Module N4

Procédures stockées

22.09

Dalibo SCOP

https://dalibo.com/formations

Procédures stockées

Module N4

TITRE : Procédures stockées
SOUS-TITRE : Module N4

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 PROCÉDURES STOCKÉES

1.1 INTRODUCTION

Oracle et PostgreSQL n'ont pas le même langage PL :

- Oracle : PL/SQL et Java
- PostgreSQL : PL/pgSQL, PL/Java, PL/Perl, PL/Python, PL/R…

Les langages de routines stockées sont différents entre Oracle et PostgreSQL. Même si PL/pgSQL est un langage assez proche de PL/SQL, cela demandera une revue des routines stockées et une réécriture (automatique ou manuelle) des routines.

1.1.1 SOMMAIRE

Ce module est organisé en cinq parties :

- Outils et méthodes
- Différences dans le code
- Conversion automatique
- Migration des procédures stockées
- Tests et validation

C'est la partie la plus importante en terme de complexité et de temps dans la migration : la conversion du code PL/SQL en code PL/pgSQL.

Ce module vous donnera les outils et la méthode pour réussir la migration de ce code. Il vous expliquera aussi les différences de syntaxe entre ces deux langages avec des exemples de cas concrets.

Ce module aborde aussi la conversion automatique du code avec Ora2Pg et détaille la façon d'importer ce code dans PostgreSQL avant d'aborder la phase de test et de validation du code.

1.2 OUTILS ET MÉTHODES

- Outils d'émulation de fonctionnalités Oracle
- Outils de conversion de code PL/SQL vers PL/pgSQL
- Outils de débogage du code PL/pgSQL

Cette partie indique les différents outils offrant une aide à la migration du code PL/SQL vers le PL/pgSQL.

- Certains outils ont fait le choix d'implémenter certaines fonctionnalités absentes ;
- D'autres ont choisi de s'affranchir complètement de la syntaxe Oracle ;
- Lors de la phase de tests, des outils de débogage peuvent s'avérer nécessaires.

1.2.1 LES OUTILS D'ÉMULATION

- Orafce :
  - nombreuses fonctions de compatibilité Oracle
  - `to_char(1 param)`, `add_month()`, `decode()`...
  - `DBMS_ALERT`, `DBMS_PIPE`, `DBMS_OUTPUT`, `DBMS_RANDOM` et `UTL_FILE`
- Migration Tool Kit :
  - réservé à EDB PostgreSQL Plus Advanced Server Migration
  - ne convertit pas le code PL/SQL

Librairie Orafce

Pour accélérer la phase de réécriture du code PL/SQL vers PL/pgSQL, il existe une bibliothèque de compatibilité nommée Orafce[2] . Cette bibliothèque libre sous licence BSD est développée par Pavel Stehule et émule le comportement de bon nombre de fonctions et modules Oracle sous PostgreSQL.

Fonctions relatives aux dates

- `add_months(date, integer)`
- `last_day(date)`
- `next_day(date, text)`
- `next_day(date, integer)`
- `months_between(date, date)`
- `trunc(date, text)`
- `round(date, text)`

Emulation de la table DUAL

[2] https://github.com/orafce/orafce

Inutile sous PostgreSQL, il suffit d'enlever la clause `FROM DUAL` de toutes les requêtes l'utilisant.

Module dbms_output

Habituellement, PostgreSQL utilise `RAISE NOTICE` pour retourner les informations aux clients. La fonction Oracle `dbms_output.put_line()` a le même but mais ce module Oracle permet en plus de gérer une file d'attente des messages.

Ce module contient les fonctions suivantes :

- `enable()`
- `disable()`
- `serveroutput()`
- `put()`
- `put_line()`
- `new_line()`
- `get_line()`
- `get_lines()`

Module utl_file

Ce module permet de lire et d'écrire dans n'importe quel fichier accessible depuis le serveur à partir du code PL/pgSQL. Ce module contient les fonctions suivantes :

- `utl_file.fclose()`
- `utl_file.fclose_all()`
- `utl_file.fcopy()`
- `utl_file.fflush()`
- `utl_file.fgetattr()`
- `utl_file.fopen()`
- `utl_file.fremove()`
- `utl_file.frename()`
- `utl_file.get_line()`
- `utl_file.get_nextline()`
- `utl_file.is_open()`
- `utl_file.new_line()`
- `utl_file.put()`
- `utl_file.put_line()`
- `utl_file.putf()`
- `utl_file.tmpdir()`

Module dbms_pipe

Ce module permet la communication entre session. Il est l'équivalent du module de même nom sous Oracle.

Module dbms_alert

Ce module permet aussi la communication entre sessions.

