integer	The data length (in bytes) as stored on server.
PRECISION	integer	The data precision (if the data is of numeric type).
RETURNED_LENGTH	integer	Actual length of data item.
RETURNED_OCTET_LENGTH	integer	Actual length of data item.
SCALE	integer	The data scale (if the data is of numeric type).
TYPE	integer	A numeric code that represents the data type of the column: 1 - SQL3_CHARACTER 2 - SQL3_NUMERIC 3 - SQL3_DECIMAL 4 - SQL3_INTEGER 5 - SQL3_SMALLINT 6 - SQL3_FLOAT 7 - SQL3_REAL 8 - SQL3_DOUBLE_PRECISION 9 - SQL3_DATE_TIME_TIMESTAMP 10 - SQL3_INTERVAL 12 - SQL3_CHARACTER_VARYING 13 - SQL3_ENUMERATED 14 - SQL3_BIT 15 - SQL3_BIT_VARYING 16 - SQL3_BOOLEAN

Example - Using a Descriptor to Return Data

The following simple application executes an SQL statement entered by an end user. The code sample demonstrates:

- how to use a SQL descriptor to execute a SELECT statement.
- how to find the data and metadata returned by the statement.

The application accepts an SQL statement from an end user, tests the statement to see if it includes the SELECT keyword, and executes the statement.

When invoking the application, an end user must provide the name of the database on which the SQL statement will be performed, and a string that contains the text of the query.

For example, a user might invoke the sample with the following command:

```
./exec_stmt edb "SELECT * FROM emp"
```

```
/************************************************************
/* exec_stmt.pgc
*
*/
#include \<stdio.h>
#include \<stdlib.h>
#include \<sql3types.h>
#include \<sqlca.h>
EXEC SQL WHENEVER SQLERROR SQLPRINT;
static void print_meta_data( char * desc_name );
char *md1 = "col field data ret";
char *md2 = "num name type len";
char *md3 = "--- -------------------- ----------------- ---";
int main( int argc, char *argv[] )
{
EXEC SQL BEGIN DECLARE SECTION;
char *db = argv[1];
char *stmt = argv[2];
int col_count;
EXEC SQL END DECLARE SECTION;
EXEC SQL CONNECT TO :db;
EXEC SQL ALLOCATE DESCRIPTOR parse_desc;
EXEC SQL PREPARE query FROM :stmt;
EXEC SQL DESCRIBE query INTO SQL DESCRIPTOR parse_desc;
EXEC SQL GET DESCRIPTOR 'parse_desc' :col_count = COUNT;
if( col_count == 0 )
{
EXEC SQL EXECUTE IMMEDIATE :stmt;
if( sqlca.sqlcode >= 0 )
EXEC SQL COMMIT;
```

```
}
else
{
int row;
EXEC SQL ALLOCATE DESCRIPTOR row_desc;
EXEC SQL DECLARE my_cursor CURSOR FOR query;
EXEC SQL OPEN my_cursor;
for( row = 0; ; row++ )
{
EXEC SQL BEGIN DECLARE SECTION;
int col;
EXEC SQL END DECLARE SECTION;
EXEC SQL FETCH IN my_cursor
INTO SQL DESCRIPTOR row_desc;
if( sqlca.sqlcode != 0 )
break;
if( row == 0 )
print_meta_data( "row_desc" );
printf("[RECORD %d]\n", row+1);
for( col = 1; col \<= col_count; col++ )
{
EXEC SQL BEGIN DECLARE SECTION;
short ind;
varchar val[40+1];
varchar name[20+1];
EXEC SQL END DECLARE SECTION;
EXEC SQL GET DESCRIPTOR 'row_desc'
VALUE :col
:val = DATA, :ind = INDICATOR, :name = NAME;
if( ind == -1 )
printf( " %-20s : \<null>\n", name.arr );
else if( ind > 0 )
```

```
printf( " %-20s : \<truncated>\n", name.arr );
else
printf( " %-20s : %s\n", name.arr, val.arr );
}
printf( "\n" );
}
printf( "%d rows\n", row );
}
exit( 0 );
}
static void print_meta_data( char *desc_name )
{
EXEC SQL BEGIN DECLARE SECTION;
char *desc = desc_name;
int col_count;
int col;
EXEC SQL END DECLARE SECTION;
static char *types[] =
{
"unused ",
"CHARACTER ",
"NUMERIC ",
"DECIMAL ",
"INTEGER ",
"SMALLINT ",
"FLOAT ",
"REAL ",
"DOUBLE ",
"DATE_TIME ",
"INTERVAL ",
"unused ",
"CHARACTER_VARYING",
```

```
"ENUMERATED ",
"BIT ",
"BIT_VARYING ",
"BOOLEAN ",
"abstract "
};
EXEC SQL GET DESCRIPTOR :desc :col_count = count;
printf( "%s\n", md1 );
printf( "%s\n", md2 );
printf( "%s\n", md3 );
for( col = 1; col \<= col_count; col++ )
{
EXEC SQL BEGIN DECLARE SECTION;
int type;
int ret_len;
varchar name[21];
EXEC SQL END DECLARE SECTION;
char *type_name;
EXEC SQL GET DESCRIPTOR :desc
VALUE :col
:name = NAME,
:type = TYPE,
:ret_len = RETURNED_OCTET_LENGTH;
if( type > 0 && type \< SQL3_abstract )
type_name = types[type];
else
type_name = "unknown";
printf( "%02d: %-20s %-17s %04d\n",
col, name.arr, type_name, ret_len );
}
printf( "\n" );
}
```

/***

The code sample begins by including the prototypes and type definitions for the C stdio and stdlib libraries, SQL data type symbols, and the SQLCA (SQL communications area) structure:

#include \<stdio.h>

#include \<stdlib.h>

#include \<sql3types.h>

#include \<sqlca.h>

The sample provides minimal error handling; when the application encounters an SQL error, it prints the error message to screen:

EXEC SQL WHENEVER SQLERROR SQLPRINT;

The application includes a forward-declaration for a function named print_meta_data() that will print the metadata found in a descriptor:

static void print_meta_data(char * desc_name);

The following code specifies the column header information that the application will use when printing the metadata:

char *md1 = "col field data ret";

char *md2 = "num name type len";

char *md3 = "--- ---------- ---------- ---";

int main(int argc, char *argv[])

{

The following declaration section identifies the host variables that will contain the name of the database to which the application will connect, the content of the SQL Statement, and a host variable that will hold the number of columns in the result set (if any).

EXEC SQL BEGIN DECLARE SECTION;

char *db = argv[1];

char *stmt = argv[2];

int col_count;

EXEC SQL END DECLARE SECTION;

The application connects to the database (using the default credentials):

EXEC SQL CONNECT TO :db;

Next, the application allocates an SQL descriptor to hold the metadata for a statement:

EXEC SQL ALLOCATE DESCRIPTOR parse_desc;

The application uses a PREPARE statement to syntax check the string provided by the user:

EXEC SQL PREPARE query FROM :stmt;

and a DESCRIBE statement to move the metadata for the query into the SQL descriptor.

```
EXEC SQL DESCRIBE query INTO SQL DESCRIPTOR parse_desc;
```

Then, the application interrogates the descriptor to discover the number of columns in the result set, and stores that in the host variable col_count.

```
EXEC SQL GET DESCRIPTOR parse_desc :col_count = COUNT;
```

If the column count is zero, the end user did not enter a SELECT statement; the application uses an EXECUTE IMMEDIATE statement to process the contents of the statement:

```
if( col_count == 0 )

{

EXEC SQL EXECUTE IMMEDIATE :stmt;
```

If the statement executes successfully, the application performs a COMMIT:

```
if( sqlca.sqlcode >= 0 )

EXEC SQL COMMIT;

}

else

{
```

If the statement entered by the user is a SELECT statement (which we know because the column count is non-zero), the application declares a variable named row.

```
int row;
```

Then, the application allocates another descriptor that holds the description and the values of a specific row in the result set:

```
EXEC SQL ALLOCATE DESCRIPTOR row_desc;
```

The application declares and opens a cursor for the prepared statement:

```
EXEC SQL DECLARE my_cursor CURSOR FOR query;

EXEC SQL OPEN my_cursor;
```

Loops through the rows in result set:

```
for( row = 0; ; row++ )

{

EXEC SQL BEGIN DECLARE SECTION;

int col;

EXEC SQL END DECLARE SECTION;
```

Then, uses a FETCH to retrieve the next row from the cursor into the descriptor:

```
EXEC SQL FETCH IN my_cursor INTO SQL DESCRIPTOR row_desc;
```

The application confirms that the FETCH did not fail; if the FETCH fails, the application has reached the end of the result set, and breaks the loop:

```
if( sqlca.sqlcode != 0 )

break;
```

The application checks to see if this is the first row of the cursor; if it is, the application prints the metadata for the row.

```
if( row == 0 )

print_meta_data( "row_desc" );
```

Next, it prints a record header containing the row number:

```
printf("[RECORD %d]\n", row+1);
```

Then, it loops through each column in the row:

```
for( col = 1; col \<= col_count; col++ )

{

EXEC SQL BEGIN DECLARE SECTION;

short ind;

varchar val[40+1];

varchar name[20+1];

EXEC SQL END DECLARE SECTION;
```

The application interrogates the row descriptor (row_desc) to copy the column value (:val), null indicator (:ind) and column name (:name) into the host variables declared above. Notice that you can retrieve multiple items from a descriptor using a comma-separated list.

```
EXEC SQL GET DESCRIPTOR row_desc

VALUE :col

:val = DATA, :ind = INDICATOR, :name = NAME;
```

If the null indicator (ind) is negative, the column value is NULL; if the null indicator is greater than 0, the column value is too long to fit into the val host variable (so we print \<truncated>); otherwise, the null indicator is 0 (meaning NOT NULL) so we print the value. In each case, we prefix the value (or \<null> or \<truncated>) with the name of the column.

```
if( ind == -1 )

printf( " %-20s : \<null>\n", name.arr );

else if( ind > 0 )

printf( " %-20s : \<truncated>\n", name.arr );

else

printf( " %-20s : %s\n", name.arr, val.arr );

}

printf( "\n" );

}
```

When the loop terminates, the application prints the number of rows fetched, and exits:

```
printf( "%d rows\n", row );

}

exit( 0 );

}
```

The print_meta_data() function extracts the metadata from a descriptor and prints the name, data type, and length of each column:

```
static void print_meta_data( char *desc_name )

{
```

The application declares host variables:

```
EXEC SQL BEGIN DECLARE SECTION;

char *desc = desc_name;

int col_count;

int col;

EXEC SQL END DECLARE SECTION;
```

The application then defines an array of character strings that map data type values (numeric) into data type names. We use the numeric value found in the descriptor to index into this array. For example, if we find that a given column is of type 2, we can find the name of that type (NUMERIC) by writing types[2].

```
static char *types[] =

{

"unused ",

"CHARACTER ",

"NUMERIC ",

"DECIMAL ",

"INTEGER ",

"SMALLINT ",

"FLOAT ",

"REAL ",

"DOUBLE ",

"DATE_TIME ",

"INTERVAL ",

"unused ",

"CHARACTER_VARYING",
```

```
"ENUMERATED ",

"BIT ",

"BIT_VARYING ",

"BOOLEAN ",

"abstract "

};
```

The application retrieves the column count from the descriptor. Notice that the program refers to the descriptor using a host variable (desc) that contains the name of the descriptor. In most scenarios, you would use an identifier to refer to a descriptor, but in this case, the caller provided the descriptor name, so we can use a host variable to refer to the descriptor.

```
EXEC SQL GET DESCRIPTOR :desc :col_count = count;
```

The application prints the column headers (defined at the beginning of this application):

```
printf( "%s\n", md1 );

printf( "%s\n", md2 );

printf( "%s\n", md3 );
```

Then, loops through each column found in the descriptor, and prints the name, type and length of each column.

```
for( col = 1; col \<= col_count; col++ )

{

EXEC SQL BEGIN DECLARE SECTION;

int type;

int ret_len;

varchar name[21];

EXEC SQL END DECLARE SECTION;

char *type_name;
```

It retrieves the name, type code, and length of the current column:

```
EXEC SQL GET DESCRIPTOR :desc

VALUE :col

:name = NAME,

:type = TYPE,

:ret_len = RETURNED_OCTET_LENGTH;
```

If the numeric type code matches a 'known' type code (that is, a type code found in the types[] array), it sets type_name to the name of the corresponding type; otherwise, it sets type_name to "unknown".

```
if( type > 0 && type \< SQL3_abstract )

type_name = types[type];
```

```
else
type_name = "unknown";
```

and prints the column number, name, type name, and length:

```
printf( "%02d: %-20s %-17s %04d\n",
col, name.arr, type_name, ret_len );
}
printf( "\n" );
}
```

If you invoke the sample application with the following command:

```
./exec_stmt test "SELECT * FROM emp WHERE empno IN(7902, 7934)"
```

The application returns:

```
col field               data             ret
num name                type             len
--- -------------------- ---------------- ---
01: empno               NUMERIC          0004
02: ename               CHARACTER_VARYING 0004
03: job                 CHARACTER_VARYING 0007
04: mgr                 NUMERIC          0004
05: hiredate            DATE_TIME        0018
06: sal                 NUMERIC          0007
07: comm                NUMERIC          0000
08: deptno              NUMERIC          0002
[RECORD 1]
  empno               : 7902
  ename               : FORD
  job                 : ANALYST
  mgr                 : 7566
  hiredate            : 03-DEC-81 00:00:00
  sal                 : 3000.00
  comm                : \<null>
  deptno              : 20
[RECORD 2]
```

```
empno          : 7934
ename          : MILLER
job            : CLERK
mgr            : 7782
hiredate       : 23-JAN-82 00:00:00
sal            : 1300.00
comm           : \<null>
deptno         : 10
2 rows
```

4.3 Building and Executing Dynamic SQL Statements

The following examples demonstrate four techniques for building and executing dynamic SQL statements. Each example demonstrates processing a different combination of statement and input types:

- The first example demonstrates processing and executing a SQL statement that does not contain a SELECT statement and does not require input variables. This example corresponds to the techniques used by Oracle Dynamic SQL Method 1.
- The second example demonstrates processing and executing a SQL statement that does not contain a SELECT statement, and contains a known number of input variables. This example corresponds to the techniques used by Oracle Dynamic SQL Method 2.
- The third example demonstrates processing and executing a SQL statement that may contain a SELECT statement, and includes a known number of input variables. This example corresponds to the techniques used by Oracle Dynamic SQL Method 3.
- The fourth example demonstrates processing and executing a SQL statement that may contain a SELECT statement, and includes an unknown number of input variables. This example corresponds to the techniques used by Oracle Dynamic SQL Method 4.

Example - Executing a Non-query Statement Without Parameters

The following example demonstrates how to use the EXECUTE IMMEDIATE command to execute a SQL statement where the text of the statement is not known until you run the application. You cannot use EXECUTE IMMEDIATE to execute a statement that returns a result set. You cannot use EXECUTE IMMEDIATE to execute a statement that contains parameter placeholders.

