ora2pgpro

ora2pgpro

Chapter 1. Description

ora2pgpro is a tool used to migrate an Oracle™ database to a Postgres Pro™ compatible schema. It connects your Oracle database, scans it automatically and extracts its structure or data, then generates SQL scripts that you can load into your Postgres Pro database. ora2pgpro can be used for anything from reverse engineering of an Oracle database to huge enterprise database migration or simply replicating some Oracle data into a Postgres Pro database. It does not require any Oracle database knowledge other than providing the parameters needed to connect to the Oracle database.

Chapter 2. Features

ora2pgpro consists of a Perl script `ora2pgpro` and a Perl module `Ora2PgPro.pm`, the only thing you have to modify is the configuration file `ora2pgpro.conf` by setting the DSN to the Oracle database and optionally the name of a schema. Once that is done you just have to set the type of export you want: `TABLE` with constraints, `VIEW`, `MVIEW`, `TABLESPACE`, `SEQUENCE`, `INDEXES`, `TRIGGER`, `GRANT`, `FUNCTION`, `PROCEDURE`, `PACKAGE`, `PARTITION`, `TYPE`, `INSERT` or `COPY`, `FDW`, `QUERY`, `SYNONYM`.

By default, ora2pgpro exports to a file that you can load into Postgres Pro with the psql client, but you can also import directly into a Postgres Pro database by setting its DSN into the configuration file. With all configuration options of `ora2pgpro.conf`, you have full control of what should be exported and how. The following features are included:

- Export full database schema (tables, views, sequences, indexes), with unique, primary, foreign key, and check constraints.
- Export grants/privileges for users and groups.
- Export range/list partitions and subpartitions.
- Export a table selection (by specifying the table names).
- Export Oracle schema to a Postgres Pro schema.
- Export predefined functions, triggers, procedures, packages, and package bodies.
- Export full data or following a `WHERE` clause.
- Full support of Oracle BLOB objects as Postgres Pro bytea.
- Export Oracle views as Postgres Pro tables.
- Export Oracle user-defined types.
- Provide some basic automatic conversion of PL/SQL code to PL/pgSQL.
- Works on any platform.
- Export Oracle tables as foreign data wrapper tables.
- Export materialized views.
- Show a report of an Oracle database content.
- Migration cost assessment of an Oracle database.
- Migration difficulty level assessment of an Oracle database.
- Migration cost assessment of PL/SQL code from a file.
- Migration cost assessment of Oracle SQL queries stored in a file.
- Export Oracle locator and spatial geometries into PostGIS.
- Export `DBLINK` as Oracle FDW.
- Export `SYNONYM`s as views.
- Export `DIRECTORY` as external table or directory for `external_file` extension.
- Dispatch a list of SQL orders over multiple Postgres Pro connections.
- Perform a diff between Oracle and Postgres Pro database for test purposes.
- Full support of Oracle packages as Postgres Pro packages.
- Export VARRAY as Postgres Pro arrays.
- Export associative arrays as collections of pg_variables.

ora2pgpro does its best to convert your Oracle database to Postgres Pro automatically, but there is still manual work to do. The Oracle-specific PL/SQL code generated for functions, procedures, packages, and triggers has to be reviewed to match the Postgres Pro syntax. You can find some useful recommendations on porting Oracle PL/SQL code to Postgres Pro PL/pgSQL in Porting from Oracle PL/SQL and Migration Tools in Postgres Pro.

Chapter 3. Installation and Setup

Install Oracle Instant Client

`perl-DBD-Oracle` and `libdbd-oracle-perl` packages shipped with ora2pgpro require Oracle Instant Client Package 12.1 (e.g. `oracle-instantclient12.1-basic-12.1.0.2.0-1.x86_64.rpm`). If you use Debian, note that the `libdbd-oracle-perl` package is not shipped with ora2pgpro as it is provided in the `contrib` section of the distribution. Its version for older systems (Debian 10, 11) uses the same version of Oracle Instant Client 12.1. The `libdbd-oracle-perl` package in Debian 12 requires Oracle Instant Client 21 or above (with package name `oracle-instantclient-basic-xx.xx.x.x.x.x86_64.rpm`). If you use a Debian-based system (Ubuntu, Astra Linux), you have to convert the RPM package to a Debian package using alien, and then install it using dpkg. You can download the packages here or from a different source and follow the installation instructions.

Connect Distribution

- Download the connection script `pgpro-repo-add.sh`. You can also do it using wget:

```
  $ wget --user your-username --ask-password https://
repo.postgrespro.ru/ora2pgpro/keys/pgpro-repo-add.sh
```

- Run it with the rights of `root`. If your distribution is supported, the repository will be connected. Note that for a Debian-based system make sure to connect not only the `main` section but also the `contrib` section since it contains the required `libdbd-oracle-perl` package.

Install Package

Install the ora2pgpro package using your package manager. Depending on your system, it can be one of the following:

- `apt-get`: Debian, Ubuntu, Astra, AltLinux
- `dnf`: Red Hat Enterprise Linux (RHEL) systems and its derivatives: Red OS, Rosa
- `yum`: older Rosa versions
- `zypper`: SLES

In any case, specify the parameters for `install ora2pgpro`.

Supported Distributions

- ALT 9/10
- ALT SP 8.2/10
- Astra Linux 1.7/1.8
- Debian 10/11/12
- Red OS Murom 7.3/8
- Red Hat Enterprise Linux 8/9
- ROSA Chrome 2021.2
- SUSE Linux Enterprise Server 15
- Ubuntu 20.04/22.04/24.04

Chapter 4. Configuration

ora2pgpro configuration can be as simple as choosing the Oracle database to export and choosing the export type.

The full control of the Oracle database migration is managed in a single configuration file named `ora2pgpro.conf`. The format of this file consists of a directive name in upper case followed by a tab character and a value. Comments are lines beginning with a #. There is no specific order to place the configuration directives, they are set at the time they are read in the configuration file.

For configuration directives that just take a single value, you can use them multiple times in the configuration file but only the last occurrence found in the file will be used. For configuration directives that allow a list of values, you can use it multiple times, the values will be appended to the list. If you use the `IMPORT` directive to load a custom configuration file, directives defined in this file will be stored from the place the `IMPORT` directive is found, so it is better to put it at the end of the configuration file.

Values set in the command line options override values from the configuration file.

Usage

First of all, make sure that libraries and binaries path include the Oracle Instant Client installation:

```
export LD_LIBRARY_PATH=/usr/lib/oracle/11.2/client64/lib
export PATH="/usr/lib/oracle/11.2/client64/bin:$PATH"
```

By default, ora2pgpro looks for `/etc/ora2pgpro/ora2pgpro.conf` configuration file, if the file exists, you can simply execute:

```
/usr/local/bin/ora2pgpro
```

If you want to call another configuration file, give the path as the command-line argument:

```
/usr/local/bin/ora2pgpro -c /etc/ora2pgpro/new_ora2pgpro.conf
```

Here are all command-line parameters available when using ora2pgpro:

```
ora2pgpro [-dhpqv --estimate_cost --dump_as_html] [--option value]
```

`-a`, `--allow`

Comma-separated list of objects to allow from export. Can be used with `SHOW_COLUMN` too.

`-b`, `--basedir`

Set the default output directory where files resulting from exports will be stored.

`-c`, `--conf`

Set an alternative configuration file other than the default `/etc/ora2pgpro/ora2pgpro.conf`.

`-C`, `--cdc_file`

File used to store/read SCN per table during export. Default: `TABLES_SCN.log` in the current directory. This is the file written by the `--cdc_ready` option.

`-d`, `--debug`

Enable verbose output.

`-D`, `--data_type`

Allow custom type replacement in command line.

`-e`, `--exclude`

Comma-separated list of objects to exclude from export. Can be used with `SHOW_COLUMN` too.

`-h`, `--help`

Print this short help.

`-g`, `--grant_object`

Extract privilege for the given object type. See possible values in `GRANT_OBJECT` configuration.

`-i`, `--input`

File containing Oracle PL/SQL code to convert with no Oracle database connection initiated.

`-j`, `--jobs`

Number of parallel processes to send data to Postgres Pro.

`-J`, `--copies`

Number of parallel connections to extract data from Oracle.

`-l`, `--log`

Set a log file. Default is `stdout`.

`-L`, `--limit`

Number of tuples extracted from Oracle and stored in memory before writing, default: 10000.

`-n`, `--namespace`

Set the Oracle schema to extract from.

`-N`, `--pg_schema`

Set Postgres Pro `search_path`.

`-o`, `--out`

Set the path to the output file where SQL will be written. Default: `output.sql` in running directory.

`-p`, `--plsql`

Enable PL/SQL to PL/pgSQL code conversion.

`-P`, `--parallel`

Number of parallel tables to extract at the same time.

`-q`, `--quiet`

Disable progress bar.

`-r`, `--relative`

Use `\ir` instead of `\i` in the psql scripts generated.

`-s`, `--source`

Set the Oracle DBI data source.

`-S`, `--scn`

Set the Oracle System Change Number (SCN) to use to export data. It will be used in the `WHERE` clause to get the data. It is used with `COPY` or `INSERT`.

`-t`, `--type`

Set the export type. It will override the one given in the configuration file (`TYPE`).

`-T`, `--temp_dir`

Set a distinct temporary directory when two or more ora2pgpro instances run in parallel.

`-u`, `--user`

Set the Oracle database connection user. `ORA2PG_USER` environment variable can be used instead.

`-v`, `--version`

Show ora2pgpro version and exit.

`-w`, `--password`

Set the password of the Oracle database user. `ORA2PG_PASSWD` environment variable can be used instead.

`-W`, `--where`

Set the `WHERE` clause to apply to the Oracle query to retrieve data. Can be used multiple times.

`--forceowner`

Force ora2pgpro to set tables and sequences owner like in Oracle database. If the value is set to a username, this one will be used as the objects owner. By default it is the user used to connect to the Postgres Pro database that will be the owner.

`--nls_lang`

Set the Oracle `NLS_LANG` client encoding.

`--client_encoding`

Set the Postgres Pro client encoding.

`--view_as_table`

Comma-separated list of views to export as table.

`--estimate_cost`

Activate the migration cost evaluation with `SHOW_REPORT`.

`--cost_unit_value`

Number of minutes for a cost evaluation unit. Default: 5 minutes, corresponds to a migration conducted by a Postgres Pro expert. Set it to 10 if this is your first migration.

`--dump_as_html`

Force ora2pgpro to dump report in HTML, used only with `SHOW_REPORT`. Default is to dump report as simple text.

`--dump_as_csv`

As above but force ora2pgpro to dump report in CSV.

`--dump_as_sheet`

Report migration assessment with one CSV line per database.

`--init_project`

Initialize a typical ora2pgpro project tree. Top directory will be created under the project base directory.

`--project_base`

Define the base directory for ora2pgpro project trees. Default is the current directory.

`--print_header`

Used with `--dump_as_sheet` to print the CSV header, especially for the first run of ora2pgpro.

`--human_days_limit`

Set the number of man-days limit where the migration assessment level switches from B to C. Default is set to 5 man-days.

`--audit_user`

Comma-separated list of usernames to filter queries in the `AUDIT_USER` table. Used only with `SHOW_REPORT` and `QUERY` export type.

`--pg_dsn`

Set the data source to Postgres Pro for direct import.

`--pg_user`

Set the Postgres Pro user to be used.

`--pg_pwd`

Set the Postgres Pro password to be used.

`--count_rows`

Force ora2pgpro to perform a real row count in `TEST`, `TEST_COUNT`, and `SHOW_TABLE` actions.

`--no_header`

Do not append ora2pgpro header to output file.

`--oracle_speed`

Use to know at which speed Oracle is able to send data. No data will be processed or written.

`--ora2pg_speed`

Use to know at which speed ora2pgpro is able to send transformed data. Nothing will be written.

`--blob_to_lo`

Export BLOBs as large objects, can only be used with action `SHOW_COLUMN`, `TABLE`, and `INSERT`.

`--cdc_ready`

Use current SCN per table to export data and register them into a file named `TABLES_SCN.log` by default. It can be changed using `-C`|`--cdc_file`.

`--lo_import`

Use psql `\lo_import` command to import BLOBs as large objects. Can be used to import data with `COPY` and import the large object manually in a second pass. It is required for BLOBs larger than 1GB.

`--mview_as_table`

Comma-separated list of materialized views to export as regular tables.

`--drop_if_exists`

Drop the object before creation if it exists.

`--offline`

Convert exported data without the connection to the Oracle database.

ora2pgpro returns 0 on success, 1 on error. It returns 2 when a child process has been interrupted and you received the warning message: “WARNING: an error occurs during data export. Please check what's happen.” Most of the time this is an OOM issue, first try reducing `DATA_LIMIT` value.

Note that performance might be improved by updating stats on Oracle:

```
DBMS_STATS.GATHER_SCHEMA_STATS
```

```
DBMS_STATS.GATHER_DATABASE_STATS
DBMS_STATS.GATHER_DICTIONARY_STATS
```

The two options `--project_base` and `--init_project`, when used, indicate that ora2pgpro has to create a project template with a work tree, a configuration file and a script to export all objects from the Oracle database. Here is a sample of the command usage:

```
ora2pgpro --project_base /app/migration/ --init_project test_project
 Creating project test_project.
 /app/migration/test_project/
  schema/
   dblinks/
   directories/
   functions/
   grants/
   mviews/
   packages/
   partitions/
   procedures/
   sequences/
   synonyms/
   tables/
   tablespaces/
   triggers/
   types/
   views/
  sources/
   functions/
   mviews/
   packages/
   partitions/
   procedures/
   triggers/
   types/
   views/
  data/
  config/
  reports/

 Generating generic configuration file
 Creating script export_schema.sh to automate all exports.
 Creating script import_all.sh to automate all imports.
```

It creates a generic configuration file where you define the Oracle database connection and a shell script called `export_schema.sh`. The `sources/` directory will contain the Oracle code, the `schema/` will contain the code ported to Postgres Pro. The `reports/` directory will contain the HTML reports with the migration cost assessment.

If you want to use your own default configuration file, use the `-c` option to give the path to that file. Rename it with the `.dist` suffix if you want ora2pgpro to apply the generic configuration values, the configuration file will be copied untouched otherwise.

Once you have set the connection to the Oracle database, you can execute the script `export_schema.sh` that will export all object type from your Oracle database and output DDL files into

the schema subdirectories. At end of the export, it will give you the command to export data later when the import of the schema will be done and verified.

You can choose to load the DDL files generated manually or use the second script `import_all.sh` to import those files interactively. If this kind of migration is not currently in progress for you, it is recommended you to use those scripts.

Oracle Database Connection

The following configuration directives control the access to the Oracle database.

`ORACLE_HOME`

Used to set `ORACLE_HOME` environment variable to the Oracle libraries required by the `DBD::Oracle` Perl module.

`ORACLE_DSN`

This directive is used to set the data source name in the form of standard DBI DSN. For example:

```
dbi:Oracle:host=oradb_host.myhost.com;sid=DB_SID;port=1521
```

or

```
dbi:Oracle:DB_SID
```

On 18c, this could be for example:

```
dbi:Oracle:host=192.168.1.29;service_name=pdb1;port=1521
```

For the second notation, the SID should be declared in the file `$ORACLE_HOME/network/admin/tnsnames.ora` or in the path given to the `TNS_ADMIN` environment variable.

`ORACLE_DSN` `ORACLE_PWD`

These two directives are used to define the user and password for the Oracle database connection. Note that if you can, it is better to log in as Oracle super admin to avoid grants problem during the database scan and be sure that nothing is missing.

If you do not supply a credential with `ORACLE_PWD`, and you have installed the `Term::ReadKey` Perl module, ora2pgpro will ask for the password interactively. If `ORACLE_USER` is not set, it will be asked interactively too.

To connect to a local Oracle instance with connections as `SYSDBA`, you have to set `ORACLE_USER` to `/` and an empty password.

`USER_GRANTS`

Set this directive to 1 if you connect to the Oracle database as a simple user and do not have enough grants to extract things from the `DBA_` tables. It will use tables `ALL_` instead.

Warning: if you use the export type `GRANT`, you must set this configuration option to 0, or it will not work.

`TRANSACTION`

This directive may be used if you want to change the default isolation level of the data export transaction. Default is to set the level to a serializable transaction to ensure data consistency. The allowed values for this directive are:
- readonly: 'SET TRANSACTION READ ONLY',
- readwrite: 'SET TRANSACTION READ WRITE',
- serializable: 'SET TRANSACTION ISOLATION LEVEL SERIALIZABLE'
- committed: 'SET TRANSACTION ISOLATION LEVEL READ COMMITTED',

`ORA_INITIAL_COMMAND`

This directive can be used to send an initial command to Oracle, just after the connection, for example, to unlock a policy before reading objects or to set some session parameters. This directive can be used multiple times.