Modules PLVdate, PLVstr, PLVchr, PLVsubst et PLVlex

Ces modules implémentent la plupart des fonctions définies dans le module PL/Vision d'Oracle.

Module dbms_assert et PLUnit

Ce module fournit des fonctions permettant de protéger les utilisateurs contre des injections SQL.

Autres fonctions

Orafce permet aussi l'utilisation de certaines fonctions disponibles sous Oracle :

- `concat()`
- `nvl()`
- `nvl2()`
- `lnnvl()`
- `decode()`
- `bitand()`
- `nanvl()`
- `sinh()`
- `cosh()`
- `tanh()`
- `substr()`

Migration Tool Kit

Cet ensemble d'outils de migration est un module propriétaire développé par la société Enterprise DB et destiné à être mis en œuvre uniquement avec la version propriétaire du serveur PostgreSQL Plus.

Il n'y a pas de conversion de code PL/SQL en PL/pgSQL. La solution tend à implémenter dans le moteur propriétaire du serveur PostgreSQL Plus les types et fonctionnalités existantes dans Oracle. La bibliothèque Orafce y est d'ailleurs intégrée.

1.2.2 LES OUTILS DE CONVERSION

- Ora2pg
  - convertisseur de code PL/SQL en PL/pgSQL sous licence GPL
  - seul outil libre

Ora2Pg[3] est le seul outil libre permettant une migration de la majorité du code PL/SQL. Couplé à Orafce, il permet de limiter considérablement la retouche du code PL/SQL pour son portage sous PostgreSQL.

1.2.3 LES OUTILS DE DÉBOGAGE

- pldebugger (ex edb-debugger)
- plpgsql_check
- SQLMaestro

Pour aider lors de la phase de test, vous pouvez utiliser le débogueur PL/SQL d'EDB qui vous indiquera à quelle ligne du code se trouve le problème et `plpgsql_check`, une extension pour les versions de PostgreSQL 9.5 et supérieures permettant de signaler des problèmes de syntaxe PL/pgSQL. Ce validateur de code SQL embarqué vous alerte si vous faites référence à des tables, colonnes ou variables inexistantes.

- `pldebugger` (anciennement edb-debugger) peut être téléchargé sur github[4] ;
- `plpgsql_check` de Pavel Stehule peut être téléchargé ici[5] .

Ces deux extensions sont des contributions en langage `c` et doivent être compilées. Si vous utilisez `pgAdmin`, `edb-debugger` est directement intégré dans la distribution.

Il existe aussi SQLMaestro[6] , un outil propriétaire qui permet l'exécution pas à pas du code PL/pgSQL.

---

[3] https://ora2pg.darold.net/
[4] https://github.com/EnterpriseDB/pldebugger
[5] https://github.com/okbob/plpgsql_check
[6] https://www.sqlmaestro.com/products/postgresql/maestro/tour/pgsql_debugger/

1.3 DIFFÉRENCES DANS LE CODE

- Généralité
- Triggers
- Routines
- Packages

Cette partie dresse une liste exhaustive des différences majeures entre Oracle et PostgreSQL en matière de code procédural embarqué.

1.3.1 GÉNÉRALITÉ - 1

- `nom_sequence.nextval` => `nextval('nom_sequence')`
- Pas de transaction autonome à moins de passer par `dblink` ou `pg_background`
- `RETURN` => `RETURNS`
- `EXECUTE IMMEDIATE` => `EXECUTE`
- `SELECT` sans `INTO` => `PERFORM`

Les séquences

L'appel aux fonctions des séquences se fait de manière différente même si les noms de fonctions sont identiques. Avec Oracle, l'appel se fait avec `nom_sequence.nom_fonction` alors qu'avec PostgreSQL, l'appel se fait en donnant le nom de la séquence en paramètre de la fonction `nom_fonction('nom_sequence')`.

Transactions autonomes

Les transactions autonomes définies par `PRAGMA AUTONOMOUS_TRANSACTION` dans Oracle n'ont pas d'équivalent sous PostgreSQL. Pour émuler cette fonctionnalité, il faut utiliser une autre connexion à la base de données, par exemple avec l'extensions `dblink`, ou `pg_background` depuis PostgreSQL 9.5.

Article Dalibo : Support des transactions autonomes dans PostgreSQL[7]

Différences de syntaxe

Il y a aussi des différences d'écriture. Dans les déclarations de fonction, `RETURN` prends un `S`. `EXECUTE` l'est toujours immédiatement, le mot clé `IMMEDIATE` n'existe donc pas.

Dans une fonction, les `SELECT` non affectés à une variable (sans `INTO`) doivent être remplacés par `PERFORM`. C'est exactement la même syntaxe qu'un `SELECT` normal, c'est simplement le mot `SELECT` qui est remplacé par `PERFORM`.

