Database Compatibility for Oracle® Developers Tools and Utilities Guide

EDB Postgres™ Advanced Server 10

November 6, 2018

Database Compatibility for Oracle® Developers
Tools and Utilities Guide
by EnterpriseDB® Corporation

EnterpriseDB Corporation, 34 Crosby Drive, Suite 100, Bedford, MA 01730, USA
T +1 781 357 3390 **F** +1 978 589 5701 **E** info@enterprisedb.com **www**.enterprisedb.com

Table of Contents

1 Introduction

The tools and utilities documented in this guide allow a developer that is accustomed to working with Oracle utilities to work with Advanced Server in a familiar environment.

The sections in this guide describe compatible tools and utilities that are supported by Advanced Server. These include:

- EDB*Loader
- EDB*Wrap
- Dynamic Runtime Instrumentation

The EDB*Plus command line client provides a user interface to Advanced Server that supports SQL*Plus commands; EDB*Plus allows you to:

- Query database objects
- Execute stored procedures
- Format output from SQL commands
- Execute batch scripts
- Execute OS commands
- Record output

For detailed installation and usage information about EDB*Plus, please see the *EDB*Plus User's Guide*, available from the EnterpriseDB website at:

https://www.enterprisedb.com/resources/product-documentation

For detailed information about the features supported by Advanced Server, please consult the complete library of Advanced Server guides available at:

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

1.1 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`".

2 EDB*Loader

EDB*Loader is a high-performance bulk data loader that provides an interface compatible with Oracle databases for Advanced Server. The EDB*Loader command line utility loads data from an input source, typically a file, into one or more tables using a subset of the parameters offered by Oracle SQL*Loader.

EDB*Loader features include:

- Support for the Oracle SQL*Loader data loading methods - conventional path load, direct path load, and parallel direct path load
- Syntax for control file directives compatible with Oracle SQL*Loader
- Input data with delimiter-separated or fixed-width fields
- Bad file for collecting rejected records
- Loading of multiple target tables
- Discard file for collecting records that do not meet the selection criteria of any target table
- Log file for recording the EDB*Loader session and any error messages
- Data loading from standard input and remote loading, particularly useful for large data sources on remote hosts

These features are explained in detail in the following sections.

Note: The following are important version compatibility restrictions between the EDB*Loader client and the database server.

- Invoking EDB*Loader is done using a client program called `edbldr`, which is used to pass parameters and directive information to the database server. **It is strongly recommended that the EDB*Loader client supplied with Advanced Server 10 be used to load data only into version 10 of the database server. In general, the EDB*Loader client and database server should be the same version.**
- It is possible to use a version 10 EDB*Loader client to load data into a 10 database server, but the new EDB*Loader features may not be available under those circumstances.
- Use of a version 10, 9.6, 9.5, 9.4 or 9.3 EDB*Loader client is not supported for database servers version 9.2 or earlier.

2.1 Data Loading Methods

As with Oracle SQL*Loader, EDB*Loader supports three data loading methods:

- Conventional path load
- Direct path load
- Parallel direct path load

Conventional path load is the default method used by EDB*Loader. Basic insert processing is used to add rows to the table.

The advantage of a conventional path load over the other methods is that table constraints and database objects defined on the table such as primary keys, not null constraints, check constraints, unique indexes, foreign key constraints, and triggers are enforced during a conventional path load.

One exception is that the Advanced Server *rules* defined on the table are not enforced. EDB*Loader can load tables on which rules are defined, but the rules are not executed. As a consequence, partitioned tables implemented using rules cannot be loaded using EDB*Loader.

Note: Advanced Server rules are created by the `CREATE RULE` command. Advanced Server rules are not the same database objects as rules and rule sets used in Oracle.

EDB*Loader also supports direct path loads. A direct path load is faster than a conventional path load, but requires the removal of most types of constraints and triggers from the table. See Section 2.5 for information on direct path loads.

Finally, EDB*Loader supports parallel direct path loads. A parallel direct path load provides even greater performance improvement by permitting multiple EDB*Loader sessions to run simultaneously to load a single table. See Section 2.5.1 for information on parallel direct path loads.

2.2 General Usage

EDB*Loader can load data files with either delimiter-separated or fixed-width fields, in single-byte or multi-byte character sets. The delimiter can be a string consisting of one or more single-byte or multi-byte characters. Data file encoding and the database encoding may be different. Character set conversion of the data file to the database encoding is supported.

Each EDB*Loader session runs as a single, independent transaction. If an error should occur during the EDB*Loader session that aborts the transaction, all changes made during the session are rolled back.

Generally, formatting errors in the data file do not result in an aborted transaction. Instead, the badly formatted records are written to a text file called the *bad file*. The reason for the error is recorded in the *log file*.

Records causing database integrity errors do result in an aborted transaction and rollback. As with formatting errors, the record causing the error is written to the bad file and the reason is recorded in the log file.

Note: EDB*Loader differs from Oracle SQL*Loader in that a database integrity error results in a rollback in EDB*Loader. In Oracle SQL*Loader, only the record causing the error is rejected. Records that were previously inserted into the table are retained and loading continues after the rejected record.

The following are examples of types of formatting errors that do not abort the transaction:

- Attempt to load non-numeric value into a numeric column
- Numeric value is too large for a numeric column
- Character value is too long for the maximum length of a character column
- Attempt to load improperly formatted date value into a date column

The following are examples of types of database errors that abort the transaction and result in the rollback of all changes made in the EDB*Loader session:

- Violation of a unique constraint such as a primary key or unique index
- Violation of a referential integrity constraint
- Violation of a check constraint
- Error thrown by a trigger fired as a result of inserting rows

2.3 Building the EDB*Loader Control File

When you invoke EDB*Loader, the list of arguments provided must include the name of a control file. The control file includes the instructions that EDB*Loader uses to load the table (or tables) from the input data file. The control file includes information such as:

- The name of the input data file containing the data to be loaded.
- The name of the table or tables to be loaded from the data file.
- Names of the columns within the table or tables and their corresponding field placement in the data file.
- Specification of whether the data file uses a delimiter string to separate the fields, or if the fields occupy fixed column positions.
- Optional selection criteria to choose which records from the data file to load into a given table.
- The name of the file that will collect illegally formatted records.
- The name of the discard file that will collect records that do not meet the selection criteria of any table.

The syntax for the EDB*Loader control file is as follows:

```
[ OPTIONS (param=value [, param=value ] ...) ]
LOAD DATA
  [ CHARACTERSET charset ]
  [ INFILE '{ data_file | stdin }' ]
  [ BADFILE 'bad_file' ]
  [ DISCARDFILE 'discard_file' ]
  [ { DISCARDMAX | DISCARDS } max_discard_recs ]
[ INSERT | APPEND | REPLACE | TRUNCATE ]
[ PRESERVE BLANKS ]
{ INTO TABLE target_table
  [ WHEN field_condition [ AND field_condition ] ...]
  [ FIELDS TERMINATED BY 'termstring'
    [ OPTIONALLY ENCLOSED BY 'enclstring' ] ]
  [ RECORDS DELIMITED BY 'delimstring' ]
  [ TRAILING NULLCOLS ]
   (field_def [, field_def ] ...)
} ...
```

where *field_def* defines a field in the specified *data_file* that describes the location, data format, or value of the data to be inserted into *column_name* of the *target_table*. The syntax of *field_def* is the following:

```
column_name {
  CONSTANT val |
  FILLER [ POSITION (start:end) ] [ fieldtype ] |
  BOUNDFILLER [ POSITION (start:end) ] [ fieldtype ] |
  [ POSITION (start:end) ] [ fieldtype ]
```

```
  [ NULLIF field_condition [ AND field_condition ] ...]
  [ PRESERVE BLANKS ] [ "expr" ]
}
```

where *fieldtype* is one of:

```
CHAR [(length)] | DATE [(length)] [ "datemask" ] |
INTEGER EXTERNAL [(length)] |
FLOAT EXTERNAL [(length)] | DECIMAL EXTERNAL [(length)] |
ZONED EXTERNAL [(length)] | ZONED [(precision[,scale])]
```

Description

The specification of *data_file*, *bad_file*, and *discard_file* may include the full directory path or a relative directory path to the file name. If the file name is specified alone or with a relative directory path, the file is then assumed to exist (in the case of *data_file*), or is created (in the case of *bad_file* or *discard_file*), relative to the current working directory from which `edbldr` is invoked.

You can include references to environment variables within the EDB*Loader control file when referring to a directory path and/or file name. Environment variable references are formatted differently on Windows systems than on Linux systems:

- On Linux, the format is `$`*ENV_VARIABLE* or `${`*ENV_VARIABLE*`}`
- On Windows, the format is `%`*ENV_VARIABLE*`%`

Where *ENV_VARIABLE* is the environment variable that is set to the directory path and/or file name.

The `EDBLDR_ENV_STYLE` environment variable instructs Advanced Server to interpret environment variable references as Windows-styled references or Linux-styled references irregardless of the operating system on which EDB*Loader resides. You can use this environment variable to create portable control files for EDB*Loader.

- On a Windows system, set `EDBLDR_ENV_STYLE` to `linux` or `unix` to instruct Advanced Server to recognize Linux-style references within the control file.
- On a Linux system, set `EDBLDR_ENV_STYLE` to `windows` to instruct Advanced Server to recognize Windows-style references within the control file.

The operating system account `enterprisedb` must have read permission on the directory and file specified by *data_file*.

The operating system account `enterprisedb` must have write permission on the directories where *`bad_file`* and *`discard_file`* are to be written.

Note: It is suggested that the file names for *`data_file`*, *`bad_file`*, and *`discard_file`* include extensions of `.dat`, `.bad`, and `.dsc`, respectively. If the provided file name does not contain an extension, EDB*Loader assumes the actual file name includes the appropriate aforementioned extension.

If an EDB*Loader session results in data format errors and the `BADFILE` clause is not specified, nor is the `BAD` parameter given on the command line when `edbldr` is invoked, a bad file is created with the name *`control_file_base`*`.bad` in the current working directory from which `edbldr` is invoked. *`control_file_base`* is the base name of the control file (that is, the file name without any extension) used in the `edbldr` session.

If all of the following conditions are true, the discard file is not created even if the EDB*Loader session results in discarded records:

- The `DISCARDFILE` clause for specifying the discard file is not included in the control file.
- The `DISCARD` parameter for specifying the discard file is not included on the command line.
- The `DISCARDMAX` clause for specifying the maximum number of discarded records is not included in the control file.
- The `DISCARDS` clause for specifying the maximum number of discarded records is not included in the control file.
- The `DISCARDMAX` parameter for specifying the maximum number of discarded records is not included on the command line.

If neither the `DISCARDFILE` clause nor the `DISCARD` parameter for explicitly specifying the discard file name are specified, but `DISCARDMAX` or `DISCARDS` is specified, then the EDB*Loader session creates a discard file using the data file name with an extension of `.dsc`.

Note: There is a distinction between keywords `DISCARD` and `DISCARDS`. `DISCARD` is an EDB*Loader command line parameter used to specify the discard file name (see Section 2.2). `DISCARDS` is a clause of the `LOAD DATA` directive that may only appear in the control file. Keywords `DISCARDS` and `DISCARDMAX` provide the same functionality of specifying the maximum number of discarded records allowed before terminating the EDB*Loader session. Records loaded into the database before termination of the EDB*Loader session due to exceeding the `DISCARDS` or `DISCARDMAX` settings are kept in the database and are not rolled back.

If one of INSERT, APPEND, REPLACE, or TRUNCATE is specified, it establishes the default action of how rows are to be added to target tables. If omitted, the default action is as if INSERT had been specified.

If the FIELDS TERMINATED BY clause is specified, then the POSITION (*start*:*end*) clause may not be specified for any *field_def*. Alternatively if the FIELDS TERMINATED BY clause is not specified, then every *field_def* must contain either the POSITION (*start*:*end*) clause, the *fieldtype*(*length*) clause, or the CONSTANT clause.

Parameters

OPTIONS *param=value*

Use the OPTIONS clause to specify *param=value* pairs that represent an EDB*Loader directive. If a parameter is specified in both the OPTIONS clause and on the command line when edbldr is invoked, the command line setting is used.

Specify one or more of the following parameter/value pairs:

DIRECT= { FALSE | TRUE }

If DIRECT is set to TRUE EDB*Loader performs a direct path load instead of a conventional path load. The default value of DIRECT is FALSE.

See Section 2.5 for information on direct path loads.

ERRORS=*error_count*

error_count specifies the number of errors permitted before aborting the EDB*Loader session. The default is 50.

FREEZE= { FALSE | TRUE }

Set FREEZE to TRUE to indicate that the data should be copied with the rows *frozen*. A tuple guaranteed to be visible to all current and future transactions is marked as frozen to prevent transaction ID wrap-around. For more information about frozen tuples, see the PostgreSQL core documentation at:

https://www.postgresql.org/docs/10/static/routine-vacuuming.html

You must specify a data-loading type of TRUNCATE in the control file when using the FREEZE option. FREEZE is not supported for direct loading.

By default, FREEZE is FALSE.

PARALLEL= { FALSE | TRUE }

Set PARALLEL to TRUE to indicate that this EDB*Loader session is one of a number of concurrent EDB*Loader sessions participating in a parallel direct path load. The default value of PARALLEL is FALSE.

When PARALLEL is TRUE, the DIRECT parameter must also be set to TRUE . See Section 2.5.1 for more information about parallel direct path loads.

ROWS=*n*

n specifies the number of rows that EDB*Loader will commit before loading the next set of *n* rows.

If EDB*Loader encounters an invalid row during a load (in which the ROWS parameter is specified), those rows committed prior to encountering the error will remain in the destination table.

SKIP=*skip_count*

skip_count specifies the number of records at the beginning of the input data file that should be skipped before loading begins. The default is 0.

SKIP_INDEX_MAINTENANCE={ FALSE | TRUE }

If SKIP_INDEX_MAINTENANCE is TRUE, index maintenance is not performed as part of a direct path load, and indexes on the loaded table are marked as invalid. The default value of SKIP_INDEX_MAINTENANCE is FALSE.

Please note: During a parallel direct path load, target table indexes are not updated, and are marked as invalid after the load is complete.

You can use the REINDEX command to rebuild an index. For more information about the REINDEX command, see the PostgreSQL core documentation available at:

https://www.postgresql.org/docs/10/static/sql-reindex.html

charset

Use the CHARACTERSET clause to identify the character set encoding of *data_file* where *charset* is the character set name. This clause is required if the data file encoding differs from the control file encoding. (The control file encoding must always be in the encoding of the client where edbldr is invoked.)

Examples of *charset* settings are UTF8, SQL_ASCII, and SJIS.

For more information about client to database character set conversion, see the PostgreSQL core documentation available at:

https://www.postgresql.org/docs/10/static/multibyte.html

data_file

File containing the data to be loaded into *target_table*. Each record in the data file corresponds to a row to be inserted into *target_table*.

If an extension is not provided in the file name, EDB*Loader assumes the file has an extension of .dat, for example, mydatafile.dat.

Note: If the DATA parameter is specified on the command line when edbldr is invoked, the file given by the command line DATA parameter is used instead.

If the INFILE clause is omitted as well as the command line DATA parameter, then the data file name is assumed to be identical to the control file name, but with an extension of .dat.

stdin

Specify stdin (all lowercase letters) if you want to use standard input to pipe the data to be loaded directly to EDB*Loader. This is useful for data sources generating a large number of records to be loaded.

bad_file

File that receives *data_file* records that cannot be loaded due to errors.

If an extension is not provided in the file name, EDB*Loader assumes the file has an extension of .bad, for example, mybadfile.bad.

Note: If the BAD parameter is specified on the command line when edbldr is invoked, the file given by the command line BAD parameter is used instead.

discard_file

File that receives input data records that are not loaded into any table because none of the selection criteria are met for tables with the WHEN clause, and there are no tables without a WHEN clause. (All records meet the selection criteria of a table without a WHEN clause.)