The EXECUTE IMMEDIATE statement parses and plans the SQL statement each time it executes, which can have a negative impact on the performance of your application. If you plan to execute the same statement repeatedly, consider using the PREPARE/EXECUTE technique described in the next example.

```
/*************************************************************/

#include \<stdio.h> #include \<string.h> #include \<stdlib.h>

static void handle_error(void);

int main(int argc, char *argv[]) { char *insertStmt;
```

```
EXEC SQL WHENEVER SQLERROR DO handle_error();

EXEC SQL CONNECT :argv[1];

insertStmt = "INSERT INTO dept VALUES(50, 'ACCTG', 'SEATTLE')";

EXEC SQL EXECUTE IMMEDIATE :insertStmt;

fprintf(stderr, "ok\n");

EXEC SQL COMMIT RELEASE;

exit(EXIT_SUCCESS); }

static void handle_error(void) { fprintf(stderr, "%s\n", sqlca.sqlerrm.sqlerrmc);

EXEC SQL WHENEVER SQLERROR CONTINUE; EXEC SQL ROLLBACK RELEASE;

exit(EXIT_FAILURE); }

/************************************************************/
```

The code sample begins by including the prototypes and type definitions for the C stdio, string, and stdlib libraries, and providing basic infrastructure for the program:

```
#include \<stdio.h> #include \<string.h> #include \<stdlib.h>

static void handle_error(void);

int main(int argc, char *argv[]) { char *insertStmt;
```

The example then sets up an error handler; ECPGPlus calls the handle_error() function whenever a SQL error occurs:

```
EXEC SQL WHENEVER SQLERROR DO handle_error();
```

Then, the example connects to the database using the credentials specified on the command line:

```
EXEC SQL CONNECT :argv[1];
```

Next, the program uses an EXECUTE IMMEDIATE statement to execute a SQL statement, adding a row to the dept table:

```
insertStmt = "INSERT INTO dept VALUES(50, 'ACCTG', 'SEATTLE')";

EXEC SQL EXECUTE IMMEDIATE :insertStmt;
```

If the EXECUTE IMMEDIATE command fails for any reason, ECPGPlus will invoke the handle_error() function (which terminates the application after displaying an error message to the user). If the EXECUTE IMMEDIATE command succeeds, the application displays a message (ok) to the user, commits the changes, disconnects from the server, and terminates the application.

```
fprintf(stderr, "ok\n");

EXEC SQL COMMIT RELEASE;

exit(EXIT_SUCCESS); }
```

ECPGPlus calls the handle_error() function whenever it encounters a SQL error. The handle_error() function prints the content of the error message, resets the error handler, rolls back any changes, disconnects from the database, and terminates the application.

```
static void handle_error(void) { fprintf(stderr, "%s\n", sqlca.sqlerrm.sqlerrmc);
```

EXEC SQL WHENEVER SQLERROR CONTINUE; EXEC SQL ROLLBACK RELEASE;

exit(EXIT_FAILURE); }

Example - Executing a Non-query Statement with a Specified Number of Placeholders

To execute a non-query command that includes a known number of parameter placeholders, you must first PREPARE the statement (providing a *statement handle*), and then EXECUTE the statement using the statement handle. When the application executes the statement, it must provide a *value* for each placeholder found in the statement.

When an application uses the PREPARE/EXECUTE mechanism, each SQL statement is parsed and planned once, but may execute many times (providing different *values* each time).

ECPGPlus will convert each parameter value to the type required by the SQL statement, if possible; if not possible, ECPGPlus will report an error.

/***/

#include \<stdio.h> #include \<string.h> #include \<stdlib.h> #include \<sqlca.h>

static void handle_error(void);

int main(int argc, char *argv[]) { char *stmtText;

EXEC SQL WHENEVER SQLERROR DO handle_error();

EXEC SQL CONNECT :argv[1];

stmtText = "INSERT INTO dept VALUES(?, ?, ?)";

EXEC SQL PREPARE stmtHandle FROM :stmtText;

EXEC SQL EXECUTE stmtHandle USING :argv[2], :argv[3], :argv[4];

fprintf(stderr, "ok\n");

EXEC SQL COMMIT RELEASE;

exit(EXIT_SUCCESS); }

static void handle_error(void) { printf("%s\n", sqlca.sqlerrm.sqlerrmc);

EXEC SQL WHENEVER SQLERROR CONTINUE; EXEC SQL ROLLBACK RELEASE;

exit(EXIT_FAILURE); } /***/

The code sample begins by including the prototypes and type definitions for the C stdio, string, stdlib, and sqlca libraries, and providing basic infrastructure for the program:

#include \<stdio.h> #include \<string.h> #include \<stdlib.h> #include \<sqlca.h>

static void handle_error(void);

int main(int argc, char *argv[]) { char *stmtText;

The example then sets up an error handler; ECPGPlus calls the handle_error() function whenever a SQL error occurs:

EXEC SQL WHENEVER SQLERROR DO handle_error();

Then, the example connects to the database using the credentials specified on the command line:

EXEC SQL CONNECT :argv[1];

Next, the program uses a PREPARE statement to parse and plan a statement that includes three parameter markers - if the PREPARE statement succeeds, it will create a statement handle that you can use to execute the statement (in this example, the statement handle is named stmtHandle). You can execute a given statement multiple times using the same statement handle.

stmtText = "INSERT INTO dept VALUES(?, ?, ?)";

EXEC SQL PREPARE stmtHandle FROM :stmtText;

After parsing and planning the statement, the application uses the EXECUTE statement to execute the statement associated with the statement handle, substituting user-provided values for the parameter markers:

EXEC SQL EXECUTE stmtHandle USING :argv[2], :argv[3], :argv[4];

If the EXECUTE command fails for any reason, ECPGPlus will invoke the handle_error() function (which terminates the application after displaying an error message to the user). If the EXECUTE command succeeds, the application displays a message (ok) to the user, commits the changes, disconnects from the server, and terminates the application.

fprintf(stderr, "ok\n");

EXEC SQL COMMIT RELEASE;

exit(EXIT_SUCCESS); }

ECPGPlus calls the handle_error() function whenever it encounters a SQL error. The handle_error() function prints the content of the error message, resets the error handler, rolls back any changes, disconnects from the database, and terminates the application.

static void handle_error(void) { printf("%s\n", sqlca.sqlerrm.sqlerrmc);

EXEC SQL WHENEVER SQLERROR CONTINUE; EXEC SQL ROLLBACK RELEASE;

exit(EXIT_FAILURE); }

Example - Executing a Query With a Known Number of Placeholders

This example demonstrates how to execute a *query* with a known number of input parameters, and with a known number of columns in the result set. This method uses the PREPARE statement to parse and plan a query, before opening a cursor and iterating through the result set.

/***/

#include \<stdio.h> #include \<string.h> #include \<stdlib.h> #include \<stdbool.h> #include \<sqlca.h>

static void handle_error(void);

int main(int argc, char *argv[]) { VARCHAR empno[10]; VARCHAR ename[20];

EXEC SQL WHENEVER SQLERROR DO handle_error();

EXEC SQL CONNECT :argv[1];

EXEC SQL PREPARE queryHandle FROM "SELECT empno, ename FROM emp WHERE deptno = ?";

EXEC SQL DECLARE empCursor CURSOR FOR queryHandle;

```
EXEC SQL OPEN empCursor USING :argv[2];

EXEC SQL WHENEVER NOT FOUND DO break;

while(true) {

EXEC SQL FETCH empCursor INTO :empno, :ename;

printf("%-10s %s\n", empno.arr, ename.arr); }

EXEC SQL CLOSE empCursor;

EXEC SQL COMMIT RELEASE;

exit(EXIT_SUCCESS); }

static void handle_error(void) { printf("%s\n", sqlca.sqlerrm.sqlerrmc);

EXEC SQL WHENEVER SQLERROR CONTINUE; EXEC SQL ROLLBACK RELEASE;

exit(EXIT_FAILURE); }

/************************************************************/
```

The code sample begins by including the prototypes and type definitions for the C stdio, string, stdlib, stdbool, and sqlca libraries, and providing basic infrastructure for the program:

```
#include \<stdio.h> #include \<string.h> #include \<stdlib.h> #include \<stdbool.h> #include \<sqlca.h>

static void handle_error(void);

int main(int argc, char *argv[]) { VARCHAR empno[10]; VARCHAR ename[20];
```

The example then sets up an error handler; ECPGPlus calls the handle_error() function whenever a SQL error occurs:

```
EXEC SQL WHENEVER SQLERROR DO handle_error();
```

Then, the example connects to the database using the credentials specified on the command line:

```
EXEC SQL CONNECT :argv[1];
```

Next, the program uses a PREPARE statement to parse and plan a query that includes a single parameter marker - if the PREPARE statement succeeds, it will create a statement handle that you can use to execute the statement (in this example, the statement handle is named stmtHandle). You can execute a given statement multiple times using the same statement handle.

```
EXEC SQL PREPARE stmtHandle FROM "SELECT empno, ename FROM emp WHERE deptno = ?";
```

The program then declares and opens the cursor, empCursor, substituting a user-provided value for the parameter marker in the prepared SELECT statement. Notice that the OPEN statement includes a USING clause: the USING clause must provide a *value* for each placeholder found in the query:

```
EXEC SQL DECLARE empCursor CURSOR FOR stmtHandle;

EXEC SQL OPEN empCursor USING :argv[2];

EXEC SQL WHENEVER NOT FOUND DO break;

while(true) {
```

The program iterates through the cursor, and prints the employee number and name of each employee in the selected department:

EXEC SQL FETCH empCursor INTO :empno, :ename;

printf("%-10s %s\n", empno.arr, ename.arr); }

The program then closes the cursor, commits any changes, disconnects from the server, and terminates the application.

EXEC SQL CLOSE empCursor;

EXEC SQL COMMIT RELEASE;

exit(EXIT_SUCCESS); }

The application calls the handle_error() function whenever it encounters a SQL error. The handle_error() function prints the content of the error message, resets the error handler, rolls back any changes, disconnects from the database, and terminates the application.

static void handle_error(void) { printf("%s\n", sqlca.sqlerrm.sqlerrmc);

EXEC SQL WHENEVER SQLERROR CONTINUE; EXEC SQL ROLLBACK RELEASE;

exit(EXIT_FAILURE); }

Example - Executing a Query With an Unknown Number of Variables

The next example demonstrates executing a query with an unknown number of input parameters and/or columns in the result set. This type of query may occur when you prompt the user for the text of the query, or when a query is assembled from a form on which the user chooses from a number of conditions (i.e., a filter).

/***/

#include \<stdio.h>

#include \<stdlib.h>

#include \<sqlda.h>

#include \<sqlcpr.h>

SQLDA *params;

SQLDA *results;

static void allocateDescriptors(int count,

int varNameLength,

int indNameLenth);

static void bindParams(void);

static void displayResultSet(void);

int main(int argc, char *argv[])

{

EXEC SQL BEGIN DECLARE SECTION;

char *username = argv[1];

```
char *password = argv[2];
char *stmtText = argv[3];
EXEC SQL END DECLARE SECTION;
EXEC SQL WHENEVER SQLERROR sqlprint;
EXEC SQL CONNECT TO test
USER :username
IDENTIFIED BY :password;
params = sqlald(20, 64, 64);
results = sqlald(20, 64, 64);
EXEC SQL PREPARE stmt FROM :stmtText;
EXEC SQL DECLARE dynCursor CURSOR FOR stmt;
bindParams();
EXEC SQL OPEN dynCursor USING DESCRIPTOR params;
displayResultSet(20);
}
static void bindParams(void)
{
EXEC SQL DESCRIBE BIND VARIABLES FOR stmt INTO params;
if (params->F \< 0)
fprintf(stderr, "Too many parameters required\n");
else
{
int i;
params->N = params->F;
for (i = 0; i \< params->F; i++)
{
char *paramName = params->S[i];
int nameLen = params->C[i];
char paramValue[255];
printf("Enter value for parameter %.*s: ",
nameLen, paramName);
fgets(paramValue, sizeof(paramValue), stdin);
```

```
params->T[i] = 1; /* Data type = Character (1) */
params->L[i] = strlen(paramValue) - 1;
params->V[i] = strdup(paramValue);
}
}
}
static void displayResultSet(void)
{
EXEC SQL DESCRIBE SELECT LIST FOR stmt INTO results;
if (results->F \< 0)
fprintf(stderr, "Too many columns returned by query\n");
else if (results->F == 0)
return;
else
{
int col;
results->N = results->F;
for (col = 0; col \< results->F; col++)
{
int null_permitted, length;
sqlnul(\&results->T[col],
\&results->T[col],
\&null_permitted);
switch (results->T[col])
{
case 2: /* NUMERIC */
{
int precision, scale;
sqlprc(\&results->L[col], \&precision, \&scale);
if (precision == 0)
precision = 38;
length = precision + 3;
```

```
break;
}
case 12: /* DATE */
{
length = 30;
break;
}
default: /* Others */
{
length = results->L[col] + 1;
break;
}
}
results->V[col] = realloc(results->V[col], length);
results->L[col] = length;
results->T[col] = 1;
}
EXEC SQL WHENEVER NOT FOUND DO break;
while (1)
{
const char *delimiter = "";
EXEC SQL FETCH dynCursor USING DESCRIPTOR results;
for (col = 0; col \< results->F; col++)
{
if (*results->I[col] == -1)
printf("%s%s", delimiter, "\<null>");
else
printf("%s%s", delimiter, results->V[col]);
delimiter = ", ";
}
printf("\n");
}
```

```
}

}

/*************************************************************/
```

The code sample begins by including the prototypes and type definitions for the C stdio and stdlib libraries. In addition, the program includes the sqlda.h and sqlcpr.h header files. sqlda.h defines the SQLDA structure used throughout this example. sqlcpr.h defines a small set of functions used to interrogate the metadata found in an SQLDA structure.

```
#include \<stdio.h>

#include \<stdlib.h>

#include \<sqlda.h>

#include \<sqlcpr.h>
```

Next, the program declares pointers to two SQLDA structures. The first SQLDA structure (params) will be used to describe the metadata for any parameter markers found in the dynamic query text. The second SQLDA structure (results) will contain both the metadata and the result set obtained by executing the dynamic query.

```
SQLDA *params;

SQLDA *results;
```

The program then declares two helper functions (defined near the end of the code sample):

```
static void bindParams(void);

static void displayResultSet(void);
```

Next, the program declares three host variables; the first two (username and password) are used to connect to the database server; the third host variable (stmtTxt) is a NULL-terminated C string containing the text of the query to execute. Notice that the values for these three host variables are derived from the command-line arguments. When the program begins execution, it sets up an error handler and then connects to the database server:

```
int main(int argc, char *argv[])

{

EXEC SQL BEGIN DECLARE SECTION;

char *username = argv[1];

char *password = argv[2];

char *stmtText = argv[3];

EXEC SQL END DECLARE SECTION;

EXEC SQL WHENEVER SQLERROR sqlprint;

EXEC SQL CONNECT TO test

USER :username

IDENTIFIED BY :password;
```

Next, the program calls the sqlald()function to allocate the memory required for each descriptor. Each descriptor contains (among other things):

- a pointer to an array of column names
- a pointer to an array of indicator names
- a pointer to an array of data types
- a pointer to an array of lengths
- a pointer to an array of data values.