Data Encryption with Oracle Server

If your Oracle Client configuration file already includes the encryption method, then `DBD:Oracle` uses those settings to encrypt the connection while you extract the data. For example, if you have configured the Oracle Client configuration file (`sqlnet.or` or `.sqlnet`) with the following information:

```
# Configure encryption of connections to Oracle
SQLNET.ENCRYPTION_CLIENT = required
SQLNET.ENCRYPTION_TYPES_CLIENT = (AES256, RC4_256)
SQLNET.CRYPTO_SEED = 'should be 10-70 random characters'
```

Any tool that uses the Oracle client to talk to the database will be encrypted if you set up session encryption like above.

For example, Perl DBI uses `DBD-Oracle`, which uses the Oracle client for actually handling database communication. If the installation of Oracle client used by Perl is set up to request encrypted connections, then your Perl connection to the Oracle database will also be encrypted.

Testing Connection

Once you have set the Oracle database DSN, you can run ora2pgpro to see if it works:

```
ora2pgpro -t SHOW_VERSION -c config/ora2pgpro.conf
```

This will show the Oracle database server version. You can test your installation as most problems take place here, the other configuration steps are more technical.

Troubleshooting

If the `output.sql` file has not exported anything other than the Postgres Pro transaction header and footer, there are two possible reasons:

- The Perl script ora2pgpro dumps an `ORA-XXX` error: that means that your DSN or login information is wrong, check the error and your settings, and try again.
- The Perl script says nothing, and the output file is empty: the user lacks permission to extract something from the database. Try to connect to Oracle as super user or take a look at the directive `USER_GRANTS` above and in the next section, especially the `SCHEMA` directive.

LOGFILE

By default, all messages are sent to the standard output. If you give a file path to that directive, all output will be appended to this file.

Oracle Schema to Export

The Oracle database export can be limited to a specific schema or namespace, this can be mandatory following the database connection user.

SCHEMA

This directive is used to set the schema name to use during export. For example, SCHEMA APPS will extract objects associated to the APPS schema.

When no schema name is provided and EXPORT_SCHEMA is enabled, ora2pgpro will export all objects from all schemas of the Oracle instance with their names prefixed with the schema name.

EXPORT_SCHEMA

By default, the Oracle schema is not exported into the Postgres Pro database, and all objects are created under the default Postgres Pro namespace. If you also want to export this schema and create all objects under this namespace, set the EXPORT_SCHEMA directive to 1. This will set the schema search_path at the top of the export SQL file to the schema name set in the SCHEMA directive with the default pg_catalog schema. If you want to change this path, use the directive PG_SCHEMA.

CREATE_SCHEMA

Enable/disable the CREATE SCHEMA command at starting of the output file. It is enabled by default and concerns the TABLE export type.

COMPILE_SCHEMA

By default ora2pgpro will only export valid PL/SQL code. You can force Oracle to compile again the invalidated code to get a chance to have it obtain the valid status and then be able to export it.

Enable this directive to force Oracle to compile schema before exporting code. When this directive is enabled, and SCHEMA is set to a specific schema name, only invalid objects in this schema will be recompiled. If SCHEMA is not set, then all schemas will be recompiled. To force recompile invalid objects in a specific schema, set COMPILE_SCHEMA to the schema name you want to recompile.

This will ask to Oracle to validate the PL/SQL code that could have been invalidated after an export/import, for example. The VALID or INVALID status applies to functions, procedures, packages, and user-defined types. It also concerns disabled triggers.

EXPORT_INVALID

If the above configuration directive is not enough to validate your PL/SQL code, enable this configuration directive to allow export of all PL/SQL code even if it is marked as invalid. The VALID or INVALID status applies to functions, procedures, packages and user-defined types.

PG_SCHEMA

Allows you to define/force the Postgres Pro schema to use. By default, if you set EXPORT_SCHEMA to 1, the Postgres Pro search_path will be set to the schema name exported set as the value of the SCHEMA directive.

The value can be a comma-separated list of schema names but not when using the TABLE export type because in this case it will generate the CREATE SCHEMA statement, and it does not support multiple

schema names. For example, if you set PG_SCHEMA to something like user_schema, public, the search path will be set like this:

```
SET search_path = user_schema, public;
```

This forces the use of another schema (here user_schema) than the one from Oracle schema set in SCHEMA directive.

You can also set the default search_path for the Postgres Pro user you are using to connect to the destination database by using:

```
ALTER ROLE username SET search_path TO user_schema, public;
```

In this case, you do not have to set PG_SCHEMA.

SYSUSERS

Without an explicit schema, ora2pgpro will export all objects that do not belong to the system schema or role:

```
SYSTEM,CTXSYS,DBSNMP,EXFSYS,LBACSYS,MDSYS,MGMT_VIEW,
OLAPSYS,ORDDATA,OWBSYS,ORDPLUGINS,ORDSYS,OUTLN,
SI_INFORMTN_SCHEMA,SYS,SYSMAN,WK_TEST,WKSYS,WKPROXY,
WMSYS,XDB,APEX_PUBLIC_USER,DIP,FLOWS_020100,FLOWS_030000,
FLOWS_040100,FLOWS_010600,FLOWS_FILES,MDDATA,ORACLE_OCM,
SPATIAL_CSW_ADMIN_USR,SPATIAL_WFS_ADMIN_USR,XS$NULL,PERFSTAT,
SQLTXPLAIN,DMSYS,TSMSYS,WKSYS,APEX_040000,APEX_040200,
DVSYS,OJVMSYS,GSMADMIN_INTERNAL,APPQOSSYS,DVSYS,DVF,
AUDSYS,APEX_030200,MGMT_VIEW,ODM,ODM_MTR,TRACESRV,MTMSYS,
OWBSYS_AUDIT,WEBSYS,WK_PROXY,OSE$HTTP$ADMIN,
AURORA$JIS$UTILITY$,AURORA$ORB$UNAUTHENTICATED,
DBMS_PRIVILEGE_CAPTURE,CSMIG,MGDSYS,SDE,DBSFWUSER
```

Following your Oracle installation, you may have several other system roles defined. To append these users to the schema exclusion list, set the SYSUSERS configuration directive to a comma-separated list of system users to exclude. For example:

```
SYSUSERS        INTERNAL,SYSDBA,BI,HR,IX,OE,PM,SH
```

This will add users INTERNAL and SYSDBA to the schema exclusion list.

FORCE_OWNER

By default the owner of the database objects is the one you are using to connect to Postgres Pro using psql. If you use another user (postgres, for example), you can force ora2pgpro to set the object owner to be the one used in the Oracle database by setting the directive to 1, or to a completely different username by setting the directive value to that username.

FORCE_SECURITY_INVOKER

ora2pgpro use the function security privileges set in Oracle, and it is often defined as SECURITY DEFINER. If you want to override those security privileges for all functions and use SECURITY INVOKER instead, enable this directive.

`USE_TABLESPACE`

When enabled, this directive forces ora2pgpro to export all tables, indexes constraints and indexes using the tablespace name defined in Oracle database. This works only with tablespaces that are not `TEMP`, `USERS`, and `SYSTEM`.

`WITH_OID`

Activating this directive will force ora2pgpro to add `WITH OIDS` when creating tables or views as tables. Default is the same as in Postgres Pro, disabled.

`NO_FUNCTION_METADATA`

Force ora2pgpro to not look for function declaration. Note that this will prevent ora2pgpro to rewrite function replacement call if needed. Do not enable it unless looking forward at function breaks other export.

Export Type

The export action is performed following a single configuration directive `TYPE`, some others add more control on what should be really exported.

`TYPE`

Here are the different values of the `TYPE` directive, default is `TABLE`:

- `TABLE`: Extract all tables with indexes, primary keys, unique keys, foreign keys, and check constraints.
- `VIEW`: Extract only views.
- `GRANT`: Extract roles converted to Postgres Pro groups, users, and grants on all objects.
- `SEQUENCE`: Extract all sequences and their last positions.
- `TABLESPACE`: Extract storage spaces for tables and indexes.
- `TRIGGER`: Extract triggers defined following actions.
- `FUNCTION`: Extract functions.
- `PROCEDURE`: Extract procedures.
- `PACKAGE`: Extract packages and package bodies.
- `INSERT`: Extract data as `INSERT` statement.
- `COPY`: Extract data as `COPY` statement.
- `PARTITION`: Extract range and list Oracle partitions with subpartitions.
- `TYPE`: Extract user-defined Oracle type.
- `FDW`: Export Oracle tables as foreign tables for oracle_fdw.
- `MVIEW`: Export materialized views.
- `QUERY`: Try to automatically convert Oracle SQL queries.
- `DBLINK`: Generate Oracle foreign data wrapper server to use as `dblink`.
- `SYNONYM`: Export Oracle synonyms as views on other schema's objects.
- `DIRECTORY`: Export Oracle directories as `external_file` extension objects.
- `LOAD`: Dispatch a list of queries over multiple Postgres Pro connections.
- `TEST`: Perform a diff between Oracle and Postgres Pro database.
- `TEST_COUNT`: Perform a row count diff between Oracle and Postgres Pro table.
- `TEST_VIEW`: Perform a count on both sides of number of rows returned by views.
- `TEST_DATA`: Perform data validation check on rows on both sides.
- `SHOW_VERSION`: Display Oracle version.
- `SHOW_SCHEMA`: Display the list of schemas available in the database.

- SHOW_TABLE: Display the list of tables available.
- SHOW_COLUMN: Display the list of tables columns available and the ora2pgpro conversion type from Oracle to Postgres Pro that will be applied. It will also warn you if there are Postgres Pro reserved words in Oracle object names.
- SHOW_REPORT: Show a detailed report of the Oracle database content to evaluate the content of the database to migrate, in terms of objects and cost to end the migration.

Only one type of export can be performed at the same time so the TYPE directive must be unique. If you have more than one, only the last found in the file will be registered.

Some export type can not or should not be loaded directly into the Postgres Pro database and still require little manual editing. This is the case for GRANT, TABLESPACE, TRIGGER, FUNCTION, PROCEDURE, TYPE, QUERY, and PACKAGE export types, especially if you have PL/SQL code or Oracle-specific SQL in it.

For TABLESPACE, you must ensure that the file path exists in the system and for SYNONYM, ensure that the object's owners and schemas correspond to the new Postgres Pro database design.

Note that you can chain multiple export by giving to the TYPE directive a comma-separated list of export type, but in this case you must not use COPY or INSERT with other export type.

ora2pgpro will convert Oracle partition using table inheritance, trigger, and functions. For more information, see Table Partitioning.

The TYPE export allows export of user-defined Oracle types. If you do not use the `--plsql` command-line parameter, it dumps Oracle user type as-is, else ora2pgpro will try to convert it to Postgres Pro syntax.

Here is an example of the SHOW_COLUMN output:

```
[2] TABLE CURRENT_SCHEMA (1 rows) (Warning: 'CURRENT_SCHEMA' is a
 reserved word in PostgreSQL)
  CONSTRAINT : NUMBER(22) => bigint (Warning: 'CONSTRAINT' is a
 reserved word in PostgreSQL)
  FREEZE : VARCHAR2(25) => varchar(25) (Warning: 'FREEZE' is a
 reserved word in PostgreSQL)
 ...
 [6] TABLE LOCATIONS (23 rows)
  LOCATION_ID : NUMBER(4) => smallint
  STREET_ADDRESS : VARCHAR2(40) => varchar(40)
  POSTAL_CODE : VARCHAR2(12) => varchar(12)
  CITY : VARCHAR2(30) => varchar(30)
  STATE_PROVINCE : VARCHAR2(25) => varchar(25)
  COUNTRY_ID : CHAR(2) => char(2)
```

Those extraction keywords are used only to display the requested information and exit. This allows you to quickly know on what you are going to work with.

The SHOW_COLUMN allows another ora2pgpro command-line option: `--allow relname` or `-a relname` to limit the displayed information to the given table.

The SHOW_ENCODING export type will display the NLS_LANG and CLIENT_ENCODING values that ora2pgpro will use, and the real encoding of the Oracle database with the corresponding client encoding that could be used with Postgres Pro.

ora2pgpro allows you to export your Oracle table definition to be used with the oracle_fdw foreign data wrapper. By using type FDW, your Oracle tables will be exported as follows:

```
CREATE FOREIGN TABLE oratab (
  id        integer           NOT NULL,
  text      character varying(30),
  floating  double precision  NOT NULL
 ) SERVER oradb OPTIONS (table 'ORATAB');
```

Now you can use the table like a regular Postgres Pro table.

There is also a more advanced report with migration cost, see the section called “Migration Cost Assessment”.

ESTIMATE_COST

Activate the migration cost evaluation. Must only be used with SHOW_REPORT, FUNCTION, PROCEDURE, PACKAGE, and QUERY export type. Default is disabled. You may want to use the --estimate_cost command-line option instead to activate this functionality. Note that enabling this directive will force PLSQL_PGSQL activation.

COST_UNIT_VALUE

Set the value in minutes of the migration cost evaluation unit. Default is five minutes per unit. See --cost_unit_value to change the unit value in the command line.

DUMP_AS_HTML

By default, when using SHOW_REPORT, the migration report is generated as simple text, enabling this directive will force ora2pgpro to create a report in HTML format.

HUMAN_DAYS_LIMIT

Use this directive to redefine the number of man-days limit where the migration assessment level must switch from B to C. Default is set to 10 man-days.

JOBS

This configuration directive adds multiprocess support to COPY, FUNCTION, and PROCEDURE export type, the value is the number of processes to use. By default multiprocess is disabled.

This directive is used to set the number of cores to be used to parallelize data import into Postgres Pro. During FUNCTION or PROCEDURE export type each function will be translated to PL/pgSQL using a new process, the performances gain can be very important when you have many functions to convert.

There is no limitation in parallel processing other than the number of cores and the Postgres Pro I/O performance capabilities.

Doesn't work under Windows Operating System, it is simply disabled.

ORACLE_COPIES

This configuration directive adds multiprocess support to extract data from Oracle. The value is the number of processes to use to parallelize the select query. By default parallel query is disabled.

The parallelism is built on splitting the query following of the number of cores given as value to ORACLE_COPIES as follows:

```
SELECT * FROM MYTABLE WHERE ABS(MOD(COLUMN, ORACLE_COPIES)) =
 CUR_PROC
```

Here COLUMN is a technical key like a primary or unique key where split will be based and the current core used by the query (CUR_PROC). You can also force the column name to use with the DEFINED_PK configuration directive.

Doesn't work under Windows Operating System, it is simply disabled.

DEFINED_PK

This directive is used to define the technical key to be used to split the query between number of cores set with the ORACLE_COPIES variable. For example:

```
DEFINED_PK      EMPLOYEES:employee_id
```

The parallel query that will be used supposing that -J or ORACLE_COPIES is set to 8:

```
SELECT * FROM EMPLOYEES WHERE ABS(MOD(employee_id, 8)) = N
```

Here N is the current process forked starting from 0.

PARALLEL_TABLES

This directive is used to define the number of tables that will be processed in parallel for data extraction. The limit is the number of cores on your machine. ora2pgpro will open one database connection for each parallel table extraction. This directive, when more than 1, will invalidate ORACLE_COPIES but not JOBS, so the real number of process that will be used is PARALLEL_TABLES * JOBS.

Note that this directive when set more that 1 will also automatically enable the FILE_PER_TABLE directive if you are exporting to files. This is used to export tables and views in separate files.

Use PARALLEL_TABLES to use parallelism with COPY, INSERT, and TEST_DATA actions. It is also useful with TEST, TEST_COUNT, and SHOW_TABLE if --count_rows is used for real row count.

DEFAULT_PARALLELISM_DEGREE

You can force ora2pgpro to use /*+ PARALLEL(tbname, degree) */ hint in each query used to export data from Oracle by setting a value more than 1 to this directive. A value of 0 or 1 disables the use of the parallel hint. Default is disabled.

FDW_SERVER

This directive is used to set the name of the foreign data server that is used in the CREATE SERVER name FOREIGN DATA WRAPPER oracle_fdw command. This name will then be used in the CREATE FOREIGN TABLE commands and to import data using oracle_fdw. By default no foreign server is defined. This only concerns export types FDW, COPY, and INSERT. For export type FDW, the default value is orcl.

FDW_IMPORT_SCHEMA

Schema where foreign tables for data migration will be created. If you use several instances of ora2pgpro for data migration through the foreign data wrapper, you might need to change the name of the schema for each instance. Default: ora2pg_fdw_import.

DROP_FOREIGN_SCHEMA

By default ora2pgpro drops the temporary schema `ora2pg_fdw_import` used to import the Oracle foreign schema before each new import. If you want to preserve the existing schema because of modifications or the use of a third-party server, disable this directive.

EXTERNAL_TO_FDW

This directive, enabled by default, allows to export Oracle external tables as file_fdw foreign tables. To not export these tables at all, set the directive to 0.

INTERNAL_DATE_MAX

Internal timestamps retrieved from custom type are extracted in the following format: `01-JAN-77 12.00.00.000000 AM`. It is impossible to know the exact century that must be used, so by default any year below 49 will be added to 2000 and others to 1900. You can use this directive to change the default value 49. This is only relevant if you have a user-defined type with a column `timestamp`.