If an extension is not provided in the file name, EDB*Loader assumes the file has an extension of .dsc, for example, mydiscardfile.dsc.

Note: If the DISCARD parameter is specified on the command line when edbldr is invoked, the file given by the command line DISCARD parameter is used instead.

{ DISCARDMAX | DISCARDS } *max_discard_recs*

Maximum number of discarded records that may be encountered from the input data records before terminating the EDB*Loader session. (A discarded record is described in the preceding description of the *discard_file* parameter.) Either keyword DISCARDMAX or DISCARDS may be used preceding the integer value specified by *max_discard_recs*.

For example, if *max_discard_recs* is 0, then the EDB*Loader session is terminated if and when a first discarded record is encountered. If *max_discard_recs* is 1, then the EDB*Loader session is terminated if and when a second discarded record is encountered.

When the EDB*Loader session is terminated due to exceeding *max_discard_recs*, prior input data records that have been loaded into the database are retained. They are not rolled back.

INSERT | APPEND | REPLACE | TRUNCATE

Specifies how data is to be loaded into the target tables. If one of INSERT, APPEND, REPLACE, or TRUNCATE is specified, it establishes the default action for all tables, overriding the default of INSERT.

INSERT

Data is to be loaded into an empty table. EDB*Loader throws an exception and does not load any data if the table is not initially empty.

Note: If the table contains rows, the TRUNCATE command must be used to empty the table prior to invoking EDB*Loader. EDB*Loader throws an exception if the DELETE command is used to empty the table instead of

the TRUNCATE command. Oracle SQL*Loader allows the table to be emptied by using either the DELETE or TRUNCATE command.

APPEND

Data is to be added to any existing rows in the table. The table may be initially empty as well.

REPLACE

The REPLACE keyword and TRUNCATE keywords are functionally identical. The table is truncated by EDB*Loader prior to loading the new data.

Note: Delete triggers on the table are not fired as a result of the REPLACE operation.

TRUNCATE

The table is truncated by EDB*Loader prior to loading the new data. Delete triggers on the table are not fired as a result of the TRUNCATE operation.

PRESERVE BLANKS

For all target tables, retains leading white space when the optional enclosure delimiters are not present and leaves trailing white space intact when fields are specified with a predetermined size. When omitted, the default behavior is to trim leading and trailing white space.

target_table

Name of the table into which data is to be loaded. The table name may be schema-qualified (for example, enterprisedb.emp). The specified target must not be a view.

field_condition

Conditional clause taking the following form:

```
[ ( ] { (start:end) | column_name } { = | != | <> }
  'val' [ ) ]
```


This conditional clause is used for the WHEN clause, which is part of the INTO TABLE *target_table* clause, and the NULLIF clause, which is part of the field definition denoted as *field_def* in the syntax diagram.

start and *end* are positive integers specifying the column positions in *data_file* that mark the beginning and end of a field that is to be compared with the constant *val*. The first character in each record begins with a *start* value of 1.

column_name specifies the name assigned to a field definition of the data file as defined by *field_def* in the syntax diagram.

Use of either (*start*:*end*) or *column_name* defines the portion of the record in *data_file* that is to be compared with the value specified by '*val*' to evaluate as either true or false.

All characters used in the *field_condition* text (particularly in the *val* string) must be valid in the database encoding. (For performing data conversion, EDB*Loader first converts the characters in *val* string to the database encoding and then to the data file encoding.)

In the WHEN *field_condition* [AND *field_condition*] clause, if all such conditions evaluate to TRUE for a given record, then EDB*Loader attempts to insert that record into *target_table*. If the insert operation fails, the record is written to *bad_file*.

If for a given record, none of the WHEN clauses evaluate to TRUE for all INTO TABLE clauses, the record is written to *discard_file*, if a discard file was specified for the EDB*Loader session.

See the description of the NULLIF clause in this Parameters list for the effect of *field_condition* on this clause.

termstring

String of one or more characters that separates each field in *data_file*. The characters may be single-byte or multi-byte as long as they are valid in the database encoding. Two consecutive appearances of *termstring* with no intervening character results in the corresponding column set to null.

enclstring

String of one or more characters used to enclose a field value in *data_file*. The characters may be single-byte or multi-byte as long as they are valid in the

database encoding. Use *enclstring* on fields where *termstring* appears as part of the data.

delimstring

String of one or more characters that separates each record in *data_file*. The characters may be single-byte or multi-byte as long as they are valid in the database encoding. Two consecutive appearances of *delimstring* with no intervening character results in no corresponding row loaded into the table. The last record (in other words, the end of the data file) must also be terminated by the *delimstring* characters, otherwise the final record is not loaded into the table.

Note: The RECORDS DELIMITED BY '*delimstring*' clause is not compatible with Oracle databases.

TRAILING NULLCOLS

If TRAILING NULLCOLS is specified, then the columns in the column list for which there is no data in *data_file* for a given record, are set to null when the row is inserted. This applies only to one or more consecutive columns at the end of the column list.

If fields are omitted at the end of a record and TRAILING NULLCOLS is not specified, EDB*Loader assumes the record contains formatting errors and writes it to the bad file.

column_name

Name of a column in *target_table* into which a field value defined by *field_def* is to be inserted. If the field definition includes the FILLER or BOUNDFILLER clause, then *column_name* is not required to be the name of a column in the table. It can be any identifier name since the FILLER and BOUNDFILLER clauses prevent the loading of the field data into a table column.

CONSTANT *val*

Specifies a constant that is type-compatible with the column data type to which it is assigned in a field definition. Single or double quotes may enclose *val*. If *val* contains white space, then enclosing quotation marks must be used.

The use of the CONSTANT clause completely determines the value to be assigned to a column in each inserted row. No other clause may appear in the same field definition.

If the TERMINATED BY clause is used to delimit the fields in *data_file*, there must be no delimited field in *data_file* corresponding to any field definition with a CONSTANT clause. In other words, EDB*Loader assumes there is no field in *data_file* for any field definition with a CONSTANT clause.

FILLER

Specifies that the data in the field defined by the field definition is not to be loaded into the associated column if the identifier of the field definition is an actual column name in the table. In such case, the column is set to null. Use of the FILLER or BOUNDFILLER clause is the only circumstance in which the field definition does not have to be identified by an actual column name.

Unlike the BOUNDFILLER clause, an identifier defined with the FILLER clause must not be referenced in a SQL expression. See the discussion of the *expr* parameter.

BOUNDFILLER

Specifies that the data in the field defined by the field definition is not to be loaded into the associated column if the identifier of the field definition is an actual column name in the table. In such case, the column is set to null. Use of the FILLER or BOUNDFILLER clause is the only circumstance in which the field definition does not have to be identified by an actual column name.

Unlike the FILLER clause, an identifier defined with the BOUNDFILLER clause may be referenced in a SQL expression. See the discussion of the *expr* parameter.

POSITION (*start*:*end*)

Defines the location of the field in a record in a fixed-width field data file. *start* and *end* are positive integers. The first character in the record has a start value of 1.

```
CHAR [(length)] | DATE [(length)] [ "datemask" ] |
INTEGER EXTERNAL [(length)] |
FLOAT EXTERNAL [(length)] | DECIMAL EXTERNAL [(length)] |
ZONED EXTERNAL [(length)] | ZONED [(precision[,scale])]
```

Field type that describes the format of the data field in *data_file*.

Note: Specification of a field type is optional (for descriptive purposes only) and has no effect on whether or not EDB*Loader successfully inserts the data in the

field into the table column. Successful loading depends upon the compatibility of the column data type and the field value. For example, a column with data type NUMBER(7,2) successfully accepts a field containing 2600, but if the field contains a value such as 26XX, the insertion fails and the record is written to *bad_file*.

Please note that ZONED data is not human-readable; ZONED data is stored in an internal format where each digit is encoded in a separate nibble/nybble/4-bit field. In each ZONED value, the last byte contains a single digit (in the high-order 4 bits) and the sign (in the low-order 4 bits).

length

Specifies the length of the value to be loaded into the associated column.

If the POSITION (*start*:*end*) clause is specified along with a *fieldtype*(*length*) clause, then the ending position of the field is overridden by the specified *length* value. That is, the length of the value to be loaded into the column is determined by the *length* value beginning at the *start* position, and not by the *end* position of the POSITION (*start*:*end*) clause. Thus, the value to be loaded into the column may be shorter than the field defined by POSITION (*start*:*end*), or it may go beyond the *end* position depending upon the specified *length* size.

If the FIELDS TERMINATED BY '*termstring*' clause is specified as part of the INTO TABLE clause, and a field definition contains the *fieldtype*(*length*) clause, then a record is accepted as long as the specified *length* values are greater than or equal to the field lengths as determined by the *termstring* characters enclosing all such fields of the record. If the specified *length* value is less than a field length as determined by the enclosing *termstring* characters for any such field, then the record is rejected.

If the FIELDS TERMINATED BY '*termstring*' clause is not specified, and the POSITION (*start*:*end*) clause is not included with a field containing the *fieldtype*(*length*) clause, then the starting position of this field begins with the next character following the ending position of the preceding field. The ending position of the preceding field is either the end of its *length* value if the preceding field contains the *fieldtype*(*length*) clause, or by its *end* parameter if the field contains the POSITION (*start*:*end*) clause without the *fieldtype*(*length*) clause.

precision

Use *precision* to specify the length of the ZONED value.

If the *precision* value specified for ZONED conflicts with the length calculated by the server based on information provided with the POSITION clause, EDB*Loader will use the value specified for *precision*.

scale

scale specifies the number of digits to the right of the decimal point in a ZONED value.

datemask

Specifies the ordering and abbreviation of the day, month, and year components of a date field.

Note: If the DATE field type is specified along with a SQL expression for the column, then *datemask* must be specified after DATE and before the SQL expression. See the following discussion of the *expr* parameter.

NULLIF *field_condition* [AND *field_condition*] ...

Note: See the description of *field_condition* previously listed in this Parameters section for the syntax of *field_condition*.

If all field conditions evaluate to TRUE, then the column identified by *column_name* in the field definition is set to null. If any field condition evaluates to FALSE, then the column is set to the appropriate value as would normally occur according to the field definition.

PRESERVE BLANKS

For the column on which this option appears, retains leading white space when the optional enclosure delimiters are not present and leaves trailing white space intact when fields are specified with a predetermined size. When omitted, the default behavior is to trim leading and trailing white space.

expr

A SQL expression returning a scalar value that is type-compatible with the column data type to which it is assigned in a field definition. Double quotes must enclose *expr*. *expr* may contain a reference to any column in the field list (except for fields with the FILLER clause) by prefixing the column name by a colon character (:).

expr may also consist of a SQL SELECT statement. If a SELECT statement is used then the following rules must apply: 1) The SELECT statement must be

enclosed within parentheses `(SELECT ...)`. 2) The select list must consist of exactly one expression following the `SELECT` keyword. 3) The result set must not return more than one row. If no rows are returned, then the returned value of the resulting expression is null. The following is the syntax for use of the `SELECT` statement:

```
"(SELECT expr [ FROM table_list [ WHERE condition ] ])"
```

Note: Omitting the `FROM table_list` clause is not compatible with Oracle databases. If no tables need to be specified, use of the `FROM DUAL` clause is compatible with Oracle databases.

Examples

The following are some examples of control files and their corresponding data files.

Delimiter-Separated Field Data File

The following control file uses a delimiter-separated data file that appends rows to the `emp` table:

```
LOAD DATA
  INFILE    'emp.dat'
    BADFILE 'emp.bad'
  APPEND
  INTO TABLE emp
    FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"'
    TRAILING NULLCOLS
  (
    empno,
    ename,
    job,
    mgr,
    hiredate,
    sal,
    deptno,
    comm
  )
```

In the preceding control file, the `APPEND` clause is used to allow the insertion of additional rows into the `emp` table.

The following is the corresponding delimiter-separated data file:

```
9101,ROGERS,CLERK,7902,17-DEC-10,1980.00,20
9102,PETERSON,SALESMAN,7698,20-DEC-10,2600.00,30,2300.00
9103,WARREN,SALESMAN,7698,22-DEC-10,5250.00,30,2500.00
9104,"JONES, JR.",MANAGER,7839,02-APR-09,7975.00,20
```

The use of the `TRAILING NULLCOLS` clause allows the last field supplying the `comm` column to be omitted from the first and last records. The `comm` column is set to null for the rows inserted from these records.

The double quotation mark enclosure character surrounds the value `JONES, JR.` in the last record since the comma delimiter character is part of the field value.

The following query displays the rows added to the table after the EDB*Loader session:

```
SELECT * FROM emp WHERE empno > 9100;

 empno |   ename    |   job    | mgr  |      hiredate       |   sal   |  comm   | deptno
-------+------------+----------+------+---------------------+---------+---------+-------
-
  9101 | ROGERS     | CLERK    | 7902 | 17-DEC-10 00:00:00 | 1980.00 |         |     20
  9102 | PETERSON   | SALESMAN | 7698 | 20-DEC-10 00:00:00 | 2600.00 | 2300.00 |     30
  9103 | WARREN     | SALESMAN | 7698 | 22-DEC-10 00:00:00 | 5250.00 | 2500.00 |     30
  9104 | JONES, JR. | MANAGER  | 7839 | 02-APR-09 00:00:00 | 7975.00 |         |     20
(4 rows)
```

Fixed-Width Field Data File

The following example is a control file that loads the same rows into the `emp` table, but uses a data file containing fixed-width fields:

```
LOAD DATA
  INFILE        'emp_fixed.dat'
    BADFILE     'emp_fixed.bad'
  APPEND
  INTO TABLE emp
    TRAILING NULLCOLS
  (
    empno       POSITION (1:4),
    ename       POSITION (5:14),
    job         POSITION (15:23),
    mgr         POSITION (24:27),
    hiredate    POSITION (28:38),
    sal         POSITION (39:46),
    deptno      POSITION (47:48),
    comm        POSITION (49:56)
  )
```

In the preceding control file, the `FIELDS TERMINATED BY` and `OPTIONALLY ENCLOSED BY` clauses are absent. Instead, each field now includes the `POSITION` clause.