When you allocate an SQLDA descriptor, you specify the maximum number of columns you expect to find in the result set (for SELECT-list descriptors) or the maximum number of parameters you expect to find the dynamic query text (for bind-variable descriptors) - in this case, we specify that we expect no more than 20 columns and 20 parameters. You must also specify a maximum length for each column (or parameter) name and each indicator variable name - in this case, we expect names to be no more than 64 bytes long.

See Section 7.4 for a complete description of the SQLDA structure.

```
params = sqlald(20, 64, 64);

results = sqlald(20, 64, 64);
```

After allocating the SELECT-list and bind descriptors, the program prepares the dynamic statement and declares a cursor over the result set.

```
EXEC SQL PREPARE stmt FROM :stmtText;

EXEC SQL DECLARE dynCursor CURSOR FOR stmt;
```

Next, the program calls the bindParams() function. The bindParams() function examines the bind descriptor (params) and prompt the user for a value to substitute in place of each parameter marker found in the dynamic query.

```
bindParams();
```

Finally, the program opens the cursor (using the parameter values supplied by the user, if any) and calls the displayResultSet() function to print the result set produced by the query.

```
EXEC SQL OPEN dynCursor USING DESCRIPTOR params;

displayResultSet();

}
```

The bindParams() function determines whether the dynamic query contains any parameter markers, and, if so, prompts the user for a value for each parameter and then binds that value to the corresponding marker. The DESCRIBE BIND VARIABLE statement populates the params SQLDA structure with information describing each parameter marker.

```
static void bindParams(void)

{

EXEC SQL DESCRIBE BIND VARIABLES FOR stmt INTO params;
```

If the statement contains no parameter markers, params->F will contain 0. If the statement contains more parameters than will fit into the descriptor, params->F will contain a negative number (in this case, the absolute value of params->F indicates the number of parameter markers found in the statement). If params->F contains a positive number, that number indicates how many parameter markers were found in the statement.

```
if (params->F \< 0)

fprintf(stderr, "Too many parameters required\n");

else

{
```

```
int i;

params->N = params->F;
```

Next, the program executes a loop that prompts the user for a value, iterating once for each parameter marker found in the statement.

```
for (i = 0; i \< params->F; i++)

{

char *paramName = params->S[i];

int nameLen = params->C[i];

char paramValue[255];

printf("Enter value for parameter %.*s: ",

nameLen, paramName);

fgets(paramValue, sizeof(paramValue), stdin);
```

After prompting the user for a value for a given parameter, the program *binds* that value to the parameter by setting params->T[i] to indicate the data type of the value (see Section 7.3 for a list of type codes), params->L[i] to the length of the value (we subtract one to trim off the trailing new-line character added by fgets()), and params->V[i] to point to a copy of the NULL-terminated string provided by the user.

```
params->T[i] = 1; /* Data type = Character (1) */

params->L[i] = strlen(paramValue) + 1;

params->V[i] = strdup(paramValue);

}

}

}
```

The displayResultSet() function loops through each row in the result set and prints the value found in each column. displayResultSet() starts by executing a DESCRIBE SELECT LIST statement - this statement populates an SQLDA descriptor (results) with a description of each column in the result set.

```
static void displayResultSet(void)

{

EXEC SQL DESCRIBE SELECT LIST FOR stmt INTO results;
```

If the dynamic statement returns no columns (that is, the dynamic statement is not a SELECT statement), results->F will contain 0. If the statement returns more columns than will fit into the descriptor, results->F will contain a negative number (in this case, the absolute value of results->F indicates the number of columns returned by the statement). If results->F contains a positive number, that number indicates how many columns where returned by the query.

```
if (results->F \< 0)

fprintf(stderr, "Too many columns returned by query\n");

else if (results->F == 0)

return;
```

```
else

{

int col;

results->N = results->F;
```

Next, the program enters a loop, iterating once for each column in the result set:

```
for (col = 0; col \< results->F; col++)

{

int null_permitted, length;
```

To decode the type code found in results->T, the program invokes the sqlnul() function (see the description of the T member of the SQLDA structure in Section 7.4). This call to sqlnul() modifies results->T[col] to contain only the type code (the nullability flag is copied to null_permitted). This step is necessary because the DESCRIBE SELECT LIST statement encodes the type of each column and the nullability of each column into the T array.

```
sqlnul(\&results->T[col],

\&results->T[col],

\&null_permitted);
```

After decoding the actual data type of the column, the program modifies the results descriptor to tell ECPGPlus to return each value in the form of a NULL-terminated string. Before modifying the descriptor, the program must compute the amount of space required to hold each value. To make this computation, the program examines the maximum length of each column (results->V[col]) and the data type of each column (results->T[col]).

For numeric values (where results->T[col] = 2), the program calls the sqlprc() function to extract the precision and scale from the column length. To compute the number of bytes required to hold a numeric value in string form, displayResultSet() starts with the precision (that is, the maximum number of digits) and adds three bytes for a sign character, a decimal point, and a NULL terminator.

```
switch (results->T[col])

{

case 2: /* NUMERIC */

{

int precision, scale;

sqlprc(\&results->L[col], \&precision, \&scale);

if (precision == 0)

precision = 38;

length = precision + 3;

break;

}
```

For date values, the program uses a somewhat arbitrary, hard-coded length of 30. In a real-world application, you may want to more carefully compute the amount of space required.

```
case 12: /* DATE */
```

```
{

length = 30;

break;

}
```

For a value of any type other than date or numeric, displayResultSet() starts with the maximum column width reported by DESCRIBE SELECT LIST and adds one extra byte for the NULL terminator. Again, in a real-world application you may want to include more careful calculations for other data types.

```
default: /* Others */

{

length = results->L[col] + 1;

break;

}

}
```

After computing the amount of space required to hold a given column, the program allocates enough memory to hold the value, sets results->L[col] to indicate the number of bytes found at results->V[col], and set the type code for the column (results->T[col]) to 1 to instruct the upcoming FETCH statement to return the value in the form of a NULL-terminated string.

```
results->V[col] = malloc(length);

results->L[col] = length;

results->T[col] = 1;

}
```

At this point, the results descriptor is configured such that a FETCH statement can copy each value into an appropriately sized buffer in the form of a NULL-terminated string.

Next, the program defines a new error handler to break out of the upcoming loop when the cursor is exhausted.

```
EXEC SQL WHENEVER NOT FOUND DO break;

while (1)

{

const char *delimiter = "";
```

The program executes a FETCH statement to fetch the next row in the cursor into the results descriptor. If the FETCH statement fails (because the cursor is exhausted), control transfers to the end of the loop because of the EXEC SQL WHENEVER directive found before the top of the loop.

```
EXEC SQL FETCH dynCursor USING DESCRIPTOR results;
```

The FETCH statement will populate the following members of the results descriptor:

- *results->I[col] will indicate whether the column contains a NULL value (-1) or a non-NULL value (0). If the value non-NULL but too large to fit into the space provided, the value is truncated and *results->I[col] will contain a positive value.
- results->V[col] will contain the value fetched for the given column (unless *results->I[col] indicates that the column value is NULL).

- results->L[col] will contain the length of the value fetched for the given column

Finally, displayResultSet() iterates through each column in the result set, examines the corresponding NULL indicator, and prints the value. The result set is not aligned - instead, each value is separated from the previous value by a comma.

```
for (col = 0; col \< results->F; col++)

{

if (*results->I[col] == -1)

printf("%s%s", delimiter, "\<null>");

else

printf("%s%s", delimiter, results->V[col]);

delimiter = ", ";

}

printf("\n");

}

}

}

/*************************************************************/
```

4.4 Error Handling

ECPGPlus provides two methods to detect and handle errors in embedded SQL code:

- A client application can examine the sqlca data structure for error messages, and supply customized error handling for your client application.
- A client application can include EXEC SQL WHENEVER directives to instruct the ECPGPlus compiler to add error-handling code.

Error Handling with sqlca

sqlca (SQL communications area) is a global variable used by ecpglib to communicate information from the server to the client application. After executing a SQL statement (for example, an INSERT or SELECT statement) you can inspect the contents of sqlca to determine if the statement has completed successfully or if the statement has failed.

sqlca has the following structure:

```
struct

{
```

```
char sqlcaid[8];

long sqlabc;

long sqlcode;

struct

{

int sqlerrml;

char sqlerrmc[SQLERRMC_LEN];

} sqlerrm;

char sqlerrp[8];

long sqlerrd[6];

char sqlwarn[8];

char sqlstate[5];

} sqlca;
```

Use the following directive to implement sqlca functionality:

```
EXEC SQL INCLUDE sqlca;
```

If you include the ecpg directive, you do not need to #include the sqlca.h file in the client application's header declaration.

The Advanced Server sqlca structure contains the following members:

sqlcaid

> sqlcaid contains the string: "SQLCA".

sqlabc

> sqlabc contains the size of the sqlca structure.

sqlcode

> The sqlcode member has been deprecated with SQL 92; Advanced Server supports sqlcode for backward compatibility, but you should use the sqlstate member when writing new code.
>
> sqlcode is an integer value; a positive sqlcode value indicates that the client application has encountered a harmless processing condition, while a negative value indicates a warning or error.
>
> If a statement processes without error, sqlcode will contain a value of 0. If the client application encounters an error (or warning) during a statement's execution, sqlcode will contain the last code returned.
>
> The SQL standard defines only a positive value of 100, which indicates that he most recent SQL statement processed returned/affected no rows. Since the SQL standard does not define other sqlcode values, please be aware that the values assigned to each condition may vary from database to database.

sqlerrm is a structure embedded within sqlca, composed of two members:

> sqlerrml
>
> sqlerrml contains the length of the error message currently stored in sqlerrmc.

> sqlerrmc
>
> sqlerrmc contains the null-terminated message text associated with the code stored in sqlstate. If a message exceeds 149 characters in length, ecpglib will truncate the error message.

sqlerrp

sqlerrp contains the string "NOT SET".

sqlerrd is an array that contains six elements:

> sqlerrd[1] contains the OID of the processed row (if applicable).
>
> sqlerrd[2] contains the number of processed or returned rows.
>
> sqlerrd[0], sqlerrd[3], sqlerrd[4] and sqlerrd[5] are unused.

sqlwarn is an array that contains 8 characters:

> sqlwarn[0] contains a value of 'W' if any other element within sqlwarn is set to 'W'.
>
> sqlwarn[1] contains a value of 'W' if a data value was truncated when it was stored in a host variable.
>
> sqlwarn[2] contains a value of 'W' if the client application encounters a non-fatal warning.
>
> sqlwarn[3], sqlwarn[4], sqlwarn[5], sqlwarn[6], and sqlwarn[7] are unused.

sqlstate

> sqlstate is a 5 character array that contains a SQL-compliant status code after the execution of a statement from the client application. If a statement processes without error, sqlstate will contain a value of 00000. Please note that sqlstate is *not* a null-terminated string.
>
> sqlstate codes are assigned in a hierarchical scheme:
>
> - The first two characters of sqlstate indicate the general class of the condition.
> - The last three characters of sqlstate indicate a specific status within the class.

> If the client application encounters multiple errors (or warnings) during an SQL statement's execution sqlstate will contain the last code returned.

The following table lists the sqlstate and sqlcode values, as well as the symbolic name and error description for the related condition:

sqlstate	sqlcode (Deprecated)	Symbolic Name	Description
YE001	-12	ECPG_OUT_OF_MEMORY	Virtual memory is exhausted.
YE002	-200	ECPG_UNSUPPORTED	The preprocessor has generated an unrecognized item. Could indicate incompatibility between the preprocessor and the library.
07001, or 07002	-201	ECPG_TOO_MANY_ARGUMENTS	The program specifies more variables than the command expects.
07001, or 07002	-202	ECPG_TOO_FEW_ARGUMENTS	The program specified fewer variables than the command expects.

sqlstate	sqlcode (Deprecated)	Symbolic Name	Description
21000	-203	ECPG_TOO_MANY_MATCHES	The SQL command has returned multiple rows, but the statement was prepared to receive a single row.
42804	-204	ECPG_INT_FORMAT	The host variable (defined in the C code) is of type INT, and the selected data is of a type that cannot be converted into an INT. ecpglib uses the strtol() function to convert string values into numeric form.
42804	-205	ECPG_UINT_FORMAT	The host variable (defined in the C code) is an unsigned INT, and the selected data is of a type that cannot be converted into an unsigned INT. ecpglib uses the strtoul() function to convert string values into numeric form.
42804	-206	ECPG_FLOAT_FORMAT	The host variable (defined in the C code) is of type FLOAT, and the selected data is of a type that cannot be converted into an FLOAT. ecpglib uses the strtod() function to convert string values into numeric form.
42804	-211	ECPG_CONVERT_BOOL	The host variable (defined in the C code) is of type BOOL, and the selected data cannot be stored in a BOOL.
YE002	-2-1	ECPG_EMPTY	The statement sent to the server was empty.
22002	-213	ECPG_MISSING_INDICATOR	A NULL indicator variable has not been supplied for the NULL value returned by the server (the client application has received an unexpected NULL value).
42804	-214	ECPG_NO_ARRAY	The server has returned an array, and the corresponding host variable is not capable of storing an array.
42804	-215	ECPG_DATA_NOT_ARRAY	The server has returned a value that is not an array into a host variable that expects an array value.
08003	-220	ECPG_NO_CONN	The client application has attempted to use a non-existent connection.
YE002	-221	ECPG_NOT_CONN	The client application has attempted to use an allocated, but closed connection.
26000	-230	ECPG_INVALID_STMT	The statement has not been prepared.
33000	-240	ECPG_UNKNOWN_DESCRIPTOR	The specified descriptor is not found.
07009	-241	ECPG_INVALID_DESCRIPTOR_INDEX	The descriptor index is out-of-range.
YE002	-242	ECPG_UNKNOWN_DESCRIPTOR_ITEM	The client application has requested an invalid descriptor item (internal error).
07006	-243	ECPG_VAR_NOT_NUMERIC	A dynamic statement has returned a numeric value for a non-numeric host variable.
07006	-244	ECPG_VAR_NOT_CHAR	A dynamic SQL statement has returned a CHAR value, and the host variable is not a CHAR.

sqlstate	sqlcode (Deprecated)	Symbolic Name	Description
	-400	ECPG_PGSQL	The server has returned an error message; the resulting message contains the error text.
08007	-401	ECPG_TRANS	The server cannot start, commit or rollback the specified transaction.
08001	-402	ECPG_CONNECT	The client application's attempt to connect to the database has failed.
02000	100	ECPG_NOT_FOUND	The last command retrieved or processed no rows, or you have reached the end of a cursor.

EXEC SQL WHENEVER

Use the EXEC SQL WHENEVER directive to implement simple error handling for client applications compiled with ECPGPlus. The syntax of the directive is:

> EXEC SQL WHENEVER *condition action*;

This directive instructs the ECPG compiler to insert error-handling code into your program.

The code instructs the client application that it should perform a specified action if the client application detects a given condition. The *condition* may be one of the following:

> SQLERROR
>
> A SQLERROR condition exists when sqlca.sqlcode is less than zero.
>
> SQLWARNING
>
> A SQLWARNING condition exists when sqlca.sqlwarn[0] contains a 'W'.
>
> NOT FOUND
>
> A NOT FOUND condition exists when sqlca.sqlcode is ECPG_NOT_FOUND (when a query returns no data).