AUDIT_USER

Set the comma-separated list of usernames that must be used to filter queries from the `DBA_AUDIT_TRAIL` table. Default is to not scan this table and to never look for queries. This parameter is used only with `SHOW_REPORT` and `QUERY` export types with no input file for queries. Note that queries will be normalized before output unlike when a file is given at input using the `-i` option.

FUNCTION_CHECK

Disable this directive if you want to disable `check_function_bodies`.

```
SET check_function_bodies = false;
```

It disables validation of the function body string during `CREATE FUNCTION`. Default is to use the `postgresql.conf` setting that enables it by default.

ENABLE_BLOB_EXPORT

Exporting BLOB takes time, in some circumstances you may want to export all data except the BLOB columns. In this case disable this directive and the BLOB columns will not be included into data export. Take care that the target bytea column do not have a `NOT NULL` constraint.

DATA_EXPORT_ORDER

By default data export order will be done by sorting on table names. If you have huge tables at the end of alphabetic order and you are using multiprocess, it can be better to set the sort order on size so that multiple small tables can be processed before the largest tables finish. In this case set this directive to size. Possible values are `name` and `size`. Note that export type `SHOW_TABLE` and `SHOW_COLUMN` will use this sort order too, not only `COPY` or `INSERT` export type.

Limiting Objects to Export

You may want to export only a part of an Oracle database, here is a set of configuration directives that will allow you to control what parts of the database should be exported.

ALLOW

This directive allows you to set a list of objects on which the export must be limited, excluding all other objects in the same type of export. The value is a space or comma-separated list of objects names to export. You can include valid regex into the list. For example:

```
ALLOW           EMPLOYEES SALE_.* COUNTRIES .*_GEOM_SEQ
```

This will export objects with name `EMPLOYEES`, `COUNTRIES`, all objects beginning with `SALE_`, and all objects with a name ending by `_GEOM_SEQ`. The object depends on the export type. Note that regex will not work with 8i database, you must use the `%` placeholder instead, ora2pgpro will use the `LIKE` operator.

This is the manner to declare global filters that will be used with the current export type. You can also use extended filters that will be applied to specific objects or only on their related export type. For example:

```
ora2pgpro -p -c ora2pgpro.conf -t TRIGGER -a 'TABLE[employees]'
```

This will limit the export of triggers to those defined on table `employees`. If you want to extract all triggers but not some `INSTEAD OF` triggers:

```
ora2pgpro -c ora2pgpro.conf -t TRIGGER -e 'VIEW[trg_view_.*]'
```

Or a more complex form:

```
ora2pgpro -p -c ora2pgpro.conf -t TABLE -a 'TABLE[EMPLOYEES]' \
        -e 'INDEX[emp_.*];CKEY[emp_salary_min]'
```

This command will export the definition of the `employees` table but will exclude all index beginning with `emp_` and the `CHECK` constraint called `emp_salary_min`.

When exporting partition, you can exclude some partition tables by using the following command:

```
ora2pgpro -p -c ora2pgpro.conf -t PARTITION -e
 'PARTITION[PART_199.* PART_198.*]'
```

This will exclude partitioned tables for years 1980 to 1999 from the export but not the main partition table. The trigger will also be adapted to exclude those tables.

With `GRANT` export, you can use this extended form to exclude some users from the export, or limit the export to some others:

```
ora2pgpro -p -c ora2pgpro.conf -t GRANT -a 'USER1 USER2'
or
ora2pgpro -p -c ora2pgpro.conf -t GRANT -a 'GRANT[USER1 USER2]'
```

The latter will limit export grants to users `USER1` and `USER2`. But if you do not want to export grants on some functions for these users, for example:

```
ora2pgpro -p -c ora2pgpro.conf -t GRANT -a 'USER1 USER2' -e
 'FUNCTION[adm_.*];PROCEDURE[adm_.*]'
```

Oracle does not allow the use of look-ahead expression so you may want to exclude some objects that match the `ALLOW` regexp you have defined. For example, if you want to export all table starting with

E but not those starting with EXP, it is not possible to do that in a single expression. This is why you can start a regular expression with the ! character to exclude object matching the regexp given just after. The previous example can be written as follows:

```
ALLOW   E.* !EXP.*
```

It will be translated into the following in the object search expression:

```
REGEXP_LIKE(..., '^E.*$') AND NOT REGEXP_LIKE(..., '^EXP.*$')
```

EXCLUDE

This directive is the opposite of the previous one, it allows you to define a space or comma-separated list of object name to exclude from the export. You can include a valid regular expression into the list. For example:

```
EXCLUDE          EMPLOYEES TMP_.* COUNTRIES
```

This will exclude objects with names EMPLOYEES, COUNTRIES, and all tables beginning with tmp_.

For example, you can ban from export unwanted functions with this directive:

```
EXCLUDE          write_to_.* send_mail_.*
```

This example will exclude all functions, procedures, or functions in a package with the name beginning with those regex. Note that regex will not work with 8i database, you must use the % placeholder instead, ora2pgpro will use the NOT LIKE operator. See above (directive ALLOW) for the extended syntax.

NO_EXCLUDED_TABLE

By default ora2pgpro excludes from export some Oracle “garbage” tables that should never be part of an export. This behavior generates a lot of REGEXP_LIKE expressions which are slowing down the export when looking at tables. To disable this behavior, enable this directive, but you will have to exclude or clean up later by yourself the unwanted tables. The regexp used to exclude the table are defined in the array @EXCLUDED_TABLES in lib/Ora2Pgpro.pm. Note that this behavior is independent to the EXCLUDE configuration directive.

VIEW_AS_TABLE

Set which view to export as a table. By default none. The value must be a list of view names or regexp separated by space or comma. If the object name is a view and the export type is TABLE, the view will be exported as a CREATE TABLE statement. If export type is COPY or INSERT, the corresponding data will be exported. See Exporting Views as Postgres Pro Tables for more details.

MVIEW_AS_TABLE

Set which materialized view to export as a table. By default none. Value must be a list of materialized view names or regexp separated by space or comma. If the object name is a materialized view and the export type is TABLE, the view will be exported as a CREATE TABLE statement. If export type is COPY or INSERT, the corresponding data will be exported.

NO_VIEW_ORDERING

By default ora2pgpro tries to order views to avoid error at the import time with nested views. With a huge number of views, this can take a very long time, you can bypass this ordering by enabling this directive.

GRANT_OBJECT

When exporting grants, you can specify a comma-separated list of objects for which privileges will be exported. Default is export for all objects. Here are the possibles values TABLE, VIEW, MATERIALIZED VIEW, SEQUENCE, PROCEDURE, FUNCTION, PACKAGE BODY, TYPE, SYNONYM, DIRECTORY. Only one object type is allowed at a time. For example, set it to TABLE if you just want to export privileges on tables. You can use the -g option to overwrite it.

WHERE

This directive allows you to specify a WHERE clause filter when dumping the contents of tables. Value constructs as follows: TABLE_NAME[WHERE_CLAUSE], or if you have only one where clause for each table just put the where clause as the value. Both are possible too. Here are some examples:

```
# Global where clause applying to all tables included in the export
WHERE  1=1

# Apply the where clause only on table TABLE_NAME
WHERE  TABLE_NAME[ID1='001']

# Applies two different clause on tables TABLE_NAME and OTHER_TABLE
# and a generic where clause on DATE_CREATE to all other tables
WHERE  TABLE_NAME[ID1='001' OR ID1='002] DATE_CREATE > '2001-01-01'
 OTHER_TABLE[NAME='test']
```

Any where clause not included into a table name bracket clause will be applied to all exported table including the tables defined in the WHERE clause. These WHERE clauses are very useful if you want to archive some data or to the opposite only export some recent data.

To be able to test data import quickly, it is useful to limit data export to the first thousand tuples of each table. For Oracle, define the following clause:

```
WHERE   ROWNUM < 1000
```

This can also be restricted to some tables data export.

The command-line option -W or --where will override this directive for the global part and per table if the table names are the same.

TOP_MAX

This directive is used to limit the number of items shown in the top N lists like the top list of tables per number of rows and the top list of largest tables in megabytes. By default it is set to 10 items.

LOG_ON_ERROR

Enable this directive if you want to continue direct data import on error. When ora2pgpro receives an error in the COPY or INSERT statement from Postgres Pro, it will log the statement to a file called

TABLENAME_error.log in the output directory and continue to the next bulk of data. Like this you can try to fix the statement and manually reload the error log file. Default is disabled: abort import on error.

REPLACE_QUERY

Sometimes you may want to extract data from an Oracle table but you need a custom query for that. Not just a SELECT * FROM table like ora2pgpro does but a more complex query. This directive allows you to overwrite the query used by ora2pgpro to extract data. The format is TABLENAME[SQL_QUERY]. If you have multiple table to extract by replacing the ora2pgpro query, you can define multiple REPLACE_QUERY lines.

```
REPLACE_QUERY   EMPLOYEES[SELECT e.id,e.fisrtname,lastname FROM
 EMPLOYEES e JOIN EMP_UPDT u ON (e.id=u.id AND u.cdate>'2014-08-01
 00:00:00')]
```

Control of Full Text Search Export

Several directives can be used to control the way ora2pgpro will export the Oracle text search indexes. By default CONTEXT indexes will be exported to Postgres Pro FTS indexes but CTXCAT indexes will be exported as indexes using the pg_trgm extension.

CONTEXT_AS_TRGM

Force ora2pgpro to translate Oracle text indexes into Postgres Pro indexes using the pg_trgm extension. Default is to translate CONTEXT indexes into FTS indexes and CTXCAT indexes using pg_trgm. Most of the time using pg_trgm is enough, this is what this directive is for. You need to create the pg_trgm extension into the destination database before importing the objects.

FTS_INDEX_ONLY

By default, ora2pgpro creates a function-based index to translate Oracle text indexes.

```
CREATE INDEX ON t_document
  USING gin(to_tsvector('pg_catalog.french', title));
```

You will have to rewrite the CONTAIN() clause using to_tsvector(), for example:

```
SELECT id,title FROM t_document
  WHERE to_tsvector(title) @@ to_tsquery('search_word');
```

To force ora2pgpro to create an extra tsvector column with dedicated triggers for FTS indexes, disable this directive. In this case, ora2pgpro will add the column as follows:

```
ALTER TABLE t_document ADD COLUMN tsv_title tsvector;
```

Then update the column to compute FTS vectors if the data have been loaded before UPDATE t_document SET tsv_title = to_tsvector('pg_catalog.french', coalesce(title,''));. To automatically update the column when a modification in the title column appears, ora2pgpro adds the following trigger:

```
CREATE FUNCTION tsv_t_document_title() RETURNS trigger AS $$
BEGIN
       IF TG_OP = 'INSERT' OR new.title != old.title THEN
               new.tsv_title :=
               to_tsvector('pg_catalog.french',
 coalesce(new.title,''));
       END IF;
       return new;
END
$$ LANGUAGE plpgsql;
CREATE TRIGGER trig_tsv_t_document_title BEFORE INSERT OR UPDATE
 ON t_document
 FOR EACH ROW EXECUTE PROCEDURE tsv_t_document_title();
```

When the Oracle text index is defined over multiple columns, ora2pgpro will use `setweight()` to set a weight in the order of the column declaration.

`FTS_CONFIG`

Use this directive to force text search configuration to use. When it is not set, ora2pgpro will autodetect the stemmer used by Oracle for each index and `pg_catalog.english` if the information is not found.

`USE_UNACCENT`

If you want to perform your text search in an accent-insensitive way, enable this directive. ora2pgpro will create a helper function over `unaccent()` and the pg_trgm indexes using this function. With FTS, ora2pgpro will redefine your text search configuration, for example:

```
CREATE TEXT SEARCH CONFIGURATION fr (COPY = french);
ALTER TEXT SEARCH CONFIGURATION fr
        ALTER MAPPING FOR hword, hword_part, word WITH unaccent,
 french_stem;
```

Then set the `FTS_CONFIG` `ora2pgpro.conf` directive to `fr` instead of `pg_catalog.english`.

When enabled, ora2pgpro will create the wrapper function:

```
CREATE OR REPLACE FUNCTION unaccent_immutable(text)
RETURNS text AS
$$
    SELECT public.unaccent('public.unaccent', $1);
$$ LANGUAGE sql IMMUTABLE
   COST 1;
```

The indexes are exported as follows:

```
CREATE INDEX t_document_title_unaccent_trgm_idx ON t_document
    USING gin (unaccent_immutable(title) gin_trgm_ops);
```

In your queries, you will need to use the same function in the search to be able to use the function-based index. Example:

```
SELECT * FROM t_document
        WHERE unaccent_immutable(title) LIKE '%donnees%';
```

USE_LOWER_UNACCENT

Same as above but call `lower()` in the `unaccent_immutable()` function:

```
CREATE OR REPLACE FUNCTION unaccent_immutable(text)
RETURNS text AS
$$
    SELECT lower(public.unaccent('public.unaccent', $1));
$$ LANGUAGE sql IMMUTABLE;
```

Modifying Object Structure

One of the main advantages of ora2pgpro is its flexibility to replicate Oracle database into Postgres Pro database with a different structure or schema. There are configuration directives that allow you to map those differences.

REORDERING_COLUMNS

Enable this directive to reorder columns and minimize the footprint on disc, so that more rows fit on a data page, which is the most important factor for speed. Default is `disabled`, which means that the same order as in Oracle tables definition is used, that should be enough for most usages. This directive is only used with `TABLE` export.

MODIFY_STRUCT

This directive allows you to limit the columns extracted for a given table. The value consists of a space-separated list of table names with a set of columns in parenthesis as follows: `MODIFY_STRUCT NOM_TABLE(nomcol1,nomcol2,...) ....` For example:

```
MODIFY_STRUCT   T_TEST1(id,dossier) T_TEST2(id,fichier)
```

This will only extract the columns `id` and `dossier` from the table `T_TEST1` and columns `id` and `fichier` from the table `T_TEST2`. This directive can only be used with `TABLE`, `COPY`, or `INSERT` export. With `TABLE` export, the `CREATE TABLE` DDL statement will respect the new list of columns, and all indexes or foreign keys pointing to or from a column removed will not be exported.

EXCLUDE_COLUMNS

Instead of redefining the table structure with `MODIFY_STRUCT`, you may want to exclude some columns from the table export. The value consists of a space-separated list of table names with a set of columns in parenthesis as follows: `EXCLUDE_COLUMNS NOM_TABLE(nomcol1,nomcol2,...) ....` For example:

```
EXCLUDE_COLUMNS T_TEST1(id,dossier) T_TEST2(id,fichier)
```

This will exclude from the export the columns `id` and `dossier` from the table `T_TEST1` and columns `id` and `fichier` from the table `T_TEST2`. This directive can only be used with `TABLE`, `COPY`, or `INSERT` export. With `TABLE` export, the `CREATE TABLE` DDL statement will respect

the new list of columns, and all indexes or foreign key pointing to or from a column removed will not be exported.

REPLACE_TABLES

This directive allows you to remap a list of Oracle table names to Postgres Pro table names during export. The value is a list of space-separated values with the following structure:

```
REPLACE_TABLES  ORIG_TBNAME1:DEST_TBNAME1 ORIG_TBNAME2:DEST_TBNAME2
```

Oracle tables ORIG_TBNAME1 and ORIG_TBNAME2 will be respectively renamed to DEST_TBNAME1 and DEST_TBNAME2.

REPLACE_COLS

Like table names, the names of the columns can be remapped to different ones using the following syntax: REPLACE_COLS ORIG_TBNAME(ORIG_COLNAME1:NEW_COLNAME1,ORIG_COLNAME2:NEW_COLNAME2). For example:

```
REPLACE_COLS    T_TEST(dico:dictionary,dossier:folder)
```

This renames the Oracle columns dico and dossier from table T_TEST to dictionary and folder.

REPLACE_AS_BOOLEAN

If you want to change the type of some Oracle columns to Postgres Pro boolean during the export, you can define here a list of tables and columns separated by space as follows:

```
REPLACE_AS_BOOLEAN      TB_NAME1:COL_NAME1 TB_NAME1:COL_NAME2
 TB_NAME2:COL_NAME2
```

The values set in the boolean columns list will be replaced with the t and f following the default replacement values and those additionally set in directive BOOLEAN_VALUES.

Note that if you have modified the table name with REPLACE_TABLES and/or the column name, you need to use the name of the original table and/or column.

```
REPLACE_COLS            TB_NAME1(OLD_COL_NAME1:NEW_COL_NAME1)
REPLACE_AS_BOOLEAN      TB_NAME1:OLD_COL_NAME1
```

You can also give a type and precision to convert all fields of that type as a boolean automatically. For example:

```
REPLACE_AS_BOOLEAN      NUMBER:1 CHAR:1 TB_NAME1:COL_NAME1
 TB_NAME1:COL_NAME2
```

This will also replace any field of type number(1) or char(1) as a boolean in all exported tables.

BOOLEAN_VALUES

Use this to add additional definition of the possible boolean values used in Oracle fields. You must set a space-separated list of TRUE:FALSE values. By default here are the values recognized by ora2pgpro:

```
BOOLEAN_VALUES          yes:no y:n 1:0 true:false enabled:disabled
```

Any values defined here will be added to the default list.