The following is the corresponding data file containing fixed-width fields:

```
9101ROGERS    CLERK    790217-DEC-10   1980.0020
9102PETERSON  SALESMAN 769820-DEC-10   2600.0030 2300.00
9103WARREN    SALESMAN 769822-DEC-10   5250.0030 2500.00
9104JONES, JR.MANAGER  783902-APR-09   7975.0020
```

Single Physical Record Data File – RECORDS DELIMITED BY Clause

The following example is a control file that loads the same rows into the `emp` table, but uses a data file with one physical record. Each individual record that is to be loaded as a row in the table is terminated by the semicolon character (`;`) specified by the `RECORDS DELIMITED BY` clause.

```
LOAD DATA
  INFILE    'emp_recdelim.dat'
    BADFILE 'emp_recdelim.bad'
  APPEND
  INTO TABLE emp
    FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"'
    RECORDS DELIMITED BY ';'
    TRAILING NULLCOLS
  (
    empno,
    ename,
    job,
    mgr,
    hiredate,
    sal,
    deptno,
    comm
  )
```

The following is the corresponding data file. The content is a single, physical record in the data file. The record delimiter character is included following the last record (that is, at the end of the file).

```
9101,ROGERS,CLERK,7902,17-DEC-10,1980.00,20,;9102,PETERSON,SALESMAN,7698,20-
DEC-10,2600.00,30,2300.00;9103,WARREN,SALESMAN,7698,22-DEC-
10,5250.00,30,2500.00;9104,"JONES, JR.",MANAGER,7839,02-APR-09,7975.00,20,;
```

FILLER Clause

The following control file illustrates the use of the `FILLER` clause in the data fields for the `sal` and `comm` columns. EDB*Loader ignores the values in these fields and sets the corresponding columns to null.

```
LOAD DATA
  INFILE        'emp_fixed.dat'
    BADFILE     'emp_fixed.bad'
  APPEND
  INTO TABLE emp
    TRAILING NULLCOLS
  (
    empno       POSITION (1:4),
    ename       POSITION (5:14),
    job         POSITION (15:23),
    mgr         POSITION (24:27),
    hiredate    POSITION (28:38),
    sal         FILLER POSITION (39:46),
    deptno      POSITION (47:48),
```

```
    comm        FILLER POSITION (49:56)
  )
```

Using the same fixed-width data file as in the prior fixed-width field example, the resulting rows in the table appear as follows:

```
SELECT * FROM emp WHERE empno > 9100;

 empno |      ename       |   job    | mgr  |      hiredate       | sal | comm | deptno
-------+------------------+----------+------+---------------------+-----+------+--------
  9101 | ROGERS           | CLERK    | 7902 | 17-DEC-10 00:00:00 |     |      |     20
  9102 | PETERSON         | SALESMAN | 7698 | 20-DEC-10 00:00:00 |     |      |     30
  9103 | WARREN           | SALESMAN | 7698 | 22-DEC-10 00:00:00 |     |      |     30
  9104 | JONES, JR.       | MANAGER  | 7839 | 02-APR-09 00:00:00 |     |      |     20
(4 rows)
```

BOUNDFILLER Clause

The following control file illustrates the use of the `BOUNDFILLER` clause in the data fields for the `job` and `mgr` columns. EDB*Loader ignores the values in these fields and sets the corresponding columns to null in the same manner as the `FILLER` clause. However, unlike columns with the `FILLER` clause, columns with the `BOUNDFILLER` clause are permitted to be used in an expression as shown for column `jobdesc`.

```
LOAD DATA
  INFILE    'emp.dat'
    BADFILE 'emp.bad'
  APPEND
  INTO TABLE empjob
    FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"'
    TRAILING NULLCOLS
  (
    empno,
    ename,
    job         BOUNDFILLER,
    mgr         BOUNDFILLER,
    hiredate    FILLER,
    sal         FILLER,
    deptno      FILLER,
    comm        FILLER,
    jobdesc     ":job || ' for manager ' || :mgr"
  )
```

The following is the delimiter-separated data file used in this example.

```
9101,ROGERS,CLERK,7902,17-DEC-10,1980.00,20
9102,PETERSON,SALESMAN,7698,20-DEC-10,2600.00,30,2300.00
9103,WARREN,SALESMAN,7698,22-DEC-10,5250.00,30,2500.00
9104,"JONES, JR.",MANAGER,7839,02-APR-09,7975.00,20
```

The following table is loaded using the preceding control file and data file.

```
CREATE TABLE empjob (
    empno           NUMBER(4) NOT NULL CONSTRAINT empjob_pk PRIMARY KEY,
    ename           VARCHAR2(10),
```

```
    job             VARCHAR2(9),
    mgr             NUMBER(4),
    jobdesc         VARCHAR2(25)
);
```

The resulting rows in the table appear as follows:

```
SELECT * FROM empjob;

 empno |   ename    | job | mgr |          jobdesc
-------+------------+-----+-----+---------------------------
  9101 | ROGERS     |     |     | CLERK for manager 7902
  9102 | PETERSON   |     |     | SALESMAN for manager 7698
  9103 | WARREN     |     |     | SALESMAN for manager 7698
  9104 | JONES, JR. |     |     | MANAGER for manager 7839
(4 rows)
```

Field Types with Length Specification

The following example is a control file that contains the field type clauses with the length specification:

```
LOAD DATA
  INFILE        'emp_fixed.dat'
    BADFILE     'emp_fixed.bad'
  APPEND
  INTO TABLE emp
    TRAILING NULLCOLS
  (
    empno       CHAR(4),
    ename       CHAR(10),
    job         POSITION (15:23) CHAR(9),
    mgr         INTEGER EXTERNAL(4),
    hiredate    DATE(11) "DD-MON-YY",
    sal         DECIMAL EXTERNAL(8),
    deptno      POSITION (47:48),
    comm        POSITION (49:56) DECIMAL EXTERNAL(8)
  )
```

Note that the `POSITION` clause and the *`fieldtype`*`(`*`length`*`)` clause can be used individually or in combination as long as each field definition contains at least one of the two clauses.

The following is the corresponding data file containing fixed-width fields:

```
9101ROGERS    CLERK    790217-DEC-10   1980.0020
9102PETERSON  SALESMAN 769820-DEC-10   2600.0030 2300.00
9103WARREN    SALESMAN 769822-DEC-10   5250.0030 2500.00
9104JONES, JR.MANAGER  783902-APR-09   7975.0020
```

The resulting rows in the table appear as follows:

```
SELECT * FROM emp WHERE empno > 9100;

 empno |   ename    |   job    | mgr  |      hiredate      |   sal   |  comm   | deptno
```



```
-------+------------+----------+------+--------------------+---------+---------+-------
-
  9101 | ROGERS     | CLERK    | 7902 | 17-DEC-10 00:00:00 | 1980.00 |         |     20
  9102 | PETERSON   | SALESMAN | 7698 | 20-DEC-10 00:00:00 | 2600.00 | 2300.00 |     30
  9103 | WARREN     | SALESMAN | 7698 | 22-DEC-10 00:00:00 | 5250.00 | 2500.00 |     30
  9104 | JONES, JR. | MANAGER  | 7839 | 02-APR-09 00:00:00 | 7975.00 |         |     20
(4 rows)
```

NULLIF Clause

The following example uses the `NULLIF` clause on the `sal` column to set it to null for employees of job `MANAGER` as well as on the `comm` column to set it to null if the employee is not a `SALESMAN` and is not in department `30`. In other words, a `comm` value is accepted if the employee is a `SALESMAN` or is a member of department `30`.

The following is the control file:

```
LOAD DATA
  INFILE        'emp_fixed_2.dat'
    BADFILE     'emp_fixed_2.bad'
  APPEND
  INTO TABLE emp
    TRAILING NULLCOLS
  (
    empno       POSITION (1:4),
    ename       POSITION (5:14),
    job         POSITION (15:23),
    mgr         POSITION (24:27),
    hiredate    POSITION (28:38),
    sal         POSITION (39:46) NULLIF job = 'MANAGER',
    deptno      POSITION (47:48),
    comm        POSITION (49:56) NULLIF job <> 'SALESMAN' AND deptno <> '30'
  )
```

The following is the corresponding data file:

```
9101ROGERS    CLERK    790217-DEC-10   1980.0020
9102PETERSON  SALESMAN 769820-DEC-10   2600.0030 2300.00
9103WARREN    SALESMAN 769822-DEC-10   5250.0030 2500.00
9104JONES, JR.MANAGER  783902-APR-09   7975.0020
9105ARNOLDS   CLERK    778213-SEP-10   3750.0030  800.00
9106JACKSON   ANALYST  756603-JAN-11   4500.0040 2000.00
9107MAXWELL   SALESMAN 769820-DEC-10   2600.0010 1600.00
```

The resulting rows in the table appear as follows:

```
SELECT empno, ename, job, NVL(TO_CHAR(sal),'--null--') "sal",
  NVL(TO_CHAR(comm),'--null--') "comm", deptno FROM emp WHERE empno > 9100;

 empno |   ename    |   job    |   sal    |   comm   | deptno
-------+------------+----------+----------+----------+--------
  9101 | ROGERS     | CLERK    | 1980.00  | --null-- |     20
  9102 | PETERSON   | SALESMAN | 2600.00  | 2300.00  |     30
  9103 | WARREN     | SALESMAN | 5250.00  | 2500.00  |     30
  9104 | JONES, JR. | MANAGER  | --null-- | --null-- |     20
  9105 | ARNOLDS    | CLERK    | 3750.00  | 800.00   |     30
```

```
  9106 | JACKSON    | ANALYST  | 4500.00  | --null-- |     40
  9107 | MAXWELL    | SALESMAN | 2600.00  | 1600.00  |     10
(7 rows)
```

Note that the `sal` column for employee `JONES, JR.` is null since the job is `MANAGER`.

The `comm` values from the data file for employees `PETERSON`, `WARREN`, `ARNOLDS`, and `MAXWELL` are all loaded into the `comm` column of the `emp` table since these employees are either `SALESMAN` or members of department `30`.

The `comm` value of `2000.00` in the data file for employee `JACKSON` is ignored and the `comm` column of the `emp` table set to null since this employee is neither a `SALESMAN` nor is a member of department `30`.

SELECT Statement in a Field Expression

The following example uses a `SELECT` statement in the expression of the field definition to return the value to be loaded into the column.

```
LOAD DATA
  INFILE        'emp_fixed.dat'
    BADFILE     'emp_fixed.bad'
  APPEND
  INTO TABLE emp
    TRAILING NULLCOLS
  (
    empno       POSITION (1:4),
    ename       POSITION (5:14),
    job         POSITION (15:23) "(SELECT dname FROM dept WHERE deptno = :deptno)",
    mgr         POSITION (24:27),
    hiredate    POSITION (28:38),
    sal         POSITION (39:46),
    deptno      POSITION (47:48),
    comm        POSITION (49:56)
  )
```

The content of the `dept` table used in the `SELECT` statement is the following:

```
SELECT * FROM dept;

 deptno |   dname    |   loc
--------+------------+----------
     10 | ACCOUNTING | NEW YORK
     20 | RESEARCH   | DALLAS
     30 | SALES      | CHICAGO
     40 | OPERATIONS | BOSTON
(4 rows)
```

The following is the corresponding data file:

```
9101ROGERS    CLERK    790217-DEC-10   1980.0020
9102PETERSON  SALESMAN 769820-DEC-10   2600.0030 2300.00
9103WARREN    SALESMAN 769822-DEC-10   5250.0030 2500.00
9104JONES, JR.MANAGER  783902-APR-09   7975.0020
```


The resulting rows in the table appear as follows:

```
SELECT * FROM emp WHERE empno > 9100;

 empno |   ename    |   job    | mgr  |      hiredate       |   sal   |  comm   | deptno
-------+------------+----------+------+---------------------+---------+---------+-------
-
  9101 | ROGERS     | RESEARCH | 7902 | 17-DEC-10 00:00:00 | 1980.00 |         |     20
  9102 | PETERSON   | SALES    | 7698 | 20-DEC-10 00:00:00 | 2600.00 | 2300.00 |     30
  9103 | WARREN     | SALES    | 7698 | 22-DEC-10 00:00:00 | 5250.00 | 2500.00 |     30
  9104 | JONES, JR. | RESEARCH | 7839 | 02-APR-09 00:00:00 | 7975.00 |         |     20
(4 rows)
```

Note that the `job` column contains the value from the `dname` column of the `dept` table returned by the `SELECT` statement instead of the job name from the data file.

Multiple INTO TABLE Clauses

The following example illustrates the use of multiple `INTO TABLE` clauses. For this example, two empty tables are created with the same data definition as the `emp` table. The following `CREATE TABLE` commands create these two empty tables, while inserting no rows from the original `emp` table:

```
CREATE TABLE emp_research AS SELECT * FROM emp WHERE deptno = 99;
CREATE TABLE emp_sales AS SELECT * FROM emp WHERE deptno = 99;
```

The following control file contains two `INTO TABLE` clauses. Also note that there is no `APPEND` clause so the default operation of `INSERT` is used, which requires that tables `emp_research` and `emp_sales` be empty.

```
LOAD DATA
  INFILE        'emp_multitbl.dat'
    BADFILE     'emp_multitbl.bad'
    DISCARDFILE 'emp_multitbl.dsc'
  INTO TABLE emp_research
    WHEN (47:48) = '20'
    TRAILING NULLCOLS
  (
    empno       POSITION (1:4),
    ename       POSITION (5:14),
    job         POSITION (15:23),
    mgr         POSITION (24:27),
    hiredate    POSITION (28:38),
    sal         POSITION (39:46),
    deptno      CONSTANT '20',
    comm        POSITION (49:56)
  )
  INTO TABLE emp_sales
    WHEN (47:48) = '30'
    TRAILING NULLCOLS
  (
    empno       POSITION (1:4),
    ename       POSITION (5:14),
    job         POSITION (15:23),
    mgr         POSITION (24:27),
    hiredate    POSITION (28:38),
```

```
    sal         POSITION (39:46),
    deptno      CONSTANT '30',
    comm        POSITION (49:56) "ROUND(:comm + (:sal * .25), 0)"
  )
```

The `WHEN` clauses specify that when the field designated by columns 47 thru 48 contains `20`, the record is inserted into the `emp_research` table and when that same field contains `30`, the record is inserted into the `emp_sales` table. If neither condition is true, the record is written to the discard file named `emp_multitbl.dsc`.

The `CONSTANT` clause is given for column `deptno` so the specified constant value is inserted into `deptno` for each record. When the `CONSTANT` clause is used, it must be the only clause in the field definition other than the column name to which the constant value is assigned.

Finally, column `comm` of the `emp_sales` table is assigned a SQL expression. Column names may be referenced in the expression by prefixing the column name with a colon character (:).

The following is the corresponding data file:

```
9101ROGERS    CLERK    790217-DEC-10   1980.0020
9102PETERSON  SALESMAN 769820-DEC-10   2600.0030 2300.00
9103WARREN    SALESMAN 769822-DEC-10   5250.0030 2500.00
9104JONES, JR.MANAGER  783902-APR-09   7975.0020
9105ARNOLDS   CLERK    778213-SEP-10   3750.0010
9106JACKSON   ANALYST  756603-JAN-11   4500.0040
```

Since the records for employees `ARNOLDS` and `JACKSON` contain `10` and `40` in columns 47 thru 48, which do not satisfy any of the `WHEN` clauses, EDB*Loader writes these two records to the discard file, `emp_multitbl.dsc`, whose content is shown by the following:

```
9105ARNOLDS   CLERK    778213-SEP-10   3750.0010
9106JACKSON   ANALYST  756603-JAN-11   4500.0040
```

The following are the rows loaded into the `emp_research` and `emp_sales` tables:

```
SELECT * FROM emp research;

empno |   ename    |   job   | mgr  |      hiredate       |   sal   | comm | deptno
-------+------------+---------+------+---------------------+---------+------+--------
  9101 | ROGERS     | CLERK   | 7902 | 17-DEC-10 00:00:00 | 1980.00 |      |  20.00
  9104 | JONES, JR. | MANAGER | 7839 | 02-APR-09 00:00:00 | 7975.00 |      |  20.00
(2 rows)

SELECT * FROM emp sales;

empno |  ename   |   job    | mgr  |      hiredate       |   sal   |  comm   | deptno
-------+----------+----------+------+---------------------+---------+---------+--------
  9102 | PETERSON | SALESMAN | 7698 | 20-DEC-10 00:00:00 | 2600.00 | 2950.00 |  30.00
  9103 | WARREN   | SALESMAN | 7698 | 22-DEC-10 00:00:00 | 5250.00 | 3813.00 |  30.00
(2 rows)
```

2.4 Invoking EDB*Loader

You must have superuser privileges to run EDB*Loader. Use the following command to invoke EDB*Loader from the command line:

```
edbldr [ -d dbname ] [ -p port ] [ -h host ]
[ USERID={ username/password | username/ | username | / } ]
  CONTROL=control_file
[ DATA=data_file ]
[ BAD=bad_file ]
[ DISCARD=discard_file ]
[ DISCARDMAX=max_discard_recs ]
[ LOG=log_file ]
[ PARFILE=param_file ]
[ DIRECT={ FALSE | TRUE } ]
[ FREEZE={ FALSE | TRUE } ]
[ ERRORS=error_count ]
[ PARALLEL={ FALSE | TRUE } ]
[ ROWS=n ]
[ SKIP=skip_count ]
[ SKIP_INDEX_MAINTENANCE={ FALSE | TRUE } ]
[ edb_resource_group=group_name ]
```

Description

If the `-d` option, the `-p` option, or the `-h` option are omitted, the defaults for the database, port, and host are determined according to the same rules as other Advanced Server utility programs such as `edb-psql`, for example.