[7] https://blog.dalibo.com/2016/08/19/Support_des_transactions_autonomes_dans_PostgreSQL.html

1.3.2 GÉNÉRALITÉ - 2

- `REVERSE LOOP` => inversion des bornes
- Une fonction doit avoir un langage
- `CONNECT BY` n'existe pas, utiliser `WITH RECURSIVE`
- `REF CURSOR` doit être remplacé par `REFCURSOR`
- `nom_curseur%ROWTYPE` doit être remplacé par `RECORD`
- `BULK COLLECT` => Array
- Les chaînes vides sont équivalentes à NULL sous Oracle

Boucle inversée

Dans les ordres `REVERSE LOOP`, les bornes minimales et maximales doivent être inversées sous PostgreSQL, car cela indique qu'à chaque pas la valeur sera décrémentée et non incrémentée.

Sous Oracle, on écrit :

```
FOR v IN REVERSE min .. max LOOP
```

et avec PostgreSQL, on écrira :

```
FOR v IN REVERSE max .. min LOOP
```

Langage d'une fonction

Une fonction doit impérativement déclarer le langage qu'elle utilise (SQL, PL/pgSQL, C, PL/Perl, etc.) :

```
CREATE FUNCTION add(integer, integer) RETURNS integer
AS $$
    select $1 + $2;
$$
LANGUAGE SQL
IMMUTABLE
RETURNS NULL ON NULL INPUT;

CREATE FUNCTION perl_max (integer, integer) RETURNS integer
AS $$
    if ($_[0] > $_[1]) { return $_[0]; }
    return $_[1];
$$
LANGUAGE plperl;
```

CONNECT BY

L'instruction `CONNECT BY` n'existe pas sous PostgreSQL. Il faudra réécrire entièrement la requête à l'aide d'une requête récursive. Par exemple, soit une table définie comme suit :

```
create table books (
   author_id int not null,
   id int not null,
   parent_id int,
   title varchar2(50)
);
```

Voici une requête `CONNECT BY` Oracle :

```
 SELECT author_id, id, title
    FROM books
   WHERE author_id = 2
   START WITH id = 1
   CONNECT BY PRIOR id = parent_id;
```

et voici sa traduction pour PostgreSQL :

```
WITH RECURSIVE recurs_query (author_id, id, title) AS (
  SELECT author_id, id, title
    FROM books
   WHERE id = 1
   UNION ALL
  SELECT tn.author_id, tn.id, tn.title
    FROM recurs_query tp, books tn
   WHERE tp.id = tn.parent_id
)
SELECT author_id, id, title
  FROM recurs_query
 WHERE author_id = 2;
```

Les curseurs

Au niveau des curseurs, leurs références est de type `REFCURSOR` au lieu de `REF CURSOR`.

Par exemple la déclaration d'une référence sur un curseur se fait de la façon suivante sous Oracle :

```
TYPE return_cur IS REF CURSOR RETURN ma_table%ROWTYPE;
p_retcur return_cur;
```

Alors que sous PostgreSQL, cela s'écrit de la sorte :

```
return_cur REFCURSOR;
```

Le type retourné lors de la manipulation des curseurs est un enregistrement `RECORD` et non pas `nom_curseur%ROWTYPE` sous Oracle. Avec PostgreSQL, il est possible à la lecture

du curseur de placer cet enregistrement dans une cible qui peut être une variable ligne, une variable record ou une liste de variables simples séparées par des virgules.

BULK COLLECT

La notion de `BULK COLLECT` n'existe pas sous PostgreSQL. En fait, il s'agit de charger dans un tableau le résultat d'une requête et de parcourir ensuite ce tableau. Par exemple, ce code Oracle

```
CREATE PROCEDURE tousLesAuteurs
IS
    TYPE my_array IS varray(100) OF varchar(25);
    temp_arr my_array;
BEGIN
    SELECT nom BULK COLLECT INTO temp_arr FROM auteurs ORDER BY nom;
    FOR i IN temp_arr.first .. temp_arr.last LOOP
        DBMS_OUTPUT.put_line(i || ') nom: ' || temp_arr..(i));
    END LOOP;
END tousLesAuteurs;
```

peut être traduit sous PostgreSQL de la façon suivante :

```
CREATE PROCEDURE tousLesAuteurs()
AS $$
DECLARE
    temp_arr varchar(25)[];
BEGIN
    temp_arr := (SELECT nom FROM auteurs ORDER BY nom);
    FOR i IN array_lower(temp_arr,1) .. array_upper(temp_arr,1) LOOP
        RAISE NOTICE '% ) nom: %', i,  temp_arr(i);
    END LOOP;
END;
$$ LANGUAGE plpgsql;
```

Chaines vides et NULL

Oracle traite les chaînes vides comme NULL, c'est-à-dire qu'il ne fait pas la différence entre `NULL` et `''`.

