Module N2

Schéma et données

22.09

Dalibo SCOP

https://dalibo.com/formations

Schéma et données

Module N2

TITRE : Schéma et données
SOUS-TITRE : Module N2

REVISION: 22.09
DATE: 02 septembre 2022

Remerciements : Ce manuel de formation est une aventure collective qui se transmet au sein de notre société depuis des années. Nous remercions chaleureusement ici toutes les personnes qui ont contribué directement ou indirectement à cet ouvrage, notamment : Jean-Paul Argudo, Alexandre Anriot, Carole Arnaud, Alexandre Baron, David Bidoc, Sharon Bonan, Franck Boudehen, Arnaud Bruniquel, Damien Clochard, Christophe Courtois, Marc Cousin, Gilles Darold, Jehan-Guillaume de Rorthais, Ronan Dunklau, Vik Fearing, Stefan Fercot, Pierre Giraud, Nicolas Gollet, Dimitri Fontaine, Florent Jardin, Virginie Jourdan, Luc Lamarle, Denis Laxalde, Guillaume Lelarge, Benoit Lobréau, Jean-Louis Louër, Thibaut Madelaine, Adrien Nayrat, Alexandre Pereira, Flavie Perette, Robin Portigliatti, Thomas Reiss, Maël Rimbault, Julien Rouhaud, Stéphane Schildknecht, Julien Tachoires, Nicolas Thauvin, Be Hai Tran, Christophe Truffier, Cédric Villemain, Thibaud Walkowiak, Frédéric Yhuel.

LICENCE CREATIVE COMMONS BY-NC-SA 2.0 FR

Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions

Vous êtes autorisé à :

- Partager, copier, distribuer et communiquer le matériel par tous moyens et sous tous formats
- Adapter, remixer, transformer et créer à partir du matériel

Dalibo ne peut retirer les autorisations concédées par la licence tant que vous appliquez les termes de cette licence selon les conditions suivantes :

Attribution : Vous devez créditer l'œuvre, intégrer un lien vers la licence et indiquer si des modifications ont été effectuées à l'œuvre. Vous devez indiquer ces informations par tous les moyens raisonnables, sans toutefois suggérer que Dalibo vous soutient ou soutient la façon dont vous avez utilisé ce document.

Pas d'Utilisation Commerciale : Vous n'êtes pas autorisé à faire un usage commercial de ce document, tout ou partie du matériel le composant.

Partage dans les Mêmes Conditions : Dans le cas où vous effectuez un remix, que vous transformez, ou créez à partir du matériel composant le document original, vous devez diffuser le document modifié dans les même conditions, c'est à dire avec la même licence avec laquelle le document original a été diffusé.

Pas de restrictions complémentaires : Vous n'êtes pas autorisé à appliquer des conditions légales ou des mesures techniques qui restreindraient légalement autrui à utiliser le document dans les conditions décrites par la licence.

Note : Ceci est un résumé de la licence. Le texte complet est disponible ici :

https://creativecommons.org/licenses/by-nc-sa/2.0/fr/legalcode

Pour toute demande au sujet des conditions d'utilisation de ce document, envoyez vos questions à contact@dalibo.com[1] !

[1] mailto:contact@dalibo.com

Chers lectrices & lecteurs,

Nos formations PostgreSQL sont issues de nombreuses années d'études, d'expérience de terrain et de passion pour les logiciels libres. Pour Dalibo, l'utilisation de PostgreSQL n'est pas une marque d'opportunisme commercial, mais l'expression d'un engagement de longue date. Le choix de l'Open Source est aussi le choix de l'implication dans la communauté du logiciel.

Au-delà du contenu technique en lui-même, notre intention est de transmettre les valeurs qui animent et unissent les développeurs de PostgreSQL depuis toujours : partage, ouverture, transparence, créativité, dynamisme... Le but premier de nos formations est de vous aider à mieux exploiter toute la puissance de PostgreSQL mais nous espérons également qu'elles vous inciteront à devenir un membre actif de la communauté en partageant à votre tour le savoir-faire que vous aurez acquis avec nous.

Nous mettons un point d'honneur à maintenir nos manuels à jour, avec des informations précises et des exemples détaillés. Toutefois malgré nos efforts et nos multiples relectures, il est probable que ce document contienne des oublis, des coquilles, des imprécisions ou des erreurs. Si vous constatez un souci, n'hésitez pas à le signaler via l'adresse formation@dalibo.com !

Table des Matières

1 SCHÉMA ET DONNÉES

1.1 INTRODUCTION

Ce module est organisé en trois parties :

- Configuration d'Ora2Pg
- Migration du schéma
- Migration des données

Ce module a pour but de montrer la configuration et l'utilisation d'Ora2Pg.

1.2 CONFIGURATION D'ORA2PG

Étapes de la configuration :

- Syntaxe du fichier de configuration
- Connexion et schéma Oracle
- Validation de la configuration
- La base Oracle vue par Ora2Pg
- Estimation de la charge de migration
- Création d'une configuration générique

Nous allons aborder ici les différentes étapes de la configuration d'Ora2Pg :

- la configuration en elle-même ;
- comment se connecter à la base Oracle ?
- comment valider la configuration ?
- que contient la base de données et comment Ora2Pg va l'exporter ?
- comment estimer la charge de la migration ?
- comment créer un fichier de configuration générique ?

1.2.1 STRUCTURE DU FICHIER

Structure

- Fichier de configuration simple
- Les lignes en commentaires débutent par un dièse (`#`)
- Les variables sont en majuscules
- Plusieurs paramètres sont du type binaire : `0` pour désactivé et `1` pour activé

Chaque ligne non commentée doit commencer par l'une des clés de configuration. Il y en a environ 86.

La valeur de cette clé est variable. La directive de configuration et sa valeur doivent être séparées par une ou plusieurs tabulations.

Lorsque la valeur est une liste, le séparateur des éléments de la liste est généralement le caractère espace.

```
SKIP        fkeys pkeys ukeys indexes checks
```

Toutes les clés dont la valeur peut être une liste peuvent être répétées plusieurs fois, exemple :

```
SKIP        fkeys pkeys ukeys
SKIP        indexes checks
```

Pour les autres, si elles sont répétées, la dernière valeur indiquée sera la valeur prise en compte.

1.2.2 CONFIGURATION LOCALE

- `IMPORT fichier.conf`
- `ORACLE_HOME /path/.../`
- `DEBUG [0|1]`
- `LOGFILE /path/.../migration.log`

IMPORT

Cette variable permet d'inclure un fichier de configuration dans le fichier `ora2pg.conf`. Ainsi on peut définir les variables communes à toutes les configurations dans un seul fichier, qu'on inclut dans tous les autres.

Par exemple :

```
IMPORT common.conf
```

Le fichier de configuration importé est chargé au moment où la directive `IMPORT` apparaît dans le fichier de configuration. Si les directives importées se retrouvent aussi plus loin dans le fichier de configuration, elles seront écrasées.

ORACLE_HOME

Cette variable très connue dans le monde Oracle permet de déterminer où se trouve le répertoire contenant toutes les bibliothèques Oracle ainsi que les autres fichiers d'un client (ou d'un serveur) Oracle.

Par exemple, pour un serveur Oracle 18c Express Edition, le `ORACLE_HOME` ressemble à cela :

```
ORACLE_HOME /opt/oracle/product/18c/dbhomeXE
```

Pour un client de la même version, on peut avoir :

```
ORACLE_HOME /usr/lib/oracle/18.5/client64
```

Si la variable d'environnement `ORACLE_HOME` était définie au moment de l'installation, ce paramètre possède alors déjà la bonne valeur.

DEBUG

Lorsque `DEBUG` est positionné à `1`, Ora2Pg va envoyer tous les messages d'information, y compris les messages d'erreur, sur la console.

Si cette variable est positionnée à `0`, alors Ora2Pg restera muet.

Il est recommandé de le désactiver par défaut et, s'il doit être activé, de rediriger la sortie standard dans un fichier ou d'utiliser un fichier de traces en donnant le chemin complet à la directive `LOGFILE`.

LOGFILE

La valeur de cette directive correspond à un fichier dans lequel seront ajoutés tous les messages retournés par Ora2Pg. Ceci permet notamment de garder la trace complète des messages de la migration pour s'assurer qu'il n'y a pas eu de messages d'erreur.

1.2.3 CONNEXION À ORACLE

- ORACLE_DSN
  - dbi:Oracle:host=serveur;sid=INSTANCE
- ORACLE_USER
  - system
- ORACLE_PWD
  - manager
- SCHEMA
  - NOM_SCHEMA versus SYSUSERS
- USER_GRANTS [0|1]
  - l'utilisateur Oracle a-t-il les droits DBA ?

ORACLE_DSN

Cette variable permet de déterminer la chaîne de connexion au serveur Oracle. On y trouve en particulier :

- le connecteur DBI à utiliser : `dbi:Oracle`
- le nom du serveur (ou son adresse IP) : `host=`
- le nom de l'instance Oracle: `sid=`

Voici par exemple la chaîne de connexion permettant de se connecter à l'instance DB_SID sur le serveur Oracle `oracle_server` :

```
ORACLE_DSN      dbi:Oracle:host=oracle_server;sid=DB_SID
```

Il est possible aussi d'utiliser une notation plus simple :

```
ORACLE_DSN      dbi:Oracle:DB_SID
```

mais ceci implique que l'instance DB_SID (dans cet exemple) soit connue et accessible par la machine où Ora2Pg va fonctionner. Pour cela, il suffit de déclarer son alias dans le fichier `$ORACLE_HOME/network/admin/tnsnames.ora` :

```
$ cat <<EOF >> $ORACLE_HOME/network/admin/tnsnames.ora
  XE = (DESCRIPTION =
    (ADDRESS = (PROTOCOL = TCP) (HOST = 192.168.1.10) (port = 1521))
    (CONNECT_DATA = (SERVER = DEDICATED) (SERVICE_NAME = PDB_NAME))
  )
EOF
```

On peut tester cela simplement avec des outils comme `tnsping` ou encore `sqlplus`.

Pour MySQL un datasource typique sera de la forme :

```
ORACLE_DSN  dbi:mysql:host=192.168.1.10;database=sakila;port=3306
```

la partie SID propre à Oracle database est remplacée ici par `database`.

ORACLE_USER et ORACLE_PWD

On définit avec ces variables l'utilisateur et le mot de passe avec lesquels Ora2Pg va se connecter au serveur `Oracle` pour en extraire des informations (schéma, données, etc.).

Il est préférable que cet utilisateur soit déclaré comme un `SYSDBA` dans `Oracle`, c'est-à-dire un utilisateur privilégié de type DBA (un peu comme l'utilisateur `postgres` l'est généralement pour un serveur PostgreSQL).

L'export des droits (`GRANT`) sur les objets de la base de données et les `TABLESPACES` ne peuvent être réalisés que par un utilisateur privilégié.

SCHEMA

Cette variable permet de déterminer le schéma ou utilisateur Oracle dont les objets ou données seront exportés. Le paramètre `ORACLE_USER` défini précédemment dans le fichier de configuration doit avoir les droits nécessaires sur les objets de ce schéma.

Par exemple, pour exporter les objets du schéma `HR` de la base de données de démonstration d'Oracle 10g XE (Express Edition) :

```
SCHEMA  HR
```

Si aucun schéma n'est précisé, les objets ou données de tous les schémas de l'instance seront exportés hormis ceux définis dans le paramètre `SYSUSERS`.

SYSUSERS

Ce paramètre permet d'exclure, à l'origine, tous les utilisateurs système d'Oracle et leur schéma qui, parfois, contiennent des tables systèmes qui sont superflues pour une migration vers PostgreSQL.

À ce jour, les utilisateurs ignorés par Ora2Pg sont les suivants :

```
CTXSYS DBSNMP EXFSYS LBACSYS MDSYS MGMT_VIEW OLAPSYS ORDDATA OWBSYS
ORDPLUGINS ORDSYS OUTLN SI_INFORMTN_SCHEMA SYS SYSMAN SYSTEM 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_040200 DVSYS OJVMSYS GSMADMIN_INTERNAL APPQOSSYS APEX_040000
```

On peut utiliser cette fonctionnalité d'une manière détournée pour ignorer les objets appartenant à d'autres utilisateurs.

Tout utilisateur spécifié dans la clause `SYSUSERS` sera ignoré, en plus des utilisateurs ignorés par défaut (voir liste ci-dessus).

Par exemple, si on veut ignorer les objets des utilisateurs `RECETTE` et `DEV` :

```
SYSUSERS      RECETTE,DEV
```

USER_GRANTS

Ce paramètre est par défaut à `0` car Ora2Pg part du principe que l'on utilise un utilisateur privilégié (membre du groupe `DBA`, comme `SYSTEM`) pour, par exemple, exporter la définition des objets.

En effet, Ora2Pg utilise intensivement les vues de type `DBA_...`. Or, un utilisateur non privilégié n'a pas accès à ces vues, réservées aux administrateurs de la base de données. On peut alors configurer `USER_GRANTS` à `1` pour utiliser un utilisateur `Oracle` non DBA. Dans ce cas, Ora2Pg utilisera les vues de type `ALL_...` pour récupérer la définition des objets.

À noter, qu'alors, cela ne fonctionnera pas avec les types d'export `GRANT` et `TABLESPACE` qui doivent impérativement être réalisés par un utilisateur avec les privilèges `DBA`. L'analyse de requêtes applicatives dans la table `DBA_AUDIT_TRAIL` (export type `QUERY`), nécessite aussi ce privilège.

Dans la mesure où le fichier `ora2pg.conf` va contenir des informations sensibles, il est recommandé de prendre garde aux droits qui sont associés à ce fichier et, si possible, de positionner des droits à `0` pour tout utilisateur autre que le propriétaire et le groupe associés au fichier :

```
$ chown 660 /etc/ora2pg/ora2pg.conf
```

1.3 VALIDATION DE LA CONFIGURATION

Cette étape de validation de la configuration permet d'obtenir des informations sur la base Oracle :

- Liste des tables et colonnes
- Recherche de l'encodage de la base
- Création d'un rapport de migration
- Estimation du coût de migration

1.3.1 DÉCOUVERTE DE LA BASE

Certaines informations sont disponibles immédiatement, sans plus de configuration :

- `SHOW_VERSION` affiche la version de l'instance Oracle.
- `SHOW_SCHEMA` liste les schémas définis sous Oracle.
- `SHOW_TABLE` affiche la liste des tables de la base Oracle.
- `SHOW_COLUMN` affiche la liste des colonnes par table d'une base Oracle.

Pour tester que les paramètres de connexion à l'instance Oracle sont les bons, on peut utiliser les actions de rapports simples d'Ora2Pg qui ne nécessitent que la configuration des variables de connexion.

Par exemple, pour l'instance d'exemple fournie par Oracle XE et le schéma HR, la commande :

```
ora2pg -t SHOW_SCHEMA
```

permettra de lister tous les schémas de l'instance Oracle pour trouver la bonne valeur à donner à la directive `SCHEMA` dans le fichier de configuration.

La commande :

```
ora2pg -t SHOW_TABLE
```

donne la liste des tables qui seront exportées et le nombre d'enregistrements pour chaque table :

```
[1] TABLE COUNTRIES (owner: HR, 25 rows)
[2] TABLE DEPARTMENTS (owner: HR, 27 rows)
[3] TABLE EMPLOYEES (owner: HR, 107 rows)
[4] TABLE JOBS (owner: HR, 19 rows)
[5] TABLE JOB_HISTORY (owner: HR, 10 rows)
[6] TABLE LOCATIONS (owner: HR, 23 rows)
[7] TABLE REGIONS (owner: HR, 4 rows)
```

Si des tables sont non loguées (*unlogged tables*), correspondent à des tables externes ou sont partitionnées, Ora2Pg l'affichera à côté du nom de la table.

```
[19] UNLOGGED TABLE REGIONS (owner: HR, 4 rows)
[20] PARTITIONED TABLE SALES_PARTITIONED (owner: HR, 0 rows) - 2 partitions
```

L'utilisation de l'action `SHOW_COLUMN` :

```
ora2pg -t SHOW_COLUMN -a COUNTRIES
```

renvoie le détail des colonnes de la table `COUNTRIES` et notamment les correspondances des types de colonnes qui seront utilisés pour la migration :

```
[1] TABLE COUNTRIES (owner: HR, 25 rows)
    COUNTRY_ID : CHAR(2) => char(2)
    COUNTRY_NAME : VARCHAR2(40) => varchar(40)
    REGION_ID : NUMBER(22) => bigint
```

S'il s'agit d'une table contenant des objets géométriques avec une contrainte sur le type d'objet, Ora2Pg donnera son équivalent PostGIS :

```
[1] TABLE TRAJETS (owner: HR, 1 rows)
    MKT_ID : NUMBER(22) => bigint
    NAME : VARCHAR2(32) => varchar(32)
    START_POINT : SDO_GEOMETRY => geometry(POINT,4326)
    FINISH_POINT : SDO_GEOMETRY => geometry(GEOMETRY,4326) - POINT,LINESTRING
```

1.3.2 GESTION DE L'ENCODAGE - 1

Recherche de l'encodage utilisé par l'instance Oracle :

- `SHOW_ENCODING` : affiche les valeurs utilisées par Ora2Pg pour
  - `NLS_LANG`
  - `CLIENT_ENCODING`
- `NLS_LANG`
  - `AMERICAN_AMERICA.AL32UTF8`
  - `French_France.WE8ISO8895P1`...
- `NLS_NCHAR`
  - `AL32UTF8`...

SHOW_ENCODING

```
ora2pg -t SHOW_ENCODING -c ../ora2pg.conf
```

Ceci retournera les valeurs `NLS_LANG`, `NLS_NCHAR` et `CLIENT_ENCODING`, qui seront utilisées par Ora2Pg, mais aussi l'encodage réel de la base Oracle et de l'encodage correspondant dans PostgreSQL. Par exemple :

```
Current encoding settings that will be used by Ora2Pg:
    Oracle NLS_LANG AMERICAN_AMERICA.AL32UTF8
    Oracle NLS_NCHAR AL32UTF8
    Oracle NLS_TIMESTAMP_FORMAT YYYY-MM-DD HH24:MI:SS.FF6
    Oracle NLS_DATE_FORMAT YYYY-MM-DD HH24:MI:SS
    PostgreSQL CLIENT_ENCODING UTF8
    Perl output encoding ''
Showing current Oracle encoding and possible PostgreSQL client encoding:
    Oracle NLS_LANG AMERICAN_AMERICA.AL32UTF8
    Oracle NLS_NCHAR AL32UTF8
```

```
Oracle NLS_TIMESTAMP_FORMAT YYYY-MM-DD HH24:MI:SS.FF6
Oracle NLS_DATE_FORMAT YYYY-MM-DD HH24:MI:SS
PostgreSQL CLIENT_ENCODING UTF8
```

NLS_LANG et NLS_CHAR

Par défaut, Ora2Pg va utiliser l'encodage `AMERICAN_AMERICA.AL32UTF8` au niveau du client Oracle. Il est toutefois possible de le changer et de forcer sa valeur avec la variable de configuration `NLS_LANG`. De même, la variable de session `NLS_NCHAR` a la valeur `AL32UTF8` par défaut.