Any parameter listed in the preceding syntax diagram except for the `-d` option, `-p` option, `-h` option, and the `PARFILE` parameter may be specified in a *parameter file*. The parameter file is specified on the command line when `edbldr` is invoked using `PARFILE=param_file`. Some parameters may be specified in the `OPTIONS` clause in the control file. See the description of the control file in Section 2.3.

The specification of *control_file*, *data_file*, *bad_file*, *discard_file*, *log_file*, and *param_file* may include the full directory path or a relative directory path to the file name. If the file name is specified alone or with a relative directory path, the file is assumed to exist (in the case of *control_file*, *data_file*, or *param_file*), or to be created (in the case of *bad_file*, *discard_file*, or *log_file*) relative to the current working directory from which `edbldr` is invoked.

Note: The control file must exist in the character set encoding of the client where `edbldr` is invoked. If the client is in a different encoding than the database encoding, then the `PGCLIENTENCODING` environment variable must be set on the client to the

client's encoding prior to invoking edbldr. This must be done to ensure character set conversion is properly done between the client and the database server.

The operating system account used to invoke edbldr must have read permission on the directories and files specified by *control_file*, *data_file*, and *param_file*.

The operating system account enterprisedb must have write permission on the directories where *bad_file*, *discard_file*, and *log_file* are to be written.

Note: It is suggested that the file names for *control_file*, *data_file*, *bad_file*, *discard_file*, and *log_file* include extensions of .ctl, .dat, .bad, .dsc, and .log, respectively. If the provided file name does not contain an extension, EDB*Loader assumes the actual file name includes the appropriate aforementioned extension.

Parameters

dbname

Name of the database containing the tables to be loaded.

port

Port number on which the database server is accepting connections.

host

IP address of the host on which the database server is running.

USERID={ *username*/*password* | *username*/ | *username* | / }

EDB*Loader connects to the database with *username*. *username* must be a superuser. *password* is the password for *username*.

If the USERID parameter is omitted, EDB*Loader prompts for *username* and *password*. If USERID=*username*/ is specified, then EDB*Loader 1) uses the password file specified by environment variable PGPASSFILE if PGPASSFILE is set, or 2) uses the .pgpass password file (pgpass.conf on Windows systems) if PGPASSFILE is not set. If USERID=*username* is specified, then EDB*Loader prompts for *password*. If USERID=/ is specified, the connection is attempted using the operating system account as the user name.

Note: The Advanced Server connection environment variables PGUSER and PGPASSWORD are ignored by EDB*Loader. See the PostgreSQL core documentation for information on the PGPASSFILE environment variable and the password file.

CONTROL=*control_file*

control_file specifies the name of the control file containing EDB*Loader directives. If a file extension is not specified, an extension of .ctl is assumed. See Section 2.3 for a description of the control file.

DATA=*data_file*

data_file specifies the name of the file containing the data to be loaded into the target table. If a file extension is not specified, an extension of .dat is assumed. See Section 2.3 for a description of the *data_file*.

Note: Specifying a *data_file* on the command line overrides the INFILE clause specified in the control file.

BAD=*bad_file*

bad_file specifies the name of a file that receives input data records that cannot be loaded due to errors. See Section 2.3 for a description of the *bad_file*.

Note: Specifying a *bad_file* on the command line overrides any BADFILE clause specified in the control file.

DISCARD=*discard_file*

discard_file is the name of the file that receives input data records that do not meet any table's selection criteria. See the description of *discard_file* in Section 2.3.

Note: Specifying a *discard_file* using the command line DISCARD parameter overrides the DISCARDFILE clause in the control file.

DISCARDMAX=*max_discard_recs*

max_discard_recs is the maximum number of discarded records that may be encountered from the input data records before terminating the EDB*Loader session. See the description of *max_discard_recs* in Section2.3.

Note: Specifying *max_discard_recs* using the command line DISCARDMAX parameter overrides the DISCARDMAX or DISCARDS clause in the control file.

LOG=*log_file*

log_file specifies the name of the file in which EDB*Loader records the results of the EDB*Loader session.

If the LOG parameter is omitted, EDB*Loader creates a log file with the name *control_file_base*.log in the directory from which edbldr is invoked. *control_file_base* is the base name of the control file used in the EDB*Loader session. The operating system account enterprisedb must have write permission on the directory where the log file is to be written.

PARFILE=*param_file*

param_file specifies the name of the file that contains command line parameters for the EDB*Loader session. Any command line parameter listed in this section except for the -d, -p, and -h options, and the PARFILE parameter itself, can be specified in *param_file* instead of on the command line.

Any parameter given in *param_file* overrides the same parameter supplied on the command line before the PARFILE option. Any parameter given on the command line that appears after the PARFILE option overrides the same parameter given in *param_file*.

Note: Unlike other EDB*Loader files, there is no default file name or extension assumed for *param_file*, though by Oracle SQL*Loader convention, .par is typically used, but not required, as an extension.

DIRECT= { FALSE | TRUE }

If DIRECT is set to TRUE EDB*Loader performs a direct path load instead of a conventional path load. The default value of DIRECT is FALSE.

See Section 2.5 for information on direct path loads.

FREEZE= { FALSE | TRUE }

Set FREEZE to TRUE to indicate that the data should be copied with the rows *frozen*. A tuple guaranteed to be visible to all current and future transactions is marked as frozen to prevent transaction ID wrap-around. For more information about frozen tuples, see the PostgreSQL core documentation at:

https://www.postgresql.org/docs/10/static/routine-vacuuming.html

You must specify a data-loading type of TRUNCATE in the control file when using the FREEZE option. FREEZE is not supported for direct loading.

By default, FREEZE is FALSE.

ERRORS=*error_count*

error_count specifies the number of errors permitted before aborting the EDB*Loader session. The default is 50.

PARALLEL= { FALSE | TRUE }

Set PARALLEL to TRUE to indicate that this EDB*Loader session is one of a number of concurrent EDB*Loader sessions participating in a parallel direct path load. The default value of PARALLEL is FALSE.

When PARALLEL is TRUE, the DIRECT parameter must also be set to TRUE . See Section 2.5.1 for more information about parallel direct path loads.

ROWS=*n*

n specifies the number of rows that EDB*Loader will commit before loading the next set of *n* rows.

SKIP=*skip_count*

Number of records at the beginning of the input data file that should be skipped before loading begins. The default is 0.

SKIP_INDEX_MAINTENANCE= { FALSE | TRUE }

If set to TRUE, index maintenance is not performed as part of a direct path load, and indexes on the loaded table are marked as invalid. The default value of SKIP_INDEX_MAINTENANCE is FALSE.

Please note: During a parallel direct path load, target table indexes are not updated, and are marked as invalid after the load is complete.

You can use the REINDEX command to rebuild an index. For more information about the REINDEX command, see the PostgreSQL core documentation available at:

https://www.postgresql.org/docs/10/static/sql-reindex.html

edb_resource_group=*group_name*

group_name specifies the name of an EDB Resource Manager resource group to which the EDB*Loader session is to be assigned.

Any default resource group that may have been assigned to the session (for example, a database user running the EDB*Loader session who had been assigned a default resource group with the `ALTER ROLE ... SET edb_resource_group` command) is overridden by the resource group given by the `edb_resource_group` parameter specified on the `edbldr` command line.

Examples

In the following example EDB*Loader is invoked using a control file named `emp.ctl` located in the current working directory to load a table in database `edb`:

```
$ /opt/edb/as10/bin/edbldr -d edb USERID=enterprisedb/password
CONTROL=emp.ctl
EDB*Loader: Copyright (c) 2007-2017, EnterpriseDB Corporation.

Successfully loaded (4) records
```

In the following example, EDB*Loader prompts for the user name and password since they are omitted from the command line. In addition, the files for the bad file and log file are specified with the `BAD` and `LOG` command line parameters.

```
$ /opt/edb/as10/bin/edbldr -d edb CONTROL=emp.ctl BAD=/tmp/emp.bad
LOG=/tmp/emp.log
Enter the user name : enterprisedb
Enter the password :
EDB*Loader: Copyright (c) 2007-2017, EnterpriseDB Corporation.

Successfully loaded (4) records
```

The following example runs EDB*Loader with the same parameters as shown in the preceding example, but using a parameter file located in the current working directory. The `SKIP` and `ERRORS` parameters are altered from their defaults in the parameter file as well. The parameter file, `emp.par`, contains the following:

```
CONTROL=emp.ctl
BAD=/tmp/emp.bad
LOG=/tmp/emp.log
SKIP=1
ERRORS=10
```

EDB*Loader is invoked with the parameter file as shown by the following:

```
$ /opt/edb/as10/bin/edbldr -d edb PARFILE=emp.par
Enter the user name : enterprisedb
Enter the password :
EDB*Loader: Copyright (c) 2007-2017, EnterpriseDB Corporation.

Successfully loaded (3) records
```

2.4.1 Exit Codes

When EDB*Loader exits, it will return one of the following codes:

Exit Code	Description
0	Indicates that all rows loaded successfully.
1	Indicates that EDB*Loader encountered command line or syntax errors, or aborted the load operation due to an unrecoverable error.
2	Indicates that the load completed, but some (or all) rows were rejected or discarded.
3	Indicates that EDB*Loader encountered fatal errors (such as OS errors). This class of errors is equivalent to the `FATAL` or `PANIC` severity levels of PostgreSQL errors.

2.5 Direct Path Load

During a direct path load, EDB*Loader writes the data directly to the database pages, which is then synchronized to disk. The insert processing associated with a conventional path load is bypassed, thereby resulting in a performance improvement.

Bypassing insert processing reduces the types of constraints that may exist on the target table. The following types of constraints are permitted on the target table of a direct path load:

- Primary key
- Not null constraints
- Indexes (unique or non-unique)

The restrictions on the target table of a direct path load are the following:

- Triggers are not permitted
- Check constraints are not permitted
- Foreign key constraints on the target table referencing another table are not permitted
- Foreign key constraints on other tables referencing the target table are not permitted
- The table must not be partitioned
- Rules may exist on the target table, but they are not executed

Note: Currently, a direct path load in EDB*Loader is more restrictive than in Oracle SQL*Loader. The preceding restrictions do not apply to Oracle SQL*Loader in most cases. The following restrictions apply to a control file used in a direct path load:

- Multiple table loads are not supported. That is, only one `INTO TABLE` clause may be specified in the control file.
- SQL expressions may not be used in the data field definitions of the `INTO TABLE` clause.
- The `FREEZE` option is not supported for direct path loading.

To run a direct path load, add the `DIRECT=TRUE` option as shown by the following example:

```
$ /opt/edb/as10/bin/edbldr -d edb USERID=enterprisedb/password
CONTROL=emp.ctl DIRECT=TRUE
EDB*Loader: Copyright (c) 2007-2017, EnterpriseDB Corporation.

Successfully loaded (4) records
```


2.5.1 Direct Loading Limitations

During a direct load, EDB*Loader has the following limitations:

- Direct loading only supports loading data to empty tables.
- Direct loading into multiple tables simultaneously is also not supported currently.
- If you are using PITR, you must take a full backup of the database when loading is complete. During a direct load EDB*Loader bypasses any WAL file updates, making a PITR recovery after a load invalid until the full backup is updated.
- Only unique constraints are enforced during the load; other constraints (i.e. NULL or referential integrity constraints) are not enforced.

Note: These limitations do not apply while using conventional loading (i.e. `direct=false`).

2.6 Parallel Direct Path Load

The performance of a direct path load can be further improved by distributing the loading process over two or more sessions running concurrently. Each session runs a direct path load into the same table.

Since the same table is loaded from multiple sessions, the input records to be loaded into the table must be divided amongst several data files so that each EDB*Loader session uses its own data file and the same record is not loaded more than once into the table.

The target table of a parallel direct path load is under the same restrictions as a direct path load run in a single session.

The restrictions on the target table of a direct path load are the following:

- Triggers are not permitted
- Check constraints are not permitted
- Foreign key constraints on the target table referencing another table are not permitted
- Foreign key constraints on other tables referencing the target table are not permitted
- The table must not be partitioned
- Rules may exist on the target table, but they are not executed

In addition, the `APPEND` clause must be specified in the control file used by each EDB*Loader session.

To run a parallel direct path load, run EDB*Loader in a separate session for each participant of the parallel direct path load. Invocation of each such EDB*Loader session must include the `DIRECT=TRUE` and `PARALLEL=TRUE` parameters.

Each EDB*Loader session runs as an independent transaction so if one of the parallel sessions aborts and rolls back its changes, the loading done by the other parallel sessions are not affected.

Note: In a parallel direct path load, each EDB*Loader session reserves a fixed number of blocks in the target table in a round-robin fashion. Some of the blocks in the last allocated chunk may not be used, and those blocks remain uninitialized. A subsequent use of the `VACUUM` command on the target table may show warnings regarding these uninitialized blocks such as the following:

```
WARNING:  relation "emp" page 98264 is uninitialized --- fixing

WARNING:  relation "emp" page 98265 is uninitialized --- fixing

WARNING:  relation "emp" page 98266 is uninitialized --- fixing
```

This is an expected behavior and does not indicate data corruption.

Indexes on the target table are not updated during a parallel direct path load and are therefore marked as invalid after the load is complete. You must use the `REINDEX` command to rebuild the indexes.

The following example shows the use of a parallel direct path load on the `emp` table.

Note: If you attempt a parallel direct path load on the sample `emp` table provided with Advanced Server, you must first remove the triggers and constraints referencing the `emp` table. In addition the primary key column, `empno`, was expanded from `NUMBER(4)` to `NUMBER` in this example to allow for the insertion of a larger number of rows.

The following is the control file used in the first session:

```
LOAD DATA
  INFILE    '/home/user/loader/emp_parallel_1.dat'
  APPEND
  INTO TABLE emp
    FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"'
    TRAILING NULLCOLS
  (
    empno,
    ename,
    job,
    mgr,
    hiredate,
    sal,
    deptno,
    comm
```



```
  )
```

The `APPEND` clause must be specified in the control file for a parallel direct path load.

The following shows the invocation of EDB*Loader in the first session. The `DIRECT=TRUE` and `PARALLEL=TRUE` parameters must be specified.

```
$ /opt/edb/as10/bin/edbldr -d edb USERID=enterprisedb/password
CONTROL=emp_parallel_1.ctl DIRECT=TRUE PARALLEL=TRUE
WARNING:  index maintenance will be skipped with PARALLEL load
EDB*Loader: Copyright (c) 2007-2017, EnterpriseDB Corporation.
```

The control file used for the second session appears as follows. Note that it is the same as the one used in the first session, but uses a different data file.

```
LOAD DATA
  INFILE    '/home/user/loader/emp_parallel_2.dat'
  APPEND
  INTO TABLE emp
    FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"'
    TRAILING NULLCOLS
  (
    empno,
    ename,
    job,
    mgr,
    hiredate,
    sal,
    deptno,
    comm
  )
```

The preceding control file is used in a second session as shown by the following:

```
$ /opt/edb/as10/bin/edbldr -d edb USERID=enterprisedb/password
CONTROL=emp_parallel_2.ctl DIRECT=TRUE PARALLEL=TRUE
WARNING:  index maintenance will be skipped with PARALLEL load
EDB*Loader: Copyright (c) 2007-2017, EnterpriseDB Corporation.
```

EDB*Loader displays the following message in each session when its respective load operation completes:

```
Successfully loaded (10000) records
```

The following query shows that the index on the `emp` table has been marked as `INVALID`:

```
SELECT index_name, status FROM user_indexes WHERE table_name = 'EMP';

 index_name | status
------------+---------
 EMP_PK     | INVALID
(1 row)
```


Note: `user_indexes` is the view of indexes compatible with Oracle databases owned by the current user.