REPLACE_ZERO_DATE

When ora2pgpro finds a “zero” date `0000-00-00 00:00:00`, it is replaced by NULL. This could be a problem if your column is defined with the `NOT NULL` constraint. If you can not remove the constraint, use this directive to set an arbitral date that will be used instead. You can also use `-INFINITY` if you do not want to use a fake date.

INDEXES_SUFFIX

Add the given value as suffix to index names. Useful, if you have indexes with same name as tables. For example:

```
INDEXES_SUFFIX          _idx
```

This will add `_idx` at the end of all index names. Not so common but can help.

INDEXES_RENAMING

Enable this directive to rename all indexes in the following format: `tablename_columns_names`. Could be very useful for databases that have the same index name multiple times or that use the same name as a table, which is not allowed by Postgres Pro. Disabled by default.

USE_INDEX_OPCLASS

Operator classes `text_pattern_ops`, `varchar_pattern_ops`, and `bpchar_pattern_ops` support B-tree indexes on the corresponding types. The difference from the default operator classes is that the values are compared strictly character by character rather than according to the locale-specific collation rules. This makes these operator classes suitable for use by queries involving pattern-matching expressions (LIKE or POSIX regular expressions) when the database does not use the standard `C` locale. If you enable, with value 1, this will force ora2pgpro to export all indexes defined on varchar2() and char() columns using those operators. If you set it to a value greater than 1, it will only change indexes on columns where the character limit is greater or equal than this value. For example, set it to 128 to create these kind of indexes on columns of type varchar2(N) where N >= 128.

RENAME_PARTITION

Enable this directive if you want that your partition tables will be renamed. Disabled by default. If you have multiple partitioned tables, when exported to Postgres Pro some partitions could have the same name but different parent tables. This is not allowed, a table name must be unique, in this case enable this directive. The following rules are applied by default:

- Partition: "tablename"_part"pos" where "pos" is the partition number.
- Subpartition: "tablename"_part"pos"_subpart"pos".
- Partition/subpartition: "tablename"_part_default "tablename"_part"pos"_subpart_default.

DISABLE_PARTITION

If you do not want to reproduce the partitioning like in Oracle and want to export all partitioned Oracle data into the main single table in Postgres Pro, enable this directive. ora2pgpro will export all data into the main table. Default is to use partitioning, ora2pgpro will export data from each partition and import them into the Postgres Pro dedicated partition table.

DISABLE_UNLOGGED

By default ora2pgpro exports Oracle tables with the NOLOGGING attribute as UNLOGGED tables. You may want to fully disable this feature because you will lose all data from unlogged tables in case of a Postgres Pro crash. Set it to 1 to export all tables as normal tables.

Oracle Spatial to PostGIS

ora2pgpro fully exports spatial objects from Oracle databases. There are some configuration directives that could be used to control the export.

AUTODETECT_SPATIAL_TYPE

By default, ora2pgpro is looking at indexes to see the spatial constraint type and dimensions defined under Oracle. Those constraints are passed at index creation using for example:

```
CREATE INDEX ... INDEXTYPE IS MDSYS.SPATIAL_INDEX
PARAMETERS('sdo_indx_dims=2, layer_gtype=point');
```

If those Oracle constraints parameters are not set, the default is to export those columns as generic type GEOMETRY to be able to receive any spatial type.

The AUTODETECT_SPATIAL_TYPE directive allows to force ora2pgpro to autodetect the real spatial type and dimensions used in a spatial column otherwise a non-constrained geometry type is used. Enabling this feature will force ora2pgpro to scan a sample of 50000 column to look at the GTYPE used. You can increase or reduce the sample size by setting the value of AUTODETECT_SPATIAL_TYPE to the desired number of line to scan. The directive is enabled by default.

For example, in the case of a column named shape and defined with Oracle type SDO_GEOMETRY, with AUTODETECT_SPATIAL_TYPE disabled, it will be converted as:

```
shape geometry(GEOMETRY) or shape geometry(GEOMETRYZ, 4326)
```

If the directive is enabled and the column just contains a single geometry type that uses a single dimension with a two or three dimensional polygon:

```
shape geometry(POLYGON, 4326) or shape geometry(POLYGONZ, 4326)
```

CONVERT_SRID

This directive allows you to control the automatic conversion of Oracle SRID to standard EPSG. If enabled, ora2pgpro will use the Oracle function sdo_cs.map_oracle_srid_to_epsg() to convert all SRIDs. Enabled by default.

If the SDO_SRID returned by Oracle is NULL, it will be replaced by the default value 8307 converted to its EPSG value: 4326 (see DEFAULT_SRID).

If the value is more than 1, all SRIDs will be forced to this value, in this case DEFAULT_SRID will not be used when Oracle returns a null value and the value will be forced to CONVERT_SRID.

Note that it is also possible to set the EPSG value on Oracle side when sdo_cs.map_oracle_srid_to_epsg() returns NULL if your want to force the value:

```
 system@db> UPDATE sdo_coord_ref_sys SET legacy_code=41014 WHERE
srid = 27572;
```

DEFAULT_SRID

Use this directive to override the default EPSG SRID to be used: 4326. Can be overwritten by CONVERT_SRID, see above.

GEOMETRY_EXTRACT_TYPE

This directive can take three values: WKT (default), WKB, and INTERNAL. When it is set to WKT, ora2pgpro will use SDO_UTIL.TO_WKTGEOMETRY() to extract the geometry data. When it is set to WKB, ora2pgpro will use the binary output using SDO_UTIL.TO_WKBGEOMETRY(). If those two extract types are calls at Oracle side, they are slow and you can easily reach Out Of Memory when you have a lot of rows. Also WKB is not able to export 3D geometry and some geometries like CURVEPOLYGON. In this case you may use the INTERNAL extraction type. It will use a Pure Perl library to convert the SDO_GEOMETRY data into a WKT representation, the translation is done on ora2pgpro side. This is a work in progress, validate your exported data geometries before use. Default spatial object extraction type is INTERNAL.

POSTGIS_SCHEMA

Use this directive to add a specific schema to the search path to look for PostGIS functions.

ST_SRID_FUNCTION

Oracle function to use to extract the SRID from ST_Geometry meta information. Default: ST_SRID, for example, it should be set to sde.st_srid for ArcSDE.

ST_DIMENSION_FUNCTION

Oracle function to use to extract the dimension from ST_Geometry meta information. Default: ST_DIMENSION, for example it should be set to sde.st_dimention for ArcSDE.

ST_GEOMETRYTYPE_FUNCTION

Oracle function to use to extract the geometry type from a ST_Geometry column. Default: ST_GEOMETRYTYPE, for example it should be set to sde.st_geometrytype for ArcSDE.

ST_ASBINARY_FUNCTION

Oracle function to be used to convert an ST_Geometry value into WKB format. Default: ST_ASBINARY, for example it should be set to sde.st_asbinary for ArcSDE.

ST_ASTEXT_FUNCTION

Oracle function to be used to convert an ST_Geometry value into WKT format. Default: ST_ASTEXT, for example it should be set to sde.st_astext for ArcSDE.

Postgres Pro Import

By default conversion to Postgres Pro format is written to file output.sql.

```
psql mydb < output.sql
```

This command will import content of the file output.sql into Postgres Pro mydb database.

DATA_LIMIT

When you are performing INSERT/COPY export, ora2pgpro proceeds by chunks of DATA_LIMIT tuples for speed improvement. Tuples are stored in memory before being written to disk, so if you want speed and have enough system resources you can raise this limit to an higher value for example: 100000 or 1000000. A value of 0 means that the chunk will be set to the default: 10000.

BLOB_LIMIT

When ora2pgpro detects a table with some BLOB, it will automatically reduce the value of this directive by dividing it by 10 until its value is below 1000. You can control this value by setting BLOB_LIMIT. Exporting BLOB use lot of resources, setting it to a too high value can produce OOM.

OUTPUT

The ora2pgpro output filename can be changed with this directive. Default value is output.sql. If you set the file name with extension .gz or .bz2, the output will be automatically compressed. This requires that the Compress::Zlib Perl module is installed if the filename extension is .gz and that the bzip2 system command is installed for the .bz2 extension.

OUTPUT_DIR

You can define a base directory where the file will be written. The directory must exist.

BZIP2

This directive allows you to specify the full path to the bzip2 program if it can not be found in the PATH environment variable.

FILE_PER_CONSTRAINT

Allow object constraints to be saved in a separate file during schema export. The file will be named CONSTRAINTS_OUTPUT, where OUTPUT is the value of the corresponding configuration directive. You can use .gz or .bz2 extension to enable compression. Default is to save all data in the OUTPUT file. This directive is usable only with TABLE export type.

The constraints can be imported quickly into Postgres Pro using the LOAD export type to parallelize their creation over multiple (-j or JOBS) connections.

FILE_PER_INDEX

Allow indexes to be saved in a separate file during schema export. The file will be named INDEXES_OUTPUT, where OUTPUT is the value of the corresponding configuration directive. You can use .gz or .bz2 file extension to enable compression. Default is to save all data in the OUTPUT file. This directive is usable only with TABLE AND TABLESPACE export type. With the TABLESPACE export, it is used to write ALTER INDEX ... TABLESPACE ... into a separate file named TBSP_INDEXES_OUTPUT that can be loaded at end of the migration after the indexes creation to move the indexes.

The indexes can be imported quickly into Postgres Pro using the LOAD export type to parallelize their creation over multiple (-j or JOBS) connections.

FILE_PER_FKEYS

Allow foreign key declaration to be saved in a separate file during schema export. By default foreign keys are exported into the main output file or in the CONSTRAINT_output.sql file. When enabled, foreign keys will be exported into a file named FKEYS_output.sql.

FILE_PER_TABLE

Allow data export to be saved in one file per table/view. The files will be named as table-name_OUTPUT, where OUTPUT is the value of the corresponding configuration directive. You can still use .gz or .bz2 extension in the OUTPUT directive to enable compression. Default 0 will save all data in one file, set it to 1 to enable this feature. This is usable only during INSERT or COPY export type.

FILE_PER_FUNCTION

Allow functions, procedures and triggers to be saved in one file per object. The files will be named as objectname_OUTPUT, where OUTPUT is the value of the corresponding configuration directive. You can still use .gz or .bz2 extension in the OUTPUT directive to enable compression. Default 0 will save all in one single file, set it to 1 to enable this feature. This is usable only during the corresponding export type, the package body export has a special behavior.

When export type is PACKAGE, and you enabled this directive, ora2pgpro will create a directory per package, named with the lower-case name of the package, and will create one file per function/procedure into that directory. If the configuration directive is not enabled, it will create one file per package as packagename_OUTPUT, where OUTPUT is the value of the corresponding directive.

TRUNCATE_TABLE

If this directive is set to 1, a TRUNCATE TABLE instruction will be added before loading data. This is usable only during INSERT or COPY export type.

When activated, the instruction will be added only if there is no global DELETE clause or not one specific to the current table (see below).

DELETE

Support for include a DELETE FROM ... WHERE clause filter before importing data and perform a deletion of some lines instead of truncating tables. Value is constructed as follows: TABLE_NAME[DELETE_WHERE_CLAUSE], or if you have only one WHERE clause for all tables just put the DELETE clause as a single value. Both are possible too. Here are some examples:

```
DELETE  1=1    # Apply to all tables and delete all tuples
DELETE  TABLE_TEST[ID1='001']   # Apply only on table TABLE_TEST
DELETE  TABLE_TEST[ID1='001' OR ID1='002] DATE_CREATE >
 '2001-01-01' TABLE_INFO[NAME='test']
```

The last applies two different delete where clause on tables TABLE_TEST and TABLE_INFO and a generic DELETE clause on DATE_CREATE to all other tables. If TRUNCATE_TABLE is enabled, it will be applied to all tables not covered by the DELETE definition. These DELETE clauses might be useful with regular updates.

STOP_ON_ERROR

Set this parameter to 0 to not include the call to \set ON_ERROR_STOP ON in all SQL scripts generated by ora2pgpro. By default this order is always present so that the script will immediately abort when an error is encountered.

COPY_FREEZE

Enable this directive to use COPY FREEZE instead of a simple COPY to export data with rows already frozen. This is intended as a performance option for initial data loading. Rows will be frozen only

if the table being loaded has been created or truncated in the current sub-transaction. This will only work with export to file and when `-J` or `ORACLE_COPIES` is not set or defaults to 1. It can be used with direct import into Postgres Pro under the same condition, but `-j` or `JOBS` must also be unset or default to 1.

`CREATE_OR_REPLACE`

By default ora2pgpro uses `CREATE OR REPLACE` in functions and views DDL, if you need not to override existing functions or views disable this configuration directive, DDL will not include `OR REPLACE`.

`DROP_IF_EXISTS`

To add a `DROP` *`OBJECT`* `IF EXISTS` before creating the object, enable this directive. Can be useful in an iterative work. Default is disabled.

`EXPORT_GTT`

Postgres Pro does not support Global Temporary Table natively but you can use the pgtt extension to emulate this behavior. Enable this directive to export global temporary table.

`NO_HEADER`

Enabling this directive will prevent ora2pgpro to print its header into output files. Only the translated code will be written.

`PSQL_RELATIVE_PATH`

By default ora2pgpro use `\i` psql command to execute generated SQL files if you want to use a relative path following the script execution file enabling this option will use `\ir`. See psql help for more information.

`DATA_VALIDATION_ROWS`

Number of rows that must be retrieved on both sides for data validation. Default it to compare the 10000 first rows. A value of 0 means to compare all rows.

`DATA_VALIDATION_ORDERING`

Order of rows between both sides is different once the data have been modified. In this case data must be ordered using a primary key or a unique index, that means that a table without such objects can not be compared. If the validation is done just after the data migration without any data modification, the validation can be done on all tables without any ordering.

`DATA_VALIDATION_ERROR`

Stop validating data from a table after a certain amount of row mismatch. Default is to stop after 10 rows validation errors.

`TRANSFORM_VALUE`

Use this directive to specify which transformation should be applied to a column when exporting data. Value must be a semicolon-separated list of the following:

`TABLE[COLUMN_NAME,` *`code in SELECT target list`*`]`

For example, to replace string “Oracle” by “PostgreSQL” in a varchar2 column, use the following:

```
TRANSFORM_VALUE
 ERROR_LOG_SAMPLE[DBMS_TYPE:regexp_replace("DBMS_TYPE",'Oracle','PostgreSQL')]
```

To replace all Oracle char(0) in a string with a space character:

```
TRANSFORM_VALUE   CLOB_TABLE[CHARDATA:translate("CHARDATA", chr(0),
 ' ')]
```

The expression will be applied in the SQL statement used to extract data from the source database.

When using ora2pgpro the export type `INSERT` or `COPY` to dump data to the file and that `FILE_PER_TABLE` is enabled, you will be warned that ora2pgpro will not export data again if the file already exists. This is to prevent downloading twice the table with a huge amount of data. To force the download of data from these tables, you have to remove the existing output file first.

If you want to import data on the fly to the Postgres Pro database, you have three configuration directives to set the Postgres Pro database connection. This is only possible with `COPY` or `INSERT` export type as for the database schema there is no real interest to do that.

`PG_DSN`

Use this directive to set the Postgres Pro data source namespace using `DBD::Pg` Perl module as follows:

```
dbi:Pg:dbname=pgdb;host=localhost;port=5432
```

It will connect to the database `pgdb` on `localhost` at TCP port 5432.

Note that this directive is only used for data export, other export needs to be imported manually through the use of psql or any other Postgres Pro client.

To use SSL-encrypted connection, you must add `sslmode=require` to the connection string like this:

```
dbi:Pg:dbname=pgdb;host=localhost;port=5432;sslmode=require
```

`PG_USER`, `PG_PWD`

These two directives are used to set the login user and password. If you do not supply a credential with `PG_PWD`, and you have installed the `Term::ReadKey` Perl module, ora2pgpro will ask for the password interactively. If `PG_USER` is not set, it will be asked interactively too.

`SYNCHRONOUS_COMMIT`

Specifies whether the transaction commit will wait for WAL records to be written to disk before the command returns a success indication to the client. This is the equivalent to setting `synchronous_commit` directive of the `postgresql.conf` file. This is only used when you load data directly to Postgres Pro, the default is off to disable the synchronous commit to gain speed at writing data.

PG_INITIAL_COMMAND

This directive can be used to send an initial command to Postgres Pro, just after the connection, for example, to set some session parameters. This directive can be used multiple times.

Column Type Control

PG_NUMERIC_TYPE

If set to 1, replace portable numeric types with Postgres Pro internal types. Oracle data type NUMBER(p,s) is approximatively converted to real and float Postgres Pro data types. If you have monetary fields or do not want rounding issues with the extra decimals, you should preserve the same numeric(p,s) Postgres Pro data type. Do that only if you need exactness because using numeric(p,s) is slower than using real or double.

PG_INTEGER_TYPE

If set to 1, replace portable numeric type into Postgres Pro internal type. Oracle data type NUMBER(p) or NUMBER are converted to a smallint, integer, or bigint Postgres Pro data type following the value of the precision. NUMBER without precision is set to DEFAULT_NUMERIC (see below).