Il est fortement conseillé de conserver le comportement par défaut d'Ora2Pg pour éviter les erreurs liées à l'encodage, mais on peut le changer si l'on veut éviter le coût de l'encodage ou qu'une table Oracle ne respecte pas l'encodage lors de l'export des données. Dans ce cas, le `NLS_LANG` doit correspondre au paramétrage obtenu lorsqu'on ouvre une session sur Oracle avec l'utilisateur Oracle spécifié dans la configuration d'Ora2Pg. Pour cela, on se connecte à l'instance avec cet utilisateur, et on peut lire le paramétrage `NLS` (acronyme de *National Language Support*) comme suit :

```
$ sqlplus hr/secret@xe

SQL> set pages 80;
SQL> select * from nls_session_parameters;

PARAMETER                      VALUE
------------------------------ ----------------------------------------
NLS_LANGUAGE                   FRENCH
NLS_TERRITORY                  FRANCE
NLS_CURRENCY                   €
NLS_ISO_CURRENCY               FRANCE
NLS_NUMERIC_CHARACTERS         ,
NLS_CALENDAR                   GREGORIAN
NLS_DATE_FORMAT                DD/MM/RR
NLS_DATE_LANGUAGE              FRENCH
NLS_SORT                       FRENCH
NLS_TIME_FORMAT                HH24:MI:SSXFF
NLS_TIMESTAMP_FORMAT           DD/MM/RR HH24:MI:SSXFF
NLS_TIME_TZ_FORMAT             HH24:MI:SSXFF TZR
NLS_TIMESTAMP_TZ_FORMAT        DD/MM/RR HH24:MI:SSXFF TZR
NLS_DUAL_CURRENCY              €
NLS_COMP                       BINARY
NLS_LENGTH_SEMANTICS           BYTE
NLS_NCHAR_CONV_EXCP            FALSE

17 ligne(s) sélectionnée(s).
```

On peut aussi exécuter une requête pour récupérer le paramétrage de l'instance toute entière avec :

```
SELECT * FROM nls_instance_parameters ;
```

Ce paramétrage au niveau instance se modifie avec l'ordre `ALTER SYSTEM`, ainsi qu'au niveau de la base de données :

```
SELECT * FROM nls_database_parameters;
```

Ce paramétrage au niveau base de données ne se modifie pas, il est défini lors de la création de la base de données avec un `SET`.

1.3.3 GESTION DE L'ENCODAGE - 2

- `CLIENT_ENCODING`
  - `utf8`, `latin1`, `latin9`
- `BINMODE`
  - `utf8`, `raw`

CLIENT_ENCODING

Par défaut la valeur de cette directive est `UTF8`, c'est celle qui correspond à l'encodage unicode utilisé pour extraire les données d'Oracle.

Si le `NLS_LANG` a été modifié dans le fichier de configuration alors pour que la conversion des données en provenance d'Oracle vers PostgreSQL soit exacte, il faut définir l'encodage à utiliser par le client PostgreSQL. Ainsi, si la variable `NLS_LANG` côté connexion Oracle est `FRENCH_FRANCE.WE8ISO8859P1`, il faudra utiliser l'encodage `LATIN1` côté client PostgreSQL pour ne pas avoir de problème de conversion d'encodage des données.

Pour vous aider à trouver le jeu de caractères dans PostgreSQL correspondant à celui sous Oracle, vous pouvez consulter ce document, 22.3. Character Set Support[2] , qui fait partie de la documentation officielle de PostgreSQL.

BINMODE

Par défaut le paramètre est positionné à `utf8` si `NLS_LANG` utilise un encodage unicode. Il n'est donc normalement pas nécessaire de modifier cette variable de configuration. Lors de l'utilisation d'un encodage unicode, il est indispensable de le positionner à la valeur `utf8` pour éviter les erreurs d'écriture Perl de type `Wide character in print`.

---

[2] https://www.postgresql.org/docs/current/static/multibyte.html

1.3.4 RAPPORT DE MIGRATION

- Rapport exhaustif du contenu de la base Oracle

```
ora2pg -t SHOW_REPORT
ora2pg -t SHOW_REPORT --dump_as_html
```

- Estimation du coût de migration

```
ora2pg -t SHOW_REPORT --estimate_cost
ora2pg -t SHOW_REPORT --estimate_cost --dump_as_html
```

Rapport sur le contenu de la base Oracle

Ora2Pg dispose d'un mode d'analyse du contenu de la base Oracle afin de générer un rapport sur son contenu et présenter ce qui peut ou ne peut pas être exporté.

L'outil parcourt l'intégralité des objets, les dénombre, extrait les particularités de chacun d'eux et dresse un bilan exhaustif de ce qu'il a rencontré. Pour activer le mode « analyse et rapport », il faut utiliser l'export de type `SHOW_REPORT` par la commande suivante :

```
ora2pg -t SHOW_REPORT
```

Voici un exemple de rapport obtenu avec cette commande :

```
--------------------------------------------------------------------------------
Ora2Pg v20.0 - Database Migration Report
--------------------------------------------------------------------------------
Version Oracle Database 12c Enterprise Edition Release 12.1.0.2.0
Schema  HR
Size    28.56 MB

--------------------------------------------------------------------------------
Object  Number  Invalid Comments    Details
--------------------------------------------------------------------------------
DATABASE LINK   2   0   Database links will be exported as SQL/MED PostgreSQL's
                Foreign Data Wrapper (FDW) extensions using oracle_fdw.
GLOBAL TEMPORARY TABLE  0   0   Global temporary table are not supported by
                PostgreSQL and will not be exported. You will have to
                rewrite some application code to match the PostgreSQL
                temporary table behavior.
INDEX   28  0   19 index(es) are concerned by the export, others are automatically
                generated and will do so on PostgreSQL. Bitmap will be
                exported as btree_gin index(es) and hash index(es) will be
                exported as b-tree index(es) if any. Domain index are exported
                as b-tree but commented to be edited to mainly use FTS.
                Cluster, 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.
                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.
                3 function based b-tree index(es).
                13 b-tree index(es).
                3 spatial index index(es).
INDEX PARTITION 2   0   Only local indexes partition are exported, they are
                build on the column used for the partitioning.
JOB 0   0   Job are not exported. You may set external cron job with them.
PROCEDURE   3   1   Total size of procedure code: 1870 bytes.
SEQUENCE    3   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 0   0   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.
TABLE   22  0    2 check constraint(s). 1 unknown types. Total number of rows: 421.
                Top 10 of tables sorted by number of rows:.
                sg_infrastructure_route has 188 rows.
                employees has 107 rows. departments has 27 rows. countries has
                25 rows.
                locations has 23 rows. jobs has 19 rows. job_history has 10 rows.
                error_log_sample has 6 rows. emptyclob has 4 rows. regions has
                4 rows.
                Top 10 of largest tables:...
TABLE PARTITION 5   0   Partitions are exported using table inheritance and
                check constraint.
                Hash and Key partitions are not supported by PostgreSQL and will not
                be exported.
                5 RANGE partitions..
TRIGGER 3   1   Total size of trigger code: 736 bytes.
VIEW    1   0   Views are fully supported but can use specific functions.
------------------------------------------------------------------------------
Total   69  2
------------------------------------------------------------------------------
```

D'autres paramètres ne peuvent pas être analysés par Ora2Pg comme l'usage de l'application. Il existe aussi d'autres objets qui ne sont pas exportés directement par Ora2Pg comme les objets `DIMENSION` des fonctionnalités `OLAP` d'Oracle dans la mesure où ils n'ont pas d'équivalent dans PostgreSQL.

Évaluer la charge de migration d'une base Oracle

Pour déterminer le coût en jours/personne de la migration, Ora2Pg dispose d'une directive de configuration nommée `ESTIMATE_COST`. Celle-ci peut aussi être activée en ligne

de commande : `--estimate_cost`. Cette fonctionnalité n'est disponible qu'avec le type d'export `SHOW_REPORT`.

```
ora2pg -t SHOW_REPORT  --estimate_cost
```

Le rapport généré est identique à celui généré par `SHOW_REPORT`, mais cette fonctionnalité provoque en plus l'exploration des objets de la base de données, du code source des vues, triggers et routines stockées (fonctions, procédures et paquets de fonctions), puis donne un score à chaque objet et à chaque routine suivant le volume de code et la complexité de réécriture manuelle de ce code. En effet, la réécriture d'une routine comportant un `CONNECT BY` ne prend pas le même temps que la réécriture d'une routine comportant des appels à `GOTO`.

```
-------------------------------------------------------------------------------
Ora2Pg v20.0 - Database Migration Report
-------------------------------------------------------------------------------
Version Oracle Database 12c Enterprise Edition Release 12.1.0.2.0
Schema  HR
Size    28.56 MB

-------------------------------------------------------------------------------
Object  Number  Invalid Estimated cost  Comments    Details
-------------------------------------------------------------------------------
DATABASE LINK   2   0   6   Database links will be exported as SQL/MED PostgreSQL's
                    Foreign Data Wrapper (FDW) extensions using oracle_fdw.
GLOBAL TEMPORARY TABLE  0   0   0   Global temporary table are not supported by
                    PostgreSQL and will not be exported. You will have to
                    rewrite some application code to match the PostgreSQL
                    temporary table behavior.
INDEX   28  0   5.3 19 index(es) are concerned by the export, others are
                    automatically
                    generated and will do so on PostgreSQL. Bitmap will be
                    exported as btree_gin index(es) and hash index(es) will be
                    exported as b-tree index(es) if any. Domain index are exported
                    as b-tree but commented to be edited to mainly use FTS.
                    Cluster, 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.  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.
                    3 function based b-tree index(es). 13 b-tree index(es).
                    3 spatial index index(es).
INDEX PARTITION 2   0   0   Only local indexes partition are exported, they are
                    build on the column used for the partitioning.
```

```
JOB 0   0   0   Job are not exported. You may set external cron job with them.
PROCEDURE   3   1   16  Total size of procedure code: 1870 bytes.
                    add_job_history: 3. test_dupl_vazba: 7. secure_dml: 3.
SEQUENCE    3   0   1   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 0   0   0   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.
TABLE   22  0   2.4  2 check constraint(s). 1 unknown types. Total number of
                    rows: 421.
                    Top 10 of tables sorted by number of rows:.
                    sg_infrastructure_route
                    has 188 rows. employees has 107 rows. departments has 27 rows.
                    countries has 25 rows. locations has 23 rows. jobs has 19 rows.
                    job_history has 10 rows. error_log_sample has 6 rows.
                    regions has 4 rows.
                    emptyclob has 4 rows. Top 10 of largest tables:.
TABLE PARTITION 5   0   1   Partitions are exported using table inheritance
                    and check constraint.
                    Hash and Key partitions are not supported by PostgreSQL
                    and will not be exported.
                    5 RANGE partitions..
TRIGGER 3   1   9.3 Total size of trigger code: 736 bytes.  cisvpolpre_bi:
                    3.3. update_job_history: 3.
VIEW    1   0   1   Views are fully supported but can use specific functions.
--------------------------------------------------------------------------------
Total   69  2   42.0042.00 cost migration units means approximatively 1 man-day(s).
                    The migration unit was set to 5 minute(s)

--------------------------------------------------------------------------------
Migration level : B-5
--------------------------------------------------------------------------------

Migration levels:
    A - Migration that might be run automatically
    B - Migration with code rewrite and a human-days cost up to 5 days
    C - Migration with code rewrite and a human-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
```

```
-------------------------------------------------------------------------------

Details of cost assessment per function
Function test_dupl_vazba total estimated cost: 7
    CONCAT => 9 (cost: 0.1)
    TEST => 2
    SIZE => 1
    TO_CHAR => 1 (cost: 0.1)
    PRAGMA => 1 (cost: 3)
Function add_job_history total estimated cost: 3
    TEST => 2
    SIZE => 1
Function secure_dml total estimated cost: 3
    TEST => 2
    SIZE => 1
-------------------------------------------------------------------------------

Details of cost assessment per trigger
Trigger cisvpolpre_bi total estimated cost: 3.3
    CONCAT => 3 (cost: 0.1)
    TEST => 2
    SIZE => 1
Trigger update_job_history total estimated cost: 3
    TEST => 2
    SIZE => 1
-------------------------------------------------------------------------------
```

En fin de rapport, Ora2Pg affiche le nombre total d'objets rencontrés, les objets invalides et un nombre correspondant au nombre d'unités de coût de migration qu'il aura estimé nécessaire en fonction du code détecté (voir l'*Annexe 3 : Méthode de valorisation de la charge de migration*). Cette unité vaut par défaut 5 minutes, cela correspond au temps moyen que mettrait un spécialiste pour porter le code. Dans l'exemple ci-dessus, on a donc une estimation par Ora2Pg d'une migration ayant un coût de 162,5 unités multipliées par 5 minutes, ce qui correspond en gros à 2 jours/personne.

Il est possible d'ajuster le coût de l'unité en utilisant l'option :

```
--cost_unit_value
```

ou de la fixer avec la directive de configuration `COST_UNIT_VALUE`, comme suit :

```
ora2pg -t SHOW_REPORT  --estimate_cost --cost_unit_value 5
```

L'ajustement de cette valeur est à faire en fonction de l'expérience de l'équipe en charge de la migration. Pour la première migration, il est tout à fait raisonnable de doubler ce coût dans le fichier de configuration `ora2pg.conf` :

```
COST_UNIT_VALUE       10
```

ou en ligne de commande :

```
ora2pg -t SHOW_REPORT  --estimate_cost --cost_unit_value 10
```

Dans ce mode de rapport, Ora2Pg affiche aussi les détails du coût de migration estimé par routine.

Il est possible d'obtenir un rapport au format HTML en activant la directive `DUMP_AS_HTML` :

```
DUMP_AS_HTML       1
```

ou en utilisant l'option `--dump_as_html` en ligne de commande :

```
ora2pg -t SHOW_REPORT  --estimate_cost --cost_unit_value 10 --dump_as_html
```

Ora2Pg propose un exemple de rapport en HTML sur son site : Ora2Pg - Database Migration Report[3]

Par défaut, Ora2Pg affiche les dix tables les plus volumineuses en terme de nombre de lignes et le top dix des tables les plus volumineuses en taille (hors partitions). Le nombre de table affichées peut être contrôlé avec la directive de configuration `TOP_MAX`.

L'action `SHOW_REPORT` renvoie le rapport sur la sortie standard (`stdout`), il est donc conseillé de renvoyer la sortie dans un fichier pour pouvoir le consulter dans l'application adaptée à son format. Par exemple :

```
ora2pg -t SHOW_REPORT --estimate_cost --dump_as_html > report.html
```

1.4 CONFIGURATION GÉNÉRIQUE

Le but du fichier de configuration générique est multiple :

- éviter de faire des allers/retours en édition sur ce fichier
- éviter d'avoir une multitude de fichiers de configuration dédiés à chaque opération
- utiliser la souplesse des options de ligne de commande

Le but est d'avoir un fichier de configuration générique qui sera utilisé pour tous les types d'export et d'utiliser la souplesse des options en ligne de commande du script `ora2pg`.

---

[3] https://ora2pg.darold.net/report.html

1.4.1 FICHIERS DE SORTIE

Utilisation de fichiers de sortie dédiés

- `FILE_PER_CONSTRAINT` 1
- `FILE_PER_INDEX` 1
- `FILE_PER_FKEYS` 1
- `FILE_PER_TABLE` 1
- `FILE_PER_FUNCTION` 1

On commande d'abord à Ora2Pg de créer des fichiers de sortie différents pour les contraintes (`FILE_PER_CONSTRAINT`), les index (`FILE_PER_INDEX`) et les clés étrangères (`FILE_PER_FKEY`). Cela nous permettra de ne les importer qu'à la fin de la migration pour ne pas être gêné ou ralenti lors de l'import de données.

On peut aussi générer un fichier différent par table (`FILE_PER_TABLE`) lors de l'export des données et par routine (`FILE_PER_FUNCTION`) pour permettre un traitement individualisé.

1.4.2 ORDRES SQL ADDITIONNELS

- Ajout d'ordres SQL :
  - `DISABLE_TRIGGERS` 1
  - `TRUNCATE_TABLE` 1
  - `DISABLE_SEQUENCE` 1
  - `COMPILE_SCHEMA` [0|1]
- Désactivation de la conversion automatique du PL/SQL :
  - `PLSQL_PGSQL` 0

La désactivation des triggers pour chaque table avant l'import des données est réalisée, peu importe s'ils ont été importés auparavant ou non. Cela évitera leur déclenchement s'ils ont été importés et n'aura pas d'effet si ce n'est pas le cas, `DISABLE_TRIGGERS` doit donc être activé. Il est toutefois préférable de ne charger les triggers qu'à la fin.

Les deux directives `TRUNCATE_TABLE` et `DISABLE_SEQUENCE` permettent de déterminer le comportement lors de l'export des données, à savoir respectivement l'ajout des ordres SQL de troncature des tables avant le chargement des données et la désactivation des ordres de réinitialisation des séquences après le chargement, ces dernières n'étant importées qu'à la fin.

`COMPILE_SCHEMA` permet de demander à Oracle de vérifier à nouveau le code PL/SQL et de valider ce qui doit l'être. Par exemple, un trigger a pu être ajouté et faire appel à une fonction avant qu'elle ne soit créée, et sera dans ce cas marqué invalide par Oracle.

L'activation de cette variable permet de forcer Oracle à revalider le code. Par défaut ce comportement n'est pas activé, le code valide seul sera exporté.

La conversion automatique du code des routines stockées est désactivée pour pouvoir obtenir les sources du code. On utilisera l'option `-p` lors de l'exécution d'`ora2pg` afin de l'activer.

1.4.3 COMPORTEMENT CÔTÉ POSTGRESQL

Utilisation d'un schéma sous PostgreSQL ?

- `EXPORT_SCHEMA` [0|1]
- `PG_SCHEMA` nom_du_shema
- `CREATE_SCHEMA` 0

Il faut ensuite se poser la question de savoir si l'on souhaite recréer le schéma ou l'utilisateur Oracle sous lequel seront créés tous les objets dans PostgreSQL. Si la réponse est oui, il faut activer la directive `EXPORT_SCHEMA` et désactiver la directive `CREATE_SCHEMA` car la création du schéma peut se faire de manière manuelle lors de la création de la base de données et de son propriétaire.