Queries on the `emp` table will not utilize the index unless it is rebuilt using the `REINDEX` command as shown by the following:

```
REINDEX INDEX emp_pk;
```

A subsequent query on `user_indexes` shows that the index is now marked as `VALID`:

```
SELECT index_name, status FROM user_indexes WHERE table_name = 'EMP';

 index_name | status
------------+--------
 EMP_PK     | VALID
(1 row)
```

2.7 Remote Loading

EDB*Loader supports a feature called *remote loading*. In remote loading, the database containing the table to be loaded is running on a database server on a different host than from where EDB*Loader is invoked with the input data source.

This feature is useful if you have a large amount of data to be loaded, and you do not want to create a large data file on the host running the database server.

In addition, you can use the standard input feature to pipe the data from the data source such as another program or script, directly to EDB*Loader, which then loads the table in the remote database. This bypasses the process of having to create a data file on disk for EDB*Loader.

Performing remote loading along with using standard input requires the following:

- The `edbldr` program must be installed on the client host on which it is to be invoked with the data source for the EDB*Loader session.
- The control file must contain the clause `INFILE 'stdin'` so you can pipe the data directly into EDB*Loader's standard input. See Section 2.3 for information on the `INFILE` clause and the EDB*Loader control file.
- All files used by EDB*Loader such as the control file, bad file, discard file, and log file must reside on, or are created on, the client host on which `edbldr` is invoked.
- When invoking EDB*Loader, use the `-h` option to specify the IP address of the remote database server. See Section 2.4 for information on invoking EDB*Loader.

- Use the operating system pipe operator (|) or input redirection operator (<) to supply the input data to EDB*Loader.

The following example loads a database running on a database server at `192.168.1.14` using data piped from a source named `datasource`.

```
datasource | ./edbldr -d edb -h 192.168.1.14 USERID=enterprisedb/password
CONTROL=remote.ctl
```

The following is another example of how standard input can be used:

```
./edbldr -d edb -h 192.168.1.14 USERID=enterprisedb/password
CONTROL=remote.ctl < datasource
```

2.8 Updating a Table with a Conventional Path Load

You can use EDB*Loader with a conventional path load to update the rows within a table, merging new data with the existing data. When you invoke EDB*Loader to perform an update, the server searches the table for an existing row with a matching primary key:

- If the server locates a row with a matching key, it replaces the existing row with the new row.

- If the server does not locate a row with a matching key, it adds the new row to the table.

To use EDB*Loader to update a table, the table must have a primary key. Please note that you cannot use EDB*Loader to `UPDATE` a partitioned table.

To perform an `UPDATE`, use the same steps as when performing a conventional path load:

1. Create a data file that contains the rows you wish to `UPDATE` or `INSERT`.

2. Define a control file that uses the `INFILE` keyword to specify the name of the data file. For information about building the EDB*Loader control file, see Section 2.3.

3. Invoke EDB*Loader, specifying the database name, connection information, and the name of the control file. For information about invoking EDB*Loader, see Section 2.4.

The following example uses the `emp` table that is distributed with the Advanced Server sample data. By default, the table contains:

```
edb=# select * from emp;
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)
```

The following control file (`emp_update.ctl`) specifies the fields in the table in a comma-delimited list. The control file performs an `UPDATE` on the `emp` table:

```
LOAD DATA
  INFILE 'emp_update.dat'
  BADFILE 'emp_update.bad'
  DISCARDFILE 'emp_update.dsc'
UPDATE INTO TABLE emp
FIELDS TERMINATED BY ","
(empno, ename, job, mgr, hiredate, sal, comm, deptno)
```

The data that is being updated or inserted is saved in the `emp_update.dat` file. `emp_update.dat` contains:

```
7521,WARD,MANAGER,7839,22-FEB-81 00:00:00,3000.00,0.00,30
7566,JONES,MANAGER,7839,02-APR-81 00:00:00,3500.00,0.00,20
7903,BAKER,SALESMAN,7521,10-JUN-13 00:00:00,1800.00,500.00,20
7904,MILLS,SALESMAN,7839,13-JUN-13 00:00:00,1800.00,500.00,20
7654,MARTIN,SALESMAN,7698,28-SEP-81 00:00:00,1500.00,400.00,30
```

Invoke EDB*Loader, specifying the name of the database (`edb`), the name of a database superuser (and their associated password) and the name of the control file (`emp_update.ctl`):

```
edbldr -d edb userid=user_name/password control=emp_update.ctl
```

After performing the update, the `emp` table contains:

```
edb=# select * from emp;
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  |MANAGER  | 7839 | 22-FEB-81 00:00:00 | 3000.00 |0.00   |     30
7566 |JONES |MANAGER  | 7839 | 02-APR-81 00:00:00 | 3500.00 |0.00   |     20
7654 |MARTIN|SALESMAN | 7698 | 28-SEP-81 00:00:00 | 1500.00 |400.00 |     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
7903 |BAKER |SALESMAN |7521  | 10-JUN-13 00:00:00 | 1800.00 |500.00 |     20
7904 |MILLS |SALESMAN |7839  |13-JUN-13 00:00:00  |1800.00  |500.00 |     20
7934 |MILLER|CLERK    | 7782 | 23-JAN-82 00:00:00 | 1300.00 |        |     10
(16 rows)
```

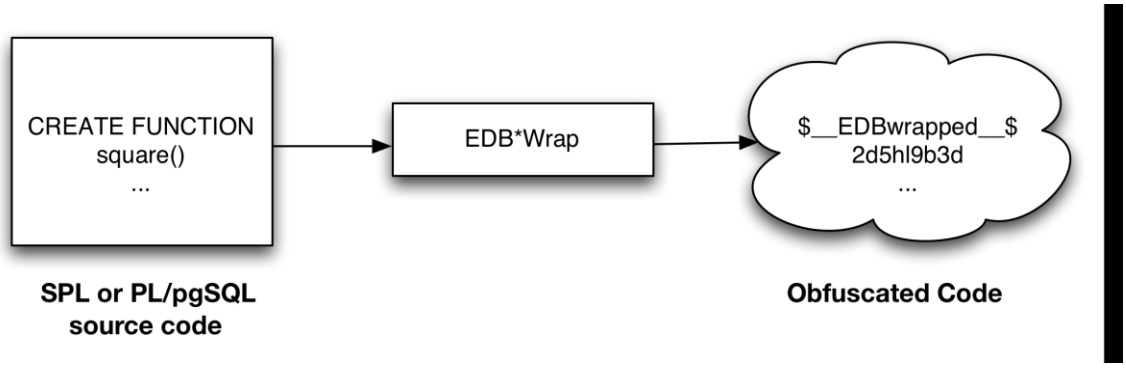
The rows containing information for the three employees that are currently in the `emp` table are updated, while rows are added for the new employees (`BAKER` and `MILLS`)

3 EDB*Wrap

The EDB*Wrap utility protects proprietary source code and programs (functions, stored procedures, triggers, and packages) from unauthorized scrutiny. The EDB*Wrap program translates a file that contains SPL or PL/pgSQL source code (the plaintext) into a file that contains the same code in a form that is nearly impossible to read. Once you have the obfuscated form of the code, you can send that code to the PostgreSQL server and the server will store those programs in obfuscated form. While EDB*Wrap does obscure code, table definitions are still exposed.

Everything you wrap is stored in obfuscated form. If you wrap an entire package, the package body source, as well as the prototypes contained in the package header and the functions and procedures contained in the package body are stored in obfuscated form.

If you wrap a `CREATE PACKAGE` statement, you hide the package API from other developers. You may want to wrap the package body, but not the package header so users can see the package prototypes and other public variables that are defined in the package body. To allow users to see what prototypes the package contains, use EDBWrap to obfuscate only the `CREATE PACKAGE BODY` statement in the edbwrap input file, omitting the `'CREATE PACKAGE'` statement. The package header source will be stored plaintext, while the package body source and package functions and procedures will be stored obfuscated.



Once wrapped, source code and programs cannot be unwrapped or debugged. Reverse engineering is possible, but would be very difficult.

The entire source file is wrapped into one unit. Any `psql` meta-commands included in the wrapped file will not be recognized when the file is executed; executing an obfuscated file that contains a psql meta-command will cause a syntax error. `edbwrap` does not validate SQL source code - if the plaintext form contains a syntax error, `edbwrap` will not complain. Instead, the server will report an error and abort the entire file when you try to execute the obfuscated form.

3.1 Using EDB*Wrap to Obfuscate Source Code

EDB*Wrap is a command line utility; it accepts a single input source file, obfuscates the contents and returns a single output file. When you invoke the `edbwrap` utility, you must provide the name of the file that contains the source code to obfuscate. You may also specify the name of the file where `edbwrap` will write the obfuscated form of the code. `edbwrap` offers three different command-line styles. The first style is compatible with Oracle's `wrap` utility:

```
edbwrap iname=input_file [oname=output_file]
```

The `iname=input_file` argument specifies the name of the input file; if `input_file` does not contain an extension, `edbwrap` will search for a file named *`input_file`*`.sql`

The `oname=`*`output_file`* argument (which is optional) specifies the name of the output file; if *`output_file`* does not contain an extension, `edbwrap` will append `.plb` to the name.

If you do not specify an output file name, `edbwrap` writes to a file whose name is derived from the input file name: `edbwrap` strips the suffix (typically `.sql`) from the input file name and adds `.plb`.

`edbwrap` offers two other command-line styles that may feel more familiar:

```
edbwrap --iname input_file [--oname output_file]
edbwrap -i input_file [-o output_file]
```

You may mix command-line styles; the rules for deriving input and output file names are identical regardless of which style you use.

Once `edbwrap` has produced a file that contains obfuscated code, you typically feed that file into the PostgreSQL server using a client application such as `edb-psql`. The server executes the obfuscated code line by line and stores the source code for SPL and PL/pgSQL programs in wrapped form.

In summary, to obfuscate code with EDB*Wrap, you:

1. Create the source code file.
2. Invoke EDB*Wrap to obfuscate the code.
3. Import the file as if it were in plaintext form.

The following sequence demonstrates `edbwrap` functionality.

First, create the source code for the `list_emp` procedure (in plaintext form):

```
[bash] cat listemp.sql
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;
/
```

You can import the `list_emp` procedure with a client application such as `edb-psql`:

```
[bash] edb-psql edb
Welcome to edb-psql 8.4.3.2, the EnterpriseDB interactive terminal.

Type:  \copyright for distribution terms
       \h for help with SQL commands
       \? for help with edb-psql commands
       \g or terminate with semicolon to execute query
       \q to quit

edb=# \i listemp.sql
CREATE PROCEDURE
```

You can view the plaintext source code (stored in the server) by examining the `pg_proc` system table:

```
edb=# SELECT prosrc FROM pg_proc WHERE proname = 'list_emp';
                            prosrc
------------------------------------------------------------------

     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
(1 row)

edb=# quit
```

Next, obfuscate the plaintext file with EDB*Wrap:

```
[bash] edbwrap -i listemp.sql
EDB*Wrap Utility: Release 8.4.3.2

Copyright (c) 2004-2013 EnterpriseDB Corporation.  All Rights Reserved.

Using encoding UTF8 for input
Processing listemp.sql to listemp.plb

Examining the contents of the output file (listemp.plb) file reveals that the
code is obfuscated:

[bash] cat listemp.plb
$__EDBwrapped__$
UTF8
d+6DL30RVaGjYMIzkuoSzAQgtBw7MhYFuAFkBsfYfhdJ0rjwBv+bHr1FCyH6j9SgH
movU+bYI+jR+hR2jbzq3sovHKEyZIp9y3/GckbQgualRhIlGpyWfE0dltDUpkYRLN
/OUXmk0/P4H6EI98sAHevGDhOWI+58DjJ44qhZ+l5NNEVxbWDztpb/s5sdx4660qQ
Ozx3/gh8VkqS2JbcxYMpjmrwVr6fAXfb68Ml9mW2Hl7fNtxcb5kjSzXvfWR2XYzJf
KFNrEhbL1DTVlSEC5wE6lGlwhYvXOf22m1R2IFns0MtF9fwcnBWAs1YqjR00j6+fc
er/f/efAFh4=
$__EDBwrapped__$
```

You may notice that the second line of the wrapped file contains an encoding name (in this case, the encoding is UTF8). When you obfuscate a file, edbwrap infers the encoding of the input file by examining the locale. For example, if you are running edbwrap while your locale is set to en_US.utf8, edbwrap assumes that the input file is encoded in UTF8. Be sure to examine the output file after running edbwrap; if the locale contained in the wrapped file does not match the encoding of the input file, you should change your locale and rewrap the input file.

You can import the obfuscated code into the PostgreSQL server using the same tools that work with plaintext code:

```
[bash] edb-psql edb
Welcome to edb-psql 8.4.3.2, the EnterpriseDB interactive terminal.

Type:  \copyright for distribution terms
       \h for help with SQL commands
       \? for help with edb-psql commands
       \g or terminate with semicolon to execute query
       \q to quit

edb=# \i listemp.plb
CREATE PROCEDURE

Now, the pg_proc system table contains the obfuscated code:

edb=# SELECT prosrc FROM pg_proc WHERE proname = 'list_emp';
                                    prosrc
-------------------------------------------------------------------
 $__EDBwrapped__$
 UTF8
 dw4B9Tz69J3WOsy0GgYJQa+G2sLZ3IOyxS8pDyuOTFuiYe/EXiEatwwG3h3tdJk
 ea+AIp35dS/4idbN8wpegM3s994dQ3R97NgNHfvTQnO2vtd4wQtsQ/Zc4v4Lhfj
```

```
 nlV+A4UpHI5oQEnXeAch2LcRD87hkU0uo1ESeQV8IrXaj9BsZr+ueROnwhGs/Ec
 pva/tRV4m9RusFn0wyr38u4Z8w4dfnPW184Y3o6It4b3aH07WxTkWrMLmOZW1jJ
 Nu6u4o+ezO64G9QKPazgehslv4JB9NQnuocActfDSPMY7R7anmgw
 $__EDBwrapped__$
(1 row)
```

Invoke the obfuscated code in the same way that you would invoke the plaintext form:

edb=# 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

EDB-SPL Procedure successfully completed
edb=# quit
```

When you use `pg_dump` to back up a database, wrapped programs remain obfuscated in the archive file.

Be aware that audit logs produced by the Postgres server will show wrapped programs in plaintext form. Source code is also displayed in plaintext in SQL error messages generated during the execution of a program.

Note: At this time, the bodies of the objects created by the following statements will not be stored in obfuscated form:

```
CREATE [OR REPLACE] TYPE type_name AS OBJECT
CREATE [OR REPLACE] TYPE type_name UNDER type_name
CREATE [OR REPLACE] TYPE BODY type_name
```


4 Dynamic Runtime Instrumentation Tools Architecture (DRITA)

The Dynamic Runtime Instrumentation Tools Architecture (DRITA) allows a DBA to query catalog views to determine the *wait events* that affect the performance of individual sessions or the system as a whole. DRITA records the number of times each event occurs as well as the time spent waiting; you can use this information to diagnose performance problems. DRITA offers this functionality, while consuming minimal system resources.

DRITA compares *snapshots* to evaluate the performance of a system. A snapshot is a saved set of system performance data at a given point in time. Each snapshot is identified by a unique ID number; you can use snapshot ID numbers with DRITA reporting functions to return system performance statistics.

4.1 Configuring and Using DRITA

Advanced Server's `postgresql.conf` file includes a configuration parameter named `timed_statistics` that controls the collection of timing data. The valid parameter values are `TRUE` or `FALSE`; the default value is `FALSE`.

This is a dynamic parameter which can be modified in the `postgresql.conf` file, or while a session is in progress. To enable DRITA, you must either:

Modify the `postgresql.conf` file, setting the `timed_statistics` parameter to `TRUE`.

or

Connect to the server with the EDB-PSQL client, and invoke the command:

```
SET timed_statistics = TRUE
```

After modifying the `timed_statistics` parameter, take a starting snapshot. A snapshot captures the current state of each timer and event counter. The server will compare the starting snapshot to a later snapshot to gauge system performance.

Use the `edbsnap()` function to take the beginning snapshot:

```
edb=# SELECT * FROM edbsnap();
       edbsnap
----------------------
 Statement processed.
(1 row)
```

Then, run the workload that you would like to evaluate; when the workload has completed (or at a strategic point during the workload), take another snapshot:

```
edb=# SELECT * FROM edbsnap();
       edbsnap
----------------------
 Statement processed.
(1 row)
```

You can capture multiple snapshots during a session. Then, use the DRITA functions and reports to manage and compare the snapshots to evaluate performance information.

4.2 DRITA Functions

You can use DRITA functions to gather wait information and manage snapshots. DRITA functions are fully supported by Advanced Server 10 whether your installation is made compatible with Oracle databases or is made in PostgreSQL-compatible mode.

4.2.1 get_snaps()

The `get_snaps()` function returns a list of the current snapshots. The signature is:

```
get_snaps()
```

The following example demonstrates using the `get_snaps()` function to display a list of snapshots:

```
edb=# SELECT * FROM get_snaps();
          get_snaps
------------------------------
 1  11-FEB-10 10:41:05.668852
 2  11-FEB-10 10:42:27.26154
 3  11-FEB-10 10:45:48.999992
 4  11-FEB-10 11:01:58.345163
 5  11-FEB-10 11:05:14.092683
 6  11-FEB-10 11:06:33.151002
 7  11-FEB-10 11:11:16.405664
 8  11-FEB-10 11:13:29.458405
 9  11-FEB-10 11:23:57.595916
 10 11-FEB-10 11:29:02.214014
 11 11-FEB-10 11:31:44.244038
(11 rows)
```

The first column in the result list displays the snapshot identifier; the second column displays the date and time that the snapshot was captured.

4.2.2 sys_rpt()

The `sys_rpt()` function returns system wait information. The signature is:

```
sys_rpt(beginning_id, ending_id, top_n)
```

Parameters

beginning_id

beginning_id is an integer value that represents the beginning session identifier.

ending_id

ending_id is an integer value that represents the ending session identifier.

top_n

top_n represents the number of rows to return

This example demonstrates a call to the `sys_rpt()` function:

```
edb=# SELECT * FROM sys_rpt(9, 10, 10);
                                    sys_rpt
-------------------------------------------------------------------------------
 WAIT NAME                                COUNT      WAIT TIME       % WAIT
 -------------------------------------------------------------------------------
 wal write                                21250      104.723772      36.31
 db file read                             121407     72.143274       25.01
 wal flush                                84185      51.652495       17.91
 wal file sync                            712        29.482206       10.22
 infinitecache write                      84178      15.814444       5.48
 db file write                            84177      14.447718       5.01
 infinitecache read                       672        0.098691        0.03
 db file extend                           190        0.040386        0.01
 query plan                               52         0.024400        0.01
 wal insert lock acquire                  4          0.000837        0.00
(12 rows)
```

The information displayed in the result set includes:

Column Name	Description
WAIT NAME	The name of the wait.
COUNT	The number of times that the wait event occurred.
WAIT TIME	The time of the wait event in milliseconds.
% WAIT	The percentage of the total wait time used by this wait for this session.

4.2.3 sess_rpt()

The `sess_rpt()` function returns session wait information. The signature is:

```
sess_rpt(beginning_id, ending_id, top_n)
```

Parameters

beginning_id

beginning_id is an integer value that represents the beginning session identifier.

ending_id

ending_id is an integer value that represents the ending session identifier.

top_n

top_n represents the number of rows to return

The following example demonstrates a call to the `sess_rpt()` function:

```
SELECT * FROM sess_rpt(18, 19, 10);

                              sess_rpt
-------------------------------------------------------------------------------
ID    USER       WAIT NAME              COUNT TIME(ms)    %WAIT SES  %WAIT ALL
 -------------------------------------------------------------------------------

 17373 enterprise db file read           30   0.175713   85.24      85.24
 17373 enterprise query plan             18   0.014930   7.24       7.24
 17373 enterprise wal flush              6    0.004067   1.97       1.97
 17373 enterprise wal write              1    0.004063   1.97       1.97
 17373 enterprise wal file sync          1    0.003664   1.78       1.78
 17373 enterprise infinitecache read     38   0.003076   1.49       1.49
 17373 enterprise infinitecache write    5    0.000548   0.27       0.27
 17373 enterprise db file extend         190  0.04.386   0.03       0.03
 17373 enterprise db file write          5    0.000082   0.04       0.04
 (11 rows)
```

The information displayed in the result set includes:

Column Name	Description
`ID`	The processID of the session.
`USER`	The name of the user incurring the wait.
`WAIT NAME`	The name of the wait event.
`COUNT`	The number of times that the wait event occurred.
`TIME (ms)`	The length of the wait event in milliseconds.
`% WAIT SES`	The percentage of the total wait time used by this wait for this session.
`% WAIT ALL`	The percentage of the total wait time used by this wait (for all sessions).

4.2.4 sessid_rpt()

The `sessid_rpt()` function returns session ID information for a specified backend. The signature is:


```
sessid_rpt(beginning_id, ending_id, backend_id)
```

Parameters

beginning_id

beginning_id is an integer value that represents the beginning session identifier.

ending_id

ending_id is an integer value that represents the ending session identifier.

backend_id

backend_id is an integer value that represents the backend identifier.

The following code sample demonstrates a call to `sessid_rpt()`:

```
SELECT * FROM sessid_rpt(18, 19, 17373);

                              sessid_rpt
-------------------------------------------------------------------------------
 ID    USER       WAIT NAME              COUNT TIME(ms)  %WAIT SES   %WAIT ALL
 ------------------------------------------------------------------------------
 17373 enterprise db file read            30   0.175713  85.24       85.24
 17373 enterprise query plan              18   0.014930  7.24        7.24
 17373 enterprise wal flush               6    0.004067  1.97        1.97
 17373 enterprise wal write               1    0.004063  1.97        1.97
 17373 enterprise wal file sync           1    0.003664  1.78        1.78
 17373 enterprise infinitecache read      38   0.003076  1.49        1.49
 17373 enterprise infinitecache write     5    0.000548  0.27        0.27
 17373 enterprise db file extend          190  0.040386  0.03        0.03
 17373 enterprise db file write           5    0.000082  0.04        0.04
(11 rows)
```

The information displayed in the result set includes:

Column Name	Description
ID	The process ID of the wait.
USER	The name of the user that owns the session.
WAIT NAME	The name of the wait event.
COUNT	The number of times that the wait event occurred.
TIME (ms)	The length of the wait in milliseconds.
% WAIT SES	The percentage of the total wait time used by this wait for this session.
% WAIT ALL	The percentage of the total wait time used by this wait (for all sessions).

4.2.5 sesshist_rpt()

The `sesshist_rpt()` function returns session wait information for a specified backend. The signature is:

```
sesshist_rpt(snapshot_id, session_id)
```

Parameters

snapshot_id

snapshot_id is an integer value that identifies the snapshot.

session_id

session_id is an integer value that represents the session.

The following example demonstrates a call to the `sesshist_rpt()` function:

```
edb=# SELECT * FROM sesshist_rpt (9, 5531);
                              sesshist_rpt
-------------------------------------------------------------------------------
 ID    USER       SEQ  WAIT NAME
   ELAPSED(ms)   File  Name                 # of Blk   Sum of Blks
 -------------------------------------------------------------------------------
 5531 enterprise 1     db file read
   18546        14309  session_waits_pk     1          1
 5531 enterprise 2     infinitecache read
   125          14309  session_waits_pk     1          1
 5531 enterprise 3     db file read
   376          14304  edb$session_waits    0          1
 5531 enterprise 4     infinitecache read
   166          14304  edb$session_waits    0          1
 5531 enterprise 5     db file read
   7978         1260   pg_authid            0          1
 5531 enterprise 6     infinitecache read
   154          1260   pg_authid            0          1
 5531 enterprise 7     db file read
   628          14302  system_waits_pk      1          1
 5531 enterprise 8     infinitecache read
   463          14302  system_waits_pk      1          1
 5531 enterprise 9     db file read
   3446         14297  edb$system_waits     0          1
 5531 enterprise 10    infinitecache read
   187          14297  edb$system_waits     0          1
 5531 enterprise 11    db file read
   14750        14295  snap_pk              1          1
 5531 enterprise 12    infinitecache read
   416          14295  snap_pk              1          1
 5531 enterprise 13    db file read
   7139         14290  edb$snap             0          1
 5531 enterprise 14    infinitecache read
   158          14290  edb$snap             0          1
 5531 enterprise 15    db file read
   27287        14288  snapshot_num_seq     0          1
```

```
 5531 enterprise 16    infinitecache read
(17 rows)
```

The information displayed in the result set includes:

Column Name	Description
ID	The system-assigned identifier of the wait.
USER	The name of the user that incurred the wait.
SEQ	The sequence number of the wait event.
WAIT NAME	The name of the wait event.
ELAPSED (ms)	The length of the wait event in milliseconds.
File	The relfilenode number of the file.
Name	If available, the name of the file name related to the wait event.
# of Blk	The block number read or written for a specific instance of the event .
Sum of Blks	The number of blocks read.

4.2.6 purgesnap()

The `purgesnap()` function purges a range of snapshots from the snapshot tables. The signature is:

```
purgesnap(beginning_id, ending_id)
```

Parameters

`beginning_id`

`beginning_id` is an integer value that represents the beginning session identifier.

`ending_id`

`ending_id` is an integer value that represents the ending session identifier.

`purgesnap()` removes all snapshots between *`beginning_id`* and *`ending_id`* (inclusive):

```
SELECT * FROM purgesnap(6, 9);

             purgesnap
-------------------------------------
 Snapshots in range 6 to 9 deleted.
(1 row)
```


A call to the `get_snaps()` function after executing the example shows that snapshots `6` through `9` have been purged from the snapshot tables:

```
edb=# SELECT * FROM get_snaps();
          get_snaps
------------------------------
 1  11-FEB-10 10:41:05.668852
 2  11-FEB-10 10:42:27.26154
 3  11-FEB-10 10:45:48.999992
 4  11-FEB-10 11:01:58.345163
 5  11-FEB-10 11:05:14.092683
 10 11-FEB-10 11:29:02.214014
 11 11-FEB-10 11:31:44.244038
(7 rows)
```

4.2.7 truncsnap()

Use the `truncsnap()` function to delete all records from the snapshot table. The signature is:

```
truncsnap()
```

For example:

```
SELECT * FROM truncsnap();

      truncsnap
----------------------
 Snapshots truncated.
(1 row)
```

A call to the `get_snaps()` function after calling the `truncsnap()` function shows that all records have been removed from the snapshot tables:

```
SELECT * FROM get_snaps();
 get_snaps
-----------
(0 rows)
```

4.3 Simulating Statspack AWR Reports

The functions described in this section return information comparable to the information contained in an Oracle Statspack/AWR (Automatic Workload Repository) report. When taking a snapshot, performance data from system catalog tables is saved into history tables. The reporting functions listed below report on the differences between two given snapshots.

- `stat_db_rpt()`
- `stat_tables_rpt()`
- `statio_tables_rpt()`
- `stat_indexes_rpt()`
- `statio_indexes_rpt()`

The reporting functions can be executed individually or you can execute all five functions by calling the `edbreport()` function.

4.3.1 edbreport()

The `edbreport()` function includes data from the other reporting functions, plus additional system information. The signature is:

```
edbreport(beginning_id, ending_id)
```

Parameters

`beginning_id`

`beginning_id` is an integer value that represents the beginning session identifier.

`ending_id`

`ending_id` is an integer value that represents the ending session identifier.

The call to the `edbreport()` function returns a composite report that contains system information and the reports returned by the other statspack functions. :

```
edb=# SELECT * FROM edbreport(9, 10);

edbreport
--------------------------------------------------------------------------------
    EnterpriseDB Report for database edb        23-AUG-15
 Version: EnterpriseDB 10.0.0 on i686-pc-linux-gnu
      Begin snapshot: 9 at 23-AUG-15 13:45:07.165123
```



```
     End snapshot:   10 at 23-AUG-15 13:45:35.653036

Size of database edb is 155 MB
     Tablespace: pg_default Size: 179 MB Owner: enterprisedb
     Tablespace: pg_global  Size: 435 kB Owner: enterprisedb

Schema: pg_toast_temp_1         Size: 0 bytes    Owner: enterprisedb
Schema: public                  Size: 0 bytes    Owner: enterprisedb
Schema: enterprisedb            Size: 143 MB     Owner: enterprisedb
Schema: pgagent                 Size: 192 kB     Owner: enterprisedb
Schema: dbms_job_procedure      Size: 0 bytes    Owner: enterprisedb
```

The information displayed in the report introduction includes the database name and version, the current date, the beginning and ending snapshot date and times, database and tablespace details and schema information.

```
               Top 10 Relations by pages

TABLE                                                RELPAGES
-------------------------------------------------------------------------------
pgbench_accounts                                      15874
pg_proc                                               102
edb$statio_all_indexes                                73
edb$stat_all_indexes                                  73
pg_attribute                                          67
pg_depend                                             58
edb$statio_all_tables                                 49
edb$stat_all_tables                                   47
pgbench_tellers                                       37
pg_description                                        32
```

The information displayed in the `Top 10 Relations by pages` section includes:

Column Name	Description
TABLE	The name of the table.
RELPAGES	The number of pages in the table.

```
               Top 10 Indexes by pages

INDEX                                                RELPAGES
-------------------------------------------------------------------------------
pgbench_accounts_pkey                                 2198
pg_depend_depender_index                              32
pg_depend_reference_index                             31
pg_proc_proname_args_nsp_index                        30
pg_attribute_relid_attnam_index                       23
pg_attribute_relid_attnum_index                       17
pg_description_o_c_o_index                            15
edb$statio_idx_pk                                     11
edb$stat_idx_pk                                       11
pg_proc_oid_index                                     9
```


The information displayed in the `Top 10 Indexes by pages` section includes:

Column Name	Description
INDEX	The name of the index.
RELPAGES	The number of pages in the index.

```
                Top 10 Relations by DML

 SCHEMA          RELATION                        UPDATES   DELETES   INSERTS
 ------------------------------------------------------------------------------
 enterprisedb    pgbench_accounts                10400     0         1000000
 enterprisedb    pgbench_tellers                 10400     0         100
 enterprisedb    pgbench_branches                10400     0         10
 enterprisedb    pgbench_history                 0         0         10400
 pgagent         pga_jobclass                    0         0         6
 pgagent         pga_exception                   0         0         0
 pgagent         pga_job                         0         0         0
 pgagent         pga_jobagent                    0         0         0
 pgagent         pga_joblog                      0         0         0
 pgagent         pga_jobstep                     0         0         0
```

The information displayed in the `Top 10 Relations by DML` section includes:

Column Name	Description
SCHEMA	The name of the schema in which the table resides.
RELATION	The name of the table.
UPDATES	The number of UPDATES performed on the table.
DELETES	The number of DELETES performed on the table.
INSERTS	The number of INSERTS performed on the table.

```
   DATA from pg_stat_database

 DATABASE   NUMBACKENDS  XACT COMMIT  XACT ROLLBACK   BLKS READ  BLKS HIT
BLKS ICACHE HIT      HIT RATIO  ICACHE HIT RATIO
 ------------------------------------------------------------------------------
 edb        0            142          0              78         10446
    0                99.26      0.00
```

The information displayed in the `DATA from pg_stat_database` section of the report includes:

Column Name	Description
DATABASE	The name of the database.
NUMBACKENDS	Number of backends currently connected to this database. This is the only column in this view that returns a value reflecting current state; all other columns return the accumulated values since the last reset.
XACT COMMIT	Number of transactions in this database that have been committed.
XACT ROLLBACK	Number of transactions in this database that have been rolled back.
BLKS READ	Number of disk blocks read in this database.
BLKS HIT	Number of times disk blocks were found already in the buffer cache (when a read was not necessary).

Column Name	Description
BLKS ICACHE HIT	The number of blocks found in Infinite Cache.
HIT RATIO	The percentage of times that a block was found in the shared buffer cache.
ICACHE HIT RATIO	The percentage of times that a block was found in Infinite Cache.