DEFAULT_NUMERIC

NUMBER without precision is converted by default to bigint only if PG_INTEGER_TYPE is true. You can overwrite this value to any Postgres Pro type, like integer or float.

DATA_TYPE

If you are experiencing any problem in data type schema conversion with this directive, you can take full control of the correspondence between Oracle and Postgres Pro types to redefine data type translation used in ora2pgpro. The syntax is a comma-separated list of `Oracle datatype:Postgres Pro datatype`. Here is the default list used:

```
DATA_TYPE
 VARCHAR2:varchar,NVARCHAR2:varchar,NVARCHAR:varchar,NCHAR:char,DATE:timestamp(
 RAW:bytea,CLOB:text,NCLOB:text,BLOB:bytea,BFILE:bytea,RAW(16):uuid,RAW(32):uui
 precision,DEC:decimal,DECIMAL:decimal,DOUBLE PRECISION:double
 precision,INT:integer,INTEGER:integer,REAL:real,SMALLINT:smallint,BINARY_FLOAT
 precision,BINARY_DOUBLE:double
 precision,TIMESTAMP:timestamp,XMLTYPE:xml,BINARY_INTEGER:integer,PLS_INTEGER:i
 WITH TIME ZONE:timestamp with time zone,TIMESTAMP WITH LOCAL TIME
 ZONE:timestamp with time zone
```

The directive and the list definition must be a single line.

Note that when RAW(16) and RAW(32) columns is found or that the RAW column has SYS_GUID() as the default value, ora2pgpro will automatically translate the type of the column into uuid, which might be the right translation in most of cases. In this case data will be automatically migrated as Postgres Pro uuid data type provided by the uuid-ossp extension.

If you want to replace a type with a precision and scale, you need to escape the comma with a backslash. For example, if you want to replace all NUMBER(*,0) into bigint instead of numeric(38), add the following:

```
DATA_TYPE       NUMBER(*\,0):bigint
```

You do not have to recopy all default type conversion but just the one you want to rewrite.

There is a special case with BFILE when they are converted to type TEXT, they will just contain the full path to the external file. If you set the destination type to BYTEA, the default, ora2pgpro will export the content of the BFILE as bytea. The third case is when you set the destination type to EFILE, in this case, ora2pgpro will export it as an EFILE record: (DIRECTORY, FILENAME). Use the `DIRECTORY` export type to export the existing directories as well as the privileges on those directories.

There is no SQL function available to retrieve the path to BFILE. ora2pgpro has to create one using the `DBMS_LOB` package.

```
CREATE OR REPLACE FUNCTION ora2pg_get_bfilename( p_bfile IN BFILE )
RETURN VARCHAR2
AS
    l_dir   VARCHAR2(4000);
    l_fname VARCHAR2(4000);
    l_path  VARCHAR2(4000);
BEGIN
    dbms_lob.FILEGETNAME( p_bfile, l_dir, l_fname );
    SELECT directory_path INTO l_path FROM all_directories
        WHERE directory_name = l_dir;
    l_dir := rtrim(l_path,'/');
    RETURN l_dir || '/' || l_fname;
END;
```

This function is only created if ora2pgpro found a table with a BFILE column and that the destination type is TEXT. The function is dropped at the end of export. This concerns both `COPY` and `INSERT` export types.

There is no SQL function available to retrieve BFILE as an EFILE record, then ora2pgpro have to create one using the `DBMS_LOB` package.

```
CREATE OR REPLACE FUNCTION ora2pg_get_efile( p_bfile IN BFILE )
RETURN VARCHAR2
AS
    l_dir   VARCHAR2(4000);
    l_fname VARCHAR2(4000);
BEGIN
    dbms_lob.FILEGETNAME( p_bfile, l_dir, l_fname );
    RETURN '(' || l_dir || ',' || l_fnamei || ')';
END;
```

This function is only created if ora2pgpro found a table with a BFILE column and that the destination type is EFILE. The function is dropped at the end of the export. This concerns both `COPY` and `INSERT` export types.

To set the destination type, use the `DATA_TYPE` configuration directive:

```
DATA_TYPE       BFILE:EFILE
```

The EFILE type is a user-defined type created by the extension that can be found here: external_file This is a port of the BFILE Oracle type to Postgres Pro.

There is no SQL function available to retrieve the content of a BFILE. ora2pgpro have to create one using the DBMS_LOB package.

```
CREATE OR REPLACE FUNCTION ora2pg_get_bfile( p_bfile IN BFILE )
 RETURN
BLOB
  AS
        filecontent BLOB := NULL;
        src_file BFILE := NULL;
        l_step PLS_INTEGER := 12000;
        l_dir   VARCHAR2(4000);
        l_fname VARCHAR2(4000);
        offset NUMBER := 1;
  BEGIN
    IF p_bfile IS NULL THEN
      RETURN NULL;
    END IF;

DBMS_LOB.FILEGETNAME( p_bfile, l_dir, l_fname );
src_file := BFILENAME( l_dir, l_fname );
IF src_file IS NULL THEN
    RETURN NULL;
END IF;

    DBMS_LOB.FILEOPEN(src_file, DBMS_LOB.FILE_READONLY);
    DBMS_LOB.CREATETEMPORARY(filecontent, true);
    DBMS_LOB.LOADBLOBFROMFILE (filecontent, src_file,
 DBMS_LOB.LOBMAXSIZE, offset, offset);
    DBMS_LOB.FILECLOSE(src_file);
    RETURN filecontent;
END;
```

This function is only created if ora2pgpro found a table with a BFILE column and that the destination type is bytea (the default). The function is dropped at the end of the export. This concerns both COPY and INSERT export type.

About the ROWID and UROWID, they are converted into OID by logical default but this will throw an error at data import. There is no equivalent data type so you might want to use the DATA_TYPE directive to change the corresponding type in Postgres Pro. You should consider replacing this data type by a bigserial (autoincremented sequence), text or uuid data type.

MODIFY_TYPE

Sometimes you need to force the destination type, for example a column exported as timestamp by ora2pgpro can be forced into type date. Value is a comma-separated list of TABLE:COLUMN:TYPE structure. If you need to use comma or space inside type definition, you will have to backslash them.

```
MODIFY_TYPE     TABLE1:COL3:varchar,TABLE1:COL4:decimal(9\,6)
```

Type of table1.col3 will be replaced by a varchar and table1.col4 by a decimal with precision and scale.

If the column's type is a user-defined type, ora2pgpro will autodetect the composite type and will export its data using `ROW()`. Some Oracle user defined types are just array of a native type, in this case you may want to transform this column in simple array of a Postgres Pro native type. To do so, just redefine the destination type as wanted and ora2pgpro will also transform the data as an array. For example, with the following definition in Oracle:

```
CREATE OR REPLACE TYPE mem_type IS VARRAY(10) of VARCHAR2(15);
CREATE TABLE club (Name VARCHAR2(10),
        Address VARCHAR2(20),
        City VARCHAR2(20),
        Phone VARCHAR2(8),
        Members mem_type
);
```

Here custom type mem_type is just a string array and can be translated into the following in Postgres Pro:

```
CREATE TABLE club (
        name varchar(10),
        address varchar(20),
        city varchar(20),
        phone varchar(8),
        members text[]
) ;
```

To do so, just use the directive as follow:

```
MODIFY_TYPE     CLUB:MEMBERS:text[]
```

ora2pgpro will take care to transform all data of this column in the correct format. Only arrays of characters and numerics types are supported.

`TO_NUMBER_CONVERSION`

By default Oracle calls to function `TO_NUMBER` will be translated as a cast into numeric. For example, `TO_NUMBER('10.1234')` is converted into a Postgres Pro call `to_number('10.1234')::numeric`. If you want you can cast the call to integer or bigint by changing the value of the configuration directive. If you need better control of the format, just set it as value, for example: `TO_NUMBER_CONVERSION 99999999999999999999.9999999999` will convert the code above as: `TO_NUMBER('10.1234', '99999999999999999999.9999999999')` Any value of the directive that is not numeric, integer, or bigint will be taken as a mask format. If set to none, no conversion will be done.

`VARCHAR_TO_TEXT`

By default varchar2 without size constraint is tranlated into text. If you want to keep the varchar name, disable this directive.

`FORCE_IDENTITY_BIGINT`

Usually identity column must be bigint to correspond to an auto increment sequence so ora2pgpro always forces it to be a bigint. If, for any reason you want ora2pgpro to respect the `DATA_TYPE` you have set for identity column, then disable this directive.

TO_CHAR_NOTIMEZONE

If you want ora2pgpro to remove any timezone information into the format part of the TO_CHAR() function, enable this directive. Disabled by default.

Taking Export Under Control

The following other configuration directives interact directly with the export process and give you fine granularity in database export control.

SKIP

For TABLE export, you may not want to export all schema constraints, the SKIP configuration directive allows you to specify a space or comma-separated list of constraints that should not be exported. Possible values are:

- fkeys: turn off foreign key constraints
- pkeys: turn off primary keys
- ukeys: turn off unique column constraints
- indexes: turn off all other index types
- checks: turn off check constraints

```
SKIP    indexes,checks
```

For example, this will remove indexes and check constraints from export.

PKEY_IN_CREATE

Enable this directive if you want to add primary key definition inside the CREATE TABLE statement. If disabled (the default), primary key definition will be added with an ALTER TABLE statement.

KEEP_PKEY_NAMES

By default, names of the primary and unique keys in the source Oracle database are ignored, and key names are autogenerated in the target Postgres Pro database with the Postgres Pro internal default naming rules. If you want to preserve Oracle primary and unique key names, set this option to 1.

FKEY_ADD_UPDATE

This directive allows you to add an ON UPDATE CASCADE option to a foreign key when ON DELETE CASCADE is defined or always. Oracle does not support this feature, you have to use a trigger to operate the ON UPDATE CASCADE. As Postgres Pro has this feature, you can choose how to add the foreign key option. This directive has the following values:
- never: the default that means that foreign keys will be declared exactly like in Oracle
- delete: the ON UPDATE CASCADE option will be added only if the ON DELETE CASCADE is already defined on the foreign keys
- always: forces all foreign keys to be defined using the update option

FKEY_DEFERRABLE

When exporting tables, ora2pgpro normally exports constraints as they are, if they are non-deferrable, they are exported as non-deferrable. However, non-deferrable constraints will probably cause problems when attempting to import data to Postgres Pro. The FKEY_DEFERRABLE option set to 1 will cause all foreign key constraints to be exported as deferrable.

DEFER_FKEY

In addition to exporting data when the DEFER_FKEY option is set to 1, it will add a command to defer all foreign key constraints during data export, and the import will be done in a single transaction. This will work only if foreign keys have been exported as deferrable and you are not using direct import to Postgres Pro (PG_DSN is not defined). Constraints will then be checked at the end of the transaction.

This directive can also be enabled if you want to force all foreign keys to be created as deferrable and initially deferred during schema export (TABLE export type).

DROP_FKEY

If deferring foreign keys is not possible due to the amount of data in a single transaction, you have not exported foreign keys as deferrable or you are using direct import to Postgres Pro, you can use the DROP_FKEY directive. It will drop all foreign keys before all data import and recreate them at the end of the import.

DROP_INDEXES

This directive allows you to gain lot of speed improvement during data import by removing all indexes that are not an automatic index (indexes of primary keys) and recreate them at the end of data import. It is far better to not import indexes and constraints before having imported all data.

DISABLE_TRIGGERS

This directive is used to disable triggers on all tables in COPY or INSERT export modes. Available values are USER (disable user-defined triggers only) and ALL (includes RI system triggers). Default is 0: do not add SQL statements to disable trigger before data import.

If you want to disable triggers during data migration, set the value to USER if you are connected as non-superuser, and ALL if you are connected as Postgres Pro superuser. A value of 1 is equal to USER.

DISABLE_SEQUENCE

If set to 1, it disables alter of sequences on all tables during COPY or INSERT export mode. This is used to prevent the update of sequence during data migration. Default is 0, alter sequences.

NOESCAPE

By default all data that are not of type date or time are escaped. If you experience any problem with that you can set it to 1 to disable character escaping during data export. This directive is only used during a COPY export. See STANDARD_CONFORMING_STRINGS for enabling/disabling escape with INSERT statements.

STANDARD_CONFORMING_STRINGS

This controls whether ordinary string literals ('...') treat backslashes literally, as specified in SQL standard. This is on by default, causing ora2pgpro to use the escape string syntax (E'...') if this parameter is not set to 0. This is the exact behavior of the same option in Postgres Pro. This directive is only used during data export to build INSERT statements. See NOESCAPE for enabling/disabling escape in COPY statements.

TRIM_TYPE

If you want to convert CHAR(n) from Oracle into varchar(n) or text in Postgres Pro using directive DATA_TYPE, you might want to do some trimming on the data. By default, ora2pgpro will auto-detect this conversion and remove any whitespace at both leading and trailing position. If you just want

to remove the leadings character set the value to `LEADING`. If you just want to remove the trailing character, set the value to `TRAILING`. Default value is `BOTH`.

`TRIM_CHAR`

The default trimming character is space, use this directive if you need to change the character that will be removed. For example, set it to `-` if you have a leading `-` in the char(n) field. To use space as trimming charger, comment this directive, this is the default value.

`PRESERVE_CASE`

If you want to preserve the case of Oracle object names, set this directive to 1. By default, ora2pgpro will convert all Oracle object names to lower case. It is not recommended to enable this, unless you will always have to double-quote object names on all your SQL scripts.

`ORA_RESERVED_WORDS`

Allow escaping of column name using Oracle reserved words. Value is a list of comma-separated reserved words. Default: `audit,comment,references`.

`USE_RESERVED_WORDS`

Enable this directive if you have table or column names that are a reserved word for Postgres Pro. ora2pgpro will double quote the name of the object.

`GEN_USER_PWD`

Set this directive to 1 to replace default password by a random password for all extracted user during a `GRANT` export.

`PG_SUPPORTS_MVIEW`

In Postgres Pro, materialized views are supported with the SQL syntax `CREATE MATERIALIZED VIEW`. To force ora2pgpro to use the native Postgres Pro support you must enable this configuration - enable by default. If you want to use the old style with table and a set of function, you should disable it.

`PG_VERSION`

Set the Postgres Pro major version number of the target database. For example: 11 or 16. Default is the current major version at the time of a new release.

`BITMAP_AS_GIN`

Use btree_gin extension to create bitmap like indexes. You will need to create the extension. Default is to create GIN index, when disabled, a B-tree index will be created.

`LONGREADLEN`

Use this directive to set how the database handles the `LongReadLen` attribute to a value that will be larger than the expected size of the LOBs. The default is 1MB which may not be enough to extract BLOBs or CLOBs. If the size of the LOB exceeds the `LONGREADLEN` `DBD::Oracle` will return a “ORA-24345: A Truncation” error. Default: 1023*1024 bytes.

Important

If you increase the value of this directive, take care that `DATA_LIMIT` probably needs to be reduced. Even if you only have a 1MB blob, trying to read 10000 of them (the default

DATA_LIMIT) all at once will require 10GB of memory. You may extract data from those tables separately and set a DATA_LIMIT to 500 or lower, otherwise you may experience out of memory.

LONGTRUNKOK

If you want to bypass the "ORA-24345: A Truncation" error, set this directive to 1, it will truncate the data extracted to the LONGREADLEN value. Disabled by default so that you will be warned if your LONGREADLEN value is not high enough.

USE_LOB_LOCATOR

Disable this if you want to load full contents of BLOB and CLOB and not use LOB locators. In this case you will have to set LONGREADLEN to the right value. Note that this will not improve speed of BLOB export as most of the time is always consumed by the bytea escaping and in this case export is done line-by-line and not by chunk of DATA_LIMIT rows. Default is enabled, it uses LOB locators.

LOB_CHUNK_SIZE

Oracle recommends reading from and writing to a LOB in batches using a multiple of the LOB chunk size. This chunk size defaults to 8k (8192). Recent tests shown that the best performances can be reach with higher value like 512K or 4Mb.

A quick benchmark with 30120 rows with different size of BLOB (200x5Mb, 19800x212k, 10000x942K, 100x17Mb, 20x156Mb), with DATA_LIMIT=100, LONGREADLEN=170Mb and a total table size of 20GB gives:

```
no lob locator  : 22m46,218s (1365 sec., avg: 22 recs/sec)
chunk size 8k   : 15m50,886s (951 sec., avg: 31 recs/sec)
chunk size 512k : 1m28,161s (88 sec., avg: 342 recs/sec)
chunk size 4Mb  : 1m23,717s (83 sec., avg: 362 recs/sec)
```

In conclusion it can be more than 10 time faster with LOB_CHUNK_SIZE set to 4Mb. Depending on the size of most BLOB you may want to adjust the value here. For example, if you have a majority of small lobs bellow 8K, using 8192 is better to not waste space. Default value for LOB_CHUNK_SIZE is 512000.

XML_PRETTY

Force the use getStringVal() instead of getClobVal() for XML data export. Default is 1, enabled for backward compatibility. Set it to 0 to use extract method as CLOB. Note that XML value extracted with getStringVal() must not exceed VARCHAR2 size limit (4000), otherwise it will return an error.