Le schéma utilisé pour définir le `search_path` à la création des objets sera celui donné comme valeur de la variable `SCHEMA` par défaut ou celui défini par la variable `PG_SCHEMA` si vous souhaitez changer de nom de schéma ou que vous devez accéder à d'autres schémas lors de l'import des objets.

À ce stade, il est possible de ne plus toucher au fichier de configuration en dehors de particularités de la base Oracle obligeant à modifier certaines variables. Dans ce cas, il sera préférable de travailler sur une copie du fichier ou d'utiliser la directive `INCLUDE` en fin de fichier de configuration.

1.4.4 VERSIONS DE POSTGRESQL

- Indiquer la version majeure cible de PostgreSQL
  - `PG_VERSION` 9.6
  - `PG_VERSION` 11
- Par défaut : 11
- Autres contrôles lié à la version :
  - `BITMAP_AS_GIN` : export des index bitmap en `btree_gin`
  - `STANDARD_CONFORMING_STRINGS` échappement dans les chaines de caractères

Ora2pg considère toujours que vous utilisez la dernière version officielle de PostgreSQL disponible à la sortie de la version d'Ora2Pg que vous utilisez. Cependant il est possible que vous ayez besoin de migrer dans une base PostgreSQL d'une version antérieure, mais toutes les fonctionnalités supportées par Ora2Pg n'y existent pas forcément encore.

Pour pouvoir contrôler cela il est nécessaire de positionner la directive `PG_VERSION` à la dernière version majeur de PostgreSQL. Ora2Pg adaptera l'export en fonction des fonctionnalités développées dans chaque version.

Dans les versions antérieures à Ora2Pg v20.0, ce comportement adaptatif à la version de PostgreSQL peut être contrôlé avec les directives suivantes.

- `PG_SUPPORTS_INSTEADOF` pour le support de la clause `INSTEAD OF` dans les définitions de triggers ;
- `PG_SUPPORTS_IFEXISTS` permet d'ajouter les ordres `IF NOT EXISTS` lors de l'import des données ;
- `PG_SUPPORTS_MVIEW` pour l'utilisation des vues matérialisées natives à partir de PostgreSQL 9.3 ;
- `PG_SUPPORTS_CHECKOPTION` permet d'ajouter la clause `CHECK OPTION` aux vues ;
- `PG_SUPPORTS_NAMED_OPERATOR` pour l'activation des paramètres nommés dans l'appel des fonctions ;
- `PG_SUPPORTS_PARTITION` permet l'utilisation du partitionnement déclaratif au lieu du partitionnement par héritage ;
- `PG_SUPPORTS_IDENTITY` permet l'utilisation des colonnes `IDENTITY` plutôt que `serial` ou `bigserial`.

Elles sont maintenant obsolète.

D'autres directives permettent d'activer ou de désactiver certaines fonctionnalités :

- `BITMAP_AS_GIN` pour autoriser l'export des index bitmap dans leur équivalent avec l'extension `btree_gin` ;
- `STANDARD_CONFORMING_STRINGS` pour l'échappement dans les chaines de caractères.

La valeur de `STANDARD_CONFORMING_STRINGS` doit correspondre à la valeur de la variable `standard_conforming_string` dans le fichier `postgresql.conf`.

Par défaut donc, tous ces paramètres sont activés.

1.4.5 BASES SPATIALES

La base contient des champs de type `SDO_GEOMETRY`.

- Faut-il utiliser les contraintes sur les géométries ?
  - `AUTODETECT_SPATIAL_TYPE` [0|1]
- Quel système de référence spatial par défaut ?
  - `DEFAULT_SRID` 4326
  - `CONVERT_SRID` [0|1|N]
- PostGIS est-il installé dans un schéma spécifique ?
  - `POSTGIS_SCHEMA` schema_name
- Format d'export des géométries :
  - `GEOMETRY_EXTRACT_TYPE` [INTERNAL|WKT|WKB]

Les colonnes ayant pour type Oracle spatial `SDO_GEOMETRY`, peuvent contenir n'importe quel type de géométrie. Le type équivalent pour PostGIS est `geometry`.

```
CREATE TABLE test_geom (
    id bigint,
    shape geometry(GEOMETRY, 4326)
);
```

Dans ce cas, elles pourront aussi contenir n'importe quel type de géométrie.

Il peut être intéressant d'avoir une contrainte sur le type des géométries pouvant être insérées dans la colonne si c'est toujours le même type d'objet géométrique qui doit être utilisé.

Dans ce cas, en activant la directive `AUTODETECT_SPATIAL_TYPE`, Ora2Pg cherchera d'abord s'il existe une contrainte géométrique sur la colonne pour déterminer le type. S'il n'y a pas d'index de contrainte alors il cherchera dans la colonne Oracle si les données sont toutes du même type. Dans ce dernier cas, Ora2Pg prend comme échantillon les 50 000 premières géométries de la colonne (ou la valeur de `AUTODETECT_SPATIAL_TYPE` si elle est supérieure à 1). Si les objets spatiaux de l'échantillon sont tous du même type, alors la contrainte est appliquée.

```
CREATE TABLE test_geom (
    id bigint,
    shape geometry(POLYGON, 4326)
);
```

Le système de référence spatial (SRID) utilisé va être la valeur retournée depuis la table des métadonnées spatial Oracle (`ALL_SDO_GEOM_METADATA`) ou, si la valeur retournée est nulle, la valeur donnée à la directive de configuration `DEFAULT_SRID`. Voici à peu de chose près la requête utilisée :

```
SELECT COALESCE(SRID, $DEFAULT_SRID)
  FROM ALL_SDO_GEOM_METADATA
 WHERE TABLE_NAME='$table' AND COLUMN_NAME='$colname';
```

Si la directive `CONVERT_SRID` est activée alors la conversion en ESPG est demandée et dans ce cas la requête utilisée par Ora2Pg pour obtenir le SRID sera la suivante :

```
SELECT COALESCE(sdo_cs.map_oracle_srid_to_epsg(SRID), $DEFAULT_SRID)
  FROM ALL_SDO_GEOM_METADATA
 WHERE TABLE_NAME='$table' AND COLUMN_NAME='$colname';
```

Si l'extension PostGIS a été installée dans un schéma spécifique, les appels aux fonctions de l'extension devront être préfixées par le nom du schéma. Pour éviter cela, il est préférable de positionner le nom du schéma PostGIS dans la directive `POSTGIS_SCHEMA` et celui-ci sera ajouté au `search_path` lors de la création des objets.

Pour l'export des géométries, il est préférable d'utiliser le type `INTERNAL` pour la directive `GEOMETRY_EXTRACT_TYPE`. Cela évite d'utiliser les fonctions Oracle pour extraire la géométrie au format texte (`WKT`) ou binaire (`WKB`). Ces modes nécessitent l'utilisation de fonctions Oracle (`SDO_UTIL.TO_WKTGEOMETRY()`) et (`SDO_UTIL.TO_WKBGEOMETRY()`) qui sont lentes et ont la particularité de planter l'export dès que le volume est important.

1.4.6 CONFIGURATION LIÉE AUX LOB

L'export des champs CLOB et BLOB sont contrôlés par :

- `LONGREADLEN` 1047552
- `LONGTRUNCOK` 0
- `NO_LOB_LOCATOR` 0
- `BLOB_LIMIT` 500

Lors de l'export des LOB, si la directive `NO_LOB_LOCATOR` est activée, il se peut que vous rencontriez l'erreur Oracle :

```
ORA-24345: A Truncation or null fetch error occurred
(DBD SUCCESS_WITH_INFO: OCIStmtFetch, LongReadLen too small
 and/or LongTruncOk not set)
```

La solution est d'augmenter la valeur du paramètre `LONGREADLEN`, par défaut 1 Mo, à la taille du plus grand enregistrement de la colonne. Vous avez aussi la possibilité de tronquer les données en activant `LONGTRUNCOK`, ce qui ne remontera plus d'erreur mais bien évidement tronquera certaines données dont la taille dépasse la valeur de `LONGREADLEN`.

Il est conseillé de laisser Ora2Pg gérer l'export des LOB en utilisant des pointeurs sur les

enregistrements (*LOB Locator*) lui permettant de récupérer les données de ces champs en plusieurs fois. Cela évite la contrainte de recherche de la bonne valeur à attribuer à `LONGREADLEN`.

Lors de l'export de champs LOB, il est important de diminuer très fortement la valeur de `DATA_LIMIT` en fonction de la vitesse maximale d'export pour éviter les dépassements de mémoire. Pour permettre à Ora2Pg d'extraire ces données avec les autres en adaptant automatiquement le `DATA_LIMIT` à une valeur plus faible lorsqu'il s'agit d'un LOB, la directive `BLOB_LIMIT` est disponible.

```
BLOB_LIMIT      500
```

La valeur de 500, voire moins, n'est pas rare avec ce type d'objet. Si cette directive n'est pas définie, par défaut, Ora2Pg est capable de détecter qu'il s'agit d'une table avec un champ BLOB et de diminuer automatiquement la valeur de `DATA_LIMIT` en la divisant par 10 jusqu'à ce qu'elle soit inférieure ou égale à 1000.

Une bonne pratique consiste donc à positionner une valeur à la directive `BLOB_LIMIT` pour forcer Ora2Pg à utiliser cette valeur pour les tables avec BLOB et continuer à utiliser la valeur de `DATA_LIMIT` pour les tables sans BLOB.

1.5 MIGRATION DU SCHÉMA

Étapes :

- Organisation de l'espace de travail
- Utilisation de la configuration générique
- Export du schéma Oracle
- Import du schéma dans PostgreSQL

Nous allons aborder ici les différentes étapes à réaliser pour mettre en œuvre de façon optimale l'export du schéma :

- Comment s'y retrouver dans tous les fichiers générés et ne pas écraser le précédent export ?
- Comment utiliser la configuration générique ?
- Et enfin l'export complet du schéma Oracle en des ordres DDL PostgreSQL ?

1.5.1 ORGANISATION DE L'ESPACE DE TRAVAIL

- Arborescence d'un projet de migration
  - dossier de la configuration
  - dossier du schéma source Oracle
  - dossier du schéma converti à PostgreSQL
  - dossier des fichiers de données exportées

```
ora2pg --init_project dirname --project_base dirname
```

Il est important d'organiser l'espace de travail de son projet de migration. Sans cela, on se retrouve très vite avec une multitude de fichiers dont le contenu devient très vite énigmatique.

Dans la mesure où, par défaut, Ora2Pg fait tous ses exports dans un même fichier nommé `output.sql`, vous pouvez aussi très facilement écraser le précédent export si vous omettez de renommer le fichier.

À minima, il est conseillé d'avoir :

- un répertoire dédié au stockage du ou des fichiers de configuration ;
- un répertoire dédié aux fichiers des données exportées ;
- un répertoire de stockage des sources du code Oracle ;
- un répertoire des objets et code convertis à la syntaxe PostgreSQL.

L'export du code source du code SQL et PL/SQL dans des fichiers dans un espace de stockage particulier est très important. Cela permet, lors de la phase de migration des routines stockées, de vérifier qu'Ora2Pg n'a pas corrompu du code et de comparer le code.

Pour créer une arborescence de travail destinée à recevoir les fichiers du projet de migration, on peut s'aider d'ora2pg en exécutant la commande suivante :

```
ora2pg --init_project mydb_project --project_base /opt/ora2pg
```

Voici l'arborescence générée par Ora2Pg :

```
/opt/ora2pg/mydb_project/
├── config
│   └── ora2pg.conf
├── data
├── export_schema.sh
├── import_all.sh
├── reports
├── 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
```

La commande utilisée pour la génération automatique de l'espace de travail a permis de générer un fichier de configuration générique `config/ora2pg.conf` et un script shell `export_schema.sh`. Ce script peut être utilisé pour générer automatiquement tous les types d'export en dehors de l'export des données. Voici son contenu :

```
#!/bin/sh
#------------------------------------------------------------------------------
#
# Generated by Ora2Pg, the Oracle database Schema converter, version 20.0
#
#------------------------------------------------------------------------------
EXPORT_TYPE="TABLE PACKAGE VIEW GRANT SEQUENCE TRIGGER FUNCTION PROCEDURE
             TABLESPACE PARTITION TYPE MVIEW DBLINK SYNONYM DIRECTORY"
SOURCE_TYPE="PACKAGE VIEW TRIGGER FUNCTION PROCEDURE PARTITION TYPE MVIEW"
namespace="."

ora2pg -t SHOW_TABLE -c $namespace/config/ora2pg.conf > $namespace/reports/tables.txt
ora2pg -t SHOW_COLUMN -c $namespace/config/ora2pg.conf > $namespace/reports/columns.txt
ora2pg -t SHOW_REPORT -c $namespace/config/ora2pg.conf --dump_as_html --estimate_cost \
                                                    > $namespace/reports/report.html

for etype in $(echo $EXPORT_TYPE | tr " " "\n")
do
```

```
        ltype=`echo $etype | tr '[:upper:]' '[:lower:]'`
        ltype=`echo $ltype | sed 's/y$/ie/'`
        echo "Running: ora2pg -p -t $etype -o $ltype.sql -b $namespace/schema/${ltype}s \
                                            -c $namespace/config/ora2pg.conf"
        ora2pg -p -t $etype -o $ltype.sql -b $namespace/schema/${ltype}s \
                                            -c $namespace/config/ora2pg.conf
    ret=`grep "Nothing found" $namespace/schema/${ltype}s/$ltype.sql 2> /dev/null`
    if [ ! -z "$ret" ]; then
        rm $namespace/schema/${ltype}s/$ltype.sql
    fi
done

for etype in $(echo $SOURCE_TYPE | tr " " "\n")
do
        ltype=`echo $etype | tr '[:upper:]' '[:lower:]'`
        ltype=`echo $ltype | sed 's/y$/ie/'`
        echo "Running: ora2pg -t $etype -o $ltype.sql -b $namespace/sources/${ltype}s \
                                            -c $namespace/config/ora2pg.conf"
        ora2pg -t $etype -o $ltype.sql -b $namespace/sources/${ltype}s \
                                             -c $namespace/config/ora2pg.conf
    ret=`grep "Nothing found" $namespace/sources/${ltype}s/$ltype.sql 2> /dev/null`
    if [ ! -z "$ret" ]; then
        rm $namespace/sources/${ltype}s/$ltype.sql
    fi
done

echo
echo
echo "To extract data use the following command:"
echo
echo "ora2pg -t COPY -o data.sql -b $namespace/data -c $namespace/config/ora2pg.conf"
echo

exit 0
```

Une fois la connexion à la base Oracle paramétrée dans le fichier de configuration générique, il suffit d'exécuter ce script pour que tous les export soient réalisés. Le script réalisera même le rapport sur la base au format HTML.

Ora2Pg aura aussi créé un script `import_all.sh` utilisé pour l'import dans PostgreSQL des divers objets exportés et disponibles sous forme de fichiers dans l'espace de travail après exécution du script `export_schema.sh`. Si les données ont aussi été exportées sous forme de fichiers dans l'espace de travail, le script permet de les charger dans PostgreSQL, sinon il permettra de les charger directement depuis Oracle en utilisant les options de parallélisme d'Ora2Pg.

Pour les bases MySQL, il est nécessaire d'ajouter l'option `-m` ou `--mysql` pour indiquer à Ora2Pg qu'il s'agit d'un projet de migration de base MySQL.

1.5.2 UTILISATION DE LA CONFIGURATION GÉNÉRIQUE

- Fichier `ora2pg.conf` générique
  - création du fichier ora2pg.conf générique dans le dossier de configuration
- Utilisation des options de ligne de commande du script `ora2pg`
  - `-t` type d'export
  - `-b` répertoire de stockage des fichiers
  - `-o` nom du fichier de sortie
  - `-p` conversion automatique du code

Lors de la création par Ora2Pg du répertoire de travail, le fichier de configuration générique est créé à partir du fichier `/etc/ora2pg/ora2pg.conf.dist` et enregistré dans le répertoire `mydb_project/config/`. Les modifications appliquées à ce fichier sont celles exposée dans le chapitre *Configuration générique*. Si le fichier n'existe pas, il suffit de le copier et d'appliquer les préconisations de configuration.

On peut aussi demander à Ora2Pg d'utiliser un fichier de configuration prédéfini en le précisant avec l'option `-c config_file` lors de l'exécution de la commande `ora2pg --ini_project`, c'est alors ce fichier qui sera copié dans l'espace de travail.

Les options de connexion à Oracle peuvent être données en ligne de commande avec les options d'`ora2pg` dédiées à cet effet (`-s`, `-u` et `-n`). Les valeurs de ces paramètres seront alors appliquées dans le fichier de configuration générique.

Ensuite, le comportement d'`ora2pg` sera déterminé par les options des lignes de commande utilisées.

L'option `-t` permet de choisir le type d'action lors de l'exécution du script plutôt que d'aller modifier le fichier de configuration. Cette option peut prendre exactement les mêmes valeurs que la variable `TYPE`, c'est-à-dire `TABLE`, `VIEW`, `MVIEW`, `DBLINK`, `SYNONYM`, `DIRECTORY`, `GRANT`, `SEQUENCE`, `TRIGGER`, `PACKAGE`, `FUNCTION`, `PROCEDURE`, `PARTITION`, `TYPE`, `INSERT`, `COPY`, `TABLESPACE`, `SHOW_SCHEMA`, `SHOW_TABLE`, `SHOW_COLUMN`, `SHOW_ENCODING`, `KETTLE`, `QUERY`, `LOAD`, `TEST`, `TEST_VIEW` et `FDW`.

L'option `-b` va permettre d'utiliser l'arborescence de l'espace de travail créé auparavant pour stocker les fichiers générés dans leur espace de stockage respectif. Elle correspond à la variable `OUTPUT_DIR` du fichier de configuration.

Le nom des fichiers de sortie est défini à partir de l'option `-o` correspondant à la directive `OUTPUT`.

L'option `-p` est utilisée pour provoquer la conversion automatique du code SQL et PL/SQL.