You can specify that the client application perform one of the following *actions* if it encounters one of the previous conditions:

> CONTINUE
>
> Specify CONTINUE to instruct the client application to continue processing, ignoring the current condition. CONTINUE is the default action.
>
> DO CONTINUE
>
> An action of DO CONTINUE will generate a CONTINUE statement in the emitted C code that if it encounters the condition, skips the rest of the code in the loop and continues with the next iteration. You can only use it within a loop.
>
> GOTO *label* or GO TO *label*
>
> Use a C goto statement to jump to the specified *label*.
>
> SQLPRINT

> Print an error message to stderr (standard error), using the sqlprint() function. The sqlprint() function prints sql error, followed by the contents of sqlca.sqlerrm.sqlerrmc.
>
> STOP
>
> Call exit(1) to signal an error, and terminate the program.
>
> DO BREAK
>
> Execute the C break statement. Use this action in loops, or switch statements.
>
> CALL *name*(*args*) or DO *name*(*args*)
>
> Invoke the C function specified by the name *parameter*, using the parameters specified in the *args* parameter.

Example:

The following code fragment prints a message if the client application encounters a warning, and aborts the application if it encounters an error:

```
EXEC SQL WHENEVER SQLWARNING SQLPRINT;

EXEC SQL WHENEVER SQLERROR STOP;
```

Please Note: The ECPGPlus compiler processes your program from top to bottom, even though the client application may not *execute* from top to bottom. The compiler directive is applied to each line in order, and remains in effect until the compiler encounters another directive.

If the control of the flow within your program is not top-to-bottom, you should consider adding error-handling directives to any parts of the program that may be inadvertently missed during compilation.

4.5 Reference

The sections that follow describe ecpgPlus language elements:

- C-Preprocessor Directives
- Supported C Data Types
- Type Codes
- The SQLDA Structure
- ECPGPlus Statements

C-preprocessor Directives

The ECPGPlus C-preprocessor enforces two behaviors that are dependent on the mode in which you invoke ECPGPlus:

- PROC mode
- non-PROC mode

Compiling in PROC mode

In PROC mode, ECPGPlus allows you to:

- Declare host variables outside of an EXEC SQL BEGIN/END DECLARE SECTION.
- Use any C variable as a host variable as long as it is of a data type compatible with ECPG.

When you invoke ECPGPlus in PROC mode (by including the -C PROC keywords), the ECPG compiler honors the following C-preprocessor directives:

> #include
>
> #if *expression*
>
> #ifdef *symbolName*
>
> #ifndef *symbolName*
>
> #else
>
> #elif *expression*
>
> #endif
>
> #define *symbolName expansion *#define *symbolName*([*macro arguments*]) *expansion*
>
> #undef *symbolName*
>
> #defined(*symbolName*)

Pre-processor directives are used to effect or direct the code that is received by the compiler. For example, using the following code sample:

```
#if HAVE_LONG_LONG == 1

#define BALANCE_TYPE long long

#else

#define BALANCE_TYPE double

#endif

...

BALANCE_TYPE customerBalance;
```

If you invoke ECPGPlus with the following command-line arguments:

```
ecpg –C PROC –DHAVE_LONG_LONG=1
```

ECPGPlus will copy the entire fragment (without change) to the output file, but will only send the following tokens to the ECPG parser:

```
long long customerBalance;
```

On the other hand, if you invoke ECPGPlus with the following command-line arguments:

```
ecpg –C PROC –DHAVE_LONG_LONG=0
```

The ECPG parser will receive the following tokens:

```
double customerBalance;
```

If your code uses preprocessor directives to filter the code that is sent to the compiler, the complete code is retained in the original code, while the ECPG parser sees only the processed token stream.

You can also use compatible syntax when executing the following preprocessor directives with an EXEC directive:

> EXEC ORACLE DEFINE
>
> EXEC ORACLE UNDEF
>
> EXEC ORACLE INCLUDE
>
> EXEC ORACLE IFDEF
>
> EXEC ORACLE IFNDEF
>
> EXEC ORACLE ELIF
>
> EXEC ORACLE ELSE
>
> EXEC ORACLE ENDIF EXEC ORACLE OPTION

For example, if your code includes the following:

> EXEC ORACLE IFDEF HAVE_LONG_LONG; #define BALANCE_TYPE long long EXEC ORACLE ENDIF; BALANCE_TYPE customerBalance;

If you invoke ECPGPlus with the following command-line arguments:

> ecpg –C PROC DEFINE=HAVE_LONG_LONG=1

ECPGPlus will send the following tokens to the output file, and the ECPG parser:

> long long customerBalance;

Please Note: the EXEC ORACLE pre-processor directives only work if you specify -C PROC on the ECPG command line.

Using the SELECT_ERROR Precompiler Option

When using ECPGPlus in compatible mode, you can use the SELECT_ERROR precompiler option to instruct your program how to handle result sets that contain more rows than the host variable can accommodate. The syntax is:

> SELECT_ERROR={YES|NO}

The default value is YES; a SELECT statement will return an error message if the result set exceeds the capacity of the host variable. Specify NO to instruct the program to suppress error messages when a SELECT statement returns more rows than a host variable can accommodate.

Use SELECT_ERROR with the EXEC ORACLE OPTION directive.

Compiling in non-PROC mode

If you do not include the -C PROC command-line option:

- C preprocessor directives are copied to the output file without change.
- You must declare the type and name of each C variable that you intend to use as a host variable within an EXEC SQL BEGIN/END DECLARE section.

When invoked in non-PROC mode, ECPG implements the behavior described in the PostgreSQL Core documentation.

Supported C Data Types

An ECPGPlus application must deal with two sets of data types: SQL data types (such as SMALLINT, DOUBLE PRECISION and CHARACTER VARYING) and C data types (like short, double and varchar[*n*]). When an application fetches data from the server, ECPGPlus will map each SQL data type to the type of the C variable into which the data is returned.

In general, ECPGPlus can convert most SQL server types into similar C types, but not all combinations are valid. For example, ECPGPlus will try to convert a SQL character value into a C integer value, but the conversion may fail (at execution time) if the SQL character value contains non-numeric characters. The reverse is also true; when an application sends a value to the server, ECPGPlus will try to convert the C data type into the required SQL type. Again, the conversion may fail (at execution time) if the C value cannot be converted into the required SQL type.

ECPGPlus can convert any SQL type into C character values (char[*n*] or varchar[*n*]). Although it is safe to convert any SQL type to/from char[*n*] or varchar[*n*], it is often convenient to use more natural C types such as int, double, or float.

The supported C data types are:

- short
- int
- unsigned int
- long long int
- float
- double
- char[n+1]
- varchar[n+1]
- bool
- and any equivalent created by a typedef

In addition to the numeric and character types supported by C, the pgtypeslib run-time library offers custom data types (and functions to operate on those types) for dealing with date/time and exact numeric values:

- timestamp
- interval
- date
- decimal
- numeric

To use a data type supplied by pgtypeslib, you must #include the proper header file.

Type Codes

The following table contains the type codes for *external* data types. An external data type is used to indicate the type of a C host variable. When an application binds a value to a parameter or binds a buffer to a SELECT-list item, the type code in the corresponding SQLDA descriptor (*descriptor*->T[*column*]) should be set to one of the following values:

Type Code	Host Variable Type (C Data Type)
1, 2, 8, 11, 12, 15, 23, 24, 91, 94, 95, 96, 97	char[]
3	int
4, 7, 21	float
5, 6	null-terminated string (char[length+1])
9	varchar
22	double
68	unsigned int

The following table contains the type codes for *internal* data types. An internal type code is used to indicate the type of a value as it resides in the database. The DESCRIBE SELECT LIST statement populates the data type array (*descriptor*->T[*column*]) using the following values.

Internal Type Code	Server Type
1	VARCHAR2
2	NUMBER
8	LONG
11	ROWID
12	DATE
23	RAW
24	LONG RAW
96	CHAR
100	BINARY FLOAT
101	BINARY DOUBLE
104	UROWID
187	TIMESTAMP
188	TIMESTAMP W/TIMEZONE
189	INTERVAL YEAR TO MONTH
190	INTERVAL DAY TO SECOND
232	TIMESTAMP LOCAL_TZ

The SQLDA Structure

Oracle Dynamic SQL method 4 uses the SQLDA data structure to hold the data and metadata for a dynamic SQL statement. A SQLDA structure can describe a set of input parameters corresponding to the parameter markers found in the text of a dynamic statement or the result set of a dynamic statement. The layout of the SQLDA structure is:

```
struct SQLDA

{

int N; /* Number of entries */

char **V; /* Variables */

int *L; /* Variable lengths */

short *T; /* Variable types */

short **I; /* Indicators */

int F; /* Count of variables discovered by DESCRIBE */

char **S; /* Variable names */

short *M; /* Variable name maximum lengths */

short *C; /* Variable name actual lengths */

char **X; /* Indicator names */

short *Y; /* Indicator name maximum lengths */
```

```
short *Z; /* Indicator name actual lengths */

};
```

Parameters

N - *maximum number of entries*

The N structure member contains the maximum number of entries that the SQLDA may describe. This member is populated by the sqlald() function when you allocate the SQLDA structure. Before using a descriptor in an OPEN or FETCH statement, you must set N to the *actual* number of values described.

V - *data values*

The V structure member is a pointer to an array of data values.

> For a SELECT-list descriptor, V points to an array of values returned by a FETCH statement (each member in the array corresponds to a column in the result set).
>
> For a bind descriptor, V points to an array of parameter values (you must populate the values in this array before opening a cursor that uses the descriptor).

Your application must allocate the space required to hold each value. See the displayResultSet() function for an example of how to allocate space for SELECT-list values (Section 5.4, Executing a Query with an Unknown Number of Variables).

L - *length of each data value*

The L structure member is a pointer to an array of lengths. Each member of this array must indicate the amount of memory available in the corresponding member of the V array. For example, if V[5] points to a buffer large enough to hold a 20-byte NULL-terminated string, L[5] should contain the value 21 (20 bytes for the characters in the string plus 1 byte for the NULL-terminator). Your application must set each member of the L array.

T - *data types*

The T structure member points to an array of data types, one for each column (or parameter) described by the descriptor.

> For a bind descriptor, you must set each member of the T array to tell ECPGPlus the data type of each parameter.
>
> For a SELECT-list descriptor, the DESCRIBE SELECT LIST statement sets each member of the T array to reflect the type of data found in the corresponding column.

You may change any member of the T array before executing a FETCH statement to force ECPGPlus to convert the corresponding value to a specific data type. For example, if the DESCRIBE SELECT LIST statement indicates that a given column is of type DATE, you may change the corresponding T member to request that the next FETCH statement return that value in the form of a NULL-terminated string. Each member of the T array is a numeric type code (see Section 7.3 for a list of type codes). The type codes returned by a DESCRIBE SELECT LIST statement differ from those expected by a FETCH statement. After executing a DESCRIBE SELECT LIST statement, each member of T encodes a data type *and* a flag indicating whether the corresponding column is nullable. You can use the sqlnul() function to extract the type code and nullable flag from a member of the T array. The signature of the sqlnul() function is as follows:

```
void sqlnul(unsigned short *valType, unsigned short *typeCode, int *isNull)
```

For example, to find the type code and nullable flag for the third column of a descriptor named results, you would invoke sqlnul() as follows:

```
sqlnul(\&results->T[2], \&typeCode, \&isNull);
```

I - *indicator variables*

The I structure member points to an array of indicator variables. This array is allocated for you when your application calls the sqlald() function to allocate the descriptor.

> For a SELECT-list descriptor, each member of the I array indicates whether the corresponding column contains a NULL (non-zero) or non-NULL (zero) value.
>
> For a bind parameter, your application must set each member of the I array to indicate whether the corresponding parameter value is NULL.

F - *number of entries*

The F structure member indicates how many values are described by the descriptor (the N structure member indicates the *maximum* number of values which may be described by the descriptor; F indicates the actual number of values). The value of the F member is set by ECPGPlus when you execute a DESCRIBE statement. F may be positive, negative, or zero.

> For a SELECT-list descriptor, F will contain a positive value if the number of columns in the result set is equal to or less than the maximum number of values permitted by the descriptor (as determined by the N structure member); 0 if the statement is *not* a SELECT statement, or a negative value if the query returns more columns than allowed by the N structure member.
>
> For a bind descriptor, F will contain a positive number if the number of parameters found in the statement is less than or equal to the maximum number of values permitted by the descriptor (as determined by the N structure member); 0 if the statement contains no parameters markers, or a negative value if the statement contains more parameter markers than allowed by the N structure member.

If F contains a positive number (after executing a DESCRIBE statement), that number reflects the count of columns in the result set (for a SELECT-list descriptor) or the number of parameter markers found in the statement (for a bind descriptor). If F contains a negative value, you may compute the absolute value of F to discover how many values (or parameter markers) are required. For example, if F contains -24 after describing a SELECT list, you know that the query returns 24 columns.

S - *column/parameter names*

The S structure member points to an array of NULL-terminated strings.

> For a SELECT-list descriptor, the DESCRIBE SELECT LIST statement sets each member of this array to the name of the corresponding column in the result set.
>
> For a bind descriptor, the DESCRIBE BIND VARIABLES statement sets each member of this array to the name of the corresponding bind variable.

In this release, the name of each bind variable is determined by the left-to-right order of the parameter marker within the query - for example, the name of the first parameter is always ?0, the name of the second parameter is always ?1, and so on.

M - *maximum column/parameter name length*

The M structure member points to an array of lengths. Each member in this array specifies the *maximum* length of the corresponding member of the S array (that is, M[0] specifies the maximum length of the column/parameter name found at S[0]). This array is populated by the sqlald() function.

C - *actual column/parameter name length*

The C structure member points to an array of lengths. Each member in this array specifies the *actual* length of the corresponding member of the S array (that is, C[0] specifies the actual length of the column/parameter name found at S[0]).

This array is populated by the DESCRIBE statement.

X - *indicator variable names*

The X structure member points to an array of NULL-terminated strings - each string represents the name of a

NULL indicator for the corresponding value.

This array is not used by ECPGPlus, but is provided for compatibility with Pro*C applications.

Y - *maximum indicator name length*

The Y structure member points to an array of lengths. Each member in this array specifies the *maximum* length of the corresponding member of the X array (that is, Y[0] specifies the maximum length of the indicator name found at X[0]).

This array is not used by ECPGPlus, but is provided for compatibility with Pro*C applications.

Z - *actual indicator name length*

The Z structure member points to an array of lengths. Each member in this array specifies the *actual* length of the corresponding member of the X array (that is, Z[0] specifies the actual length of the indicator name found at X[0]).

This array is not used by ECPGPlus, but is provided for compatibility with Pro*C applications.

ECPGPlus Statements

An embedded SQL statement allows your client application to interact with the server, while an embedded directive is an instruction to the ECPGPlus compiler.

You can embed any Advanced Server SQL statement in a C program. Each statement should begin with the keywords EXEC SQL, and must be terminated with a semi-colon (;). Within the C program, a SQL statement takes the form:

> EXEC SQL *sql_command_body*;

Where *sql_command_body* represents a standard SQL statement. You can use a host variable anywhere that the SQL statement expects a value expression. For more information about substituting host variables for value expressions, please see Section 3.1.2, *Declaring Host Variables*.