ENABLE_MICROSECOND

Set it to O if you want to disable export of millisecond from Oracle timestamp columns. By default milliseconds are exported with the use of the following format:

```
'YYYY-MM-DD HH24:MI:SS.FF'
```

Disabling will force the use of the following Oracle format:

```
to_char(..., 'YYYY-MM-DD HH24:MI:SS')
```

By default milliseconds are exported.

`DISABLE_COMMENT`

Set this to 1 if you do not want to export comment associated to tables and columns definition. Default is enabled.

Special Options to Handle Character Encoding

`NLS_LANG`

By default, ora2pgpro will set `NLS_LANG` to `AMERICAN_AMERICA.AL32UTF8` and `NLS_NCHAR` to `AL32UTF8`. It is not recommended to change those settings but in some case it could be useful. Using your own settings with those configuration directive will change the client encoding at Oracle side by setting the environment variables `$ENV{NLS_LANG}` and `$ENV{NLS_NCHAR}`.

`BINMODE`

By default, ora2pgpro will force Perl to use UTF8 I/O encoding. This is done through a call to the Perl pragma:

```
use open ':utf8';
```

You can override this encoding by using the `BINMODE` directive, for example you can set it to `:locale` to use your locale or `iso-8859-7`.

```
use open ':locale';
use open ':encoding(iso-8859-7)';
```

If you have changed the `NLS_LANG` to non-UTF8 encoding, you might want to set this directive. Most of the time, leave this directive commented.

`CLIENT_ENCODING`

By default, Postgres Pro client encoding is automatically set to UTF8 to avoid encoding issues. If you have changed the value of `NLS_LANG`, you might have to change the encoding of the Postgres Pro client.

For the Postgres Pro supported character sets, see Character Set Support.

`FORCE_PLSQL_ENCODING`

To force UTF8 encoding of the PL/SQL code exported, enable this directive. Could be helpful in some rare condition.

PL/SQL-to-PL/pgSQL Conversion

Automatic code conversion from Oracle PL/SQL to Postgres Pro PL/pgSQL is a work in progress in ora2pgpro and you have to do manual work. The Perl code used for automatic conversion is all stored in a specific Perl Module named `Ora2Pgpro/PLSQL.pm`.

`PLSQL_PGSQL`

Enable/disable PL/SQL to PL/pgSQL conversion. Enabled by default.

NULL_EQUAL_EMPTY

ora2pgpro can replace all conditions with a test on NULL by a call to the `coalesce()` function to mimic the Oracle behavior where empty string are considered equal to NULL.

```
(field1 IS NULL) is replaced by (coalesce(field1::text, '') = '')
(field2 IS NOT NULL) is replaced by (field2 IS NOT NULL AND
 field2::text <> '')
```

You might want this replacement to be sure that your application will have the same behavior but if you have control on you application a better way is to change it to transform empty string into NULL because Postgres Pro makes the difference.

EMPTY_LOB_NULL

Force `empty_clob()` and `empty_blob()` to be exported as NULL instead as empty string for the first one and `\x` for the second. If NULL is allowed in your column, this might improve data export speed if you have a lot of empty lobs. Default is to preserve the exact data from Oracle.

PACKAGE_AS_SCHEMA

If you do not want to export package as schema but as simple functions you might also want to replace all calls to `package_name.function_name`. If you disable the `PACKAGE_AS_SCHEMA` directive, then ora2pgpro will replace all call to `package_name.function_name()` by `package_name_function_name()`. Default is to use a schema to emulate package.

The replacement will be done in all kind of DDL or code that is parsed by the PL/SQL to PL/pgSQL converter. `PLSQL_PGSQL` must be enabled or `-p` used in command line.

REWRITE_OUTER_JOIN

Enable this directive if the rewrite of Oracle native syntax (+) of `OUTER JOIN` is broken. This will force ora2pgpro to not rewrite such code, default is to try to rewrite simple form of the right outer join for the moment.

UUID_FUNCTION

By default ora2pgpro will convert call to SYS_GUID() Oracle function with a call to uuid_generate_v4 from uuid-ossp extension. You can redefined it to use the gen_random_uuid function from pgcrypto extension by changing the function name. Default to uuid_generate_v4.

Note that when a RAW(16) and RAW(32) columns is found or that the RAW column has "SYS_GUID()" as default value ora2pgpro will automatically translate the type of the column into uuid which might be the right translation in most of the case. In this case data will be automatically migrated as Postgres Pro uuid data type provided by the "uuid-ossp" extension.

FUNCTION_STABLE

By default Oracle functions are marked as `STABLE` as they can not modify data unless when used in PL/SQL with variable assignment or as conditional expression. You can force ora2pgpro to create these function as `VOLATILE` by disabling this configuration directive.

COMMENT_COMMIT_ROLLBACK

By default call to `COMMIT`/`ROLLBACK` are kept untouched by ora2pgpro to force the user to review the logic of the function. Once it is fixed in Oracle source code or you want to comment these calls, enable the following directive.

COMMENT_SAVEPOINT

It is common to see SAVEPOINT calls inside PL/SQL procedure together with a ROLLBACK TO savepoint_name. When COMMENT_COMMIT_ROLLBACK is enabled, you may want to also comment SAVEPOINT calls, in this case enable it.

STRING_CONSTANT_REGEXP

ora2pgpro replaces all string constants during the PL/SQL to PL/pgSQL translation, string constants are all text included between single quote. If you have a string placeholder used in dynamic call to queries, you can set a list of regexp to be temporary replaced to not break the parser. For example:

```
STRING_CONSTANT_REGEXP          <placeholder value=".*">
```

The list of regexp must use the semicolon as separator.

ALTERNATIVE_QUOTING_REGEXP

To support the Alternative Quoting Mechanism ('Q' or 'q') for String Literals set the regexp with the text capture to use to extract the text part. For example with a variable declared as

```
c_sample VARCHAR2(100 CHAR) := q'{This doesn't work.}';
```

the regexp to use must be:

```
ALTERNATIVE_QUOTING_REGEXP      q'{(.*)}'
```

ora2pgpro will use the $$ delimiter, with the example the result will be:

```
c_sample varchar(100) := $$This doesn't work.$$;
```

The value of this configuration directive can be a list of regexp separated by a semicolon. The capture part (in parenthesis) is mandatory in each regexp if you want to restore the string constant.

USE_ORAFCE

If you want to use functions defined in the orafce library and prevent ora2pgpro to translate calls to these functions, enable this directive.

By default ora2pgpro rewrite add_month(), add_year(), date_trunc() and to_char() functions, but you may prefer to use the orafce version of these function that do not need any code transformation.

INCLUDE_PACKAGES

Contains the comma-separated list of packages to allow to be exported. Used only with the PACKAGE export type. Only the last occurrence found in the file will be used.

EXCLUDE_PACKAGES

Contains the comma-separated list of packages to exclude from export. Used only with the PACKAGE export type. Only the last occurrence found in the file will be used.

`POSTGRESPRO_ATX`

If set to 1, enables export of autonomous transactions directly as Postgres Pro autonomous transactions.

Other Configuration Directives

`DEBUG`

Setting it to 1 enables verbose output.

`IMPORT`

You can define common ora2pgpro configuration directives into a single file that can be imported into other configuration files with the `IMPORT` configuration directive as follows:

```
IMPORT  commonfile.conf
```

This will import all configuration directives defined into `commonfile.conf` into the current configuration file.

Chapter 5. Usage Examples

Materialized Views

Materialized views are exported as snapshot "Snapshot Materialized Views" as Postgres Pro only supports full refresh.

When exporting materialized view, ora2pgpro will first add the SQL code to create the `materialized_views` table:

```
CREATE TABLE materialized_views (
        mview_name text NOT NULL PRIMARY KEY,
        view_name text NOT NULL,
        iname text,
        last_refresh TIMESTAMP WITH TIME ZONE
);
```

All materialized views will have an entry in this table. It then adds the PL/pgSQL code to create tree functions:

- `create_materialized_view(text, text, text)` is used to create a materialized view
- `drop_materialized_view(text)` is used to delete a materialized view
- `refresh_full_materialized_view(text)` is used to refresh a view

Then it adds the SQL code to create the view and the materialized view:

```
CREATE VIEW mviewname_mview AS
SELECT ... FROM ...;

SELECT create_materialized_view('mviewname','mviewname_mview', change
 with the name of the column to be used for the index);
```

The first argument is the name of the materialized view, the second the name of the view on which the materialized view is based and the third is the column name on which the index should be build (aka most of the time the primary key). This column is not automatically deduced so you need to replace its name.

As said above, ora2pgpro only supports snapshot materialized views so the table will be entirely refreshed by issuing first a truncate of the table and then by load again all data from the view:

```
refresh_full_materialized_view('mviewname');
```

To drop the materialized view, call the `drop_materialized_view()` function with the name of the materialized view as parameter.

Exporting Views as Postgres Pro Tables

You can export any Oracle view as a Postgres Pro table simply by setting TYPE configuration option to `TABLE` to have the corresponding create table statement. Or use type `COPY` or `INSERT` to export the corresponding data. To allow that you have to specify your views in the VIEW_AS_TABLE configuration option.

Then, if ora2pgpro finds the view, it extracts its schema (if TYPE=TABLE) into a Postgres Pro CREATE TABLE form, then it will extract the data (if TYPE=COPY or TYPE=INSERT) following the view schema.

For example, with the following view:

```
CREATE OR REPLACE VIEW product_prices (category_id, product_count,
 low_price, high_price) AS
SELECT  category_id, COUNT(*) as product_count,
    MIN(list_price) as low_price,
    MAX(list_price) as high_price
 FROM   product_information
GROUP BY category_id;
```

Setting VIEW_AS_TABLE to product_prices and using export type TABLE, will force ora2pgpro to detect columns returned types and to generate a create table statement:

```
CREATE TABLE product_prices (
        category_id bigint,
        product_count integer,
        low_price numeric,
        high_price numeric
);
```

Data will be loaded following the COPY or INSERT export type and the view declaration.

You can use the ALLOW and EXCLUDE directives in addition to filter other objects to export.

Migration Cost Assessment

Estimating the cost of a migration process from Oracle to Postgres Pro is not easy. To obtain a good assessment of this migration cost, ora2pgpro will inspect all database objects, all functions and stored procedures to detect if there are still some objects and PL/SQL code that can not be automatically converted by ora2pgpro.

ora2pgpro has a content analysis mode that inspect the Oracle database to generate a text report on what the Oracle database contains and what can not be exported.

To activate the "analysis and report" mode, you have to use the export type SHOW_REPORT like in the following command:

```
ora2pgpro -t SHOW_REPORT
```

Here is a sample report obtained with this command:

```
--------------------------------------
Ora2Pg: Oracle Database Content Report
--------------------------------------
Version Oracle Database 10g Enterprise Edition Release 10.2.0.1.0
Schema  HR
Size  880.00 MB
```

```
-------------------------------------
Object  Number  Invalid Comments
-------------------------------------
CLUSTER   2 0 Clusters are not supported and will not be exported.
FUNCTION  40  0 Total size of function code: 81992.
INDEX     435 0 232 index(es) are concerned by the export, others are
 automatically generated and will
                                 do so on PostgreSQL. 1 bitmap
 index(es). 230 b-tree index(es). 1 reversed b-tree index(es)
                                 Note that bitmap index(es) will be
 exported as b-tree index(es) if any. Cluster, domain,
                                 bitmap join and IOT indexes will not
 be exported at all. Reverse indexes are not exported
                                 too, you may use a trigram-based index
 (see pg_trgm) or a reverse() function based index
                                 and search. You may also use
 'varchar_pattern_ops', 'text_pattern_ops' or 'bpchar_pattern_ops'
                                 operators in your indexes to improve
 search with the LIKE operator respectively into
                                 varchar, text or char columns.
MATERIALIZED VIEW 1 0 All materialized view will be exported as
 snapshot materialized views, they
                                 are only updated when fully refreshed.
PACKAGE BODY  2 1 Total size of package code: 20700.
PROCEDURE 7 0 Total size of procedure code: 19198.
SEQUENCE  160 0 Sequences are fully supported, but all call to
 sequence_name.NEXTVAL or sequence_name.CURRVAL
                                 will be transformed into
 NEXTVAL('sequence_name') or CURRVAL('sequence_name').
TABLE     265 0 1 external table(s) will be exported as standard
 table. See EXTERNAL_TO_FDW configuration
                                 directive to export as file_fdw
 foreign tables or use COPY in your code if you just
                                 want to load data from external files.
 2 binary columns. 4 unknown types.
TABLE PARTITION 8 0 Partitions are exported using table inheritance
 and check constraint. 1 HASH partitions.
                                 2 LIST partitions. 6 RANGE partitions.
 Note that Hash partitions are not supported.
TRIGGER   30  0 Total size of trigger code: 21677.
TYPE      7 1 5 type(s) are concerned by the export, others are not
 supported. 2 Nested Tables.
                                 2 Object type. 1 Subtype. 1 Type Boby.
 1 Type inherited. 1 Varrays. Note that Type
                                 inherited and Subtype are converted as
 table, type inheritance is not supported.
TYPE BODY 0 3 Export of type with member method are not supported,
 they will not be exported.
VIEW      7 0 Views are fully supported, but if you have updatable
 views you will need to use
                                 INSTEAD OF triggers.
DATABASE LINK 1 0 Database links will not be exported. You may try the
 dblink perl contrib module or use
```

```
                                the SQL/MED PostgreSQL features with
 the different Foreign Data Wrapper (FDW) extensions.

Note: Invalid code will not be exported unless the EXPORT_INVALID
 configuration directive is activated.
```

Once the database can be analysed, ora2pgpro, by its ability to convert SQL and PL/SQL code from Oracle syntax to Postgres Pro, can go further by estimating the code difficulties and estimate the time necessary to operate a full database migration.

To estimate the migration cost in man-days, ora2pgpro allows you to use a configuration directive `ESTIMATE_COST` that you can also enable in the command line: `--estimate_cost`.

This feature can only be used with the `SHOW_REPORT`, `FUNCTION`, `PROCEDURE`, `PACKAGE`, and `QUERY` export types.

```
ora2pgpro -t SHOW_REPORT  --estimate_cost
```

The generated report is same as above but with a new `Estimated cost` column as follow:

```
---------------------------------------
Ora2Pg: Oracle Database Content Report
---------------------------------------
Version Oracle Database 10g Express Edition Release 10.2.0.1.0
Schema  HR
Size  890.00 MB

---------------------------------------
Object  Number  Invalid Estimated cost  Comments
---------------------------------------
DATABASE LINK  3 0 9 Database links will be exported as SQL/MED
 PostgreSQL's Foreign Data Wrapper (FDW) extensions
                                using oracle_fdw.
FUNCTION  2 0 7 Total size of function code: 369 bytes. HIGH_SALARY:
 2, VALIDATE_SSN: 3.
INDEX 21  0 11  11 index(es) are concerned by the export, others are
 automatically generated and will do so
                                on PostgreSQL. 11 b-tree index(es).
 Note that bitmap index(es) will be exported as b-tree
                                index(es) if any. Cluster, domain,
 bitmap join and IOT indexes will not be exported at all.
                                Reverse indexes are not exported too,
 you may use a trigram-based index (see pg_trgm) or a
                                reverse() function based index and
 search. You may also use 'varchar_pattern_ops', 'text_pattern_ops'
                                or 'bpchar_pattern_ops' operators in
 your indexes to improve search with the LIKE operator
                                respectively into varchar, text or
 char columns.
JOB 0 0 0 Job are not exported. You may set external cron job with
 them.
MATERIALIZED VIEW 1 0 3 All materialized view will be exported as
 snapshot materialized views, they
```

```
                                        are only updated when fully
 refreshed.
PACKAGE BODY  0 2 54  Total size of package code: 2487 bytes. Number
 of procedures and functions found
                                        inside those packages: 7.
 two_proc.get_table: 10, emp_mgmt.create_dept: 4,
                                        emp_mgmt.hire: 13,
 emp_mgmt.increase_comm: 4, emp_mgmt.increase_sal: 4,
                                        emp_mgmt.remove_dept: 3,
 emp_mgmt.remove_emp: 2.
PROCEDURE 4 0 39  Total size of procedure code: 2436 bytes.
 TEST_COMMENTAIRE: 2, SECURE_DML: 3,
                                        PHD_GET_TABLE: 24,
 ADD_JOB_HISTORY: 6.
SEQUENCE  3 0 0 Sequences are fully supported, but all call to
 sequence_name.NEXTVAL or sequence_name.CURRVAL
                                        will be transformed into
 NEXTVAL('sequence_name') or CURRVAL('sequence_name').
SYNONYM   3 0 4 SYNONYMs will be exported as views. SYNONYMs do not
 exists with PostgreSQL but a common workaround
                                        is to use views or set the
 PostgreSQL search_path in your session to access
                                        object outside the current
 schema.
                                        user1.emp_details_view_v is an
 alias to hr.emp_details_view.
                                        user1.emp_table is an alias to
 hr.employees@other_server.
                                        user1.offices is an alias to
 hr.locations.
TABLE 17  0 8.5 1 external table(s) will be exported as standard
 table. See EXTERNAL_TO_FDW configuration
                                directive to export as file_fdw
 foreign tables or use COPY in your code if you just want to
                                load data from external files. 2
 binary columns. 4 unknown types.
TRIGGER 1 1 4 Total size of trigger code: 123 bytes.
 UPDATE_JOB_HISTORY: 2.
TYPE  7 1 5 5 type(s) are concerned by the export, others are not
 supported. 2 Nested Tables. 2 Object type.
                                1 Subtype. 1 Type Boby. 1 Type
 inherited. 1 Varrays. Note that Type inherited and Subtype are
                                converted as table, type inheritance
 is not supported.
TYPE BODY 0 3 30  Export of type with member method are not supported,
 they will not be exported.
VIEW  1 1 1 Views are fully supported, but if you have updatable views
 you will need to use INSTEAD OF triggers.
-------------------------------------
Total 65  8 162.5 162.5 cost migration units means approximatively 2
 man day(s).
```

The last line shows the total estimated migration code in man-days following the number of migration units estimated for each object. This migration unit represent around five minutes for a Postgres Pro expert.