1.5.3 EXPORT DE LA STRUCTURE DE LA BASE

- Export des tables, contraintes et index

```
ora2pg -p -t TABLE -o table.sql -b schema/tables -c config/ora2pg.conf
```

- Export des séquences

```
ora2pg -t SEQUENCE -o sequences.sql -b schema/sequences -c  config/ora2pg.conf
```

- Export des vues

```
ora2pg -p -t VIEW -o views.sql -b schema/views -c config/ora2pg.conf
```

- Préservation des tablespaces Oracle : `USE_TABLESPACE`

Avec l'activation des directives `FILE_PER_INDEX`, `FILE_PER_CONSTRAINT` et `FILE_PER_FKEYS`, la commande d'extraction des définitions de tables, contraintes et index va créer quatre fichiers dans le répertoire de sortie `schema/tables` :

- `table.sql`
- `CONSTRAINTS_table.sql`
- `INDEXES_table.sql`
- `FKEYS_table.sql`

Le premier utilise le nom donné par l'option `-o` et contient les ordres `CREATE TABLE ...`. Le second utilise aussi le nom donné dans l'option `-o` mais préfixé par le mot `CONSTRAINT_` et, pour cause, il contient tous les ordres de création des contraintes : `ALTER TABLE "..." ADD CONSTRAINT ...`.

Le troisième fichier contient toutes les commandes de création des index (`CREATE INDEX ...`) définies dans Oracle à l'exception des index implicites sur les clés primaires que PostgreSQL génère automatiquement et qui n'ont donc pas besoin d'être exportées.

Le quatrième contient les ordres de création des clés étrangères pour pouvoir être créées facilement après la migration des données.

L'option de conversion de code `-p` est utilisée ici uniquement pour les index ou contraintes `CHECK` qui peuvent utiliser des fonctions à convertir.

Les séquences sont, quant à elles, exportées dans le sous-répertoire `schema/sequences` et le fichier `sequences.sql` contenant les ordres SQL `CREATE SEQUENCE ...`. Comme les contraintes, les séquences ne doivent être importées qu'à la fin de la migration. Les séquences seront créées avec la bonne valeur de départ après import des données.

Par défaut Ora2Pg supprime toutes les informations sur les tablespaces associés aux objets exportés de la base Oracle. Si vous souhaitez préserver ces informations, notamment pour utiliser des tablespaces différents pour les tables et les index, la directive de configuration `USE_TABLESPACE` doit être activée. Les tablespaces par défaut d'Oracle (`TEMP`, `USERS` et `SYSTEM`) ne sont pas pris en compte.

1.5.4 MODIFICATION DE LA STRUCTURE DES OBJETS

- Renommer les objets
  - `REPLACE_TABLES ORIG_TABLE1:DEST_TABLE1`
  - `REPLACE_COLS TABLE1(ORIG_COL1:DEST_COL1, [..])`
  - `INDEXES_SUFFIX _idx`
  - `INDEXES_RENAMING`
- Changer les types de données
  - `DATA_TYPE NUMBER(*\,0):bigint`
  - `MODIFY_TYPE TABLE1:COL1:integer TABLE1:COL2:timestamptz`
  - `REPLACE_AS_BOOLEAN TABLE1:COL1 [..]`

REPLACE_TABLES et **REPLACE_COLS**

Il peut être nécessaire de renommer une table en particulier durant la migration. La directive `REPLACE_TABLES` autorise de lister sur une ou plusieurs lignes les tables à transformer.

```
REPLACE_TABLES     ORIG_TB_NAME1:NEW_TB_NAME1
REPLACE_TABLES     ORIG_TB_NAME2:NEW_TB_NAME2
```

L'un des cas d'usages principaux est l'utilisation d'un mot-clé réservé avec PostgreSQL, comme `windows` ou `array`, qui sont acceptés comme noms de tables dans Oracle mais sont interdits avec PostgreSQL. La requête suivante permet de connaître les mots strictement réservés.

```
select word from pg_get_keywords() where catcode = 'R';
```

Dans le même ordre d'idée, la direction `REPLACE_COLS` permet de définir les renommages pour les colonnes d'une ou de plusieurs tables.

INDEXES_SUFFIX et **INDEXES_RENAMING**

Avec Oracle, les espaces de noms entre les tables et les index sont distincts, cela signifie qu'il est possible qu'une table et un index puissent avoir le même nom. Cette situation n'est pas supportée dans PostgreSQL et Ora2Pg propose de suffixer l'ensemble des index à migrer, à l'aide de la directive `INDEXES_SUFFIX`. Par défaut, ce comportement est désactivé.

```
INDEXES_SUFFIX      _idx
```

Si les noms des index importent peu dans la gestion quotidienne du schéma, il est également possible d'activer le changement automatique des noms des index, en s'inspirant du nommage proposé nativement par PostgreSQL, à savoir `tablename_columns_names`.

```
INDEXES_RENAMING    1
```

DATA_TYPE et MODIFY_TYPE

Par défaut, Ora2Pg transpose automatiquement les types Oracle en équivalents PostgreSQL. Cependant, certains typages peuvent provoquer des comportements anormaux dans la base de données migrées, notamment les erreurs de précisions et d'arrondis sur les types numériques, ou l'absence de fuseau horaire dans le type `timestamp without timezone`.

Par exemple, la table `employees` présente une date et des champs décimaux avec des définitions variées.

```
SQL> DESC employees;
 Name                  Null?    Type
 --------------------- -------- -----------------
 EMPLOYEE_ID           NOT NULL NUMBER(6)
 FIRST_NAME                     VARCHAR2(20)
 LAST_NAME             NOT NULL VARCHAR2(25)
 EMAIL                 NOT NULL VARCHAR2(25)
 PHONE_NUMBER                   VARCHAR2(20)
 HIRE_DATE             NOT NULL DATE
 JOB_ID                NOT NULL VARCHAR2(10)
 SALARY                         NUMBER(8,2)
 COMMISSION_PCT                 NUMBER(2,2)
 MANAGER_ID                     NUMBER(6)
 DEPARTMENT_ID                  NUMBER(4)
```

Les champs `salary`, `commission_pct` et `department_id` seront respectivement convertis par Ora2Pg en `double precision`, `real` et `smallint`, afin d'optimiser au mieux le stockage des valeurs numériques.

```
CREATE TABLE employees (
  employee_id integer NOT NULL,
  first_name varchar(20),
  last_name varchar(25) NOT NULL,
  email varchar(25) NOT NULL,
  phone_number varchar(20),
  hire_date timestamp NOT NULL,
  job_id varchar(10) NOT NULL,
  salary double precision,
```

```
  commission_pct real,
  manager_id integer,
  department_id smallint
);
```

Pour assurer le respect des données, il peut être judicieux d'enrichir la directive `DATA_TYPE` pour transformer les types `NUMBER` ou `DATE` dans un type équivalent.

```
DATA_TYPE      NUMBER(8\,2):decimal,NUMBER(2\,2):decimal,DATE:timestamptz
```

```
CREATE TABLE employees (
  employee_id integer NOT NULL,
  first_name varchar(20),
  last_name varchar(25) NOT NULL,
  email varchar(25) NOT NULL,
  phone_number varchar(20),
  hire_date timestamptz NOT NULL,
  job_id varchar(10) NOT NULL,
  salary decimal,
  commission_pct decimal,
  manager_id integer,
  department_id smallint
);
```

La directive `MODIFY_TYPE` est similaire mais permet de préciser colonne par colonne les différentes transformations à opérer. Par exemple, pour la table `employees` exclusivement, la configuration sera la suivante :

```
MODIFY_TYPE    employees:hire_date:timestamptz
MODIFY_TYPE    employees:salary:decimal
MODIFY_TYPE    employees:commission_pct:decimal
```

REPLACE_AS_BOOLEAN et BOOLEAN_VALUES

Puisqu'Oracle ne dispose pas d'un type `boolean`, il est possible d'instruire Ora2Pg pour qu'il réalise les conversions vers une colonne de type `bool` en assurant la correspondance des valeurs.

```
REPLACE_AS_BOOLEAN      TABLE1:COL1 TABLE2:COL1
```

Par défaut, les traductions d'une donnée Oracle vers un booléen PostgreSQL est assurée par la directive `BOOLEAN_VALUES`, qu'il est possible d'étendre selon les jeux de données à migrer.

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

1.5.5 EXPORT DES OBJETS GLOBAUX

- Les rôles et droits

```
ora2pg -t GRANT -o users.sql -b schema/users -c config/ora2pg.conf
```

- Les tablespaces

```
ora2pg -t TABLESPACE -o tablespaces.sql -b schema/tablespaces \
-c config/ora2pg.conf
```

- Les types composites

```
ora2pg -p -t TYPE -o types.sql -b schema/types -c config/ora2pg.conf
```

Le premier export (type `GRANT`) va exporter tous les rôles et leurs droits sur les objets sous forme d'ordres SQL `CREATE ROLE ...` et `GRANT ... ON ...` dans le fichier `schema/users/users.sql`.

Le deuxième provoque la génération des ordres de création des espaces de stockage des tables ou index, `CREATE TABLESPACE ...` et les ordres de déplacement des objets dans ces espaces, `ALTER ... SET TABLESPACE ...`. Les définitions sont enregistrées dans le fichier `schema/tablespaces/tablespaces.sql`. Si la directive `FILE_PER_INDEX` est activée alors les ordre concernant les index le seront dans un fichier séparé `schema/tablespaces/INDEXES_tablespaces.sql`.

Le troisième type d'export va exporter tous les types définis par les utilisateurs (`CREATE TYPE ...`) dans le fichier `schema/types/types.sql`. La conversion de certains types utilisateurs Oracle nécessite une réécriture manuelle pour être compatible avec PostgreSQL. Ce sont les types définis par `CREATE TYPE ... AS TABLE OF ...` qui nécessitent l'écriture de fonctions définissant le comportement du type lors de la lecture et de l'écriture dans ce type. Il en va de même avec les types objets (`CREATE TYPE ... AS OBJECT ... TYPE BODY`). Les fonctions doivent être converties à la syntaxe PostgreSQL. Cette conversion est réalisée grâce à l'emploi de l'option `-p` (équivalent à l'activation de la variable `PLSQL_PGSQL`).

1.5.6 EXPORT DES ROUTINES STOCKÉES

- Export des objets avec conversion de code

```
ora2pg -p -t TRIGGER -o triggers.sql -b schema/triggers -c config/ora2pg.conf
ora2pg -p -t FUNCTION -o functions.sql -b schema/functions -c config/ora2pg.conf
ora2pg -p -t PROCEDURE -o procedures.sql -bschema/procedures -c config/ora2pg.conf
ora2pg -p -t PACKAGE -o packages.sql -b schema/packages -c config/ora2pg.conf
```

L'étape suivante de la migration du schéma consiste à exporter tous les autres types d'objets : les vues, les triggers, les fonctions et procédures stockées (les routines). Tous ces types d'export nécessitent l'emploi de l'option `-p` pour provoquer la conversion automatique du code SQL et PL/SQL.

L'import de ce type d'objet sera évoqué en détail dans le chapitre dédié à la migration du code PL/SQL.

1.5.7 EXPORT DES SOURCES PL/SQL

- Extraction du code brut d'Oracle

```
ora2pg -t TYPE -o types.sql -b sources/types -c config/ora2pg.conf
ora2pg -t VIEW -o views.sql -b sources/views -c config/ora2pg.conf
ora2pg -t MVIEW -o mviews.sql -b sources/mviews -c config/ora2pg.conf
ora2pg -t TRIGGER -o triggers.sql -b sources/triggers -c config/ora2pg.conf
ora2pg -t FUNCTION -o functions.sql -b sources/functions -c config/ora2pg.conf
ora2pg -t PROCEDURE -o procedures.sql -b sources/procedures -c config/ora2pg.conf
ora2pg -t PACKAGE -o packages.sql -b sources/packages -c config/ora2pg.conf
```

Dans la mesure où la conversion du code SQL et PL/SQL n'est pas complète, voire imparfaite, il est recommandé d'extraire le code brut pour pouvoir le comparer avec le code converti par Ora2Pg en cas de problème.

L'extraction du code brut d'Oracle se fait en n'utilisant pas l'option `-p` lors de l'exécution du script et en désactivant l'option `PLSQL_PGSQL` dans le fichier de configuration.

1.5.8 EXPORT DES PARTITIONS

- Partitions par `range` et `list` uniquement
- Partitions et sous partition

```
ora2pg -t PARTITION -o partitions.sql -b schema/partitions -c config/ora2pg.conf
```

- Paramètrage : `DISABLE_PARTITION`, `PG_SUPPORTS_PARTITION`

Depuis PostgreSQL v10, il est possible de déclarer une table comme étant partitionnée et de déclarer des partitions. La spécification d'une table partitionnée consiste en une méthode de partitionnement et une liste de colonnes ou expressions à utiliser comme la clé de partitionnement.

Toutes les lignes insérées dans la table partitionnée seront alors redirigées vers une des partitions en se basant sur la valeur de la clé de partitionnement.

Les méthodes de partitionnement supportées par Ora2Pg sont le partitionnement par intervalles (`RANGE`) et par liste (`LIST`). Le hachage (`HASH`) disponible depuis la version 11, n'est pas pris en charge.

Les partitions peuvent elles-mêmes être définies comme des tables partitionnées, en

utilisant le sous-partitionnement. Les partitions peuvent avoir leurs propres index, contraintes et valeurs par défaut, différents de ceux des autres partitions.

À partir de la version 11, les index peuvent être créés au niveau de la table partitionnée afin de propager la définition et la construction d'index aux partitions. À partir de la version 12, les clés étrangères peuvent être créées sur les tables partitionnées.

Voici un exemple avec les deux types de partitionnement Oracle supportés :

- Partition par *range*

```
CREATE TABLE sales_range
(
    salesman_id  NUMBER(5),
    salesman_name VARCHAR2(30),
    sales_amount  NUMBER(10),
    sales_date    DATE
)
PARTITION BY RANGE(sales_date)
(
  PARTITION sales_jan2000 VALUES LESS THAN(TO_DATE('02/01/2000','DD/MM/YYYY')),
  PARTITION sales_feb2000 VALUES LESS THAN(TO_DATE('03/01/2000','DD/MM/YYYY')),
);
```

- Partition par liste

```
CREATE TABLE sales_list
(
    salesman_id  NUMBER(5),
    salesman_name VARCHAR2(30),
    sales_state   VARCHAR2(20),
    sales_amount  NUMBER(10),
    sales_date    DATE
)
PARTITION BY LIST(sales_state)
(
    PARTITION sales_west VALUES('California', 'Hawaii'),
    PARTITION sales_east VALUES ('New York', 'Virginia', 'Florida'),
    PARTITION sales_other VALUES (DEFAULT)
);
```

La conversion avec le partitionnement déclaratif renvoie le résultat suivant :

- Partition par *range*

```
CREATE TABLE sales_range
(
    salesman_id  integer,
```

```
    salesman_name varchar(30),
    sales_amount  bigint,
    sales_date    timestamp
) PARTITION BY RANGE (sales_date);

CREATE TABLE sales_jan2000 PARTITION OF sales_range
  FOR VALUES FROM ('2000-01-01) TO ('2000-01-02);

CREATE TABLE sales_feb2000 PARTITION OF sales_range
  FOR VALUES FROM ('2000-01-02) TO ('2000-01-03);
```

- Partition par liste

```
CREATE TABLE sales_list
(
    salesman_id  integer,
    salesman_name varchar(30),
    sales_amount  bigint,
    sales_date    timestamp
) PARTITION BY LIST (sales_state)

CREATE TABLE sales_west PARTITION OF sales_list
    FOR VALUES IN ('California', 'Hawaii');

CREATE TABLE sales_east PARTITION OF sales_list
    FOR VALUES IN ('New York', 'Virginia', 'Florida');

CREATE TABLE sales_other PARTITION OF sales_list
    FOR VALUES DEFAULT;
);
```

1.5.9 EXPORT DES VUES MATÉRIALISÉES

```
ora2pg -t MVIEW -o mviews.sql -b schema/mviews -c config/ora2pg.conf
```

- Paramétrage : `PG_SUPPORTS_MVIEW`

Le but d'une vue matérialisée est de stocker physiquement le résultat de l'exécution d'une vue et d'utiliser par la suite ce stockage plutôt que le résultat de l'exécution de la requête. Il est possible de créer des index sur cette vue matérialisée. Elle est mise à jour soit à la demande soit au fil de l'eau.

Les vues matérialisées ne sont supportées qu'à partir de la version 9.3 pour PostgreSQL. Elles ne supportent pas toutes les fonctionnalités qu'offre Oracle : pas de mise à jour au fil de l'eau, pas de rafraîchissement incrémental à l'aide de journaux de vue matérialisée

(`MATERIALIZED VIEW LOG` sous Oracle), pas de réécriture de requête (*query rewrite*).

Passer `PG_VERSION` à une version supérieure ou égale à `9.3` permet d'exporter les vues matérialisées directement avec la syntaxe SQL. Le paramètre `PG_SUPPORTS_MVIEW` permet d'activer ou de désactiver la conversation. Cette dernière est activée par défaut.

```
CREATE MATERIALIZED VIEW emp_data_mview AS
SELECT EMPLOYEES.EMPLOYEE_ID EMPLOYEE_ID,EMPLOYEES.FIRST_NAME FIRST_NAME,
       EMPLOYEES.LAST_NAME LAST_NAME, EMPLOYEES.EMAIL
       EMAIL,EMPLOYEES.PHONE_NUMBER PHONE_NUMBER,EMPLOYEES.HIRE_DATE
       HIRE_DATE,EMPLOYEES.JOB_ID JOB_ID, EMPLOYEES.SALARY
       SALARY,EMPLOYEES.COMMISSION_PCT COMMISSION_PCT,EMPLOYEES.MANAGER_ID
       MANAGER_ID, EMPLOYEES.DEPARTMENT_ID DEPARTMENT_ID
FROM EMPLOYEES EMPLOYEES;
```

et pour le rafraîchissement, il suffira d'utiliser la commande SQL :

```
REFRESH MATERIALIZED VIEW emp_data_mview;
```

Il est possible de lever le verrou exclusif de l'opération de rafraîchissement à l'aide de l'option `CONCURRENTLY`. Pour ce faire, un index unique est requis sur l'une des colonnes de la vue matérialisée qui respecte la contrainte d'unicité.

```
CREATE UNIQUE INDEX ON emp_data_mview(employee_id);
REFRESH MATERIALIZED VIEW CONCURRENTLY emp_data_mview;
```

Certaines fonctionnalités que propose Oracle (ex : `FAST REFRESH` à l'aide des `MATERIALIZED VIEW LOG`) ne sont pas encore présentes dans les versions actuelles de PostgreSQL. Si une mise à jour au fil de l'eau est requise, il faudra forcément passer par des triggers.

```
CREATE FUNCTION fct_refresh_emp_data_mview()
RETURNS trigger LANGUAGE plpgsql
AS $$
BEGIN
  REFRESH MATERIALIZED VIEW CONCURRENTLY emp_data_mview;
  RETURN new;
END
$$;

CREATE TRIGGER trg_emp_data_mview_on_insert
AFTER INSERT ON employees FOR EACH ROW
  EXECUTE PROCEDURE fct_refresh_emp_data_mview();
```

Pour aller plus loin :

- Postgres 9.4 feature highlight - REFRESH CONCURRENTLY a materialized view[4]

[4] https://paquier.xyz/postgresql-2/postgres-9-4-feature-highlight-refresh-concurrently-a-materialized-view

- Conférence pgDay Paris 2019 : Don't forget materialized view[5] (vidéo[6])

1.5.10 EXPORT DES SYNONYMES

PostgreSQL ne possède pas d'objet de type `SYNONYM`
- Ce sont des alias vers des objets d'autres schémas ou bases de données
- Il existe deux méthodes pour les émuler sous PostgreSQL :
  - modification du `search_path`
  - utilisation de vues
- Ora2Pg utilise la deuxième méthode :

```
ora2pg -t SYNONYM -o synonyms.sql -b schema/synonyms -c config/ora2pg.conf
```

Un synonyme n'est ni plus ni moins qu'un alias vers un objet d'une base de données Oracle. Ils sont utilisés pour donner les droits d'accès à un objet dans un autre schéma ou dans une base distante auquel l'utilisateur n'aurait normalement pas accès.