ECPGPlus extends the PostgreSQL server-side syntax for some statements; for those statements, syntax differences are outlined in the following reference sections. For a complete reference to the supported syntax of other SQL commands, please refer to the *PostgreSQL Core Documentation* available at:

http://www.postgresql.org/docs/10/static/sql-commands.html

ALLOCATE DESCRIPTOR

Use the ALLOCATE DESCRIPTOR statement to allocate an SQL descriptor area:

> EXEC SQL [FOR *array_size*] ALLOCATE DESCRIPTOR *descriptor_name
>
> - [WITH MAX *variable_count*];

Where:

array_size is a variable that specifies the number of array elements to allocate for the descriptor. *array_size* may be an INTEGER value or a host variable.

descriptor_name is the host variable that contains the name of the descriptor, or the name of the descriptor. This value may take the form of an identifier, a quoted string literal, or of a host variable.

variable_count specifies the maximum number of host variables in the descriptor. The default value of *variable_count* is 100.

The following code fragment allocates a descriptor named emp_query that may be processed as an array (emp_array):

```
EXEC SQL FOR :emp_array ALLOCATE DESCRIPTOR emp_query;
```

CALL

Use the CALL statement to invoke a procedure or function on the server. The CALL statement works only on Advanced Server. The CALL statement comes in two forms; the first form is used to call a *function*:

```
EXEC SQL CALL program_name '('[actual_arguments]')' INTO [[:ret_variable][:ret_indicator]];
```

The second form is used to call a *procedure*:

```
EXEC SQL CALL program_name '('[actual_arguments]')';
```

Where:

program_name is the name of the stored procedure or function that the CALL statement invokes. The program name may be schema-qualified or package-qualified (or both); if you do not specify the schema or package in which the program resides, ECPGPlus will use the value of search_path to locate the program.

actual_arguments specifies a comma-separated list of arguments required by the program. Note that each *actual_argument* corresponds to a formal argument expected by the program. Each formal argument may be an IN parameter, an OUT parameter, or an INOUT parameter.

:*ret_variable* specifies a host variable that will receive the value returned if the program is a function.

:*ret_indicator* specifies a host variable that will receive the indicator value returned, if the program is a function.

For example, the following statement invokes the get_job_desc function with the value contained in the :ename host variable, and captures the value returned by that function in the :job host variable:

```
EXEC SQL CALL get_job_desc(:ename) INTO :job;
```

CLOSE

Use the CLOSE statement to close a cursor, and free any resources currently in use by the cursor. A client application cannot fetch rows from a closed cursor. The syntax of the CLOSE statement is:

```
EXEC SQL CLOSE [cursor_name];
```

Where:

cursor_name is the name of the cursor closed by the statement. The cursor name may take the form of an identifier or of a host variable.

The OPEN statement initializes a cursor. Once initialized, a cursor result set will remain unchanged unless the cursor is re-opened. You do not need to CLOSE a cursor before re-opening it.

To manually close a cursor named emp_cursor, use the command:

```
EXEC SQL CLOSE emp_cursor;
```

A cursor is automatically closed when an application terminates.

COMMIT

Use the COMMIT statement to complete the current transaction, making all changes permanent and visible to other users. The syntax is:

```
EXEC SQL [AT database_name] COMMIT [WORK] [COMMENT 'text'] [COMMENT 'text' RELEASE];
```

Where:

database_name is the name of the database (or host variable that contains the name of the database) in which the work resides. This value may take the form of an unquoted string literal, or of a host variable.

For compatibility, ECPGPlus accepts the COMMENT clause without error but does *not* store any text included with the COMMENT clause.

Include the RELEASE clause to close the current connection after performing the commit.

For example, the following command commits all work performed on the dept database and closes the current connection:

```
EXEC SQL AT dept COMMIT RELEASE;
```

By default, statements are committed only when a client application performs a COMMIT statement. Include the -t option when invoking ECPGPlus to specify that a client application should invoke AUTOCOMMIT functionality. You can also control AUTOCOMMIT functionality in a client application with the following statements:

```
EXEC SQL SET AUTOCOMMIT TO ON

and

EXEC SQL SET AUTOCOMMIT TO OFF
```

CONNECT

Use the CONNECT statement to establish a connection to a database. The CONNECT statement is available in two forms - one form is compatible with Oracle databases, the other is not.

The first form is compatible with Oracle databases:

```
EXEC SQL CONNECT {{:user_name IDENTIFIED BY :password} | :connection_id} [AT database_name]
[USING :database_string] [ALTER AUTHORIZATION :new_password];
```

Where:

user_name is a host variable that contains the role that the client application will use to connect to the server.

password is a host variable that contains the password associated with that role.

connection_id is a host variable that contains a slash-delimited user name and password used to connect to the database.

Include the AT clause to specify the database to which the connection is established. *database_name* is the name of the database to which the client is connecting; specify the value in the form of a variable, or as a string literal.

Include the USING clause to specify a host variable that contains a null-terminated string identifying the database to which the connection will be established.

The ALTER AUTHORIZATION clause is supported for syntax compatibility only; ECPGPlus parses the ALTER AUTHORIZATION clause, and reports a warning.

Using the first form of the CONNECT statement, a client application might establish a connection with a host variable named user that contains the identity of the connecting role, and a host variable named password that contains the associated password using the following command:

```
EXEC SQL CONNECT :user IDENTIFIED BY :password;
```

A client application could also use the first form of the CONNECT statement to establish a connection using a single host variable named :connection_id. In the following example, connection_id contains the slash-delimited role name and associated password for the user:

```
EXEC SQL CONNECT :connection_id;
```

The syntax of the second form of the CONNECT statement is:

```
EXEC SQL CONNECT TO database_name [AS connection_name] [credentials];
```

Where *credentials* is one of the following:

USER *user_name* *password *USER *user_name* IDENTIFIED BY *password* USER *user_name* USING *password*

In the second form:

database_name is the name or identity of the database to which the client is connecting. Specify *database_name* as a variable, or as a string literal, in one of the following forms:

database_name[@*hostname*][:*port*]

tcp:postgresql://*hostname*[:*port*][/*database_name*][*options*]

unix:postgresql://*hostname*[:*port*][/*database_name*][*options*]

Where:

hostname is the name or IP address of the server on which the database resides.

port is the port on which the server listens.

You can also specify a value of DEFAULT to establish a connection with the default database, using the default role name. If you specify DEFAULT as the target database, do not include a *connection_name* or *credentials*.

connection_name is the name of the connection to the database. *connection_name* should take the form of an identifier (that is, not a string literal or a variable). You can open multiple connections, by providing a unique *connection_name* for each connection.

If you do not specify a name for a connection, ecpglib assigns a name of DEFAULT to the connection. You can refer to the connection by name (DEFAULT) in any EXEC SQL statement.

CURRENT is the most recently opened or the connection mentioned in the most-recent SET CONNECTION TO statement. If you do not refer to a connection by name in an EXEC SQL statement, ECPG assumes the name of the connection to be CURRENT.

user_name is the role used to establish the connection with the Advanced Server database. The privileges of the specified role will be applied to all commands performed through the connection.

password is the password associated with the specified *user_name*.

The following code fragment uses the second form of the CONNECT statement to establish a connection to a database named edb, using the role alice and the password associated with that role, 1safepwd:

```
EXEC SQL CONNECT TO edb AS acctg_conn USER 'alice' IDENTIFIED BY '1safepwd';
```

The name of the connection is acctg_conn; you can use the connection name when changing the connection name using the SET CONNECTION statement.

DEALLOCATE DESCRIPTOR

Use the DEALLOCATE DESCRIPTOR statement to free memory in use by an allocated descriptor. The syntax of the statement is:

```
EXEC SQL DEALLOCATE DESCRIPTOR descriptor_name
```

Where:

descriptor_name is the name of the descriptor. This value may take the form of a quoted string literal, or of a host variable.

The following example deallocates a descriptor named emp_query:

```
EXEC SQL DEALLOCATE DESCRIPTOR emp_query;
```

DECLARE CURSOR

Use the DECLARE CURSOR statement to define a cursor. The syntax of the statement is:

```
EXEC SQL [AT database_name] DECLARE cursor_name CURSOR FOR (select_statement |
statement_name);
```

Where:

database_name is the name of the database on which the cursor operates. This value may take the form of an identifier or of a host variable. If you do not specify a database name, the default value of *database_name* is the default database.

cursor_name is the name of the cursor.

select_statement is the text of the SELECT statement that defines the cursor result set; the SELECT statement cannot contain an INTO clause.

statement_name is the name of a SQL statement or block that defines the cursor result set.

The following example declares a cursor named employees:

EXEC SQL DECLARE employees CURSOR FOR

SELECT

empno, ename, sal, comm

FROM

emp;

The cursor generates a result set that contains the employee number, employee name, salary and commission for each employee record that is stored in the emp table.

DECLARE DATABASE

Use the DECLARE DATABASE statement to declare a database identifier for use in subsequent SQL statements (for example, in a CONNECT statement). The syntax is:

```
EXEC SQL DECLARE database_name DATABASE;
```

Where:

database_name specifies the name of the database.

The following example demonstrates declaring an identifier for the acctg database:

```
EXEC SQL DECLARE acctg DATABASE;
```

After invoking the command declaring acctg as a database identifier, the acctg database can be referenced by name when establishing a connection or in AT clauses.

This statement has no effect and is provided for Pro*C compatibility only.

DECLARE STATEMENT

Use the DECLARE STATEMENT directive to declare an identifier for an SQL statement. Advanced Server supports two versions of the DECLARE STATEMENT directive:

```
EXEC SQL [database_name] DECLARE statement_name STATEMENT;

and

EXEC SQL DECLARE STATEMENT statement_name;
```

Where:

statement_name specifies the identifier associated with the statement.

database_name specifies the name of the database. This value may take the form of an identifier or of a host variable that contains the identifier.

A typical usage sequence that includes the DECLARE STATEMENT directive might be:

```
EXEC SQL DECLARE give_raise STATEMENT; // give_raise is now a statement handle (not prepared)

EXEC SQL PREPARE give_raise FROM :stmtText; // give_raise is now associated with a statement

EXEC SQL EXECUTE give_raise;
```

This statement has no effect and is provided for Pro*C compatibility only.

DELETE

Use the DELETE statement to delete one or more rows from a table. The syntax for the ECPGPlus DELETE statement is the same as the syntax for the SQL statement, but you can use parameter markers and host variables any place that an expression is allowed. The syntax is:

```
[FOR exec_count] DELETE FROM [ONLY] table [[AS] alias]

[USING using_list]

[WHERE condition | WHERE CURRENT OF cursor_name] [{RETURNING|RETURN} * | output_expression [[ AS] output_name] [, ...] INTO host_variable_list ]
```

Where:

Include the FOR *exec_count* clause to specify the number of times the statement will execute; this clause is valid only if the VALUES clause references an array or a pointer to an array.

table is the name (optionally schema-qualified) of an existing table. Include the ONLY clause to limit processing to the specified table; if you do not include the ONLY clause, any tables inheriting from the named table are also processed.

alias is a substitute name for the target table.

using_list is a list of table expressions, allowing columns from other tables to appear in the WHERE condition.

Include the WHERE clause to specify which rows should be deleted. If you do not include a WHERE clause in the statement, DELETE will delete all rows from the table, leaving the table definition intact.

condition is an expression, host variable or parameter marker that returns a value of type BOOLEAN. Those rows for which *condition* returns true will be deleted.

cursor_name is the name of the cursor to use in the WHERE CURRENT OF clause; the row to be deleted will be the one most recently fetched from this cursor. The cursor must be a non-grouping query on the DELETE statements target table. You cannot specify WHERE CURRENT OF in a DELETE statement that includes a Boolean condition.

The RETURN/RETURNING clause specifies an *output_expression* or *host_variable_list* that is returned by the DELETE command after each row is deleted:

> *output_expression* is an expression to be computed and returned by the DELETE command after each row is deleted. *output_name* is the name of the returned column; include * to return all columns.
>
> *host_variable_list* is a comma-separated list of host variables and optional indicator variables. Each host variable receives a corresponding value from the RETURNING clause.

For example, the following statement deletes all rows from the emp table where the sal column contains a value greater than the value specified in the host variable, :max_sal:

```
DELETE FROM emp WHERE sal > :max_sal;
```

For more information about using the DELETE statement, please see the PostgreSQL Core documentation available at:

http://www.postgresql.org/docs/10/static/sql-delete.html

DESCRIBE

Use the DESCRIBE statement to find the number of input values required by a prepared statement or the number of output values returned by a prepared statement. The DESCRIBE statement is used to analyze a SQL statement whose shape is unknown at the time you write your application.

The DESCRIBE statement populates an SQLDA descriptor; to populate a SQL descriptor, use the ALLOCATE DESCRIPTOR and DESCRIBE…DESCRIPTOR statements.

> EXEC SQL DESCRIBE BIND VARIABLES FOR *statement_name* INTO *descriptor;*
>
> or
>
> EXEC SQL DESCRIBE SELECT LIST FOR *statement_name* INTO *descriptor;*

Where:

statement_name is the identifier associated with a prepared SQL statement or PL/SQL block.

descriptor is the name of C variable of type SQLDA*. You must allocate the space for the descriptor by calling sqlald() (and initialize the descriptor) before executing the DESCRIBE statement.

When you execute the first form of the DESCRIBE statement, ECPG populates the given descriptor with a description of each input variable *required* by the statement. For example, given two descriptors:

```
SQLDA *query_values_in; SQLDA *query_values_out;
```

You might prepare a query that returns information from the emp table:

```
EXEC SQL PREPARE get_emp FROM "SELECT ename, empno, sal FROM emp WHERE empno = ?";
```

The command requires one input variable (for the parameter marker (?)).

```
EXEC SQL DESCRIBE BIND VARIABLES FOR get_emp INTO query_values_in;
```

After describing the bind variables for this statement, you can examine the descriptor to find the number of variables required and the type of each variable.

When you execute the second form, ECPG populates the given descriptor with a description of each value *returned* by the statement. For example, the following statement returns three values:

```
EXEC SQL DESCRIBE SELECT LIST FOR get_emp INTO query_values_out;
```

After describing the select list for this statement, you can examine the descriptor to find the number of returned values and the name and type of each value.

Before *executing* the statement, you must bind a variable for each input value and a variable for each output value. The variables that you bind for the input values specify the actual values used by the statement. The variables that you bind for the output values tell ECPGPlus where to put the values when you execute the statement.

This is alternate Pro*C compatible syntax for the DESCRIBE DESCRIPTOR statement.

DESCRIBE DESCRIPTOR

Use the DESCRIBE DESCRIPTOR statement to retrieve information about a SQL statement, and store that information in a SQL descriptor. Before using DESCRIBE DESCRIPTOR, you must allocate the descriptor with the ALLOCATE DESCRIPTOR statement. The syntax is:

> EXEC SQL DESCRIBE [INPUT | OUTPUT] *statement_identifier*

```
USING [SQL] DESCRIPTOR descriptor_name;
```

Where:

statement_name is the name of a prepared SQL statement.

descriptor_name is the name of the descriptor. *descriptor_name* can be a quoted string value or a host variable that contains the name of the descriptor.