If this is your first migration, you can get it higher with the configuration directive `COST_UNIT_VALUE` or the `--cost_unit_value` command line option:

```
ora2pgpro -t SHOW_REPORT  --estimate_cost --cost_unit_value 10
```

ora2pgpro is also able to give you a migration difficulty level assessment, here is a sample: Migration level: B-5

```
Migration levels:
    A - Migration that might be run automatically
    B - Migration with code rewrite and a man-days cost up to 5 days
    C - Migration with code rewrite and a man-days cost above 5 days
Technical levels:
    1 = trivial: no stored functions and no triggers
    2 = easy: no stored functions but with triggers, no manual
 rewriting
    3 = simple: stored functions and/or triggers, no manual rewriting
    4 = manual: no stored functions but with triggers or views with
 code rewriting
    5 = difficult: stored functions and/or triggers with code
 rewriting
```

This assessment consists of a letter (A or B) to specify if the migration needs manual rewriting or not, and a number from 1 to 5 to give you a technical difficulty level. You have an additional option `--human_days_limit` to specify the number of man-days limit where the migration level should be set to C to indicate that it needs a huge amount of work and a full project management with migration support. Default is 10 man-days. You can use the configuration directive `HUMAN_DAYS_LIMIT` to change this default value permanently.

This feature has been developed to help you decide which database to migrate first and the team that must be mobilized to operate the migration.

Global Oracle Migration Assessment

ora2pgpro comes with a script `ora2pgpro_scanner` that can be used when you have a huge number of instances and schema to scan for migration assessment.

```
ora2pgpro_scanner -l CSVFILE [-o OUTDIR]

-b | --binpath DIR: full path to directory where the ora2pgpro binary
 stays.
             Might be useful only on Windows OS.
-c | --config FILE: set custom configuration file to use otherwise
 ora2pgpro
             will use the default: /etc/ora2pgpro/ora2pgpro.conf.
-l | --list FILE : CSV file containing a list of databases to scan
 with
             all required information. The first line of the file
             can contain the following header that describes the
             format that must be used:
```

```
"type","schema/database","dsn","user","password"

-o | --outdir DIR : (optional) by default all reports will be dumped
 to a
             directory named 'output', it will be created
 automatically.
             If you want to change the name of this directory, set the
 name
             at second argument.

-t | --test : just try all connections by retrieving the required
 schema
              or database name. Useful to validate your CSV list file.
-u | --unit MIN : redefine globally the migration cost unit value in
 minutes.
              Default is taken from the ora2pgpro.conf (default 5
 minutes).

Here is a full example of a CSV databases list file:

"type","schema/database","dsn","user","password"
"MYSQL","sakila","dbi:mysql:host=192.168.1.10;database=sakila;port=3306","root","s
"ORACLE","HR","dbi:Oracle:host=192.168.1.10;sid=XE;port=1521","system","manager"
"MSSQL","HR","dbi:ODBC:driver=msodbcsql18;server=srv.database.windows.net;database

The CSV field separator must be a comma.

Note that if you want to scan all schemas from an Oracle instance you
 just
have to leave the schema field empty, Ora2PgPro will automatically
 detect all
available schemas and generate a report for each one. Of course you
 need to
use a connection user with enough privileges to be able to scan all
 schemas.
For example:

"ORACLE","","dbi:Oracle:host=192.168.1.10;sid=XE;port=1521","system","manager"
"MSSQL","","dbi:ODBC:driver=msodbcsql18;server=srv.database.windows.net;database=t

will generate a report for all schema in the XE instance. Note that in
 this
case the SCHEMA directive in ora2pgpro.conf must not be set.
```

It will generate a CSV file with the assessment result, one line per schema or database and a detailed HTML report for each database scanned.

Hint: Use the `-t` | `--test` option before to test all your connections in your CSV file.

For Windows users you must use the `-b` command-line option to set the directory where `ora2pgpro_scanner` stays, otherwise the ora2pgpro command calls will fail.

In the migration assessment details about functions ora2pgpro always includes per default 2 migration units for TEST and 1 unit for SIZE per 1000 characters in the code. This mean that by default it will add

15 minutes in the migration assessment per function. Obviously, if you have unitary tests or very simple functions, this will not represent the real migration time.

Migration Assessment Method

Migration unit scores given to each type of Oracle database object are defined in the Perl library `lib/Ora2Pg/PLSQL.pm` in the `%OBJECT_SCORE` variable definition.

The number of PL/SQL lines associated to a migration unit is also defined in this file in the `$SIZE_SCORE` variable value.

The number of migration units associated to each PL/SQL code difficulties can be found in the same Perl library `lib/Ora2Pg/PLSQL.pm` in the hash `%UNCOVERED_SCORE` initialization.

This assessment method is a work in progress.

Improving Indexes and Constraints Creation Speed

Using the `LOAD` export type and a file containing SQL orders to perform, it is possible to dispatch those orders over multiple Postgres Pro connections. To be able to use this feature, `PG_DSN`, `PG_USER` and `PG_PWD` must be set.

```
ora2pgpro -t LOAD -c config/ora2pgpro.conf -i schema/tables/
INDEXES_table.sql -j 4
```

This will dispatch indexes creation over 4 simultaneous Postgres Pro connections. This will considerably accelerate this part of the migration process with huge data size.

Exporting LONG RAW

If you still have columns defined as LONG RAW, ora2pgpro will not be able to export this kind of data. The OCI library fails to export them and always returns the same first record. To be able to export the data you need to transform the field as BLOB by creating a temporary table before migrating data.

```
SQL> DESC TEST_LONGRAW
 Name                  NULL ?   Type
 --------------------- -------- ----------------------------
 ID                             NUMBER
 C1                             LONG RAW
```

For example, the above Oracle table needs to be "translated" into a table using BLOB as follows:

```
CREATE TABLE test_blob (id NUMBER, c1 BLOB);
```

And then copy the data with the following `INSERT` query:

```
INSERT INTO test_blob SELECT id, to_lob(c1) FROM test_longraw;
```

Then you just have to exclude the original table from the export (see `EXCLUDE` directive) and to rename the new temporary table onthe fly using the `REPLACE_TABLES` configuration directive.

Global Variables

Oracle allows the use of global variables defined in packages. ora2pgpro will export these variables as Postgres Pro package variables.

Migration Test

The type of action called TEST allows you to check that all objects from Oracle database have been created under Postgres Pro. PG_DSN must be set to be able to check the Postgres Pro side.

Note that this feature respects the schema name limitation if EXPORT_SCHEMA and SCHEMA or PG_SCHEMA are defined. If only EXPORT_SCHEMA is set, all schemes from Oracle and Postgres Pro are scanned. You can filter to a single schema using SCHEMA and/or PG_SCHEMA but you can not filter by a list of schemas. To test a list of schema, you will have to repeat the calls to ora2pgpro by specifying a single schema each time.

```
ora2pgpro -t TEST -c config/ora2pgpro.conf > migration_diff.txt
```

For example, this command will create a file containing the report of all object and row count on both sides, Oracle and Postgres Pro, with an error section giving you the details of the differences for each kind of object. Here is a sample result:

```
[TEST INDEXES COUNT]
ORACLEDB:DEPARTMENTS:2
POSTGRES:departments:1
ORACLEDB:EMPLOYEES:6
POSTGRES:employees:6
[ERRORS INDEXES COUNT]
Table departments don't have the same number of indexes in Oracle (2)
 and in PostgreSQL (1).

[TEST UNIQUE CONSTRAINTS COUNT]
ORACLEDB:DEPARTMENTS:1
POSTGRES:departments:1
ORACLEDB:EMPLOYEES:1
POSTGRES:employees:1
[ERRORS UNIQUE CONSTRAINTS COUNT]
OK, Oracle and PostgreSQL have the same number of unique constraints.

[TEST PRIMARY KEYS COUNT]
ORACLEDB:DEPARTMENTS:1
POSTGRES:departments:1
ORACLEDB:EMPLOYEES:1
POSTGRES:employees:1
[ERRORS PRIMARY KEYS COUNT]
OK, Oracle and PostgreSQL have the same number of primary keys.

[TEST CHECK CONSTRAINTS COUNT]
ORACLEDB:DEPARTMENTS:1
POSTGRES:departments:1
ORACLEDB:EMPLOYEES:1
```

```
POSTGRES:employees:1
[ERRORS CHECK CONSTRAINTS COUNT]
OK, Oracle and PostgreSQL have the same number of check constraints.

[TEST NOT NULL CONSTRAINTS COUNT]
ORACLEDB:DEPARTMENTS:1
POSTGRES:departments:1
ORACLEDB:EMPLOYEES:1
POSTGRES:employees:1
[ERRORS NOT NULL CONSTRAINTS COUNT]
OK, Oracle and PostgreSQL have the same number of not null
 constraints.

[TEST COLUMN DEFAULT VALUE COUNT]
ORACLEDB:DEPARTMENTS:1
POSTGRES:departments:1
ORACLEDB:EMPLOYEES:1
POSTGRES:employees:1
[ERRORS COLUMN DEFAULT VALUE COUNT]
OK, Oracle and PostgreSQL have the same number of column default
 value.

[TEST IDENTITY COLUMN COUNT]
ORACLEDB:DEPARTMENTS:1
POSTGRES:departments:1
ORACLEDB:EMPLOYEES:0
POSTGRES:employees:0
[ERRORS IDENTITY COLUMN COUNT]
OK, Oracle and PostgreSQL have the same number of identity column.

[TEST FOREIGN KEYS COUNT]
ORACLEDB:DEPARTMENTS:0
POSTGRES:departments:0
ORACLEDB:EMPLOYEES:1
POSTGRES:employees:1
[ERRORS FOREIGN KEYS COUNT]
OK, Oracle and PostgreSQL have the same number of foreign keys.

[TEST TABLE COUNT]
ORACLEDB:TABLE:2
POSTGRES:TABLE:2
[ERRORS TABLE COUNT]
OK, Oracle and PostgreSQL have the same number of TABLE.

[TEST TABLE TRIGGERS COUNT]
ORACLEDB:DEPARTMENTS:0
POSTGRES:departments:0
ORACLEDB:EMPLOYEES:1
POSTGRES:employees:1
[ERRORS TABLE TRIGGERS COUNT]
OK, Oracle and PostgreSQL have the same number of table triggers.

[TEST TRIGGER COUNT]
ORACLEDB:TRIGGER:2
```

```
POSTGRES:TRIGGER:2
[ERRORS TRIGGER COUNT]
OK, Oracle and PostgreSQL have the same number of TRIGGER.

[TEST VIEW COUNT]
ORACLEDB:VIEW:1
POSTGRES:VIEW:1
[ERRORS VIEW COUNT]
OK, Oracle and PostgreSQL have the same number of VIEW.

[TEST MVIEW COUNT]
ORACLEDB:MVIEW:0
POSTGRES:MVIEW:0
[ERRORS MVIEW COUNT]
OK, Oracle and PostgreSQL have the same number of MVIEW.

[TEST SEQUENCE COUNT]
ORACLEDB:SEQUENCE:1
POSTGRES:SEQUENCE:0
[ERRORS SEQUENCE COUNT]
SEQUENCE does not have the same count in Oracle (1) and in PostgreSQL
 (0).

[TEST TYPE COUNT]
ORACLEDB:TYPE:1
POSTGRES:TYPE:0
[ERRORS TYPE COUNT]
TYPE does not have the same count in Oracle (1) and in PostgreSQL (0).

[TEST FDW COUNT]
ORACLEDB:FDW:0
POSTGRES:FDW:0
[ERRORS FDW COUNT]
OK, Oracle and PostgreSQL have the same number of FDW.

[TEST FUNCTION COUNT]
ORACLEDB:FUNCTION:3
POSTGRES:FUNCTION:3
[ERRORS FUNCTION COUNT]
OK, Oracle and PostgreSQL have the same number of functions.

[TEST SEQUENCE VALUES]
ORACLEDB:EMPLOYEES_NUM_SEQ:1285
POSTGRES:employees_num_seq:1285
[ERRORS SEQUENCE VALUES COUNT]
OK, Oracle and PostgreSQL have the same values for sequences

[TEST ROWS COUNT]
ORACLEDB:DEPARTMENTS:27
POSTGRES:departments:27
ORACLEDB:EMPLOYEES:854
POSTGRES:employees:854
[ERRORS ROWS COUNT]
OK, Oracle and PostgreSQL have the same number of rows.
```

Data Validation

Data validation consists in comparing data retrieved from a foreign table pointing to the source Oracle table and a local Postgres Pro table resulting from the data export.

To run data validation, you can use a direct connection like any other ora2pgpro action but you can also use the oracle_fdw extension provided that FDW_SERVER and PG_DSN configuration directives are set.

By default, ora2pgpro will extract the 10000 first rows from both sides, you can change this value using directive DATA_VALIDATION_ROWS. When it is set to zero, all rows of the tables will be compared.

Data validation requires that the table has a primary key or unique index and that the key columns is not a LOB. Rows will be sorted using this unique key. Due to differences in sort behavior between Oracle and Postgres Pro, if the collation of unique key columns in Postgres Pro is not C, the sort order can be different compared to Oracle. In this case the data validation will fail.

Data validation must be done before any data is modified.

ora2pgpro will stop comparing two tables after DATA_VALIDATION_ROWS is reached or that 10 errors has been encountered, result is dumped in a file named data_validation.log written in the current directory by default. The number of error before stopping the diff between rows can be controlled using the configuration directive DATA_VALIDATION_ERROR. All rows in errors are printed to the output file for your analyze.

It is possible to parallelize data validation by using -P option or the corresponding configuration directive PARALLEL_TABLES in ora2pgpro.conf.

Use of System Change Number (SCN)

ora2pgpro is able to export data as of a specific SCN. You can set it at command line using the -S or --scn option. You can give a specific SCN or if you want to use the current SCN at first connection time set the value to current. In this last case, the connection user has the SELECT ANY DICTIONARY or the SELECT_CATALOG_ROLE role, the current SCN is looked at the v$database view.

```
ora2pgpro -c ora2pgpro.conf -t COPY --scn 16605281
```

This adds the following clause to the query used to retrieve data for example:

```
AS OF SCN 16605281
```

You can also use the --scn option to use the Oracle flashback capability by specifying a timestamp expression instead of a SCN.

```
ora2pgpro -c ora2pgpro.conf -t COPY --scn "TO_TIMESTAMP('2021-12-01
 00:00:00', 'YYYY-MM-DD HH:MI:SS')"
```

This will add the following clause to the query used to retrieve data:

```
AS OF TIMESTAMP TO_TIMESTAMP('2021-12-01 00:00:00', 'YYYY-MM-DD
 HH:MI:SS')
```

Or for example to only retrive yesterday's data:

```
ora2pgpro -c ora2pgpro.conf -t COPY --scn "SYSDATE - 1"
```

Change Data Capture (CDC)

ora2pgpro does not allow to import data and to only apply changes after the first import. But you can use the `--cdc_ready` option to export data with registration of the SCN at the time of the table export. All SCN per tables are written to a file named `TABLES_SCN.log` by default, it can be changed using `-C|--cdc_file` option.

These SCN registered per table during `COPY` or `INSERT` export can be used with a CDC tool. The format of the file is `tablename:SCN` per line.

Importing BLOB as Large Objects

By default ora2pgpro imports Oracle BLOB as bytea, the destination column is created using the bytea data type. If you want to use large object instead of bytea, just add the `--blob_to_lo` option to the ora2pgpro command. It will create the destination column as data type oid and will save the BLOB as a large object using the `lo_from_bytea()` function. The OID returned by the call to `lo_from_bytea()` is inserted in the destination column instead of a bytea. Because of the use of the function, this option can only be used with actions `SHOW_COLUMN`, `TABLE` and `INSERT`. Action `COPY` is not allowed.

If you want to use `COPY` or have huge size BLOB (> 1GB) than can not be imported using `lo_from_bytea()`, you can add option `--lo_import` to the ora2pgpro command. This will allow to import data in two passes.