Voici la syntaxe de création d'un synonyme sous Oracle :

```
CREATE SYNONYM synonym_name FOR object_name [@ dblink];
```

Les synonymes n'existent pas sous PostgreSQL, il y a deux méthodes pour les émuler.

Modification du search_path

L'objet est naturellement caché à l'utilisateur car il n'appartient pas à un schéma de son `search_path` par défaut et lorsqu'on veut qu'il y ait accès, on modifie le `search_path`. Par exemple :

```
SET search_path TO other_schema,...;
```

Cette méthode peut s'avérer assez fastidieuse à mettre en place au niveau applicatif mais évite la création de vues.

Utilisation de vues

L'autre méthode consiste donc à utiliser des vues. C'est ce que générera Ora2Pg lors de l'export des synonymes.

```
ora2pg -t SYNONYM -o synonyms.sql -b schema/synonyms -c config/ora2pg.conf
```

Par exemple, un synonyme créé sous Oracle avec l'ordre :

```
CREATE SYNONYM emp_table FOR hr.employees;
```

[5] https://www.postgresql.eu/events/pgdayparis2019/schedule/session/2398-dont-forget-the-materialized-views
[6] https://www.youtube.com/watch?v=X8_ZY-XMM0E

sera exporté par Ora2Pg de la façon suivante :

```
CREATE VIEW public.emp_table AS SELECT * FROM hr.employees;
ALTER VIEW public.emp_table OWNER TO hr;
GRANT ALL ON public.emp_table TO PUBLIC;
```

La vue `public.emp_table` étant la propriété de l'utilisateur `HR`, elle permet la consultation de la table dans le schéma `HR`.

Si le synonyme pointe sur une table distante par un *dblink*, Ora2Pg créera la vue telle que précédemment mais ajoutera un message en commentaire pour signifier que la table distante doit être créée via un *Foreign Data Wrapper* ou un *dblink*. Par exemple :

```
-- You need to create foreign table hr.employees using foreign server:
-- oradblink1 (see DBLINK and FDW export type)
CREATE VIEW public.emp_table AS SELECT * FROM hr.employees;
ALTER VIEW public.emp_table OWNER TO hr;
GRANT ALL ON public.emp_table TO PUBLIC;
```

1.5.11 EXPORT DES TABLES EXTERNES

PostgreSQL ne possède pas d'objets de type `DIRECTORY` ni de tables `EXTERNAL`

- Ce sont des répertoires et fichiers de données utilisés comme des tables
- Sous PostgreSQL, il faut utiliser le *Foreign Data Wrapper* `file_fdw`
  - ne fonctionne qu'en lecture
  - ces tables doivent respecter le format CSV de `COPY`

```
ora2pg -t DIRECTORY -o directories.sql -b schema/directories -c config/ora2pg.conf
```

Les `DIRECTORY` et tables externes n'existent pas dans PostgreSQL tels que définis dans Oracle. Il est possible d'émuler les accès à des tables externes en utilisant le *Foreign Data Wrapper* `file_fdw` mais uniquement en lecture. Ces tables doivent respecter le format CSV de `COPY`. Ora2Pg exporte par défaut toute table externe en une table distante basée sur l'extension `file_fdw`. Si vous voulez exporter ces tables comme des tables normales, il suffit de désactiver la directive de configuration `EXTERNAL_TO_FDW` en lui donnant la valeur `0`.

Voici un exemple de table externe sous Oracle :

```
CREATE OR REPLACE DIRECTORY ext_directory AS '/tmp/';

CREATE TABLE ext_table (
id       NUMBER(6),
nom      VARCHAR2(20),
prenom   VARCHAR2(20),
```

```
activite CHAR(1))
ORGANIZATION EXTERNAL (
  TYPE oracle_loader
  DEFAULT DIRECTORY ext_directory
    ACCESS PARAMETERS (
    RECORDS DELIMITED BY NEWLINE
    FIELDS TERMINATED BY ','
    MISSING FIELD VALUES ARE NULL
    REJECT ROWS WITH ALL NULL FIELDS
    (id, nom, prenom, activite))
    LOCATION ('person.dat')
  )
PARALLEL
REJECT LIMIT 0
NOMONITORING;
```

Ora2Pg convertit le `DIRECTORY` en serveur FDW en utilisant l'extension `file_fdw`.

```
CREATE EXTENSION file_fdw;
CREATE SERVER ext_directory FOREIGN DATA WRAPPER file_fdw;
```

Puis, il crée la table comme une table distante rattachée au serveur préalablement défini.

```
CREATE FOREIGN TABLE ext_table (
        id integer,
        nom varchar(20),
        prenom varchar(20),
        activite char(1)
) SERVER ext_directory OPTIONS(filename '/tmp/person.dat',
                               format 'csv',
                               delimiter ',');
```

1.5.12 EXPORT DES DATABASE LINK

PostgreSQL ne possède pas d'objets de type `DATABASE LINK`

- Ce sont des objets permettant l'accès à des bases distantes
- Sous PostgreSQL il faut utiliser le *Foreign Data Wrapper* `oracle_fdw`
  - fonctionne en lecture / écriture
  - les tables distantes sont vues comme des tables locales

```
ora2pg -t DBLINK -o dblinks.sql -b schema/dblinks -c config/ora2pg.conf
```

Les `DATABASE LINK` sont des objets Oracle permettant l'accès à des objets de bases de données distantes. Ils sont créés de la manière suivante :

```
CREATE PUBLIC DATABASE LINK remote_service_name CONNECT TO scott
```

```
IDENTIFIED BY tiger USING 'remote_db_name';
```

et s'utilisent ensuite de la façon suivante :

```
SELECT * FROM employees@remote_service_name;
```

Ce type d'objet n'existe pas nativement dans PostgreSQL et nécessite l'utilisation d'une extension *Foreign Data Wrapper* en fonction du type du SGBD distant.

Ora2Pg exportera ces `DATABASE LINK` comme des bases Oracle distantes en utilisant l'extension Foreign Data Wrapper `oracle_fdw` par défaut. Il est tout à fait possible de changer l'extension si la base distante est une base PostgreSQL. Voici un exemple d'export par Ora2Pg :

```
CREATE SERVER remote_service_name FOREIGN DATA WRAPPER oracle_fdw
OPTIONS (dbserver 'remote_db_name');

CREATE USER MAPPING FOR current_user SERVER remote_service_name
OPTIONS (user 'scott', password 'tiger');
```

Pour que le lien vers la base distante puisse être utilisé, il est nécessaire de créer les tables distantes dans la base locale :

```
ora2pg -c ora2pg.conf -t FDW -a EMPLOYEES
```

et le résultat de la commande ora2pg :

```
CREATE FOREIGN TABLE employees_fdw (... ) SERVER remote_service
OPTIONS(schema 'HR', table 'EMPLOYEES');
```

Maintenant la table peut être utilisée directement au niveau SQL comme s'il s'agissait d'une table locale :

```
SELECT * FROM employees@remote_service_name;
```

Cela fonctionne en lecture et écriture depuis PostgreSQL 9.3. Le *Foreign Data Wrapper* `oracle_fdw` peut être obtenu sur le site des extensions PostgreSQL pgxn.org[7]

[7] https://pgxn.org/dist/oracle_fdw/

1.5.13 EXPORT DES BFILE ET DIRECTORY - 1

- Sous PostgreSQL il n'y a pas d'équivalent aux types `DIRECTORY` et `BFILE`
  - Ora2Pg exporte les `BFILE` en donnée `bytea` par défaut
  - Si le type `BFILE` est redéfini en `TEXT`, stockage du chemin du fichier externe

Le type `BFILE` permet de stocker des données non structurées dans des fichiers externes en dehors de la base de données (fichiers image, documents pdf, etc.). Le type `DIRECTORY` permet lui de définir des chemins sur le système de fichier qui pourront être utilisés pour le stockage de ces données externes.

Il n'existe pas de types équivalents natifs sous PostgreSQL.

Un `BFILE` est une colonne qui stocke un nom de fichier qui pointe vers un fichier externe contenant les données et le nom de l'identifiant du répertoire base dans lequel ce fichier est stocké : `(DIRECTORY, FILENAME)`

Par défaut Ora2Pg transforme le type `BFILE` en type `bytea` en chargeant le contenu du fichier directement en base sous forme d'objet binaire.

```
CREATE TABLE bfile_test (id bigint, bfilecol bytea);
COPY bfile_test (id,bfilecol) FROM STDIN;
1
1234,ALBERT,GRANT,21\\0121235,ALFRED,BLUEOS,26\\0121236,BERNY,JOL
YSE,34\\012
\.
```

Il est possible de demander à Ora2Pg de ne pas importer les données dans le champ cible, mais seulement le chemin complet (répertoire base + nom de fichier) vers le fichier. Ceci se fait en modifiant le type PostgreSQL associé au type Oracle dans la directive de configuration `DATA_TYPE` : `...,BFILE:TEXT,...`

1.5.14 EXPORT DES BFILE ET DIRECTORY - 2

- Pour avoir la même fonctionnalité : extension `external_file`
  - type `EFILE` correspondant au type `BFILE` : `(directory_name, filename)`
  - les fichiers sont stockés sur le système de fichier
  - fichier accessible en lecture / écriture
  - activé lorsque `BFILE` est redéfini en `EFILE` (directive `DATA_TYPE`)

Il existe aussi une extension PostgreSQL nommée external_file[8] qui permet d'émuler les `DIRECTORY` et `BFILE` d'Oracle. Si le type PostgreSQL associé au type Oracle dans la directive

[8] https://github.com/darold/external_file

de configuration `DATA_TYPE` est positionné à `EFILE(...,BFILE:EFILE,...)`, Ora2Pg fera les conversions nécessaires pour utiliser ce type.

Voici ce que Ora2Pg générera comme ordre SQL lorsque qu'un champ de type `BFILE` doit être converti en type `EFILE` :

```
INSERT INTO external_file.directories (directory_name, directory_path)
    VALUES ('EXT_DIR', '/data/ext/');
INSERT INTO external_file.directory_roles (directory_name, directory_role,
    directory_read, directory_write) VALUES ('EXT_DIR', 'hr', true, false);
INSERT INTO external_file.directories (directory_name, directory_path)
    VALUES ('SCOTT_DIR', '/usr/home/scott/');
INSERT INTO external_file.directory_roles(directory_name, directory_role,
    directory_read, directory_write) VALUES ('SCOTT_DIR', 'hr', true, true);
```

L'objet `DIRECTORY` est défini dans la table `external_file.directories` créée par l'extension et les privilèges d'accès à ces répertoires stockés dans une autre table, `external_file.directory_roles`.

Le type `EFILE` contient lui exactement la même chose que le type `BFILE`, à savoir `(directory_name, file_name)`.

1.5.15 RECHERCHE PLEIN TEXTE

Oracle Index Texte

- CONTEXT
  - indexation de documents volumineux
  - opérateur `CONTAINS`
- CTXCAT
  - indexation de petits documents
  - opérateur `CATSEARCH`

1.5.16 RECHERCHE PLEIN TEXTE

PostgreSQL : *Full Text Search*/Recherche Plein Texte

- correspond à CONTEXT
- opérateur `@@` équivalent à `CONTAINS`

```
SELECT to_tsvector('fat cats ate fat rats') @@ to_tsquery('fat & rat');
```

- S'appuie sur GIN ou GiST
- Extension `pg_trgm` pour les recherches `LIKE '%mot%mot%'`, équivalent de CTXCAT

L'extension `pg_trgm` apporte des classes d'opérateur pour les index GiST et GIN permettant de créer un index sur une colonne texte pour les recherches rapides par similarités. Ces index permettent notamment la recherche par trigrammes pour les requêtes à base de `LIKE`, `ILIKE`, `~` et `~*`.

Exemple :

```
CREATE TABLE test_trgm (t text);
CREATE INDEX trgm_idx ON test_trgm USING GIN (t gin_trgm_ops);

SELECT * FROM test_trgm WHERE t LIKE '%foo%bar';
SELECT * FROM test_trgm WHERE t ~ '(foo|bar)';
```

Ce type d'index peut correspondre aux index Oracle CTXCAT indexant des textes de petites tailles. Il faut toutefois réécrire les requêtes utilisant l'opérateur CATSEARCH en requêtes utilisant `LIKE` ou `ILIKE`.

L'indexation FTS est un des cas les plus fréquents d'utilisation non-relationnelle d'une base de données : les utilisateurs ont souvent besoin de pouvoir rechercher une information qu'ils ne connaissent pas parfaitement, d'une façon floue :

- Recherche d'un produit/article par rapport à sa description
- Recherche dans le contenu de livres/documents

PostgreSQL doit donc permettre de rechercher de façon efficace dans un champ texte. L'avantage de cette solution est d'être intégrée au SGBD. Le moteur de recherche est donc toujours parfaitement à jour avec le contenu de la base, puisqu'il est intégré avec le reste des transactions.

Voici un exemple succinct de mise en place de FTS :

- Création d'une configuration de dictionnaire dédiée (français sans accent) :

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

- Ajout d'une colonne vectorisée à la table `depeches`, afin de maximiser les performances de recherche :

```
ALTER TABLE depeche ADD vect_depeche tsvector;
```

- Création du contenu de vecteur pour les données de la table `depeche` :

```
UPDATE depeche set vect_depeche = (setweight(
    to_tsvector('depeches',coalesce(titre,'')), 'A'
    ) || setweight(
    to_tsvector('depeches',coalesce(texte,'')), 'C')
);
```

- Création de la fonction qui sera associée au trigger :

```
CREATE FUNCTION to_vectdepeche( )
 RETURNS trigger
 LANGUAGE plpgsql
 -- common options:  IMMUTABLE  STABLE  STRICT  SECURITY DEFINER
AS $function$
BEGIN
  NEW.vect_depeche := setweight(to_tsvector('depeches',coalesce(NEW.titre,''))
                        , 'A') ||
                      setweight(to_tsvector('depeches',coalesce(NEW.texte,''))
                        , 'C');
  return NEW;
END
$function$
;
```

Le rôle de cette fonction est d'automatiquement mettre à jour le champ `vect_depeche` par rapport à ce qui aura été modifié dans l'enregistrement. On donne aussi des poids différents aux zones `titre` et `texte` du document, pour qu'on puisse éventuellement utiliser cette information pour trier les enregistrements par pertinence lors des interrogations.

- Création du trigger :

```
CREATE TRIGGER trg_depeche before INSERT OR update ON depeche
FOR EACH ROW EXECUTE PROCEDURE to_vectdepeche();
```

Et ce trigger appelle la fonction définie précédemment à chaque insertion ou modification d'enregistrement dans la table.

NB : à partir de la v12, une colonne générée est préférable à l'alimentation par trigger.

- Création de l'index associé au vecteur :

```
 CREATE INDEX idx_gin_texte ON depeche USING gin(vect_depeche);
```

L'index permet bien sûr une recherche plus rapide.

- Collecte des statistiques sur la table :

```
ANALYZE depeche ;
```

- Utilisation :

```
SELECT titre,texte FROM depeche
    WHERE vect_depeche @@ to_tsquery('depeches','varicelle');
SELECT titre,texte FROM depeche
    WHERE vect_depeche @@ to_tsquery('depeches','varicelle & médecin');
```

La recherche plein texte PostgreSQL consiste en la mise en relation entre un vecteur (la représentation normalisée du texte à indexer) et d'une tsquery, c'est-à-dire une chaîne représentant la recherche à effectuer. Ici par exemple, la première requête recherche tous les articles mentionnant « varicelle », la seconde tous ceux parlant de « varicelle » et de « médecin ». Nous obtiendrons bien sûr aussi les articles parlant de médecine, « médecine » ayant le même radical que « médecin » et étant donc automatiquement classé comme faisant partie de la même famille.

La recherche propose bien sûr d'autres opérateurs que `&` : `|` pour « ou », `!` pour « non ». On peut effectuer des recherches de radicaux, etc. L'ensemble des opérations possibles est détaillée ici : https://docs.postgresql.fr/current/textsearch-controls.html.

On peut trier par pertinence :

```
SELECT titre,texte
FROM depeche
WHERE vect_depeche @@ to_tsquery('depeches','varicelle & médecin')
ORDER BY ts_rank_cd(vect_depeche, to_tsquery('depeches','varicelle & médecin'));
```

Ou, écrit autrement (pour éviter d'écrire deux fois `to_tsquery`) :

```
SELECT titre,ts_rank_cd(vect_depeche,query) AS rank
FROM depeche, to_tsquery('depeches','varicelle & médecin') query
WHERE query@@vect_depeche
ORDER BY rank DESC
```

Ce type d'indexation plein texte correspond à la recherche de texte Oracle basée sur des index de type CONTEXT. Il sera aussi nécessaire de réécrire les requêtes Oracle utilisant l'opérateur `CONTAINS` avec l'operateur `@@` de PostgreSQL.