If you include the INPUT clause, ECPGPlus populates the given descriptor with a description of each input variable *required* by the statement.

> For example, given two descriptors:
>
> EXEC SQL ALLOCATE DESCRIPTOR query_values_in; EXEC SQL ALLOCATE DESCRIPTOR query_values_out;

> You might prepare a query that returns information from the emp table:
>
> EXEC SQL PREPARE get_emp FROM "SELECT ename, empno, sal FROM emp WHERE empno = ?";
>
> The command requires one input variable (for the parameter marker (?)).
>
> EXEC SQL DESCRIBE INPUT get_emp USING 'query_values_in';
>
> After describing the bind variables for this statement, you can examine the descriptor to find the number of variables required and the type of each variable.
>
> If you do not specify the INPUT clause, DESCRIBE DESCRIPTOR populates the specified descriptor with the values returned by the statement.

If you include the OUTPUT clause, ECPGPlus populates the given descriptor with a description of each value *returned* by the statement.

> For example, the following statement returns three values:
>
> EXEC SQL DESCRIBE OUTPUT FOR get_emp USING 'query_values_out';
>
> After describing the select list for this statement, you can examine the descriptor to find the number of returned values and the name and type of each value.

DISCONNECT

Use the DISCONNECT statement to close the connection to the server. The syntax is:

> EXEC SQL DISCONNECT [*connection_name*][CURRENT][DEFAULT][ALL];

Where:

connection_name is the connection name specified in the CONNECT statement used to establish the connection. If you do not specify a connection name, the current connection is closed.

Include the CURRENT keyword to specify that ECPGPlus should close the most-recently used connection.

Include the DEFAULT keyword to specify that ECPGPlus should close the connection named DEFAULT. If you do not specify a name when opening a connection, ECPGPlus assigns the name, DEFAULT, to the connection.

Include the ALL keyword to instruct ECPGPlus to close all active connections.

The following example creates a connection (named hr_connection) that connects to the hr database, and then disconnects from the connection:

```
/* client.pgc*/

int main()

{

EXEC SQL CONNECT TO hr AS connection_name;

EXEC SQL DISCONNECT connection_name;

return(0);

}
```

EXECUTE

Use the EXECUTE statement to execute a statement previously prepared using an EXEC SQL PREPARE statement. The syntax is:

```
EXEC SQL [FOR array_size] EXECUTE statement_name [USING {DESCRIPTOR SQLDA_descriptor
|:host_variable [[INDICATOR] :indicator_variable]}];
```

Where:

array_size is an integer value or a host variable that contains an integer value that specifies the number of rows to be processed. If you omit the FOR clause, the statement is executed once for each member of the array.

statement_name specifies the name assigned to the statement when the statement was created (using the EXEC SQL PREPARE statement).

Include the USING clause to supply values for parameters within the prepared statement:

Include the DESCRIPTOR *SQLDA_descriptor* clause to provide an SQLDA descriptor value for a parameter.

Use a *host_variable* (and an optional *indicator_variable*) to provide a user-specified value for a parameter.

The following example creates a prepared statement that inserts a record into the emp table:

```
EXEC SQL PREPARE add_emp (numeric, text, text, numeric) AS

INSERT INTO emp VALUES($1, $2, $3, $4);
```

Each time you invoke the prepared statement, provide fresh parameter values for the statement:

```
EXEC SQL EXECUTE add_emp USING 8000, 'DAWSON', 'CLERK', 7788;

EXEC SQL EXECUTE add_emp USING 8001, 'EDWARDS', 'ANALYST', 7698;
```

EXECUTE DESCRIPTOR

Use the EXECUTE statement to execute a statement previously prepared by an EXEC SQL PREPARE statement, using an SQL descriptor. The syntax is:

```
EXEC SQL [FOR array_size] EXECUTE statement_identifier [USING [SQL] DESCRIPTOR descriptor_name]
[INTO [SQL] DESCRIPTOR descriptor_name];
```

Where:

array_size is an integer value or a host variable that contains an integer value that specifies the number of rows to be processed. If you omit the FOR clause, the statement is executed once for each member of the array.

statement_identifier specifies the identifier assigned to the statement with the EXEC SQL PREPARE statement.

Include the USING clause to specify values for any input parameters required by the prepared statement.

Include the INTO clause to specify a descriptor into which the EXECUTE statement will write the results returned by the prepared statement.

descriptor_name specifies the name of a descriptor (as a single-quoted string literal), or a host variable that contains the name of a descriptor.

The following example executes the prepared statement, give_raise, using the values contained in the descriptor stmtText:

```
EXEC SQL PREPARE give_raise FROM :stmtText;

EXEC SQL EXECUTE give_raise USING DESCRIPTOR :stmtText;
```

EXECUTE...END EXEC

Use the EXECUTE…END-EXEC statement to embed an anonymous block into a client application. The syntax is:

```
EXEC SQL [AT database_name] EXECUTE anonymous_block END-EXEC;
```

Where:

database_name is the database identifier or a host variable that contains the database identifier. If you omit the AT clause, the statement will be executed on the current default database.

anonymous_block is an inline sequence of PL/pgSQL or SPL statements and declarations. You may include host variables and optional indicator variables within the block; each such variable is treated as an IN/OUT value.

The following example executes an anonymous block:

```
EXEC SQL EXECUTE BEGIN IF (current_user = :admin_user_name) THEN
DBMS_OUTPUT.PUT_LINE('You are an administrator'); END IF; END-EXEC;
```

Please Note: the EXECUTE…END EXEC statement is supported only by Advanced Server.

EXECUTE IMMEDIATE

Use the EXECUTE IMMEDIATE statement to execute a string that contains a SQL command. The syntax is:

```
EXEC SQL [AT database_name] EXECUTE IMMEDIATE command_text;
```

Where:

database_name is the database identifier or a host variable that contains the database identifier. If you omit the AT clause, the statement will be executed on the current default database.

command_text is the command executed by the EXECUTE IMMEDIATE statement.

This dynamic SQL statement is useful when you don't know the text of an SQL statement (ie., when writing a client application). For example, a client application may prompt a (trusted) user for a statement to execute. After the user provides the text of the statement as a string value, the statement is then executed with an EXECUTE IMMEDIATE command.

The statement text may not contain references to host variables. If the statement may contain parameter markers or returns one or more values, you must use the PREPARE and DESCRIBE statements.

The following example executes the command contained in the :command_text host variable:

```
EXEC SQL EXECUTE IMMEDIATE :command_text;
```

FETCH

Use the FETCH statement to return rows from a cursor into an SQLDA descriptor or a target list of host variables. Before using a FETCH statement to retrieve information from a cursor, you must prepare the cursor using DECLARE and OPEN statements. The statement syntax is:

```
EXEC SQL [FOR array_size] FETCH cursor { USING DESCRIPTOR SQLDA_descriptor }|{ INTO target_list };
```

Where:

array_size is an integer value or a host variable that contains an integer value specifying the number of rows to fetch. If you omit the FOR clause, the statement is executed once for each member of the array.

cursor is the name of the cursor from which rows are being fetched, or a host variable that contains the name of the cursor.

If you include a USING clause, the FETCH statement will populate the specified SQLDA descriptor with the values returned by the server.

If you include an INTO clause, the FETCH statement will populate the host variables (and optional indicator variables) specified in the target_list.

The following code fragment declares a cursor named employees that retrieves the employee number, name and salary from the emp table:

```
EXEC SQL DECLARE employees CURSOR FOR

SELECT empno, ename, esal FROM emp;

EXEC SQL OPEN emp_cursor;

EXEC SQL FETCH emp_cursor INTO :emp_no, :emp_name, :emp_sal;
```

FETCH DESCRIPTOR

Use the FETCH DESCRIPTOR statement to retrieve rows from a cursor into an SQL descriptor. The syntax is:

```
EXEC SQL [FOR array_size] FETCH cursor INTO [SQL] DESCRIPTOR descriptor_name;
```

Where:

array_size is an integer value or a host variable that contains an integer value specifying the number of rows to fetch. If you omit the FOR clause, the statement is executed once for each member of the array.

cursor is the name of the cursor from which rows are fetched, or a host variable that contains the name of the cursor. The client must DECLARE and OPEN the cursor before calling the FETCH DESCRIPTOR statement.

Include the INTO clause to specify an SQL descriptor into which the EXECUTE statement will write the results returned by the prepared statement. *descriptor_name* specifies the name of a descriptor (as a single-quoted string literal), or a host variable that contains the name of a descriptor. Prior to use, the descriptor must be allocated using an ALLOCATE DESCRIPTOR statement.

The following example allocates a descriptor named row_desc that will hold the description and the values of a specific row in the result set. It then declares and opens a cursor for a prepared statement (my_cursor), before looping through the rows in result set, using a FETCH to retrieve the next row from the cursor into the descriptor:

```
EXEC SQL ALLOCATE DESCRIPTOR 'row_desc';

EXEC SQL DECLARE my_cursor CURSOR FOR query;

EXEC SQL OPEN my_cursor;

for( row = 0; ; row++ )

{
```

```
EXEC SQL BEGIN DECLARE SECTION;

int col;

EXEC SQL END DECLARE SECTION;

EXEC SQL FETCH my_cursor INTO SQL DESCRIPTOR 'row_desc';
```

GET DESCRIPTOR

Use the GET DESCRIPTOR statement to retrieve information from a descriptor. The GET DESCRIPTOR statement comes in two forms. The first form returns the number of values (or columns) in the descriptor.

> EXEC SQL GET DESCRIPTOR *descriptor_name* :*host_variable* = COUNT;

The second form returns information about a specific value (specified by the VALUE *column_number* clause).

> EXEC SQL [FOR *array_size*] GET DESCRIPTOR *descriptor_name* VALUE *column_number* {:*host_variable* = *descriptor_item* {,…}};

Where:

array_size is an integer value or a host variable that contains an integer value that specifies the number of rows to be processed. If you specify an array_size, the host_variable must be an array of that size; for example, if array_size is 10, :host_variable must be a 10-member array of host_variables. If you omit the FOR clause, the statement is executed once for each member of the array.

descriptor_name specifies the name of a descriptor (as a single-quoted string literal), or a host variable that contains the name of a descriptor.

Include the VALUE clause to specify the information retrieved from the descriptor.

> *column_number* identifies the position of the variable within the descriptor.
>
> *host_variable* specifies the name of the host variable that will receive the value of the item.
>
> *descriptor_item* specifies the type of the retrieved descriptor item.

ECPGPlus implements the following *descriptor_item* types:

- TYPE
- LENGTH
- OCTET_LENGTH
- RETURNED_LENGTH
- RETURNED_OCTET_LENGTH
- PRECISION
- SCALE
- NULLABLE
- INDICATOR
- DATA
- NAME

The following code fragment demonstrates using a GET DESCRIPTOR statement to obtain the number of columns entered in a user-provided string:

```
EXEC SQL ALLOCATE DESCRIPTOR parse_desc;

EXEC SQL PREPARE query FROM :stmt;

EXEC SQL DESCRIBE query INTO SQL DESCRIPTOR parse_desc;
```

```
EXEC SQL GET DESCRIPTOR parse_desc :col_count = COUNT;
```

The example allocates an SQL descriptor (named parse_desc), before using a PREPARE statement to syntax check the string provided by the user (:stmt). A DESCRIBE statement moves the user-provided string into the descriptor, parse_desc. The call to EXEC SQL GET DESCRIPTOR interrogates the descriptor to discover the number of columns (:col_count) in the result set.

INSERT

Use the INSERT statement to add one or more rows to a table. The syntax for the ECPGPlus INSERT statement is the same as the syntax for the SQL statement, but you can use parameter markers and host variables any place that a value is allowed. The syntax is:

> [FOR *exec_count*] INSERT INTO *table* [(*column* [, ...])]
>
> {DEFAULT VALUES |
>
> VALUES ({*expression* | DEFAULT} [, ...])[, ...] | *query*}
>
> [RETURNING * | *output_expression* [[AS] *output_name*] [, ...]]

Where:

Include the FOR *exec_count* clause to specify the number of times the statement will execute; this clause is valid only if the VALUES clause references an array or a pointer to an array.

table specifies the (optionally schema-qualified) name of an existing table.

column is the name of a column in the table. The column name may be qualified with a subfield name or array subscript. Specify the DEFAULT VALUES clause to use default values for all columns.

expression is the expression, value, host variable or parameter marker that will be assigned to the corresponding column. Specify DEFAULT to fill the corresponding column with its default value.

query specifies a SELECT statement that supplies the row(s) to be inserted.

output_expression is an expression that will be computed and returned by the INSERT command after each row is inserted. The expression can refer to any column within the table. Specify * to return all columns of the inserted row(s).

output_name specifies a name to use for a returned column.

The following example adds a row to the employees table:

```
INSERT INTO emp (empno, ename, job, hiredate)

VALUES ('8400', :ename, 'CLERK', '2011-10-31');
```

Note that the INSERT statement uses a host variable (:ename) to specify the value of the ename column.

For more information about using the INSERT statement, please see the PostgreSQL Core documentation available at:

http://www.postgresql.org/docs/10/static/sql-insert.html

OPEN

Use the OPEN statement to open a cursor. The syntax is:

> EXEC SQL [FOR *array_size*] OPEN *cursor* [USING *parameters*];
>
> Where *parameters* is one of the following:
>
> DESCRIPTOR *SQLDA_descriptor* or *host_variable* [[INDICATOR] *indicator_variable*, …]

Where:

array_size is an integer value or a host variable that contains an integer value specifying the number of rows to fetch. If you omit the FOR clause, the statement is executed once for each member of the array.

cursor is the name of the cursor being opened.

parameters is either DESCRIPTOR *SQLDA_descriptor* or a comma-separated list of host variables (and optional indicator variables) that initialize the cursor. If specifying an *SQLDA_descriptor*, the descriptor must be initialized with a DESCRIBE statement.

The OPEN statement initializes a cursor using the values provided in *parameters*. Once initialized, the cursor result set will remain unchanged unless the cursor is closed and re-opened. A cursor is automatically closed when an application terminates.

The following example declares a cursor named employees, that queries the emp table, returning the employee number, name, salary and commission of an employee whose name matches a user-supplied value (stored in the host variable, :emp_name).

```
EXEC SQL DECLARE employees CURSOR FOR

SELECT

empno, ename, sal, comm

FROM

emp

WHERE ename = :emp_name;

EXEC SQL OPEN employees;

...
```

After declaring the cursor, the example uses an OPEN statement to make the contents of the cursor available to a client application.

OPEN DESCRIPTOR

Use the OPEN DESCRIPTOR statement to open a cursor with a SQL descriptor. The syntax is:

> EXEC SQL [FOR *array_size*] OPEN *cursor* [USING [SQL] DESCRIPTOR *descriptor_name*] [INTO [SQL] DESCRIPTOR *descriptor_name*];

Where:

array_size is an integer value or a host variable that contains an integer value specifying the number of rows to fetch. If you omit the FOR clause, the statement is executed once for each member of the array.

cursor is the name of the cursor being opened.

descriptor_name specifies the name of an SQL descriptor (in the form of a single-quoted string literal) or a host variable that contains the name of an SQL descriptor that contains the query that initializes the cursor.