```
   DATA from pg_buffercache

 RELATION                              BUFFERS
 ------------------------------------------------------------------------------
-
 pgbench_accounts                      16671
 pgbench_accounts_pkey                 2745
 pgbench_history                       590
 pg_statistic                          39
 edb$statio_all_indexes                31
 edb$stat_all_indexes                  31
 edb$statio_all_tables                 21
 edb$stat_all_tables                   20
 pg_depend                             20
 pg_operator                           15
```

The information displayed in the `DATA from pg_buffercache` section of the report includes:

Column Name	Description
RELATION	The name of the table.
BUFFERS	The number of shared buffers used by the relation.

Note: In order to obtain the report for `DATA from pg_buffercache`, the `pg_buffercache` module must have been installed in the database. Perform the installation with the `CREATE EXTENSION` command.

For more information on the `CREATE EXTENSION` command please see the PostgreSQL Core documentation at:

https://www.postgresql.org/docs/10/static/sql-createextension.html

```
   DATA from pg_stat_all_tables ordered by seq scan

 SCHEMA               RELATION                       SEQ SCAN   REL TUP READ
IDX SCAN   IDX TUP READ INS    UPD    DEL
 ------------------------------------------------------------------------------
 pg_catalog           pg_class                       16         7162
546        319           0      1      0
 pg_catalog           pg_am                          13         13
0          0             0      0      0
 pg_catalog           pg_database                    4          16
42         42            0      0      0
 pg_catalog           pg_index                       4          660
145        149           0      0      0
```

```
 pg_catalog           pg_namespace                    4          100
49          49            0      0      0
 sys                  edb$snap                        1          9
0           0             1      0      0
 pg_catalog           pg_authid                       1          1
25          25            0      0      0
 sys                  edb$session_wait_history        0          0
0           0             50     0      0
 sys                  edb$session_waits               0          0
0           0             2      0      0
 sys                  edb$stat_all_indexes            0          0
0           0             165    0      0
```

The information displayed in the `DATA from pg_stat_all_tables ordered by seq scan` section includes:

Column Name	Description
SCHEMA	The name of the schema in which the table resides.
RELATION	The name of the table.
SEQ SCAN	The number of sequential scans initiated on this table..
REL TUP READ	The number of tuples read in the table.
IDX SCAN	The number of index scans initiated on the table.
IDX TUP READ	The number of index tuples read.
INS	The number of rows inserted.
UPD	The number of rows updated.
DEL	The number of rows deleted.

```
   DATA from pg_stat_all_tables ordered by rel tup read

 SCHEMA               RELATION                        SEQ SCAN   REL TUP READ
IDX SCAN   IDX TUP READ INS    UPD    DEL
 ------------------------------------------------------------------------------
 pg_catalog           pg_class                        16         7162
546         319           0      1      0
 pg_catalog           pg_index                        4          660
145         149           0      0      0
 pg_catalog           pg_namespace                    4          100
49          49            0      0      0
 pg_catalog           pg_database                     4          16
42          42            0      0      0
 pg_catalog           pg_am                           13         13
0           0             0      0      0
 sys                  edb$snap                        1          9
0           0             1      0      0
 pg_catalog           pg_authid                       1          1
25          25            0      0      0
 sys                  edb$session_wait_history        0          0
0           0             50     0      0
 sys                  edb$session_waits               0          0
0           0             2      0      0
 sys                  edb$stat_all_indexes            0          0
0           0             165    0      0
```

The information displayed in the `DATA from pg_stat_all_tables ordered by rel tup read` section includes:

Column Name	Description
SCHEMA	The name of the schema in which the table resides.
RELATION	The name of the table.
SEQ SCAN	The number of sequential scans performed on the table.
REL TUP READ	The number of tuples read from the table.
IDX SCAN	The number of index scans performed on the table.
IDX TUP READ	The number of index tuples read.
INS	The number of rows inserted.
UPD	The number of rows updated.
DEL	The number of rows deleted.

```
   DATA from pg_statio_all_tables

 SCHEMA      RELATION             HEAP     HEAP     HEAP     IDX      IDX
                                  READ     HIT      ICACHE   READ     HIT
                                                    HIT

             IDX      TOAST    TOAST    TOAST    TIDX     TIDX     TIDX
             ICACHE   READ     HIT      ICACHE   READ     HIT      ICACHE
             HIT                        HIT                        HIT
-------------------------------------------------------------------------------
 public      pgbench_accounts     92766    67215    288      59       32126
             9        0        0        0        0        0        0
 pg_catalog  pg_class             0        296      0        3        16
             0        0        0        0        0        0        0
 sys         edb$stat_all_indexes 8        125      0        4        233
             0        0        0        0        0        0        0
 sys         edb$statio_all_index 8        125      0        4        233
             0        0        0        0        0        0        0
 sys         edb$stat_all_tables  6        91       0        2        174
             0        0        0        0        0        0        0
 sys         edb$statio_all_table 6        91       0        2        174
             0        0        0        0        0        0        0
 pg_catalog  pg_namespace         3        72       0        0        0
             0        0        0        0        0        0        0
 sys         edb$session_wait_his 1        24       0        4        47
             0        0        0        0        0        0        0
 pg_catalog  pg_opclass           3        13       0        2        0
             0        0        0        0        0        0        0
 pg_catalog  pg_trigger           0        12       0        1        15
             0        0        0        0        0        0        0
```

The information displayed in the `Data from pg_statio_all_tables` section includes:

Column Name	Description
SCHEMA	The name of the schema in which the table resides.
RELATION	The name of the table.
HEAP READ	The number of heap blocks read.
HEAP HIT	The number of heap blocks hit.

Column Name	Description
HEAP ICACHE HIT	The number of heap blocks in Infinite Cache.
IDX READ	The number of index blocks read.
IDX HIT	The number of index blocks hit.
IDX ICACHE HIT	The number of index blocks in Infinite Cache.
TOAST READ	The number of toast blocks read.
TOAST HIT	The number of toast blocks hit.
TOAST ICACHE HIT	The number of toast blocks in Infinite Cache.
TIDX READ	The number of toast index blocks read.
TIDX HIT	The number of toast index blocks hit.
TIDX ICACHE HIT	The number of toast index blocks in Infinite Cache.

```
   DATA from pg_stat_all_indexes

 SCHEMA               RELATION                  INDEX
IDX SCAN   IDX TUP READ IDX TUP FETCH
 ------------------------------------------------------------------------------
 pg_catalog           pg_attribute
pg_attribute_relid_attnum_index     427        907          907
 pg_catalog           pg_class                  pg_class_relname_nsp_index
289        62           62
 pg_catalog           pg_class                  pg_class_oid_index
257        257          257
 pg_catalog           pg_statistic
pg_statistic_relid_att_inh_index    207        196          196
 enterprisedb         pgbench_accounts          pgbench_accounts_pkey
200        255          200
 pg_catalog           pg_cast                   pg_cast_source_target_index
199        50           50
 pg_catalog           pg_proc                   pg_proc_oid_index
116        116          116
 pg_catalog           edb_partition             edb_partition_partrelid_index
112        0            0
 pg_catalog           edb_policy                edb_policy_object_name_index
112        0            0
 enterprisedb         pgbench_branches          pgbench_branches_pkey
101        110          0
```

The information displayed in the `DATA from pg_stat_all_indexes` section includes:

Column Name	Description
SCHEMA	The name of the schema in which the index resides.
RELATION	The name of the table on which the index is defined.
INDEX	The name of the index.
IDX SCAN	The number of indexes scans initiated on this index.
IDX TUP READ	Number of index entries returned by scans on this index
IDX TUP FETCH	Number of live table rows fetched by simple index scans using this index.

```
   DATA from pg_statio_all_indexes
```



```
 SCHEMA                RELATION                    INDEX
IDX BLKS READ   IDX BLKS HIT    IDX BLKS ICACHE HIT
 ------------------------------------------------------------------------------
 pg_catalog            pg_attribute
pg_attribute_relid_attnum_index     0               867             0
 enterprisedb          pgbench_accounts           pgbench_accounts_pkey
1               778             0
 pg_catalog            pg_class                   pg_class_relname_nsp_index
0               590             0
 pg_catalog            pg_class                   pg_class_oid_index
0               527             0
 pg_catalog            pg_statistic
pg_statistic_relid_att_inh_index    0               441             0
 sys                   edb$stat_all_indexes       edb$stat_idx_pk
1               332             0
 sys                   edb$statio_all_indexes     edb$statio_idx_pk
1               332             0
 pg_catalog            pg_proc                    pg_proc_oid_index
0               244             0
 sys                   edb$stat_all_tables        edb$stat_tab_pk
0               241             0
 sys                   edb$statio_all_tables      edb$statio_tab_pk
0               241             0
```

The information displayed in the `DATA from pg_statio_all_indexes` section includes:

Column Name	Description
SCHEMA	The name of the schema in which the index resides.
RELATION	The name of the table on which the index is defined.
INDEX	The name of the index.
IDX BLKS READ	The number of index blocks read.
IDX BLKS HIT	The number of index blocks hit.
IDX BLKS ICACHE HIT	The number of index blocks in Infinite Cache that were hit.

```
    System Wait Information

 WAIT NAME                                COUNT      WAIT TIME       % WAIT
 ------------------------------------------------------------------------------
 query plan                               0          0.000407        100.00
 db file read                             0          0.000000        0.00
```

The information displayed in the `System Wait Information` section includes:

Column Name	Description
WAIT NAME	The name of the wait.
COUNT	The number of times that the wait event occurred.
WAIT TIME	The length of the wait time in milliseconds.
% WAIT	The percentage of the total wait time used by this wait for this session.

```
    Database Parameters from postgresql.conf
```

```
 PARAMETER                          SETTING
CONTEXT     MINVAL       MAXVAL
 ------------------------------------------------------------------------------
 allow_system_table_mods            off
postmaster
 application_name                   psql
user
 archive_command                    (disabled)
sighup
 archive_mode                       off
postmaster
 archive_timeout                    0
sighup      0            2147483647
 array_nulls                        on
user
 authentication_timeout             60
sighup      1            600
 autovacuum                         on
sighup
 autovacuum_analyze_scale_factor    0.1
sighup      0            100
 autovacuum_analyze_threshold       50
sighup      0            2147483647
 autovacuum_freeze_max_age          200000000
postmaster  100000000    2000000000
 autovacuum_max_workers             3
postmaster  1            8388607
 autovacuum_naptime                 60
sighup      1            2147483
 autovacuum_vacuum_cost_delay       20
...
```

The information displayed in the `Database Parameters from postgresql.conf` section includes:

Column Name	Description
`PARAMETER`	The name of the parameter.
`SETTING`	The current value assigned to the parameter.
`CONTEXT`	The context required to set the parameter value.
`MINVAL`	The minimum value allowed for the parameter.
`MAXVAL`	The maximum value allowed for the parameter.

4.3.2 stat_db_rpt()

The signature is:

```
stat_db_rpt(beginning_id, ending_id)
```

Parameters

`beginning_id`

`beginning_id` is an integer value that represents the beginning session identifier.

`ending_id`

`ending_id` is an integer value that represents the ending session identifier.

The following example demonstrates the `stat_db_rpt()` function:

```
SELECT * FROM stat_db_rpt(9, 10);
                              stat_db_rpt
------------------------------------------------------------------------------
   DATA from pg_stat_database

 DATABASE   NUMBACKENDS  XACT COMMIT  XACT ROLLBACK   BLKS READ  BLKS HIT
        BLKS ICACHE HIT      HIT RATIO      ICACHE HIT RATIO
------------------------------------------------------------------------------
 edb        1            21           0               92928      101217
        301                 52.05          0.15
```

The information displayed in the `DATA from pg_stat_database` section of the report includes:

Column Name	Description
`DATABASE`	The name of the database.
`NUMBACKENDS`	Number of backends currently connected to this database. This is the only column in this view that returns a value reflecting current state; all other columns return the accumulated values since the last reset.
`XACT COMMIT`	The number of transactions in this database that have been committed.
`XACT ROLLBACK`	The number of transactions in this database that have been rolled back.
`BLKS READ`	The number of blocks read.
`BLKS HIT`	The number of blocks hit.
`BLKS ICACHE HIT`	The number of blocks in Infinite Cache that were hit.
`HIT RATIO`	The percentage of times that a block was found in the shared buffer cache.
`ICACHE HIT RATIO`	The percentage of times that a block was found in Infinite Cache.

4.3.3 stat_tables_rpt()

The signature is:

```
function_name(beginning_id, ending_id, top_n, scope)
```

Parameters

`beginning_id`

`beginning_id` is an integer value that represents the beginning session identifier.

`ending_id`

`ending_id` is an integer value that represents the ending session identifier.

`top_n`

`top_n` represents the number of rows to return

`scope`

`scope` determines which tables the function returns statistics about. Specify `SYS`, `USER` or `ALL`:

- `SYS` indicates that the function should return information about system defined tables. A table is considered a system table if it is stored in one of the following schemas: `pg_catalog`, `information_schema`, `sys`, or `dbo`.
- `USER` indicates that the function should return information about user-defined tables.
- `ALL` specifies that the function should return information about all tables.

The `stat_tables_rpt()` function returns a two-part report. The first portion of the report contains:

```
SELECT * FROM stat_tables_rpt(18, 19, 10, 'ALL');

stat_tables_rpt
------------------------------------------------------------------------------
DATA from pg_stat_all_tables ordered by seq scan

SCHEMA        RELATION
    SEQ SCAN   REL TUP READ IDX SCAN   IDX TUP READ   INS    UPD    DEL
------------------------------------------------------------------------------
pg_catalog    pg_class
    8          2952         78         65             0      0      0
pg_catalog    pg_index
    4          448          23         28             0      0      0
pg_catalog    pg_namespace
    4          76           1          1              0      0      0
pg_catalog    pg_database
    3          6            0          0              0      0      0
pg_catalog    pg_authid
    2          1            0          0              0      0      0
sys           edb$snap
    1          15           0          0              1      0      0
```



```
public         accounts
    0           0            0          0              0      0      0
public         branches
    0           0            0          0              0      0      0
sys            edb$session_wait_history
    0           0            0          0              25     0      0
sys            edb$session_waits
    0           0            0          0              10     0      0
```

The information displayed in the `DATA from pg_stat_all_tables ordered by seq scan` section includes:

Column Name	Description
SCHEMA	The name of the schema in which the table resides.
RELATION	The name of the table.
SEQ SCAN	The number of sequential scans on the table.
REL TUP READ	The number of tuples read from the table.
IDX SCAN	The number of index scans performed on the table.
IDX TUP READ	The number of index tuples read from the table.
INS	The number of rows inserted.
UPD	The number of rows updated.
DEL	The number of rows deleted.

The second portion of the report contains:

```
DATA from pg_stat_all_tables ordered by rel tup read

SCHEMA       RELATION
    SEQ SCAN   REL TUP READ IDX SCAN   IDX TUP READ INS    UPD    DEL
-------------------------------------------------------------------------------
pg_catalog   pg_class
    8          2952         78         65           0      0      0
pg_catalog   pg_index
    4          448          23         28           0      0      0
pg_catalog   pg_namespace
    4          76           1          1            0      0      0
sys          edb$snap
    1          15           0          0            1      0      0
pg_catalog   pg_database
    3          6            0          0            0      0      0
pg_catalog   pg_authid
    2          1            0          0            0      0      0
public       accounts
    0          0            0          0            0      0      0
public       branches
    0          0            0          0            0      0      0
sys          edb$session_wait_history
    0          0            0          0            25     0      0
sys          edb$session_waits
    0          0            0          0            10     0      0
(29 rows)
```

The information displayed in the `DATA from pg_stat_all_tables ordered by rel tup read` section includes:

Column Name	Description
SCHEMA	The name of the schema in which the table resides.
RELATION	The name of the table.
SEQ SCAN	The number of sequential scans performed on the table.
REL TUP READ	The number of tuples read from the table.
IDX SCAN	The number of index scans performed on the table.
IDX TUP READ	The number of live rows fetched by index scans.
INS	The number of rows inserted.
UPD	The number of rows updated.
DEL	The number of rows deleted.

4.3.4 statio_tables_rpt()

The signature is:

```
statio_tables_rpt(beginning_id, ending_id, top_n, scope)
```

Parameters

`beginning_id`

`beginning_id` is an integer value that represents the beginning session identifier.

`ending_id`

`ending_id` is an integer value that represents the ending session identifier.

`top_n`

`top_n` represents the number of rows to return

`scope`

`scope` determines which tables the function returns statistics about. Specify `SYS`, `USER` or `ALL`:

- `SYS` indicates that the function should return information about system defined tables. A table is considered a system table if it is stored in one of

the following schemas: `pg_catalog`, `information_schema`, `sys`, or `dbo`.

- `USER` indicates that the function should return information about user-defined tables.
- `ALL` specifies that the function should return information about all tables.

The `statio_tables_rpt()` function returns a report that contains:

```
edb=# SELECT * FROM statio_tables_rpt(9, 10, 10, 'SYS');

                              statio_tables_rpt
------------------------------------------------------------------------------
   DATA from pg_statio_all_tables

 SCHEMA      RELATION             HEAP     HEAP     HEAP     IDX      IDX
                                  READ     HIT      ICACHE   READ     HIT
                                                    HIT

             IDX      TOAST    TOAST    TOAST    TIDX     TIDX    TIDX
             ICACHE   READ     HIT      ICACHE   READ     HIT     ICACHE
             HIT                        HIT                       HIT
------------------------------------------------------------------------------
 public      pgbench_accounts     92766    67215    288      59       32126
             9        0        0        0        0        0        0
 pg_catalog  pg_class             0        296      0        3        16
             0        0        0        0        0        0        0
 sys         edb$stat_all_indexes 8        125      0        4        233
             0        0        0        0        0        0        0
 sys         edb$statio_all_index 8        125      0        4        233
             0        0        0        0        0        0        0
 sys         edb$stat_all_tables  6        91       0        2        174
             0        0        0        0        0        0        0
 sys         edb$statio_all_table 6        91       0        2        174
             0        0        0        0        0        0        0
 pg_catalog  pg_namespace         3        72       0        0        0
             0        0        0        0        0        0        0
 sys         edb$session_wait_his 1        24       0        4        47
             0        0        0        0        0        0        0
 pg_catalog  pg_opclass           3        13       0        2        0
             0        0        0        0        0        0        0
 pg_catalog  pg_trigger           0        12       0        1        15
             0        0        0        0        0        0        0
(16 rows)
```

The information displayed in the `Data from pg_statio_all_tables` section includes:

Column Name	Description
`SCHEMA`	The name of the schema in which the relation resides.
`RELATION`	The name of the relation.
`HEAP READ`	The number of heap blocks read.
`HEAP HIT`	The number of heap blocks hit.

Column Name	Description
HEAP ICACHE HIT	The number of heap blocks in Infinite Cache.
IDX READ	The number of index blocks read.
IDX HIT	The number of index blocks hit.
IDX ICACHE HIT	The number of index blocks in Infinite Cache.
TOAST READ	The number of toast blocks read.
TOAST HIT	The number of toast blocks hit.
TOAST ICACHE HIT	The number of toast blocks in Infinite Cache.
TIDX READ	The number of toast index blocks read.
TIDX HIT	The number of toast index blocks hit.
TIDX ICACHE HIT	The number of toast index blocks in Infinite Cache.

4.3.5 stat_indexes_rpt()

The signature is:

```
stat_indexes_rpt(beginning_id, ending_id, top_n, scope)
```

Parameters

`beginning_id`

`beginning_id` is an integer value that represents the beginning session identifier.

`ending_id`

`ending_id` is an integer value that represents the ending session identifier.

`top_n`

`top_n` represents the number of rows to return

`scope`

`scope` determines which tables the function returns statistics about. Specify `SYS`, `USER` or `ALL`:

- `SYS` indicates that the function should return information about system defined tables. A table is considered a system table if it is stored in one of the following schemas: `pg_catalog`, `information_schema`, `sys`, or `dbo`.

- `USER` indicates that the function should return information about user-defined tables.

- `ALL` specifies that the function should return information about all tables.

The `stat_indexes_rpt()` function returns a report that contains:

```
edb=# SELECT * FROM stat_indexes_rpt(9, 10, 10, 'ALL');

                           stat_indexes_rpt
-------------------------------------------------------------------------------
   DATA from pg_stat_all_indexes

 SCHEMA        RELATION        INDEX
                            IDX SCAN    IDX TUP READ    IDX TUP FETCH
-------------------------------------------------------------------------------
 pg_catalog    pg_cast          pg_cast_source_target_index
                           30           7               7
 pg_catalog    pg_class         pg_class_oid_index
                           15           15              15
 pg_catalog    pg_trigger       pg_trigger_tgrelid_tgname_index
                           12           12              12
 pg_catalog    pg_attribute     pg_attribute_relid_attnum_index
                           7            31              31
 pg_catalog    pg_statistic     pg_statistic_relid_att_index
                           7            0               0
 pg_catalog    pg_database      pg_database_oid_index
                           5            5               5
 pg_catalog    pg_proc          pg_proc_oid_index
                           5            5               5
 pg_catalog    pg_operator      pg_operator_oprname_l_r_n_index
                           3            1               1
 pg_catalog    pg_type          pg_type_typname_nsp_index
                           3            1               1
 pg_catalog    pg_amop          pg_amop_opr_fam_index
                           2            3               3
(14 rows)
```

The information displayed in the `DATA from pg_stat_all_indexes` section includes:

Column Name	Description
`SCHEMA`	The name of the schema in which the relation resides.
`RELATION`	The name of the relation.
`INDEX`	The name of the index.
`IDX SCAN`	The number of indexes scanned.
`IDX TUP READ`	The number of index tuples read.
`IDX TUP FETCH`	The number of index tuples fetched.

4.3.6 statio_indexes_rpt()

The signature is:

```
statio_indexes_rpt(beginning_id, ending_id, top_n, scope)
```

Parameters

`beginning_id`

`beginning_id` is an integer value that represents the beginning session identifier.

`ending_id`

`ending_id` is an integer value that represents the ending session identifier.

`top_n`

`top_n` represents the number of rows to return

`scope`

`scope` determines which tables the function returns statistics about. Specify `SYS`, `USER` or `ALL`:

- `SYS` indicates that the function should return information about system defined tables. A table is considered a system table if it is stored in one of the following schemas: `pg_catalog`, `information_schema`, `sys`, or `dbo`.
- `USER` indicates that the function should return information about user-defined tables.
- `ALL` specifies that the function should return information about all tables.

The `statio_indexes_rpt()` function returns a report that contains:

```
edb=# SELECT * FROM statio_indexes_rpt(9, 10, 10, 'SYS');

                            statio_indexes_rpt
------------------------------------------------------------------------------
   DATA from pg_statio_all_indexes

 SCHEMA      RELATION        INDEX
                          IDX BLKS READ   IDX BLKS HIT    IDX BLKS ICACHE HIT
------------------------------------------------------------------------------
```



```
public                pgbench_accounts          pgbench_accounts_pkey
                         59                32126                     9
 sys                   edb$stat_all_indexes      edb$stat_idx_pk
                          4                  233                     0
 sys                   edb$statio_all_indexes    edb$statio_idx_pk
                          4                  233                     0
 sys                   edb$stat_all_tables       edb$stat_tab_pk
                          2                  174                     0
 sys                   edb$statio_all_tables     edb$statio_tab_pk
                          2                  174                     0
 sys                   edb$session_wait_history  session_waits_hist_pk
                          4                   47                     0
 pg_catalog            pg_cast                   pg_cast_source_target_index
                          1                   29                     0
 pg_catalog            pg_trigger                pg_trig_tgrelid_tgname_index
                          1                   15                     0
 pg_catalog            pg_class                  pg_class_oid_index
                          1                   14                     0
 pg_catalog            pg_statistic              pg_statistic_relid_att_index
                          2                   12                     0
(14 rows)
```

The information displayed in the `DATA from pg_statio_all_indexes` report includes:

Column Name	Description
SCHEMA	The name of the schema in which the relation resides.
RELATION	The name of the table on which the index is defined.
INDEX	The name of the index.
IDX BLKS READ	The number of index blocks read.
IDX BLKS HIT	The number of index blocks hit.
IDX BLKS ICACHE HIT	The number of index blocks in Infinite Cache that were hit.

4.4 Performance Tuning Recommendations

To use DRITA reports for performance tuning, review the top five events in a given report, looking for any event that takes a disproportionately large percentage of resources. In a streamlined system, user I/O will probably make up the largest number of waits. Waits should be evaluated in the context of CPU usage and total time; an event may not be significant if it takes 2 minutes out of a total measurement interval of 2 hours, if the rest of the time is consumed by CPU time. The component of response time (CPU "work" time or other "wait" time) that consumes the highest percentage of overall time should be evaluated.

When evaluating events, watch for:

Event type	Description
Checkpoint waits	Checkpoint waits may indicate that checkpoint parameters need to be adjusted, (`checkpoint_segments` and `checkpoint_timeout`).
WAL-related waits	WAL-related waits may indicate `wal_buffers` are under-sized.
SQL Parse waits	If the number of waits is high, try to use prepared statements.
db file random reads	If high, check that appropriate indexes and statistics exist.
db file random writes	If high, may need to decrease `bgwriter_delay`.
btree random lock acquires	May indicate indexes are being rebuilt. Schedule index builds during less active time.

Performance reviews should also include careful scrutiny of the hardware, the operating system, the network and the application SQL statements.

4.5 Event Descriptions

The following table lists the basic wait events that are displayed by DRITA.

Event Name	Description
add in shmem lock acquire	Obsolete/unused
bgwriter communication lock acquire	The bgwriter (background writer) process has waited for the short-term lock that synchronizes messages between the bgwriter and a backend process.
btree vacuum lock acquire	The server has waited for the short-term lock that synchronizes access to the next available vacuum cycle ID.
buffer free list lock acquire	The server has waited for the short-term lock that synchronizes access to the list of free buffers (in shared memory).
checkpoint lock acquire:	A server process has waited for the short-term lock that prevents simultaneous checkpoints.
checkpoint start lock acquire	The server has waited for the short-term lock that synchronizes access to the bgwriter checkpoint schedule.
clog control lock acquire	The server has waited for the short-term lock that synchronizes access to the commit log.
control file lock acquire	The server has waited for the short-term lock that synchronizes write access to the control file (this should usually be a low number).
db file extend	A server process has waited for the operating system while adding a new page to the end of a file.
db file read	A server process has waited for the completion of a read (from disk).
db file write	A server process has waited for the completion of a write (to disk).
db file sync	A server process has waited for the operating system to flush all changes to disk.
first buf mapping lock acquire	The server has waited for a short-term lock that synchronizes access to the shared-buffer mapping table.
freespace lock acquire	The server has waited for the short-term lock that synchronizes access to the freespace map.
Infinite Cache read	The server has waited for an Infinite Cache read request.
Infinite Cache write	The server has waited for an Infinite Cache write request.
lwlock acquire	The server has waited for a short-term lock that has not been described elsewhere in this section.
multi xact gen lock acquire	The server has waited for the short-term lock that synchronizes access to the next available multi-transaction ID (when a SELECT...FOR SHARE statement executes).
multi xact member lock acquire	The server has waited for the short-term lock that synchronizes access to the multi-transaction member file (when a SELECT...FOR SHARE statement executes).
multi xact offset lock acquire	The server has waited for the short-term lock that synchronizes access to the multi-transaction offset file (when a SELECT...FOR SHARE statement executes).
oid gen lock acquire	The server has waited for the short-term lock that synchronizes access to the next available OID (object ID).
query plan	The server has computed the execution plan for a SQL statement.
rel cache init lock acquire	The server has waited for the short-term lock that prevents simultaneous relation-cache loads/unloads.

shmem index lock acquire	The server has waited for the short-term lock that synchronizes access to the shared-memory map.
sinval lock acquire	The server has waited for the short-term lock that synchronizes access to the cache invalidation state.
sql parse	The server has parsed a SQL statement.
subtrans control lock acquire	The server has waited for the short-term lock that synchronizes access to the subtransaction log.
tablespace create lock acquire	The server has waited for the short-term lock that prevents simultaneous CREATE TABLESPACE or DROP TABLESPACE commands.
two phase state lock acquire	The server has waited for the short-term lock that synchronizes access to the list of prepared transactions.
wal insert lock acquire	The server has waited for the short-term lock that synchronizes write access to the write-ahead log. A high number may indicate that WAL buffers are sized too small.
wal write lock acquire	The server has waited for the short-term lock that synchronizes write-ahead log flushes.
wal file sync	The server has waited for the write-ahead log to sync to disk (related to the wal_sync_method parameter which, by default, is 'fsync' - better performance can be gained by changing this parameter to open_sync).
wal flush	The server has waited for the write-ahead log to flush to disk.
wal write	The server has waited for a write to the write-ahead log buffer (expect this value to be high).
xid gen lock acquire	The server has waited for the short-term lock that synchronizes access to the next available transaction ID.

When wait events occur for lightweight locks, they are displayed by DRITA as well. A *lightweight lock* is used to protect a particular data structure in shared memory.

Certain wait events can be due to the server process waiting for one of a group of related lightweight locks, which is referred to as a *lightweight lock tranche*. Individual lightweight lock tranches are not displayed by DRITA, but their summation is displayed by a single event named `other lwlock acquire`.

For a list and description of lightweight locks displayed by DRITA, please see Section 28.2, `The Statistics Collector` in the PostgreSQL core documentation available at:

https://www.postgresql.org/docs/10/static/monitoring-stats.html

Under Section 28.2.2. `Viewing Statistics`, the lightweight locks are listed in Table 28-4 `wait_event` Description where the `Wait Event Type` column designates `LWLockNamed`.

Note that the table entries where `Wait Event Type` designates `LWLockTranche` are the lightweight lock tranches.

The following example displays lightweight locks `ProcArrayLock`, `CLogControlLock`, `WALBufMappingLock`, and `XidGenLock`.


```
postgres=# select * from sys_rpt(40,70,20);
                                  sys_rpt
-------------------------------------------------------------------------------
 WAIT NAME                                 COUNT      WAIT TIME       % WAIT
 -----------------------------------------------------------------------------
 wal flush                                 56107      44.456494       47.65
 db file read                              66123      19.543968       20.95
 wal write                                 32886      12.780866       13.70
 wal file sync                             32933      11.792972       12.64
 query plan                                223576     4.539186        4.87
 db file extend                            2339       0.087038        0.09
 other lwlock acquire                      402        0.066591        0.07
 ProcArrayLock                             135        0.012942        0.01
 CLogControlLock                           212        0.010333        0.01
 WALBufMappingLock                         47         0.006068        0.01
 XidGenLock                                53         0.005296        0.01
(13 rows)
```

DRITA also displays wait events that occur that are related to certain Advanced Server product features.

These Advanced Server feature specific wait events and the `other lwlock acquire` event are listed in the following table.

Event Name	Description
`ICacheLock`	The server has waited for access related to Infinite Cache.
`BulkLoadLock`	The server has waited for access related to EDB*Loader.
`EDBResoureManagerLock`	The server has waited for access related to EDB Resource Manager.
`other lwlock acquire`	Summation of waits for lightweight lock tranches.

5 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.