1.5.17 PRÉPARATION DE L'IMPORT

- Préparation de l'import du schéma
  - création du propriétaire de la base
  - création de la base
- Si EXPORT_SCHEMA est activé
  - création du schéma
  - utilisation d'un schéma par défaut
- Création des tablespaces

La première chose à faire avant de commencer à migrer réellement la base Oracle dans une base PostgreSQL est de créer le propriétaire de la base de données, toutes les opérations se feront ensuite sous cet utilisateur. Voici comment créer le propriétaire de la base :

```
$ createuser --no-superuser --no-createrole --no-createdb myuser
```

On procède ensuite à la création de la base elle-même :

```
$ createdb -E UTF-8 --owner myuser mydb
```

Si vous avez décidé d'exporter le schéma Oracle avec la variable EXPORT_SCHEMA activée, il faut créer le schéma sous PostgreSQL :

```
$ psql -U myuser mydb -c "CREATE SCHEMA myschema;"
```

Pour faciliter ensuite l'utilisation du schéma, il est possible d'affecter un schéma par défaut à un utilisateur de sorte qu'à chaque fois qu'il se connecte à la base, ce sont les schémas donnés qui seront utilisés :

```
$ psql -U myuser mydb -c \
    "ALTER ROLE miguser SET search_path TO myschema,public;"
```

Si des tablespaces doivent être importés, les chemins doivent exister sur le système. Il faut donc s'assurer qu'ils sont présents et que PostgreSQL pourra écrire dans ces répertoires.

1.5.18 IMPORT DU SCHÉMA

Création des objets du schéma :

```
psql -U myuser -f schema/tables/tables.sql mydb >> create_mydb.log 2>&1
psql -U myuser -f schema/partitions/partitions.sql mydb >> create_mydb.log 2>&1
psql -U myuser -f schema/views/views.sql mydb >> create_mydb.log 2>&1
psql -U myuser -f schema/tablespaces/tablespaces.sql mydb >> create_mydb.log 2>&1
```

La base étant créée, il ne reste plus qu'à charger les différents objets en commençant par les tables, puis les partitions, s'il y en a, les vues et pour finir les tablespaces pour déplacer les objets dans leur espaces de stockage respectif (l'export des tablespaces contient non seulement les tablespaces, mais aussi les `ALTER TABLE` et `ALTER INDEX` déplaçant les tables et index dans leur tablespace de destination).

1.5.19 IMPORT DIFFÉRÉ

Chargement différé de certains objets :

- Séquences
- Contraintes
- Déclencheurs
- Index

Les objets susceptibles de gêner l'import des données, soit en provoquant des erreurs comme les contraintes, soit en ralentissant leur chargement comme les index, sont laissés de côté et ne seront importés qu'à la fin de la migration. Dans ce cas, il faudra lancer deux fois le script `tablespaces.sql`, une fois après le chargement des tables, une fois après le chargement des index, et ignorer les erreurs.

1.5.20 BILAN DE L'EXPORT/IMPORT

Bilan de l'export/import du schéma :

- Lecture des logs et étude des problèmes
- Sensibilité à la casse
- Encodage des valeurs de contraintes `CHECK` et conditions des index
- Possibilité de code spécifique à Oracle dans les contraintes et les index
- Champs numériques

Lors du chargement du schéma, il y a normalement assez peu d'erreurs. Du coup, elles peuvent facilement passées inaperçues. Il est donc important de bien scruter les journaux applicatifs au fur et à mesure des commandes d'import pour détecter ces erreurs.

Les types d'erreur pouvant survenir sont souvent des problèmes d'encodage dans les valeurs des contraintes `CHECK` et dans les index. Dans ce cas, il faut utiliser les ordres :

```
SET client_encoding TO autre_encodage;
```

Avec aussi la possibilité, pour les contraintes et index, de trouver du code SQL utilisant des fonctions qui ne sont pas convertibles automatiquement par Ora2Pg.

```
CREATE INDEX idx_userage ON players ( to_number(to_char('1974', user_age)) );
```

```
ALTER TABLE "actifs" ADD CONSTRAINT CHECK (WYEAR between 0 and 42);
```

Ora2Pg exporte les champs Oracle de type `NUMBER` sans précision en `bigint`. Ce n'est pas forcément le bon choix notamment lorsque ce champ contient des valeurs avec décimale. Une erreur va se produire lors de l'import des données. Il sera nécessaire alors de modifier le type de la colonne à posteriori.

1.5.21 EXEMPLE D'ERREURS

- Accents dans les noms d'objets
- Mots réservés
- Certaines conversions implicites
  - ...`CHECK (WYEAR between 0 and 9);`
  - ...`CHECK (wyear::integer between 0 and 9);`

On trouve de temps en temps des objets comportant des accents, sans compter qu'il faudra que le nom de l'objet soit toujours placé entre guillemets doubles. Il faudra aussi utiliser le bon encodage lors de la création et des appels à l'objet. Ceci génère énormément d'erreurs et il est fortement conseillé de les supprimer.

Ora2Pg ne détecte pas les noms d'objets correspondants à des mots réservés PostgreSQL. Il vous faudra, dans ce cas, modifier manuellement le code SQL en les incluant entre guillemets doubles.

```
CREATE INDEX idx_userage ON user WHERE age > 16;
```

```
CREATE INDEX idx_userage ON "user" WHERE age > 16;
```

Oracle autorise certaines conversions implicites qui ne sont plus autorisées dans PostgreSQL depuis la version 8.3 (principalement les conversions implicites entre numériques et chaînes de caractères). Dans l'exemple, `WYEAR` est une colonne de type `VARCHAR` dans Oracle et exportée comme telle dans PostgreSQL. Il faudra donc forcer sa transformation en `integer` pour que la contrainte fonctionne, sinon vous obtiendrez une erreur du genre :

```
ERROR:  operator does not exist: character varying >= integer
```

1.6 MIGRATION DES DONNÉES

Étapes :

- Export / import des données
- Problèmes rencontrés
- Restauration des séquences, contraintes, triggers et index
- Performances de l'import des données
- Utilisation du parallélisme
- Limitation des données à importer

Nous allons aborder ici les différentes étapes pour migrer de façon optimale les données :

- comment exporter les données ?
- comment importer les données ?
- quels problèmes peuvent être rencontrés lors de l'import des données ?
- application des ordres DDL de création des séquences, contraintes, triggers et index ?
- comment accélérer l'import des données dans PostgreSQL ?
- comment n'extraire qu'une partie des données ?

1.6.1 EXPORTER LES DONNÉES

- Création des fichiers de données :

```
ora2pg -t COPY -o datas.sql -b data/ -c config/ora2pg.conf
ora2pg -t INSERT -o datas.sql -b data/ -c config/ora2pg.conf
```

- Un fichier de données par table :
  - FILE_PER_TABLE 1
- Compression des fichiers de données

Il faut privilégier le premier type d'export à base d'instructions COPY plutôt que le second à base d'ordre INSERT. Il y a deux raisons à cela : l'import sera beaucoup plus rapide avec COPY et vous aurez potentiellement moins d'erreurs si vos données contiennent des caractères d'échappement (\).

Si l'option FILE_PER_TABLE est activée, Ora2Pg va créer un fichier de chargement de données par table exportée et le fichier tables.sql ne sera qu'un fichier de chargement global de ces fichiers à base d'instruction psql : \i nom_fichier.sql.

Si les directives de configuration DISABLE_TRIGGERS et DROP_FKEY ont été activées, le fichier global contient aussi les appels de désactivation/activation des triggers et de suppres-

sion/création des contraintes.

L'avantage d'avoir des fichiers de données à disposition est qu'ils peuvent être rechargés manuellement plusieurs fois en cas de problème jusqu'à trouver le correctif à apporter.

Dans la mesure où l'export de données dans des fichiers peut occuper un volume disque très important, Ora2Pg vous donne la possibilité de compresser vos données soit avec `gzip` soit avec `bzip2`. Pour le premier type de compression, il faut installer au préalable le module Perl `Compress::Zlib` et donner l'extension `.gz` au fichier de sortie :

```
ora2pg -t COPY -o datas.sql.gz -b data/ -c config/ora2pg.conf
```

Pour utiliser la compression avec `bzip2`, il suffit que le programme `bzip2` soit dans le `PATH` et il faut donner l'extension `.bz2` au fichier de sortie :

```
ora2pg -t COPY -o datas.sql.bz2 -b data/ -c config/ora2pg.conf
```

La compression se fait au fil de l'export et non à la fin lorsque le fichier est créé.

1.6.2 CAS DES DONNÉES CLOB/BLOB

- Les champs `bytea`
  - export des champs `BLOB` et `CLOB` en `bytea` très lent
  - exclusion temporaire des tables avec `LOB`
  - utilisation de la parallélisation pour ces tables

La lenteur de l'export des champs de type `LOB` dans des champs `bytea` (qui est le type correspondant sous PostgreSQL) s'explique par la taille habituellement élevée de ces données et la nécessité d'échapper l'intégralité des données.

Si la volumétrie de ce type de données est très importante, il est préférable d'exclure temporairement de l'export les tables possédant des champs de ce type en les ajoutant à la directive `EXCLUDE`. À partir de là, une fois le chargement des données des autres tables réalisé, il suffit de déplacer le nom de ces tables avec des champs `LOB` dans la directive `ALLOW` pour que l'export des données se fasse uniquement à partir de ces tables.

Pour accélérer l'échappement des données `bytea`, il faut activer l'utilisation du parallélisme. Cela permet en général d'aller deux à trois fois plus vite. Pour cela, il faut utiliser l'option `-j` en ligne de commande ou la variable `JOBS` du fichier de configuration. La valeur est le nombre de cœurs CPU que l'on veut utiliser.

Lorsque les options de parallélisation sont activées, il est important de s'assurer que la valeur de `DATA_LIMIT` corresponde à la vitesse moyenne maximal d'export d'un simple pro-

cessus. Par exemple, si Ora2Pg exporte globalement les données à une vitesse moyenne de 5000 tuples/s, c'est très certainement la valeur à donner :

```
DATA_LIMIT     5000
```

Si c'est plutôt 20 000, alors `DATA_LIMIT` devra avoir cette valeur. Ceci vous permettra d'être sûr de tirer le meilleur parti de la parallélisation. Une valeur excessive par contre peut conduire à des dépassement de ressources, une valeur trop faible forcera Ora2Pg à créer des processus inutilement.

1.6.3 CAS DES DONNÉES SPATIALES

- Le `SRID`, système spatial de référence
  - `CONVERT_SRID` converti la valeur Oracle dans la norme EPSG
  - `DEFAULT_SRID` force la valeur du SRID par défaut
- Mode d'extraction des données `GEOMETRY_EXTRACT_TYPE [WKT|WKB|INTERNAL]`

CONVERT_SRID

Oracle utilise son propre système spatial de référence SRID (*Spatial Reference System Identifier*), la norme de fait est maintenant l'ESPG (*European Petroleum Survey Group*). Oracle fournit la fonction `sdo_cs.map_oracle_srid_to_epsg()` permettant de le convertir dans cette norme lorsque c'est possible. Si la directive `CONVERT_SRID` est activée, la conversion sera effectuée.

Cette fonction retourne souvent `NULL`. Dans ce cas, Ora2Pg renvoit la valeur `8307` comme SRID par défaut ou, si `CONVERT_SRID` est activée, `4326` converti en ESPG. Il est possible de changer cette valeur par défaut en donnant la valeur du SRID à utiliser à la directive `CONVERT_SRID`. À noter que dans ce cas, `DEFAULT_SRID` ne sera pas utilisé.

DEFAULT_SRID

La directive `DEFAULT_SRID` permet de changer la valeur par défaut du SRID EPSG à utiliser si la valeur retournée est nulle. Elle vaut `4326` par défaut.

GEOMETRY_EXTRACT_TYPE

Cette directive permet d'informer Ora2Pg sur la méthode à utiliser pour extraire les données. Il existe trois possibilités :

- WKT
- WKB
- INTERNAL

La valeur `WKT` ordonne à Ora2Pg d'utiliser la fonction Oracle `SDO_UTIL.TO_WKTGEOMETRY()` pour extraire les données. Ora2Pg prend alors la représentation textuelle de la donnée géométrique renvoyée par Oracle sans transformation autre que l'ajout du SRID.

La valeur `WKB` ordonne à Ora2Pg d'utiliser la fonction Oracle `SDO_UTIL.TO_WKBGEOMETRY()` pour extraire les données. Ora2Pg prend alors la représentation binaire de la donnée géométrique renvoyée par Oracle la convertit en hexadécimal et ajoute le SRID.

L'utilisation de ces fonctions est intéressante pour obtenir les géométries telle que les voit Oracle ; le seul problème est qu'elles génèrent souvent des erreurs, sont incapables d'extraire des géométries en 3D et surtout provoquent des OOM (*Out Of Memory*) lorsque il y a un grand nombre de géométries.

Pour palier à ce problème, Ora2Pg embarque sa propre librairie *Pure Perl*, `Ora2Pg::GEOM`, permettant d'extraire les données géométriques au format WKT de manière plus rapide et surtout sans erreur. Pour utiliser cette méthode, il faut donner la valeur `INTERNAL` à la directive `GEOMETRY_EXTRACT_TYPE`.

La valeur par défaut est `INTERNAL`.

1.6.4 IMPORT DES DONNÉES

- Import des fichiers de données :

```
psql -U myuser -f data/datas.sql mydb >> data_mydb.log 2>&1
gunzip -c data/datas.sql.gz | psql  -U myuser  mydb >> data_mydb.log 2>&1
bunzip2 -c data/datas.sql.bz2 | psql  -U myuser  mydb >> data_mydb.log 2>&1
```

- Chargement direct dans PostgreSQL lors de l'export

L'import des fichiers de données se fait simplement avec l'utilisation de la commande `psql` en spécifiant l'utilisateur (`myuser`), la base de données (`mydb`) et le fichier à charger (option `-f`).

Si le fichier de données est compressé, il est nécessaire d'utiliser le programme de décompression adéquat et de renvoyer la sortie vers la commande `psql` pour permettre le chargement des données au fil de la décompression.

L'import direct des données dans la base PostgreSQL n'est activé que si la variable `PG_DSN` est définie. Dans ce cas, le chargement se fait directement lors de l'export des données sans passer par des fichiers intermédiaires.

1.6.5 RESTAURATION DES CONTRAINTES

Restauration des contraintes, triggers, séquences et index

```
psql -U myuser -f schema/tables/CONSTRAINTS_tables.sql mygdb >> create_mydb.log 2>&1
psql -U myuser -f schema/tables/INDEXES_tables.sql mygdb >> create_mydb.log 2>&1
psql -U myuser -f schema/sequences/sequences.sql mygdb >> create_mydb.log 2>&1
psql -U myuser -f schema/triggers/triggers.sql mygdb >> create_mydb.log 2>&1
```

Une fois que les données sont chargées avec succès, il est temps de créer les contraintes, index et triggers qui avaient été laissés de côté lors de la création du schéma. Ces créations se font aussi à l'aide de la commande `psql`.

Il est possible que l'import de certains codes, notamment les triggers, nécessitent la présence de certaines fonctions. Dans ce cas, il faudra les intégrer en parallèle.

1.6.6 RESTAURATION PARALLÉLISÉE DES CONTRAINTES

Action :

- `LOAD` permet de paralléliser des ordres SQL sur N processus

```
ora2pg -c config/ora2pg -t LOAD -j 4 -i schema/tables/INDEXES_tables.sql
ora2pg -c config/ora2pg -t LOAD -j 4 -i schema/tables/CONSTRAINTS_tables.sql
```

La création des contraintes et des index est une phase qui très souvent dure presque aussi longtemps que le chargement des données, voire plus longtemps en fonction du nombre.

Depuis la version 16.0 d'Ora2Pg, l'action `LOAD` permet de donner un fichier d'ordre SQL en entrée (option `-i`) et de distribuer sur plusieurs processeurs ces requêtes SQL à l'aide de l'option `-j N` d'Ora2Pg.

Il suffit dans ce cas de lui donner en entrée les fichiers relatifs à la création des contraintes et des index pour pouvoir les charger beaucoup plus rapidement.

1.6.7 PROBLÈMES D'IMPORT DES DONNÉES

- Problème d'échappement de caractères : utiliser `COPY`
- Encodage des données : `CLIENT_ENCODING`
- Erreur de type numérique : `DEFAULT_NUMERIC` ou `ALTER TABLE`
- `CLOB`, `BLOB` et `XML` : `LONGREADLEN`

Si le type d'export `INSERT` a été choisi, il arrive très souvent que cela conduise à des erreurs de caractères invalides lors de l'insertion car le caractère `backslash` n'est pas échappé si `STANDARD_CONFORMING_STRING` est activé. Le respect du standard est activé par défaut dans

PostgreSQL v9.1. Dans Ora2Pg, le même comportement survient. De ce fait, ils doivent être activés ou désactivés en même temps dans les deux configurations. Le meilleur moyen de corriger ce problème est d'utiliser le type d'export recommandé pour les données, c'est-à-dire `COPY`.

Si vous n'avez pas défini correctement les variables `NLS_LANG` et `CLIENT_ENCODING`, vous aurez aussi des erreurs de caractères invalides. Il vous faudra alors trouver les bonnes valeurs selon la méthode indiquée dans les chapitres précédents. Malgré une définition correcte de ces variables, il se peut que vous ayez encore des problèmes d'encodage, et même au sein d'une même table : certains enregistrements ne passeront pas par `COPY`. Il semble qu'Oracle soit très permissif sur les caractères qu'il est possible d'inclure dans un même jeu de caractères.

Pour l'essentiel, ces problèmes sont résolus en forçant toutes les communications à utiliser l'encodage UNICODE, c'est ce qu'Ora2Pg fait par défaut depuis la version 14.0.

Au besoin, chaque bloc d'import de données est précédé d'un appel à

```
SET client_encoding TO '...';
```

la valeur étant celle définie dans la variable `CLIENT_ENCODING` du fichier de configuration `ora2pg.conf`. Vous pourrez donc ajuster le jeux de caractères à utiliser au niveau de PostgreSQL au plus près des données.

Les erreurs de type numérique apparaissent en raison de la conversion du type Oracle `NUMBER` sans précision qui est par défaut converti dans le type donné à la variable `DEFAULT_NUMERIC`, c'est-à-dire `bigint`. Comme le type Oracle permet d'inclure aussi bien des entiers que des décimaux, une erreur va inévitablement se produire si des décimaux se trouvent dans les données importées.

Pour résoudre ce problème, il faut évaluer la quantité de champs concernés par ce problème. Si cela ne concerne que peu de champs et qu'il est possible d'avoir une valeur décimale pour ces champs, le mieux est de changer le type directement :

```
ALTER TABLE employees ALTER COLUMN real_age TYPE real;
```

Si par contre le problème se pose de manière quasi systématique, il est alors préférable de modifier le type défini dans `DEFAULT_NUMERIC` et de recommencer l'import complet.