For example, the following statement opens a cursor (named emp_cursor), using the host variable, :employees:

EXEC SQL OPEN emp_cursor USING DESCRIPTOR :employees;

PREPARE

Prepared statements are useful when a client application must perform a task multiple times; the statement is parsed, written and planned only once, rather than each time the statement is executed, saving repetitive processing time.

Use the PREPARE statement to prepare an SQL statement or PL/pgSQL block for execution. The statement is available in two forms; the first form is:

> EXEC SQL [AT *database_name*] PREPARE *statement_name* FROM *sql_statement*;

The second form is:

> EXEC SQL [AT *database_name*] PREPARE *statement_name* AS *sql_statement*;

Where:

database_name is the database identifier or a host variable that contains the database identifier against which the statement will execute. If you omit the AT clause, the statement will execute against the current default database.

statement_name is the identifier associated with a prepared SQL statement or PL/SQL block.

sql_statement may take the form of a SELECT statement, a single-quoted string literal or host variable that contains the text of an SQL statement.

To include variables within a prepared statement, substitute placeholders ($1, $2, $3, etc.) for statement values that might change when you PREPARE the statement. When you EXECUTE the statement, provide a value for each parameter. The values must be provided in the order in which they will replace placeholders.

The following example creates a prepared statement (named add_emp) that inserts a record into the emp table:

> EXEC SQL PREPARE add_emp (int, text, text, numeric) AS
>
> INSERT INTO emp VALUES($1, $2, $3, $4);

Each time you invoke the statement, provide fresh parameter values for the statement:

> EXEC SQL EXECUTE add_emp(8003, 'Davis', 'CLERK', 2000.00);
>
> EXEC SQL EXECUTE add_emp(8004, 'Myer', 'CLERK', 2000.00);

Please note: A client application must issue a PREPARE statement within each session in which a statement will be executed; prepared statements persist only for the duration of the current session.

ROLLBACK

Use the ROLLBACK statement to abort the current transaction, and discard any updates made by the transaction. The syntax is:

> EXEC SQL [AT *database_name*] ROLLBACK [WORK] [{ TO [SAVEPOINT] *savepoint* } | RELEASE]

Where:

database_name is the database identifier or a host variable that contains the database identifier against which the

statement will execute. If you omit the AT clause, the statement will execute against the current default database.

Include the TO clause to abort any commands that were executed after the specified *savepoint*; use the SAVEPOINT statement to define the *savepoint*. If you omit the TO clause, the ROLLBACK statement will abort the transaction, discarding all updates.

Include the RELEASE clause to cause the application to execute an EXEC SQL COMMIT RELEASE and close the connection.

Use the following statement to rollback a complete transaction:

```
EXEC SQL ROLLBACK;
```

Invoking this statement will abort the transaction, undoing all changes, erasing any savepoints, and releasing all transaction locks. If you include a savepoint (my_savepoint in the following example):

```
EXEC SQL ROLLBACK TO SAVEPOINT my_savepoint;
```

Only the portion of the transaction that occurred after the my_savepoint is rolled back; my_savepoint is retained, but any savepoints created after my_savepoint will be erased.

Rolling back to a specified savepoint releases all locks acquired after the savepoint.

SAVEPOINT

Use the SAVEPOINT statement to define a *savepoint*; a savepoint is a marker within a transaction. You can use a ROLLBACK statement to abort the current transaction, returning the state of the server to its condition prior to the specified savepoint. The syntax of a SAVEPOINT statement is:

```
EXEC SQL [AT database_name] SAVEPOINT savepoint_name
```

Where:

database_name is the database identifier or a host variable that contains the database identifier against which the savepoint resides. If you omit the AT clause, the statement will execute against the current default database.

savepoint_name is the name of the savepoint. If you re-use a *savepoint_name*, the original savepoint is discarded.

Savepoints can only be established within a transaction block. A transaction block may contain multiple savepoints.

To create a savepoint named my_savepoint, include the statement:

```
EXEC SQL SAVEPOINT my_savepoint;
```

SELECT

ECPGPlus extends support of the SQL SELECT statement by providing the INTO *host_variables* clause. The clause allows you to select specified information from an Advanced Server database into a host variable. The syntax for the SELECT statement is:

EXEC SQL [AT *database_name*]

SELECT

[*hint*]

[ALL | DISTINCT [ON(*expression*, ...)]]

select_list **INTO *host_variables***

[FROM from_item [, *from_item*]...]

[WHERE *condition*]

[*hierarchical_query_clause*]

[GROUP BY *expression* [, ...]]

[HAVING *condition*]

[{ UNION [ALL] | INTERSECT | MINUS } (*subquery*)]

[ORDER BY *expression* [*order_by_options*]]

[LIMIT { *count* | ALL }]

[OFFSET start [ROW | ROWS]]

[FETCH { FIRST | NEXT } [count] { ROW | ROWS } ONLY]

[FOR { UPDATE | SHARE } [OF table_name [, ...]][NOWAIT][...]]

Where:

database_name is the name of the database (or host variable that contains the name of the database) in which the table resides. This value may take the form of an unquoted string literal, or of a host variable.

host_variables is a list of host variables that will be populated by the SELECT statement. If the SELECT statement returns more than a single row, *host_variables* must be an array.

ECPGPlus provides support for the additional clauses of the SQL SELECT statement as documented in the PostgreSQL Core documentation available at:

http://www.postgresql.org/docs/10/static/sql-select.html

To use the INTO *host_variables* clause, include the names of defined host variables when specifying the SELECT statement. For example, the following SELECT statement populates the :emp_name and :emp_sal host variables with a list of employee names and salaries:

```
EXEC SQL SELECT ename, sal INTO :emp_name, :emp_sal FROM emp WHERE empno = 7988;
```

The enhanced SELECT statement also allows you to include parameter markers (question marks) in any clause where a value would be permitted. For example, the following query contains a parameter marker in the WHERE clause:

```
SELECT * FROM emp WHERE dept_no = ?;
```

This SELECT statement allows you to provide a value at run-time for the dept_no parameter marker.

SET CONNECTION

There are (at least) three reasons you may need more than one connection in a given client application:

- You may want different privileges for different statements;
- You may need to interact with multiple databases within the same client.
- Multiple threads of execution (within a client application) cannot share a connection concurrently.

The syntax for the SET CONNECTION statement is:

```
EXEC SQL SET CONNECTION connection_name;
```

Where:

connection_name is the name of the connection to the database.

To use the SET CONNECTION statement, you should open the connection to the database using the second form of the CONNECT statement; include the AS clause to specify a *connection_name*.

By default, the current thread uses the current connection; use the SET CONNECTION statement to specify a default connection for the current thread to use. The default connection is only used when you execute an EXEC SQL statement that does not explicitly specify a connection name. For example, the following statement will use the default connection because it does not include an AT *connection_name* clause. :

```
EXEC SQL DELETE FROM emp;
```

This statement will not use the default connection because it specifies a connection name using the AT *connection_name* clause:

```
EXEC SQL AT acctg_conn DELETE FROM emp;
```

For example, a client application that creates and maintains multiple connections (such as):

```
EXEC SQL CONNECT TO edb AS acctg_conn USER 'alice' IDENTIFIED BY 'acctpwd';
```

and

```
EXEC SQL CONNECT TO edb AS hr_conn USER 'bob' IDENTIFIED BY 'hrpwd';
```

Can change between the connections with the SET CONNECTION statement:

```
SET CONNECTION acctg_conn;
```

or

```
SET CONNECTION hr_conn;
```

The server will use the privileges associated with the connection when determining the privileges available to the connecting client. When using the acctg_conn connection, the client will have the privileges associated with the role, alice; when connected using hr_conn, the client will have the privileges associated with bob.

SET DESCRIPTOR

Use the SET DESCRIPTOR statement to assign a value to a descriptor area using information provided by the client application in the form of a host variable or an integer value. The statement comes in two forms; the first form is:

```
EXEC SQL [FOR array_size] SET DESCRIPTOR descriptor_name VALUE column_number descriptor_item
= host_variable;
```

The second form is:

```
EXEC SQL [FOR array_size] SET DESCRIPTOR descriptor_name COUNT = integer;
```

Where:

array_size is an integer value or a host variable that contains an integer value specifying the number of rows to fetch. If you omit the FOR clause, the statement is executed once for each member of the array.

descriptor_name specifies the name of a descriptor (as a single-quoted string literal), or a host variable that

contains the name of a descriptor.

Include the VALUE clause to describe the information stored in the descriptor.

> *column_number* identifies the position of the variable within the descriptor.
>
> *descriptor_item* specifies the type of the descriptor item.
>
> *host_variable* specifies the name of the host variable that contains the value of the item.

ECPGPlus implements the following *descriptor_item* types:

- TYPE
- LENGTH
- [REF] INDICATOR
- [REF] DATA
- [REF] RETURNED LENGTH

For example, a client application might prompt a user for a dynamically created query:

```
query_text = promptUser("Enter a query");
```

To execute a dynamically created query, you must first *prepare* the query (parsing and validating the syntax of the query), and then *describe* the *input* parameters found in the query using the EXEC SQL DESCRIBE INPUT statement.

```
EXEC SQL ALLOCATE DESCRIPTOR query_params; EXEC SQL PREPARE emp_query FROM
:query_text;

EXEC SQL DESCRIBE INPUT emp_query USING SQL DESCRIPTOR 'query_params';
```

After describing the query, the query_params descriptor contains information about each parameter required by the query.

For this example, we'll assume that the user has entered:

```
SELECT ename FROM emp WHERE sal > ? AND job = ?;,
```

In this case, the descriptor describes two parameters, one for sal > ? and one for job = ?.

To discover the number of parameter markers (question marks) in the query (and therefore, the number of values you must provide before executing the query), use:

```
EXEC SQL GET DESCRIPTOR … :host_variable = COUNT;
```

Then, you can use EXEC SQL GET DESCRIPTOR to retrieve the name of each parameter. You can also use EXEC SQL GET DESCRIPTOR to retrieve the type of each parameter (along with the number of parameters) from the descriptor, or you can supply each *value* in the form of a character string and ECPG will convert that string into the required data type.

The data type of the first parameter is numeric; the type of the second parameter is varchar. The name of the first parameter is sal; the name of the second parameter is job.

Next, loop through each parameter, prompting the user for a value, and store those values in host variables. You can use GET DESCRIPTOR … COUNT to find the number of parameters in the query.

```
EXEC SQL GET DESCRIPTOR 'query_params' :param_count = COUNT;

for(param_number = 1; param_number \<= param_count; param_number++) {
```

Use GET DESCRIPTOR to copy the name of the parameter into the param_name host variable:

```
EXEC SQL GET DESCRIPTOR 'query_params' VALUE :param_number :param_name = NAME;
```

```
reply = promptUser(param_name); if (reply == NULL) reply_ind = 1; /* NULL */ else reply_ind = 0; /* NOT NULL */
```

To associate a *value* with each parameter, you use the EXEC SQL SET DESCRIPTOR statement. For example:

```
EXEC SQL SET DESCRIPTOR 'query_params' VALUE :param_number DATA = :reply; EXEC SQL SET DESCRIPTOR 'query_params' VALUE :param_number INDICATOR = :reply_ind; }
```

Now, you can use the EXEC SQL EXECUTE DESCRIPTOR statement to execute the prepared statement on the server.

UPDATE

Use an UPDATE statement to modify the data stored in a table. The syntax is:

EXEC SQL [AT *database_name*][FOR *exec_count*] UPDATE [ONLY] *table* [[AS] *alias*]

SET {*column* = { *expression* | DEFAULT } |

(*column* [, ...]) = ({ *expression*|DEFAULT } [, ...])} [, ...]

[FROM *from_list*]

[WHERE *condition* | WHERE CURRENT OF *cursor_name*] [RETURNING * | *output_expression* [[AS] *output_name*] [, ...]]

Where:

database_name is the name of the database (or host variable that contains the name of the database) in which the table resides. This value may take the form of an unquoted string literal, or of a host variable.

Include the FOR *exec_count* clause to specify the number of times the statement will execute; this clause is valid only if the SET or WHERE clause contains an array.

ECPGPlus provides support for the additional clauses of the SQL UPDATE statement as documented in the PostgreSQL Core documentation available at:

http://www.postgresql.org/docs/10/static/sql-update.html

A host variable can be used in any clause that specifies a value. To use a host variable, simply substitute a defined variable for any value associated with any of the documented UPDATE clauses.

The following UPDATE statement changes the job description of an employee (identified by the :ename host variable) to the value contained in the :new_job host variable, and increases the employees salary, by multiplying the current salary by the value in the :increase host variable:

```
EXEC SQL UPDATE emp SET job = :new_job, sal = sal * :increase WHERE ename = :ename;
```

The enhanced UPDATE statement also allows you to include parameter markers (question marks) in any clause where an input value would be permitted. For example, we can write the same update statement with a parameter marker in the WHERE clause:

```
EXEC SQL UPDATE emp SET job = ?, sal = sal * ? WHERE ename = :ename;
```

This UPDATE statement could allow you to prompt the user for a new value for the job column and provide the amount by which the sal column is incremented for the employee specified by :ename.

WHENEVER

Use the WHENEVER statement to specify the action taken by a client application when it encounters an SQL error or warning. The syntax is:

```
EXEC SQL WHENEVER condition action;
```

The following table describes the different conditions that might trigger an *action*:

Condition	Description
NOT FOUND	The server returns a NOT FOUND condition when it encounters a SELECT that returns no rows, or when a FETCH reaches the end of a result set.
SQLERROR	The server returns an SQLERROR condition when it encounters a serious error returned by an SQL statement.
SQLWARNING	The server returns an SQLWARNING condition when it encounters a non-fatal warning returned by an SQL statement.

The following table describes the actions that result from a client encountering a *condition*:

Action	Description
CALL *function*[([*args*])]	Instructs the client application to call the named *function*.
CONTINUE	Instructs the client application to proceed to the next statement.
DO BREAK	Instructs the client application to a C break statement. A break statement may appear in a loop or a switch statement. If executed, the break statement terminate the loop or the switch statement..
DO CONTINUE	Instructs the client application to emit a C continue statement. A continue statement may only exist within a loop, and if executed, will cause the flow of control to return to the top of the loop.
DO *function*([*args*])	Instructs the client application to call the named *function*.
GOTO *label* or GO TO *label*	Instructs the client application to proceed to the statement that contains the *label*.
SQLPRINT	Instructs the client application to print a message to standard error.
STOP	Instructs the client application to stop execution.

The following code fragment prints a message if the client application encounters a warning, and aborts the application if it encounters an error:

```
EXEC SQL WHENEVER SQLWARNING SQLPRINT;

EXEC SQL WHENEVER SQLERROR STOP;
```

Include the following code to specify that a client should continue processing after warning a user of a problem:

```
EXEC SQL WHENEVER SQLWARNING SQLPRINT;
```

Include the following code to call a function if a query returns no rows, or when a cursor reaches the end of a result set:

```
EXEC SQL WHENEVER NOT FOUND CALL error_handler(__LINE__);
```

4.6 Introduction

EnterpriseDB has enhanced ECPG (the PostgreSQL pre-compiler) to create ECPGPlus. ECPGPlus allows you to include Pro*C compatible embedded SQL commands in C applications when connected to an EDB Postgres Advanced Server (Advanced Server) database. When you use ECPGPlus to compile an application, the SQL code is syntax-checked and translated into C.