1. Export data using `COPY` or `INSERT` will set the OID destination column for BLOB to value 0 and save the BLOB value into a dedicated file. It will also create a Shell script to import the BLOB files into the database using psql command `\lo_import` and to update the table OID column to the returned large object OID. The script is named `lo_import-TABLENAME.sh`.

2. Execute all scripts `lo_import-TABLENAME.sh` after setting the environment variables `PGDATABASE` and optionally `PGHOST`, `PGPORT`, `PGUSER`, etc. if they do not correspond to the default values for libpq.

You might also execute manually a `VACUUM FULL` on the table to remove the bloat created by the table update.

> **Note**
>
> Limitation: the table must have a primary key, it is used to set the `WHERE` clause to update the OID column after the large object import. Importing BLOB using this second method (`--lo_import`) is very slow so it should be reserved to rows where the BLOB > 1GB for all other rows use the option `--blob_to_lo`. To filter the rows you can use the `WHERE` configuration directive in `ora2pgpro.conf`.

Exporting Packages

You can directly export Oracle packages as Postgres Pro packages using ora2pgpro by setting the `TYPE` configuration option to `PACKAGE`. ora2pgpro finds the package, downloads and processes it, and reconstructs the abstract syntax tree (AST) representing the source code in the `sources` directory under the current output directory. Then it generates equivalent code in PL/pgSQL in the `result` directory under the current output directory. ora2pgpro aims to convert package code in such a way that the resulting

package is as functional as possible and leaves `WARNING` messages in code places that require programmer's attention.

All the package functions and procedures are public if there is no `#private` modifier, and all the public package variables are listed under the `#export` modifier of the `__init__` function. If there are no public package variables, then `__init__` will contain `#export off`.

This is an example of the `#export` and `#import` modifiers:

```
CREATE OR REPLACE FUNCTION PKGB.__INIT__() RETURNS VOID AS $$
#package
#import pkga
#export x, y
DECLARE
  x integer;
  y integer;
  z integer;
BEGIN
  x = 1;
  y = 2;
  z = PKGA.z; --Here PKGA.z = 4
  PERFORM dbms_output.put_line(x::varchar || y::varchar ||
 z::varchar);
END;
$$ LANGUAGE PLPGSQL;
```

This is an example of the `#private` modifier:

```
CREATE OR REPLACE FUNCTION PKGB.private_function(a integer) RETURNS
 integer AS $$
#package
#private
DECLARE
 b integer;
BEGIN
 b = a * 3;
 RETURN b;
END; $$ LANGUAGE PLPGSQL;

CREATE OR REPLACE FUNCTION PKGB.public_function(a integer) RETURNS
 integer AS $$
#package
BEGIN
 RETURN PKGB.private_function(a);
END; $$ LANGUAGE PLPGSQL;
```

This is how the function may be called:

```
test=# select pkgb.private_function(2);
ERROR:  private package function or procedure
 pkgb.private_function(integer) called out of its package
CONTEXT:  PL/pgSQL function pkgb.private_function(integer)
```

```
test=# select pkgb.public_function(2);
NOTICE:  124
 public_function
-----------------
               6
(1 row)
```

This is an example of an Oracle package.

```
package body pkgc is
TYPE r_customer_type IS RECORD(
    customer_name varchar2(50),
    credit_limit number(10,2)
);

TYPE t_customer_type IS VARRAY(2)
    OF r_customer_type;

PROCEDURE VARRAY_TEST AS
    t_customers t_customer_type := t_customer_type();
    rec r_customer_type;
    tmp_string varchar2(2000);
BEGIN
    t_customers.EXTEND;
    t_customers(t_customers.LAST).customer_name := 'ABC Corp';
    t_customers(t_customers.LAST).credit_limit  := 10000;
    t_customers.EXTEND;
    t_customers(t_customers.LAST).customer_name := 'XYZ Inc';
    t_customers(t_customers.LAST).credit_limit  := 20000;
    tmp_string := 'The number of customers is ' || t_customers.COUNT;
    insert into pkgc_log (id, line) values (1, tmp_string);
    FOR indx in 1 .. t_customers.COUNT LOOP
        rec := t_customers(indx);
        tmp_string := 'RECORD: ' || rec.customer_name || ', ' ||
 rec.credit_limit;
        insert into pkgc_log (id, line) values (1+ indx, tmp_string);
    END LOOP;
END;

BEGIN
  DELETE FROM pkgc_log;
END pkgc;
```

The resulting Postgres Pro package looks as follows.

```
BEGIN;
DROP SCHEMA IF EXISTS PKGC CASCADE;
CREATE SCHEMA PKGC;

CREATE TYPE PKGC.r_customer_type AS (
    customer_name varchar(50),
    credit_limit numeric(10,2)
);
```

```
CREATE DOMAIN PKGC.t_customer_type /*VARRAY(2)
    OF*/ PKGC.r_customer_type[];

CREATE OR REPLACE FUNCTION PKGC.__INIT__() RETURNS VOID AS $$
#package
#export off

BEGIN
  DELETE FROM pkgc_log;
END;
$$ LANGUAGE plpgsql;

CREATE OR REPLACE PROCEDURE PKGC.VARRAY_TEST() AS $$
#package
DECLARE
    /*WARNING: collection constructors are not supported.*/
    t_customers PKGC.t_customer_type /*:=*/ /*t_customer_type() /
*WARNING: varray constructors are not supported.*/*/;
    rec PKGC.r_customer_type;
    tmp_string varchar(2000);
BEGIN
    t_customers = array_cat(t_customers,
 array_fill(NULL::PKGC.R_CUSTOMER_TYPE, ARRAY[1]));
    t_customers[array_upper(T_CUSTOMERS, 1)].customer_name = 'ABC
 Corp';
    t_customers[array_upper(T_CUSTOMERS, 1)].credit_limit  = 10000;

    t_customers = array_cat(t_customers,
 array_fill(NULL::PKGC.R_CUSTOMER_TYPE, ARRAY[1]));
    t_customers[array_upper(T_CUSTOMERS, 1)].customer_name = 'XYZ
 Inc';
    t_customers[array_upper(T_CUSTOMERS, 1)].credit_limit  = 20000;

    tmp_string = 'The number of customers is ' ||
 array_length(T_CUSTOMERS, 1);
    insert into pkgc_log (id, line) values (1, tmp_string);

    FOR indx in 1 .. array_length(T_CUSTOMERS, 1) LOOP
        rec = t_customers[indx];
        tmp_string = 'RECORD: ' || rec.customer_name || ', ' ||
 rec.credit_limit;
        insert into pkgc_log (id, line) values (1+ indx, tmp_string);
    END LOOP;
END; $$ LANGUAGE PLPGSQL;
/*end pkgc;*/

COMMIT;
```

Exporting Associative Arrays

You can export Oracle collections of associative arrays as collections of pg_variables using ora2pgpro. ora2pgpro converts calls to collection methods into calls to pg_variables functions emulating these methods.

This is an example of an Oracle package.

```
PACKAGE ASSOC IS
    TYPE NumList IS TABLE OF INTEGER INDEX BY PLS_INTEGER;

    n NumList := NumList();
    m NumList := NumList();

    PROCEDURE TestExists;
END;

PACKAGE BODY ASSOC IS

    PROCEDURE TestExists IS
        -- local variable with the same name as the global variable m
        m NumList := NumList();
    BEGIN
        -- make some data
        n(1) := 1 * 1;
        n(2 + 1) := 9 / 3;
        n(5) := 55;
        n(7) := 77;

        m(6) := 66;
        m(8) := 88;

        -- read the data
        dbms_output.put_line('Init values:');

        dbms_output.put_line('n(1) = ' || n(1));
        dbms_output.put_line('n(n.next(1)) = ' || n(n.next(1)));

        dbms_output.put_line('m(6) = ' || m(6));
        dbms_output.put_line('ASSOC.m(6) = ' || ASSOC.m(6));

        -- call the collections DELETE method
        n.DELETE(3);
        m.DELETE(6);

        dbms_output.put_line('Checks:');

        -- global n collection
        IF n.EXISTS(1) THEN
            dbms_output.put_line('OK, element n(1) exists.');
        END IF;
        IF n.EXISTS(3) = FALSE THEN
            dbms_output.put_line('OK, element n(3) has been
 deleted.');
        END IF;
        IF n.EXISTS(5) = TRUE THEN
            dbms_output.put_line('OK, element n(5) is in place.');
        END IF;
        IF n.EXISTS(99) = FALSE THEN
```

```
            dbms_output.put_line('OK, element n(99) does not exist at
 all.');
        END IF;

        -- local m collection
        IF m.EXISTS(6) = FALSE THEN
            dbms_output.put_line('OK, element m(6) has been
 deleted.');
        END IF;

        -- global m collection
        IF ASSOC.m.EXISTS(6) = TRUE THEN
            dbms_output.put_line('OK, element ASSOC.m(6) is in
 place.');
        END IF;

    END;
BEGIN
    -- put an element to the global m collection
    m(6) := 666;
END;
```

The resulting Postgres Pro package looks as follows.

```
BEGIN;
DROP SCHEMA IF EXISTS ASSOC CASCADE;
CREATE SCHEMA ASSOC;
    /*TYPE NumList IS TABLE OF INTEGER INDEX BY PLS_INTEGER;*/

    /*n NumList := NumList();*/
    /*m NumList := NumList();*/

CREATE OR REPLACE FUNCTION ASSOC.__INIT__() RETURNS VOID AS $$
#package
BEGIN
    -- put an element to the global m collection
    perform pgv_set_elem('ASSOC', 'm', 6, 666); /*m(6) := 666;*/
END;
$$ LANGUAGE plpgsql;

    CREATE OR REPLACE PROCEDURE ASSOC.TestExists() AS $$
    #package
    DECLARE
        -- local variable with the same name as the global variable m
        /*m NumList := NumList();*/
    BEGIN
        -- make some data
        perform pgv_set_elem('ASSOC', 'n', 1, 1 * 1); /*n(1) := 1 *
 1;*/
        perform pgv_set_elem('ASSOC', 'n', 2 + 1, 9 / 3); /*n(2 +
 1) := 9 / 3;*/
        perform pgv_set_elem('ASSOC', 'n', 5, 55); /*n(5) := 55;*/
        perform pgv_set_elem('ASSOC', 'n', 7, 77); /*n(7) := 77;*/
```

```
        perform pgv_set_elem('ASSOC', 'TESTEXISTS.m', 6, 66); /
*m(6) := 66;*/
        perform pgv_set_elem('ASSOC', 'TESTEXISTS.m', 8, 88); /
*m(8) := 88;*/

        -- read the data
        CALL dbms_output.put_line('Init values:');

        CALL dbms_output.put_line('n(1) = ' || pgv_get_elem('ASSOC',
 'n', 1, NULL::INTEGER));
        CALL dbms_output.put_line('n(n.next(1)) = ' ||
 pgv_get_elem('ASSOC', 'n', pgv_next('ASSOC', 'n', 1),
 NULL::INTEGER));

        CALL dbms_output.put_line('m(6) = ' || pgv_get_elem('ASSOC',
 'TESTEXISTS.m', 6, NULL::INTEGER));
        CALL dbms_output.put_line('ASSOC.m(6) = ' ||
 pgv_get_elem('ASSOC', 'm', 6, NULL::INTEGER));

        -- call the collections DELETE method
        PERFORM pgv_remove_elem('ASSOC', 'n', 3);
        PERFORM pgv_remove_elem('ASSOC', 'TESTEXISTS.m', 6);

        CALL dbms_output.put_line('Checks:');

        -- global n collection
        IF pgv_exists_elem('ASSOC', 'n', 1) THEN
            CALL dbms_output.put_line('OK, element n(1) exists.');
        END IF;
        IF pgv_exists_elem('ASSOC', 'n', 3) = FALSE THEN
            CALL dbms_output.put_line('OK, element n(3) has been
 deleted.');
        END IF;
        IF pgv_exists_elem('ASSOC', 'n', 5) = TRUE THEN
            CALL dbms_output.put_line('OK, element n(5) is in
 place.');
        END IF;
        IF pgv_exists_elem('ASSOC', 'n', 99) = FALSE THEN
            CALL dbms_output.put_line('OK, element n(99) does not
 exist at all.');
        END IF;

        -- local m collection
        IF pgv_exists_elem('ASSOC', 'TESTEXISTS.m', 6) = FALSE THEN
            CALL dbms_output.put_line('OK, element m(6) has been
 deleted.');
        END IF;

        -- global m collection
        IF pgv_exists_elem('ASSOC', 'm', 6) = TRUE THEN
            CALL dbms_output.put_line('OK, element ASSOC.m(6) is in
 place.');
        END IF;
```

```
    END; $$ LANGUAGE PLPGSQL;
/*END;*/

COMMIT;
```

Note that in the above package you have to replace all the calls `CALL dbms_output` to `PERFORM dbms_output`.

Now let's look at the result of calling the `testexists` procedure in Oracle:

```
call assoc.testexists();
/

Init values:
n(1) = 1
n(n.next(1)) = 3
m(6) = 66
ASSOC.m(6) = 666
Checks:
OK, element n(1) exists.
OK, element n(3) has been deleted.
OK, element n(5) is in place.
OK, element n(99) does not exist at all.
OK, element m(6) has been deleted.
OK, element ASSOC.m(6) is in place.
```

And compare it with the corresponding result in Postgres Pro:

```
test=# do $$begin perform dbms_output.enable(); call
 ASSOC.TestExists(); end;$$ language plpgsql; select unnest(lines)
 from dbms_output.get_lines(15);
DO
                  unnest
-----------------------------------------
 Init values:
 n(1) = 1
 n(n.next(1)) = 3
 m(6) = 66
 ASSOC.m(6) = 666
 Checks:
 OK, element n(1) exists.
 OK, element n(3) has been deleted.
 OK, element n(5) is in place.
 OK, element n(99) does not exist at all.
 OK, element m(6) has been deleted.
 OK, element ASSOC.m(6) is in place.
(12 rows)
```

Appendix A. Release Notes

ora2pgpro 24.2.1

Release date: 2024-06-20

This release provides optimizations and bug fixes. Major changes are as follows.

- Implemented a mechanism that converts package procedure calls into function calls and vice versa. It is necessary for the orafce package support, namely the `dbms_output` package.
- Implemented conversion of private variables, procedures, and functions using the `#export`, `#private`, and `#export off` modifiers when exporting or importing packages.
- Fixed an issue that resulted in errors like “FATAL: Can't open XXX: No such file or directory” when using the `SHOW_REPORT` type without `--basedir`.
- Fixed an issue with searching for the `ora2pgpro.conf` file. Now the path to this file is generated correctly.
- Added support for Ubuntu 24/04, Astra Linux 1.8, Red OS Murom 8 and ended support for Astra Linux Orel 2.12 and Astra Linux 1.6.

ora2pgpro 24.1.1

Release date: 2024-03-25

This release provides optimizations and bug fixes. Major changes are as follows.

- Fully redesigned implementation of package export and import. Now Oracle packages are converted to Postgres Pro packages automatically as correctly as possible, with messages in code parts that require programmer's attention.
- Implemented the ability to export VARRAY as Postgres Pro arrays.
- Implemented the ability to export Oracle collections of associative arrays as collections from pg_variables.
- Added the `--offline` option that allows converting exported data without the connection to the Oracle database.

> **Note**
>
> ora2pgpro processes functions and procedures not defined in a package in the same way as previous versions. No support for VARRAY or Oracle collections of associative arrays was added. The new functionality is available in package conversion only.

ora2pgpro 23.2.1

Release date: 2023-09-22

This is the first public release of ora2pgpro™. It is shipped as a package compatible with Postgres Pro Enterprise™.

Major features are as follows:

- Export full database schema (tables, views, sequences, indexes), with unique, primary, foreign key, and check constraints.
- Export grants/privileges for users and groups.
- Export range/list partitions and subpartitions.
- Export a table selection (by specifying the table names).
- Export Oracle schema to a Postgres Pro schema.
- Export predefined functions, triggers, procedures, packages, and package bodies.
- Export full data or following a `WHERE` clause.
- Full support of Oracle BLOB objects as Postgres Pro bytea.
- Export Oracle views as Postgres Pro tables.
- Export Oracle user-defined types.
- Provide some basic automatic conversion of PL/SQL code to PL/pgSQL.
- Works on any platform.
- Export Oracle tables as foreign data wrapper tables.
- Export materialized views.
- Show a report of an Oracle database content.
- Migration cost assessment of an Oracle database.
- Migration difficulty level assessment of an Oracle database.
- Migration cost assessment of PL/SQL code from a file.
- Migration cost assessment of Oracle SQL queries stored in a file.
- Export Oracle locator and spatial geometries into PostGIS.
- Export `DBLINK` as Oracle FDW.
- Export `SYNONYM`s as views.
- Export `DIRECTORY` as external table or directory for `external_file` extension.
- Dispatch a list of SQL orders over multiple Postgres Pro connections.
- Perform a diff between Oracle and Postgres Pro database for test purposes.