Lors de l'export des `LOB`, si vous n'avez pas activé la directive `NO_LOB_LOCATOR`, il se peut que vous rencontriez l'erreur Oracle :

```
ORA-24345: A Truncation or null fetch error occurred (DBD SUCCESS_WITH_INFO:
    OCIStmtFetch, LongReadLen too small and/or LongTruncOk not set)
```

La solution est d'augmenter la valeur du paramètre `LONGREADLEN`, par défaut 1 Mo, à la taille du plus grand enregistrement de la colonne. Vous avez aussi la possibilité de tronquer les données en activant `LONGTRUNCOK` , ce qui ne remontera plus d'erreur mais bien évidement tronquera certaines données dont la taille dépasse la valeur de `LONGREADLEN`. Pour plus d'explication, voir le chapitre *Configuration liée aux LOB*.

1.6.8 PERFORMANCES DE L'IMPORT DES DONNÉES

- Type d'export `COPY`
- Import direct dans PostgreSQL

```
PG_DSN  dbi:Pg:dbname=test_db;host=localhost;port=5432
PG_USER [nom_utilisateur]
PG_PWD  [mot_de_passe]
```

- Nombre d'enregistrements traités en mémoire : `DATA_LIMIT`

La méthode la plus simple pour gagner en performances est d'utiliser la méthode `COPY` et de ne pas passer par des fichiers intermédiaires pour importer ces données. Pour envoyer directement les données extraites de la base Oracle vers la base PostgreSQL, il suffit de définir les paramètres de connexion à la base PostgreSQL dans le fichier de configuration `ora2pg.conf`.

COPY ou INSERT

Préférez toujours l'import des données à l'aide de l'ordre `COPY` plutôt qu'à base d'`INSERT`. Ce dernier est beaucoup trop lent pour les gros volumes de données. Lorsque l'import direct dans PostgreSQL est utilisé, Ora2Pg va utiliser une requête préparée et passer les valeurs de chaque ligne en paramètre, mais même avec cette méthode, le chargement avec l'instruction `COPY` reste le plus performant.

PG_DSN

Il s'agit de l'équivalent pour PostgreSQL de l' `ORACLE_DNS` pour Oracle dans le fichier de configuration d'Ora2Pg.

On détermine donc ici la chaîne de connexion à PostgreSQL, en particulier :

- le connecteur DBI à utiliser ;
- le nom de la base de données PostgreSQL, `dbname=` ;
- le nom du serveur PostgreSQL à utiliser, `host=` ;
- et le port sur lequel le serveur PostgreSQL écoute, `port=`.

Par exemple, pour la base `xe` se trouvant sur le serveur `postgresql_server:5432` :

```
PG_DSN dbi:Pg:dbname=xe;host=postgresql_server;port=5432
```

L'utilisation de cette chaîne de connexion nécessite l'installation du module Perl `DBD::Pg` et donc des bibliothèques PostgreSQL.

PG_USER

Il détermine le nom de l'utilisateur PostgreSQL qui sera utilisé pour se connecter à la base PostgreSQL désignée par le paramètre `PG_DSN`.

Exemple, pour l'utilisateur `prod` :

```
PG_USER prod
```

PG_PWD

Il détermine le mot de passe de l'utilisateur PostgreSQL désigné par `PG_USER` pour se connecter sur la base PostgreSQL désignée par `PG_DSN`.

Par exemple, si le mot de passe est « secret » :

```
PG_PWD secret
```

DATA_LIMIT

Par défaut, lorsqu'on demande à Ora2Pg d'extraire les données, il récupère les données par bloc de 10 000 lignes.

Ceci permet d'écrire dans le fichier en sortie ou de transférer les données vers une base PostgreSQL toutes les 10 000 lignes et ainsi réduire les entrées/sorties. Cependant, suivant la configuration matérielle de la machine, il peut être très intéressant de faire varier cette valeur pour gagner en performance. Par exemple, sur une machine disposant de beaucoup de mémoire, travailler sur 100 000 enregistrements à chaque fois ne doit pas poser de problème et permet d'accroître les performances de manière significative.

```
DATA_LIMIT 100000
```

Si, par contre, votre machine dispose de très peu de mémoire ou que les enregistrements sont de très grosse taille, cette valeur devra être diminuée, par exemple :

```
DATA_LIMIT 1000
```

1.6.9 UTILISER LE PARALLÉLISME

- Parallélisme pour le traitement et l'import des données dans PostgreSQL
  - `JOBS` `Ncores`
- Parallélisme pour l'extraction des données d'Oracle
  - `ORACLE_COPIES` `Ncores`
  - `DEFINED_PKEY` `EMPLOYEE:ID`
- Parallélisme par tables exportées
  - `PARALLEL_TABLES` `Ncores`
- Nombre de processus utilisés
  - `JOBS` x `ORACLE_COPIES | PARALLEL_TABLES` = Total Nombre cœurs

Ora2Pg de base n'utilise qu'un seul CPU ou cœur pour le chargement des données. Ceci est très limitant en terme de vitesse d'importation des données. Pour utiliser le parallélisme sur plusieurs cœurs, Ora2Pg dispose de deux directives de configuration : `JOBS` et `ORACLE_COPIES`, correspondant respectivement aux options `-j` et `-J` de la ligne de commande.

La première, `-j` ou `JOBS`, correspond au nombre de processus que l'on veux utiliser en parallèle pour écrire les données directement dans PostgreSQL. La seconde, `-J` ou `ORACLE_COPIES`, est utilisée pour définir le nombre de connexions à Oracle pour extraire les données en parallèle.

Toutefois, pour que les requêtes d'extraction des données de la base Oracle puissent être parallélisées, il faut qu'Ora2Pg ait connaissance d'une colonne de la table sur laquelle la division par processus peut être réalisée. Cette colonne doit être de type numérique et, de préférence, être une clé unique car Ora2Pg va scinder les données en fonction du nombre de processus demandés selon le principe de la requête suivante :

```
SELECT * FROM matable WHERE MOD(colonne, ORACLE_COPIES) = #PROCESSUS;
```

où `colonne` est la clé unique, `ORACLE_COPIES` est la valeur de la variable du même nom ou de l'option `-J` et `#PROCESSUS` est le numéro du processus parallélisé en commençant par 0.

Cette colonne est renseignée à l'aide de la directive de configuration `DEFINED_PKEY` avec pour valeur une liste de tables associées à leurs colonnes, par exemple :

```
DEFINED_PKEY    EMPLOYEE:ID JOBS:ID TARIF:ROUND(MONTANT_HT) ...
```

L'utilisation de la fonction `ROUND()` est impérative lorsque le champ n'est pas un entier. Il est à noter que l'option `-J` est sans effet si la table exportée n'a pas de colonne définie dans la directive `DEFINED_PKEY`.

En affinant les valeurs données à `-j` et `-J`, il est possible de multiplier par 6 à 10 la vitesse

de chargement des données par rapport à un chargement n' utilisant pas la parallélisation.

Les valeurs de `-j` et `-J` se multiplient entre elles. Il faut donc faire attention à ne pas dépasser le nombre de cœurs disponible sur la machine, par exemple :

```
ora2pg -t COPY -c ora2pg.conf -J 8 -j 3
```

ouvrira 8 connexions à Oracle pour extraire les données en parallèle et, pour chacune de ces connexions, 3 processus supplémentaires seront utilisés pour enregistrer les données dans PostgreSQL, ce qui donne 24 cœurs utilisés par Ora2Pg.

Ce type de parallélisme est contraignant à mettre en œuvre et peut être mis en oeuvre par exemple pour extraire des données d'une table avec de nombreux `CLOB` ou `BLOB` pour tenter d'accélérer son export.

Pour paralléliser l'export de plusieurs tables en simultané, on peut aussi utiliser la directive `PARALLEL_TABLES`. Cette variable prend comme valeur le nombre de connexions à Oracle qui devront être ouvertes pour extraire les données des différentes tables en simultané. Lorsque cette directive a une valeur supérieur à 1, la variable `FILE_PER_TABLE` est automatiquement activée.

Par défaut, ces trois options ont la valeur `1`.

```
JOBS            1
PARALLEL_TABLES  1
ORACLE_COPIES    1
```

Suivant la structure d'une table, il peut être aussi nécessaire de faire bouger la valeur de la directive `DATA_LIMIT` qui, par défaut, est à `10000`. Pour les tables dont l'export est très rapide, une valeur à `100000` est préférable, alors que pour les tables avec `LOB` et potentiellement des enregistrements de très grande taille, une valeur à `100` sera probablement nécessaire. Cette valeur est aussi relative aux performances du système. Une bonne démarche est de tester la vitesse d'export sur des tables moyennes et de positionner la valeur de `DATA_LIMIT` à ce niveau, par exemple :

```
DATA_LIMIT        60000
```

Puis, sur les tables à très faible débit, utiliser l'option de ligne de commande `-L` :

```
ora2pg -t COPY -c ora2pg.conf -J 8 -j 3 -L 100
```

La plupart du temps, 90 % des tables peuvent être exportées avec la même configuration du `DATA_LIMIT` et du parallélisme pour les insertions dans PostgreSQL seul. Par exemple, sur un serveur avec 24 cœurs et 64 Go de RAM, la commande suivante (PostgreSQL tournant sur ce même serveur) :

```
ora2pg -t COPY -c ora2pg.conf -j 16 -L 60000
```

traitera parfaitement la très grande majorité des tables. Il est à noter que l'option `-j` est sans effet si le nombre de lignes de la table en cours d'export divisé par la valeur de `-j` (dans l'exemple au dessus : 16) est inférieur à la valeur donnée dans le `DATA_LIMIT`.

Pour les autres, il faut identifier les tables avec des `CLOB` et `BLOB`, les tables avec le plus grand nombre de lignes et celles avec les plus gros volumes de données. Ensuite, il faut voir s'il est possible de multiplexer les connexions à Oracle pour accélérer l'export ainsi que la valeur qui sera le mieux adaptée au `DATA_LIMIT` en faisant des tests d'import de données.

1.6.10 LIMITATION DES DONNÉES EXPORTÉES

- Contrôle des tables à exporter
  - `ALLOW TABLE1 TABLE2 [..] TABLEN`
  - `EXCLUDE TABLE1 TABLE2 [..] TABLEN`
- Contrôle des données à exporter
  - `WHERE TABLE[condition valide] GLOBAL_CONDITION`
  - `WHERE TABLE_TEST[ID1='001']`
  - `WHERE DATE_CREATION > '2001-01-01'`
  - `REPLACE_QUERY  TABLENAME[SQL_QUERY]`

ALLOW

Par défaut, Ora2Pg exporte toutes les tables qu'il trouve, au moins dans le schéma désigné avec la directive `SCHEMA`.

On peut cependant limiter l'export à certaines objets, grâce à la directive `ALLOW`. Il suffit ici de donner une liste de noms d'objets, séparées par un espace. Les expressions régulières sont aussi permises.

Exemple :

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

EXCLUDE

C'est le pendant du paramètre `ALLOW` ci-dessus. Cette variable de configuration permet d'exclure des objets de l'extraction. Par défaut, Ora2Pg n'exclut aucun objet. Les expressions régulières sont aussi permises.

Exemple:

```
EXCLUDE EMPLOYEES TMP_.* COUNTRIES EMPLOYEES_COPIE_2010.* TEST[0-9]+
```

Attention, les expressions régulières ne fonctionnent pas avec les versions Oracle 8i, vous devez utiliser le caractère `%` à la place, Ora2Pg utilise l'opérateur `LIKE` dans ce cas.

ALLOW/EXCLUDE : Filtres étendus

Les objets filtrés par ces directives dépendent du type d'export. Les exemples précédents montrent la manière dont sont déclarés les filtres globaux, ceux qui vont s'appliquer quelque soit le type d'export utilisé. Il est possible d'utiliser un filtre sur un type d'objet uniquement en utilisant la syntaxe : `OBJECT_TYPE[FILTER]`. Par exemple :

```
ora2pg -p -c ora2pg.conf -t TRIGGER -a 'TABLE[employees]'
```

limitera l'export des triggers à ceux définis sur la table `EMPLOYEES`. Si vous voulez exporter certains triggers mais pas ceux qui on une clause `INSTEAD OF` (liés à des vues) :

```
ora2pg -c ora2pg.conf -t TRIGGER -e 'VIEW[trg_view_.*]'
```

Ou, par exemple, une forme plus complexe avec inclusion / exclusion d'éléments :

```
ora2pg -p -c ora2pg.conf -t TABLE -a 'TABLE[EMPLOYEES]' \
                                  -e 'INDEX[emp_.*];CKEY[emp_salary_min]'
```

Cette commande va exporter la définition de la table `EMPLOYEES` tout en excluant tous les index commençant par `emp_` et la contrainte `CHECK` nommée `emp_salary_min`.

Autre exemple, lors de l'export des partitions on peut vouloir exclure certaines tables :

```
ora2pg -p -c ora2pg.conf -t PARTITION -e 'PARTITION[PART_199.* PART_198.*]'
```

Ceci va exclure de l'export les tables partitionnées concernant les années 1980 à 1999 mais pas la table principale ni les autres partitions.

Avec l'export des privilèges (`GRANT`) il est possible d'utiliser cette forme étendue pour exclure certains utilisateurs de l'export ou limiter l'export à certains autres :

```
ora2pg -p -c ora2pg.conf -t GRANT -a 'USER1 USER2'
```

ou bien

```
ora2pg -p -c ora2pg.conf -t GRANT -a 'GRANT[USER1 USER2]'
```

qui limitera l'export des privilèges aux utilisateurs `USER1` et `USER2`. Mais si vous ne voulez pas exporter leurs privilèges sur certaines fonctions, alors :

```
ora2pg -p -c ora2pg.conf -t GRANT -a 'USER1 USER2' \
       -e 'FUNCTION[adm_.*];PROCEDURE[adm_.*]'
```

L'utilisation des filtres étendus en fonction de leur complexité peut nécessiter un certain temps d'apprentissage.

WHERE

Ce paramètre permet d'ajouter des filtres dans les requêtes d'extraction de données. Il n'est donc utilisé que dans le cadre d'un export de données, soit avec `TYPE [INSERT|COPY]`.

Ora2Pg ajoutera tous les filtres déclarés dans cette variable et/ou correspondant à une table donnée, lorsque cela est possible.

Il convient de créer plusieurs fichiers `ora2pg.conf` si on doit ajouter des filtres sur de nombreuses tables, car la configuration de `WHERE` peut en effet rapidement devenir illisible si elle est complexe !

- `WHERE 1=1`

Cet exemple trivial est là pour illustrer le fait que si aucune table n'est mentionnée, la clause `WHERE` sera appliquée à toutes les requêtes d'extraction. Si le champ n'existe pas pour une table donnée, il sera ignoré. Autrement dit, Ora2Pg ne s'attend pas à ce que le(s) champ(s) mentionnés sans nom existent dans toutes les tables.

Exemple:

```
WHERE DATE_CREATION > '2001-01-01'
```

Si, pour une table donnée, il existe des conditions sur ses champs (voir plus bas), alors cela prévaut sur un champ qui aurait été configuré sans spécification du nom de table.

- `WHERE TABLE_TEST[ID1='001']`

On peut bien sûr préciser une expression pour une ou plusieurs colonnes d'une table donnée.

Par exemple, si on ne veut sélectionner que les départements dans la table `DEPARTMENTS` dont le champ `ID` est strictement inférieur à `100` :

```
WHERE departments[DEPARTMENT_ID<100]
```

Cela donne :

```
COPY "departments" ("department_id","department_name",[...])
FROM stdin;
10      Administration  200     1700
20      Marketing       201     1800
30      Purchasing      114     1700
40      Human Resources 203     2400
50      Shipping        121     1500
60      IT      103     1400
70      Public Relations        204     2700
80      Sales   145     2500
```

```
90      Executive       100     1700
\.
```

- `WHERE  TABLE_TEST[ID1='001' AND ID1='002] DATE_CREATE > '2001-01-01' TABLE_INFO[NAME='test']`

On peut ainsi composer sur plusieurs champs d'une même table, et ainsi de suite pour plusieurs tables à la fois. Il suffit pour cela de respecter la convention `NOM_DE_TABLE[COLONNE... etc.]` et de séparer chaque élément par un espace.

Par exemple, si on veut restreindre les données ci-dessus aux `MANAGER_ID` strictement supérieurs à `200`, on écrira :

```
WHERE DEPARTMENTS[DEPARTMENT_ID<100 AND MANAGER_ID>200]
```

Ce qui donne comme résultat:

```
COPY "departments" ("department_id","department_name",[...])
FROM stdin;
20      Marketing       201     1800
40      Human Resources 203     2400
70      Public Relations        204     2700
\.
```

REPLACE_QUERY

Le comportement normal d'Ora2Pg est de générer automatiquement la requête d'extraction des données de la manière suivante :

```
SELECT * FROM TABLENAME [CLAUSE_WHERE];
```

Quelques fois cela n'est pas suffisant, par exemple si l'on souhaite faire une jointure sur une table d'identifiants à migrer ou tout autre requête plus complexe que ce que ne peut produire Ora2Pg. Dans ce cas il est possible de forcer Ora2Pg a utiliser la requête SQL qui lui sera donné par la directive `REPLACE_QUERY`. Par exemple :

```
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')
                ]
```

Cette requête permet de n'extraire que les enregistrements de la table `employees` qui ont été créés depuis le 1er août 2014 sachant que l'information se trouve dans la table `emp_updt`.

1.7 CONCLUSION

- Le temps de migration du schéma et des données est très rapide...
- ...il est souvent marginal par rapport au temps de la migration du code
- Préférer toujours la dernière version d'Ora2Pg
- Faites un retour d'expérience de votre migration à l'auteur

Ora2Pg est simple d'utilisation. Sa configuration permet de réaliser facilement plusieurs fois la migration, pour les différentes étapes du projet. Son auteur est en recherche permanente d'améliorations ou de corrections, n'hésitez pas à lui envoyer un mail pour lui indiquer votre ressenti sur l'outil, vos rapports de bogues, etc.

Le temps de migration du schéma et des données est rapide. Même avec une grosse volumétrie de données, le plus long concerne généralement le code, au niveau applicatif comme au niveau des routines stockées.

1.7.1 POUR ALLER PLUS LOIN

- Documentation officielle[a]
- Conférence sur Data2Pg[b]
- Wiki PostgreSQL[c]

Vous pouvez retrouver la documentation en ligne en anglais sur le site officiel d'Ora2Pg.

Dans l'éventualité où les temps de chargement sont un frein à la migration, Dalibo contribue à un outil spécialisé dans l'orchestration du chargement de données nommé Data2Pg. Son auteur, Philippe Beaudoin, a pu le présenter lors du PG Day France 2022 :

- Migration vers PostgreSQL : mener de gros volumes de données à bon port[9]

Une série de documents concernant la migration Oracle vers PostgreSQL est disponible sur le wiki PostgreSQL.