ECPGPlus supports:

- Oracle Dynamic SQL – Method 4 (ODS-M4).
- Pro*C compatible anonymous blocks.
- A CALL statement compatible with Oracle databases.

As part of ECPGPlus's Pro*C compatibility, you do not need to include the BEGIN DECLARE SECTION and END DECLARE SECTION directives.

PostgreSQL Compatibility

While most ECPGPlus statements will work with community PostgreSQL, the CALL statement, and the EXECUTE…END EXEC statement work only when the client application is connected to EDB Postgres Advanced Server.

Typographical Conventions Used in this Guide

Certain typographical conventions are used in this manual to clarify the meaning and usage of various statements, statements, programs, examples, etc. This section provides a summary of these conventions.

In the following descriptions a *term* refers to any word or group of words that are language keywords, user-supplied values, literals, etc. A term's exact meaning depends upon the context in which it is used.

- *Italic font* introduces a new term, typically, in the sentence that defines it for the first time.
- Fixed-width (mono-spaced) font is used for terms that must be given literally such as SQL statements, specific table and column names used in the examples, programming language keywords, etc. For example, SELECT * FROM emp;
- *Italic fixed-width font* is used for terms for which the user must substitute values in actual usage. For example, DELETE FROM *table_name*;

- A vertical pipe | denotes a choice between the terms on either side of the pipe. A vertical pipe is used to separate two or more alternative terms within square brackets (optional choices) or braces (one mandatory choice).

- Square brackets [ ] denote that one or none of the enclosed term(s) may be substituted. For example, [a | b], means choose one of "a" or "b" or neither of the two.

- Braces {} denote that exactly one of the enclosed alternatives must be specified. For example, { a | b }, means exactly one of "a" or "b" must be specified.

- Ellipses ... denote that the proceeding term may be repeated. For example, [a | b] ... means that you may have the sequence, "b a a b a".

5 Using the Procedural Languages

The Postgres procedural languages (PL/Perl, PL/Python, and PL/Java) are installed with by the Language Pack installer. You can also use an RPM package to add procedural language functionality to your Advanced Server installation. For more information about using an RPM package, please see the EDB Advanced Server Installation Guide, available at:

https://www.enterprisedb.com/resources/product-documentation

PL/Perl

The PL/Perl procedural language allows you to use Perl functions in Postgres applications.

You must install PL/Perl in each database (or in a template database) before creating a PL/Perl function. Use the CREATE LANGUAGE command at the EDB-PSQL command line to install PL/Perl. Open the EDB-PSQL client, establish a connection to the database in which you wish to install PL/Perl, and enter the command:

CREATE LANGUAGE plperl;

The server confirms that the language is loaded with the response:

CREATE LANGUAGE;

You can now use a Postgres client application to access the features of the PL/Perl language. The following PL/Perl example creates a function named perl_max that returns the larger of two integer values:

CREATE OR REPLACE FUNCTION perl_max (integer, integer) RETURNS integer AS $$ if ($_[0] > $_[1]) { return $_[0]; } return $_[1]; $$ LANGUAGE plperl;

Pass two values when calling the function:

SELECT perl_max(1, 2);

The server returns:

perl_max --------- 2 (1 row)

For more information about using the Perl procedural language, consult the official Postgres documentation available at:

https://www.postgresql.org/docs/10/static/plperl.html

PL/Python

The PL/Python procedural language allows you to create and execute functions written in Python within Postgres applications. The version of PL/Python used by Advanced Server and PostgreSQL is untrusted (plpython3u); it offers no restrictions on users to prevent potential security risks.

Install PL/Python in each database (or in a template database) before creating a PL/Python function. You can use the CREATE LANGUAGE command at the EDB-PSQL command line to install PL/Python. Use EDB-PSQL to connect to the database in which you wish to install PL/Python, and enter the command:

CREATE LANGUAGE plpython3u;

The server confirms that the language is loaded with the response:

CREATE LANGUAGE

After installing PL/Python in your database, you can use the features of the PL/Python language.

Please note: The indentation shown in the following example must be included as you enter the sample function in EDB-PSQL. The following PL/Python example creates a function named pymax that returns the larger of two integer values:

CREATE OR REPLACE FUNCTION pymax (a integer, b integer) RETURNS integer AS $$ if a > b: return a return

b $$ LANGUAGE plpython3u;

When calling the pymax function, pass two values as shown below:

SELECT pymax(12, 3);

The server returns:

pymax ------- 12 (1 row)

For more information about using the Python procedural language, consult the official PostgreSQL documentation available at:

https://www.postgresql.org/docs/10/static/plpython.html

PL/Tcl

The PL/Tcl procedural language allows you to use Tcl/Tk functions in applications.

You must install PL/Tcl in each database (or in a template database) before creating a PL/Tcl function. Use the CREATE LANGUAGE command at the EDB-PSQL command line to install PL/Tcl. Use the psql client to connect to the database in which you wish to install PL/Tcl, and enter the command:

CREATE LANGUAGE pltcl;

After creating the pltcl language, you can use the features of the PL/Tcl language from within your Postgres server.

The following PL/Tcl example creates a function named tcl_max that returns the larger of two integer values:

CREATE OR REPLACE FUNCTION tcl_max(integer, integer) RETURNS integer AS $$ if {[argisnull 1]} { if {[argisnull 2]} { return_null } return $2 } if {[argisnull 2]} { return $1 } if {$1 > $2} {return $1} return $2 $$ LANGUAGE pltcl;

Pass two values when calling the function:

SELECT tcl_max(1, 2);

The server returns:

tcl_max -------- 2 (1 row)

For more information about using the Tcl procedural language, consult the official Postgres documentation available at:

https://www.postgresql.org/docs/10/static/pltcl.html

5.1 Installing Language Pack

The graphical installer is available from the EnterpriseDB website or via StackBuilder Plus.

Invoking the Graphical Installer

On Windows, assume Administrator privileges, and double-click the installer icon; if prompted, provide the password associated with the Administrator account.

On a Linux host, assume superuser privileges, disable SELinux (if applicable), navigate into the directory in which the installer resides, and invoke the installer with the command:

> ./edb-languagepack-*version*.run

Where *version* identifies version and platform-specific installer information.

The installer Welcome window opens (see Figure 2.1).

Figure 2.1 – The Language Pack Welcome window.

Click Next to continue.

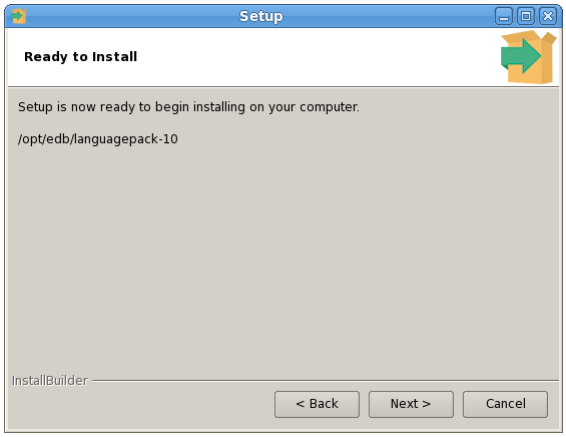

Figure 2.2 – The Language Pack Welcome window.

The Ready to Install window (see Figure 2.2) displays the Language Pack installation directory:

> On Linux 32 or 64: /opt/edb/languagepack-10/
>
> On Windows 32: C:\edb\languagepack-10\i386
>
> On Windows 64: C:\edb\languagepack-10\x64
>
> On OSX: /Library/edb/languagepack-10

You cannot modify the installation directory. Click Next to continue.

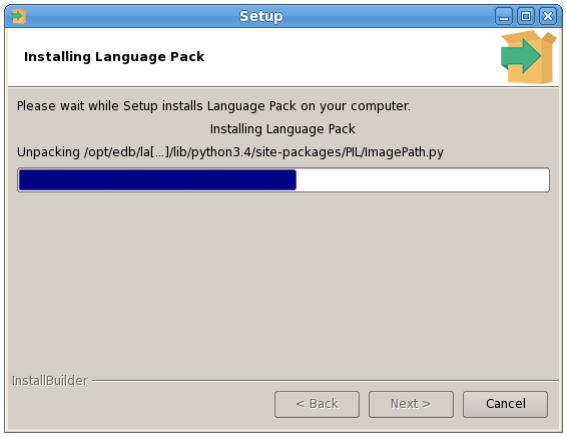

Figure 2.3 – The Language Pack Welcome window.

A progress bar marks installation progress (see Figure 2.3); click Next to continue.

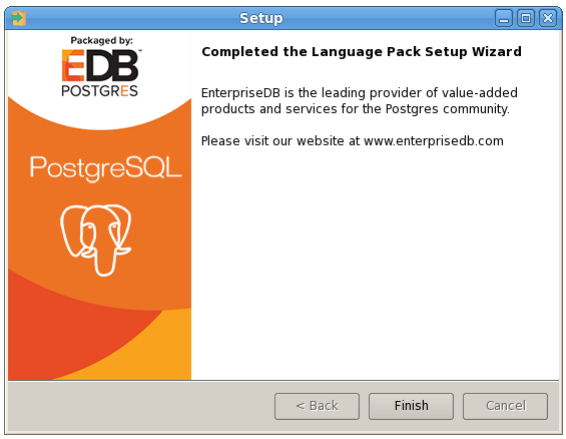

Figure 2.4 – The Language Pack Welcome window.

The installer will inform you that the Language Pack installation has completed (see Figure 2.4); click Finish to exit the installer.

Installing Language Pack with StackBuilder Plus

You can use StackBuilder Plus to download and invoke the Language Pack graphical installer. To open StackBuilder Plus, select the StackBuilder Plus menu item from the version-specific EDB Postgres sub-menu.

![image](./images/image7.png)

Figure 2.5 – The StackBuilder Plus Welcome window.

Select your server from the drop-down menu on the StackBuilder Plus Welcome window (see Figure 2.5) and click Next to continue.

Expand the Add-ons, tools and utilities node of the Categories tree control, and check the box to the left of EDB Language Pack; click Next to continue.

When prompted, provide your EnterpriseDB account credentials; if you have not registered for an account, use the provided link to register. StackBuilder Plus will confirm your package selection before downloading the installer. When the download completes, StackBuilder Plus will offer to invoke the installer for you, or to skip the installation until a more convenient time.

For details about using the graphical installer, see Section 2.1.

Configuring Language Pack on an Advanced Server Host

Configuring Language Pack on Linux

On Linux, the installer places the languages in:

```
/opt/edb/languagepack-10/
```

If you install Language Pack before Advanced Server, the Advanced Server installer will detect the Language Pack installation, and set the paths in the plLanguages.config file for you.

If you are invoking the Advanced Server installer using the --extract-only option, or if you install Language Pack after installing Advanced Server, you must manually configure the installation. The Language Pack configuration file is named:

```
/opt/edb/as10/etc/sysconfig/plLanguages.config
```

If you are installing Language Pack on a system that already hosts an Advanced Server installation, use your editor of choice to modify the plLanguages.config, changing the entries to include the locations of each language:

```
EDB_PERL_VERSION=5.24 EDB_PYTHON_VERSION=3.4 EDB_TCL_VERSION=8.6

EDB_PERL_PATH=/opt/edb/languagepack-10/Perl-5.24 EDB_PYTHON_PATH=/opt/edb/languagepack-
10/Python-3.4 EDB_TCL_PATH=/opt/edb/languagepack-10/Tcl-8.6
```

After modifying the plLanguages.config file, restart the server for the changes to take effect.

Configuring Language Pack on Windows

On Windows, the Language Pack installer places the languages in:

```
C:\edb\languagepack-10\x64
```

After installing Language Pack, you must set the following variables:

```
set PYTHONHOME=C:\edb\languagepack-10\x64\Python-3.4
```

Use the following commands to add Python, Perl and Tcl to your search path:

```
set PATH= C:\edb\LanguagePack-10\x64\Python-3.4\bin: C:\edb\LanguagePack-10\x64\Perl-5.24\bin:
C:\edb\LanguagePack-10\x64\Tcl-8.6\bin:%PATH%
```

After setting the system-specific steps required to configure Language Pack on Windows, restart the Advanced Server database server.

Configuring Language Pack on a PostgreSQL Host

After installing Language Pack on a PostgreSQL host, you must

Configuring Language Pack on Linux:

To simplify setting the value of PATH or LD_LIBRARY_PATH, you can create environment variables that identify the installation location:

```
PERLHOME=/opt/edb/languagepack-10/Perl-5.24 PYTHONHOME=/opt/edb/languagepack-10/Python-3.4
TCLHOME=/opt/edb/languagepack-10/Tcl-8.6
```

Then, instruct the Python interpreter where to find Python:

```
export PYTHONHOME
```

You can use the same environment variables when setting the value of PATH:

```
export PATH=$PYTHONHOME/bin:$PERLHOME/bin:$TCLHOME/bin:$PATH export
PATH=/opt/edb/languagepack-10/Python-3.4/bin:
```

Lastly, use the variables to tell Linux where to find the shared libraries:

```
export LD_LIBRARY_PATH= $PYTHONHOME/lib: $PERLHOME/lib/CORE: $TCLHOME/lib:
$LD_LIBRARY_PATH
```

Configuring Language Pack on Windows

On 32-bit Windows:

If you are using 32-bit Windows, you must tell the Python interpreter where to find Python:

```
set PYTHONHOME=C:\edb\languagepack-10\i386\Python-3.4
```

Then, set the path to the Language Pack installation:

```
SET PATH=C:\edb\languagepack-10\i386\Python-3.4; C:\edb\languagepack-10\i386\Perl-5.24\bin;
C:\edb\languagepack-10\i386\Tcl-8.6\bin;%PATH%
```

On 64-bit Windows:

After installing Language Pack, you must tell the Python interpreter where to find Python:

set PYTHONHOME=C:\edb\languagepack-10\x64\Python-3.3

Then, use the following commands to add Language Pack to your search path:

set PATH= C:\edb\LanguagePack-10\x64\Python-3.3\bin: C:\edb\LanguagePack-10\x64\Perl-5.20\bin:
C:\edb\LanguagePack-10\x64\Tcl-8.5\bin:%PATH%

After setting the system-specific steps required to configure Language Pack on Windows, restart the database server.

Configuring Language Pack on OSX

To simplify setting the value of PATH or LD_LIBRARY_PATH, you can create environment variables that identify the installation location:

```
PERLHOME=/Library/edb/languagepack-10/Perl-5.24 PYTHONHOME=/Library/edb/languagepack-
10/Python-3.4 TCLHOME=/Library/edb/languagepack-10/Tcl-8.6
```

Then, instruct the Python interpreter where to find Python:

```
export PYTHONHOME
```

You can use the same environment variables when setting the value of PATH:

```
export PATH=$PYTHONHOME/bin: $PERLHOME/bin: $TCLHOME/bin:$PATH
```

Lastly, use the variables to tell Linux where to find the shared libraries:

```
export DYLD_LIBRARY_PATH=$PYTHONHOME/lib: $PERLHOME/lib/CORE:$TCLHOME/lib:
$DYLD_LIBRARY_PATH
```