---

[a] https://ora2pg.darold.net/
[b] https://github.com/dalibo/data2pg
[c] https://wiki.postgresql.org/wiki/Converting_from_other_Databases_to_PostgreSQL#Oracle
[9] https://www.youtube.com/watch?v=CR67iLHTocY

1.7.2 QUESTIONS

- N'hésitez pas, c'est le moment !

1.8 QUIZ

- **https://dali.bo/n2_quiz**

1.9 TRAVAUX PRATIQUES

Création de l'espace de travail

Avec l'utilisateur `postgres`, créer une arborescence de travail destinée à recevoir les fichiers du projet de migration sous `$HOME/tp_migration`.

Configuration

Configurer la connexion Oracle

Vérifier la connexion à la base Oracle avec `sqlplus`.

- IP de l'instance : *demander au formateur*
- Utilisateur : `hr`
- Mot de passe : `phoenix`
- Nom de service : `hr`

Configurer la chaîne de connexion à la base Oracle et faire un test de connexion avec `ora2pg`.

Export/import du schéma

Exporter le schéma HR complet

Exécuter le script permettant l'exécution chaînée de tous les types d'export du schéma et des procédures stockées. Pour ces dernières l'export du code sera fait dans la version source Oracle et dans la version transformée par Ora2Pg avec la syntaxe PostgreSQL.

Import du schéma

Créer la base de données `pghr` sous l'utilisateur `migration`.

Importer uniquement les tables, les autres objets du schéma seront importés après l'import des données.

Export/import des données

Exporter les données

Exporter toutes les données de la base Oracle dans des fichiers.

Importer les données

Importer les données dans la base PostgreSQL.

Finaliser l'import du schéma

Importer les contraintes, indexes, séquences et triggers.

1.10 TRAVAUX PRATIQUES (SOLUTIONS)

Création de l'espace de travail

Pour créer une arborescence de travail destinée à recevoir les fichiers du projet de migration, on peut s'aider d'Ora2Pg en exécutant la commande suivante :

```
ora2pg --init_project tp_migration --project_base $HOME
```

Voici l'arborescence générée par Ora2Pg :

```
tp_migration/
├── config
│   └── ora2pg.conf
├── data
├── export_schema.sh
├── import_all.sh
├── reports
├── 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
```

Ora2Pg a aussi créé un script pour l'export automatique, `export_schema.sh`, un script pour automatiser l'import dans PostgreSQL, `import_all.sh` et un fichier de configuration générique `config/ora2pg.conf`.

Configuration

Configurer la connexion Oracle

> Vérifier la connexion à la base Oracle.

Il est recommandé d'ajouter les variables d'environnements dans le fichier `.bash_profile` de votre utilisateur **postgres**.

```
# ~/.bash_profile
export ORACLE_HOME=/usr/lib/oracle/19.15/client64
export LD_LIBRARY_PATH=$ORACLE_HOME/lib
export PATH=$PATH:$ORACLE_HOME/bin
```

Charger les variables :

```
source $HOME/.bash_profile

# Adapter l'IP au besoin
sqlplus hr/phoenix@192.168.1.109:1521/hr
```

> Configurer la chaîne de connexion à la base Oracle et faire un test de connexion avec Ora2Pg.

Le fichier de configuration à modifier pour définir la chaîne de connexion à la base Oracle est `tp_migration/config/ora2pg.conf`.

Normalement la directive `ORACLE_HOME` doit déjà avoir la valeur du `ORACLE_HOME` de l'installation :

```
ORACLE_HOME     /usr/lib/oracle/19.15/client64
```

Il reste donc à configurer les paramètres de connexion à l'instance `XE` d'Oracle avec l'utilisateur `HR` :

```
ORACLE_DSN      dbi:Oracle://192.168.1.109:1521/hr
ORACLE_USER     hr
ORACLE_PWD      phoenix
```

Dans la mesure où l'utilisateur `hr` n'a pas les privilèges `DBA`, il faut aussi activer la directive `USER_GRANTS` :

```
USER_GRANTS     1
```

On veut exporter le schema `HR`, il faut donc le spécifier dans la configuration et remplacer :

```
SCHEMA  CHANGE_THIS_SCHEMA_NAME
```

par

```
SCHEMA  HR
```

Test de la connexion Ora2Pg vers la base Oracle :

```
$ cd $HOME/tp_migration
$ ora2pg -d -c config/ora2pg.conf -t SHOW_SCHEMA
Ora2Pg version: 22.1
Trying to connect to database: dbi:Oracle://192.168.1.109:1521/hr
Isolation level: SET TRANSACTION ISOLATION LEVEL SERIALIZABLE
Force Oracle to compile schema HR before code extraction
Showing all schema...
SCHEMA HR
```

La connexion est opérationnelle.

Pour lister les tables de l'instance :

```
$ ora2pg -c config/ora2pg.conf -t SHOW_TABLE
[1] TABLE HR.COUNTRIES (owner: HR, 25 rows)
[2] TABLE HR.DEPARTMENTS (owner: HR, 27 rows)
[3] TABLE HR.EMPLOYEES (owner: HR, 107 rows)
[4] TABLE HR.JOBS (owner: HR, 19 rows)
[5] TABLE HR.JOB_HISTORY (owner: HR, 10 rows)
[6] TABLE HR.LOCATIONS (owner: HR, 23 rows)
[7] TABLE HR.REGIONS (owner: HR, 4 rows)
--------------------------------------------------------
Total number of rows: 215

Top 10 of tables sorted by number of rows:
  [1] TABLE HR.EMPLOYEES has 107 rows
  [2] TABLE HR.DEPARTMENTS has 27 rows
  [3] TABLE HR.COUNTRIES has 25 rows
  [4] TABLE HR.LOCATIONS has 23 rows
  [5] TABLE HR.JOBS has 19 rows
  [6] TABLE HR.JOB_HISTORY has 10 rows
  [7] TABLE HR.REGIONS has 4 rows
Top 10 of largest tables:
  [1] TABLE HR.DEPARTMENTS: 0 MB (27 rows)
  [2] TABLE HR.LOCATIONS: 0 MB (23 rows)
  [3] TABLE HR.EMPLOYEES: 0 MB (107 rows)
  [4] TABLE HR.JOB_HISTORY: 0 MB (10 rows)
  [5] TABLE HR.JOBS: 0 MB (19 rows)
  [6] TABLE HR.REGIONS: 0 MB (4 rows)
```

Cette commande affiche aussi le top 10 des tables avec le plus d'enregistrements et, si l'utilisateur de connexion a les droits suffisants, le top 10 des tables de plus gros volume.

Export/import du schéma

Exporter le schéma complet

Il ne reste plus qu'à exécuter le script :

```
$ sh export_schema.sh
```

Voici la listes des commandes exécutées par le script :

```
Running: ora2pg -p -t SEQUENCE -o sequence.sql -b ./schema/sequences
Running: ora2pg -p -t TABLE -o table.sql -b ./schema/tables
Running: ora2pg -p -t PACKAGE -o package.sql -b ./schema/packages
Running: ora2pg -p -t VIEW -o view.sql -b ./schema/views
Running: ora2pg -p -t GRANT -o grant.sql -b ./schema/grants
Running: ora2pg -p -t TRIGGER -o trigger.sql -b ./schema/triggers
Running: ora2pg -p -t FUNCTION -o function.sql -b ./schema/functions
Running: ora2pg -p -t PROCEDURE -o procedure.sql -b ./schema/procedures
Running: ora2pg -p -t TABLESPACE -o tablespace.sql -b ./schema/tablespaces
Running: ora2pg -p -t PARTITION -o partition.sql -b ./schema/partitions
Running: ora2pg -p -t TYPE -o type.sql -b ./schema/types
Running: ora2pg -p -t MVIEW -o mview.sql -b ./schema/mviews
Running: ora2pg -p -t DBLINK -o dblink.sql -b ./schema/dblinks
Running: ora2pg -p -t SYNONYM -o synonym.sql -b ./schema/synonyms
Running: ora2pg -p -t DIRECTORY -o directorie.sql -b ./schema/directories
```

Généralement l'extraction des `GRANT` et `TABLESPACE` génère une erreur si l'utilisateur n'a pas les droits DBA.

Pour obtenir le code source Oracle pour d'éventuelles vérifications :

```
Running: ora2pg -t PACKAGE -o package.sql -b ./sources/packages
Running: ora2pg -t VIEW -o view.sql -b ./sources/views
Running: ora2pg -t TRIGGER -o trigger.sql -b ./sources/triggers
Running: ora2pg -t FUNCTION -o function.sql -b ./sources/functions
Running: ora2pg -t PROCEDURE -o procedure.sql -b ./sources/procedures
Running: ora2pg -t PARTITION -o partition.sql -b ./sources/partitions
Running: ora2pg -t TYPE -o type.sql -b ./sources/types
Running: ora2pg -t MVIEW -o mview.sql -b ./sources/mviews
```

Voici l'arbre des fichiers générés :

```
tp_migration/
├── config
```

```
|   └── ora2pg.conf
├── data
├── export_schema.sh
├── import_all.sh
├── reports
|   ├── columns.txt
|   ├── report.html
|   └── tables.txt
├── schema
|   ├── dblinks
|   ├── directories
|   ├── functions
|   |   ├── EMP_SAL_RANKING_function.sql
|   |   ├── function.sql
|   |   └── LAST_FIRST_NAME_function.sql
|   ├── grants
|   ├── mviews
|   ├── packages
|   |   ├── emp_actions
|   |   |   ├── fire_employee_package.sql
|   |   |   ├── hire_employee_package.sql
|   |   |   ├── num_above_salary_package.sql
|   |   |   └── raise_salary_package.sql
|   |   ├── emp_mgmt
|   |   |   ├── create_dept_package.sql
|   |   |   ├── hire_package.sql
|   |   |   ├── increase_comm_package.sql
|   |   |   ├── increase_sal_package.sql
|   |   |   ├── remove_dept_package.sql
|   |   |   └── remove_emp_package.sql
|   |   ├── global_variables.conf
|   |   └── package.sql
|   ├── partitions
|   ├── procedures
|   |   ├── ADD_JOB_HISTORY_procedure.sql
|   |   ├── procedure.sql
|   |   └── SECURE_DML_procedure.sql
|   ├── sequences
|   |   └── sequence.sql
|   ├── synonyms
|   ├── tables
|   |   ├── CONSTRAINTS_table.sql
|   |   ├── FKEYS_table.sql
|   |   ├── INDEXES_table.sql
|   |   └── table.sql
|   ├── tablespaces
```

```
|   ├── triggers
|   |   ├── trigger.sql
|   |   └── UPDATE_JOB_HISTORY_trigger.sql
|   ├── types
|   └── views
|       ├── EMP_DETAILS_VIEW_view.sql
|       └── view.sql
└── sources
    ├── functions
    |   ├── EMP_SAL_RANKING_function.sql
    |   ├── function.sql
    |   └── LAST_FIRST_NAME_function.sql
    ├── mviews
    ├── packages
    |   ├── emp_actions_package.sql
    |   ├── emp_mgmt_package.sql
    |   └── package.sql
    ├── partitions
    ├── procedures
    |   ├── ADD_JOB_HISTORY_procedure.sql
    |   ├── procedure.sql
    |   └── SECURE_DML_procedure.sql
    ├── triggers
    |   ├── trigger.sql
    |   └── UPDATE_JOB_HISTORY_trigger.sql
    ├── types
    └── views
        ├── EMP_DETAILS_VIEW_view.sql
        └── view.sql
```

Import du schéma

Pour créer la base de données `pghr` sous l'utilisateur `migration`, il faut déjà créer l'utilisateur.

Création du propriétaire de la base :

```
createuser --no-superuser --no-createrole --no-createdb migration
```

On procède ensuite à la création de la base elle-même :

```
createdb -E UTF-8 --owner migration pghr
```

Et on importe les tables et les vues dans la base ; les autres objets seront importés à la fin.

```
psql -U migration pghr -f ./schema/tables/table.sql
psql -U migration pghr -f ./schema/views/view.sql
```

Vérification de la bonne application du script d'import :

```
psql -U migration pghr -c "\d"
```

```
               List of relations
 Schema |       Name       | Type  |   Owner
--------+------------------+-------+-----------
 public | countries        | table | migration
 public | departments      | table | migration
 public | emp_details_view | view  | migration
 public | employees        | table | migration
 public | job_history      | table | migration
 public | jobs             | table | migration
 public | locations        | table | migration
 public | regions          | table | migration
```

Export/import des données

Exporter les données

L'export de toutes les données de la base Oracle se fait en une seule commande :

```
ora2pg -t COPY -o data.sql -b ./data -c ./config/ora2pg.conf
```

```
[========================>] 7/7 tables (100.0%) end of scanning.
[========================>] 25/25 rows (100.0%) Table COUNTRIES (25 recs/sec)
[========================>] 27/27 rows (100.0%) Table DEPARTMENTS (27 recs/sec)
[========================>] 107/107 rows (100.0%) Table EMPLOYEES (107 recs/sec)
[========================>] 19/19 rows (100.0%) Table JOBS (19 recs/sec)
[========================>] 10/10 rows (100.0%) Table JOB_HISTORY (10 recs/sec)
[========================>] 23/23 rows (100.0%) Table LOCATIONS (23 recs/sec)
[========================>] 4/4 rows (100.0%) Table REGIONS (4 recs/sec)
```

Cette commande va générer un fichier par table et un fichier `data.sql` qui pourra être utilisé pour charger les données en une fois.

```
data/
├── COUNTRIES_data.sql
├── data.sql
├── DEPARTMENTS_data.sql
├── EMPLOYEES_data.sql
├── JOB_HISTORY_data.sql
├── JOBS_data.sql
├── LOCATIONS_data.sql
└── REGIONS_data.sql
```

Importer les données

Pour importer les données dans la base PostgreSQL, on peut le faire fichier par fichier mais il est plus simple d'utiliser le fichier de chargement global `data.sql`. Voici son contenu :

```
BEGIN;
ALTER TABLE countries DISABLE TRIGGER USER;
ALTER TABLE departments DISABLE TRIGGER USER;
ALTER TABLE employees DISABLE TRIGGER USER;
ALTER TABLE jobs DISABLE TRIGGER USER;
ALTER TABLE job_history DISABLE TRIGGER USER;
ALTER TABLE locations DISABLE TRIGGER USER;
ALTER TABLE regions DISABLE TRIGGER USER;

\i ./data/COUNTRIES_data.sql
\i ./data/DEPARTMENTS_data.sql
\i ./data/EMPLOYEES_data.sql
\i ./data/JOBS_data.sql
\i ./data/JOB_HISTORY_data.sql
\i ./data/LOCATIONS_data.sql
\i ./data/REGIONS_data.sql

ALTER TABLE countries ENABLE TRIGGER USER;
ALTER TABLE departments ENABLE TRIGGER USER;
ALTER TABLE employees ENABLE TRIGGER USER;
ALTER TABLE jobs ENABLE TRIGGER USER;
ALTER TABLE job_history ENABLE TRIGGER USER;
ALTER TABLE locations ENABLE TRIGGER USER;
ALTER TABLE regions ENABLE TRIGGER USER;
COMMIT;
```

Exécutons le chargement :

```
psql -U migration pghr -f ./data/data.sql -v ON_ERROR_STOP=1
```

Vérification :

```
psql pghr -c "SELECT * FROM countries"
```

```
 country_id |       country_name       | region_id
------------+--------------------------+-----------
 AR         | Argentina                |         2
 AU         | Australia                |         3
 BE         | Belgium                  |         1
 BR         | Brazil                   |         2
 CA         | Canada                   |         2
 CH         | Switzerland              |         1
 CN         | China                    |         3
 DE         | Germany                  |         1
 DK         | Denmark                  |         1
```

```
 EG         | Egypt                    |         4
 FR         | France                   |         1
 HK         | HongKong                 |         3
 IL         | Israel                   |         4
 IN         | India                    |         3
 IT         | Italy                    |         1
 JP         | Japan                    |         3
 KW         | Kuwait                   |         4
 MX         | Mexico                   |         2
 NG         | Nigeria                  |         4
 NL         | Netherlands              |         1
 SG         | Singapore                |         3
 UK         | United Kingdom           |         1
 US         | United States of America |         2
 ZM         | Zambia                   |         4
 ZW         | Zimbabwe                 |         4
(25 lignes)
```

Finaliser l'import du schéma

Importer les contraintes, les index et les séquences.

```
psql -U migration pghr -f schema/tables/CONSTRAINTS_table.sql
psql -U migration pghr -f schema/tables/FKEYS_table.sql
psql -U migration pghr -f schema/tables/INDEXES_table.sql
psql -U migration pghr -f schema/sequences/sequence.sql
```

NOTES

NOTES

NOS AUTRES PUBLICATIONS

FORMATIONS

- **DBA1 : Administration PostgreSQL**
  https://dali.bo/dba1
- **DBA2 : Administration PostgreSQL avancé**
  https://dali.bo/dba2
- **DBA3 : Sauvegarde et réplication avec PostgreSQL**
  https://dali.bo/dba3
- **DEVPG : Développer avec PostgreSQL**
  https://dali.bo/devpg
- **PERF1 : PostgreSQL Performances**
  https://dali.bo/perf1
- **PERF2 : Indexation et SQL avancés**
  https://dali.bo/perf2
- **MIGORPG : Migrer d'Oracle à PostgreSQL**
  https://dali.bo/migorpg
- **HAPAT : Haute disponibilité avec PostgreSQL**
  https://dali.bo/hapat

LIVRES BLANCS

- **Migrer d'Oracle à PostgreSQL**
- **Industrialiser PostgreSQL**
- **Bonnes pratiques de modélisation avec PostgreSQL**
- **Bonnes pratiques de développement avec PostgreSQL**

TÉLÉCHARGEMENT GRATUIT

Les versions électroniques de nos publications sont disponibles gratuitement sous licence open-source ou sous licence Creative Commons. Contactez-nous à l'adresse contact@dalibo.com pour plus d'information.

DALIBO, L'EXPERTISE POSTGRESQL

Depuis 2005, DALIBO met à la disposition de ses clients son savoir-faire dans le domaine des bases de données et propose des services de conseil, de formation et de support aux entreprises et aux institutionnels.

En parallèle de son activité commerciale, DALIBO contribue aux développements de la communauté PostgreSQL et participe activement à l'animation de la communauté francophone de PostgreSQL. La société est également à l'origine de nombreux outils libres de supervision, de migration, de sauvegarde et d'optimisation.

Le succès de PostgreSQL démontre que la transparence, l'ouverture et l'auto-gestion sont à la fois une source d'innovation et un gage de pérennité. DALIBO a intégré ces principes dans son ADN en optant pour le statut de SCOP : la société est contrôlée à 100 % par ses salariés, les décisions sont prises collectivement et les bénéfices sont partagés à parts égales.