La requête suivante sur Oracle renvoie vrai si le champ `visa` n'est pas NULL mais est vide.

```
SELECT * FROM passeports WHERE visa IS NULL;
```

Ce comportement n'est absolument pas standard et est dangereux. Il faut vraiment faire attention à ces parties de code qui, lors de la migration, peuvent provoquer des comportements aberrants de l'application.

1.3.3 TRIGGERS

- Ils doivent être séparés en fonction et trigger
- `:NEW` et `:OLD` => `NEW` et `OLD`
- `UPDATING`, `INSERTING`, `DELETING` => `TG_OP (UPDATE, INSERT, DELETE)`
- `RETURN NEW` impératif dans les triggers `BEFORE`, retour implicite sous Oracle

Les triggers sous PostgreSQL font obligatoirement appel à une fonction. Il y a donc systématiquement une déclaration de fonction et une déclaration de trigger.

```
CREATE OR REPLACE FUNCTION log_account_update() RETURNS trigger AS
...code ici...
LANGUAGE 'plpgsql';

CREATE TRIGGER log_update
    AFTER UPDATE ON accounts
    FOR EACH ROW
    WHEN (OLD.* IS DISTINCT FROM NEW.*)
    EXECUTE PROCEDURE log_account_update();
```

Les enregistrements `OLD` et `NEW` ne sont pas préfixés par le caractère `:`.

Les événements `UPDATING`, `INSERTING`, `DELETING` correspondent à la valeur de la variable `TG_OP`, qui peut valoir `UPDATE`, `INSERT` et `DELETE`.

Avec PostgreSQL, vous devez retourner les enregistrements dans les triggers avant action. Dans le cas contraire, NULL est retourné, au contraire d'Oracle pour lequel le retour est implicite. Par exemple :

```
CREATE FUNCTION gen_id () RETURNS TRIGGER AS
$$
DECLARE
    noitem integer;
BEGIN
    select max(no_produit) into noitem from produit;
    IF noitem ISNULL THEN
      noitem:=0;
    END IF;
    NEW.no_produit:=noitem+1;
    RETURN NEW;
  END;
$$
LANGUAGE 'plpgsql';

CREATE TRIGGER trig_before_ins_produit BEFORE INSERT ON produit
  FOR EACH ROW
  EXECUTE PROCEDURE gen_id();
```

Sous Oracle, nous aurions cela :

```
CREATE TRIGGER gen_id FOR produit
  BEFORE INSERT
  DECLARE noitem integer;
As
BEGIN
  select max(no_produit) into noitem from produit;
  NEW.no_produit := noitem+1;
END;
```

1.3.4 ROUTINES

- PostgreSQL supporte les fonctions et procédures stockées
  - uniquement les fonctions pour les versions 10 ou inférieures
- Il doit toujours y voir des parenthèses pour la liste des paramètres, même si elle est vide
- Les valeurs par défaut sont aussi autorisées
- PostgreSQL peut retourner un pseudo type `RECORD`, correspondant à un enregistrement
  - sous Oracle, il faut soit utiliser une référence de curseur soit définir une `TABLE FUNCTION`

Les routines PostgreSQL englobent les fonctions et les procédures. Ces dernières sont apparues avec la version 11. Avant cela, une procédure n'était ni plus ni moins qu'une fonction qui retourne `VOID`.

Avec Oracle, il est possible d'omettre les parenthèses de la section de déclaration des paramètres. Avec PostgreSQL, ces parenthèses sont obligatoires.

```
CREATE FUNCTION ma_fct () RETURNS integer AS ...
CREATE PROCEDURE ma_proc () AS ...
```

Les valeurs par défaut, les notations nommées et positionnées sont aussi supportées avec les routines PostgreSQL. Par exemple :

```
CREATE OR REPLACE PROCEDURE hello_world(
  t1 text = 'hello',
  t2 text = 'world') AS $$
BEGIN
  raise warning '% %', t1, t2;
END
$$ LANGUAGE 'plpgsql';
```

```
CALL hello_world();
-- WARNING:  hello world

CALL hello_world(t2 => 'dalibo');
-- WARNING:  hello dalibo
```

Pour retourner un jeu d'enregistrements depuis une procédure stockée sous Oracle, c'est un peu complexe. Il faut soit utiliser une référence de curseur soit définir une `TABLE FUNCTION`. Avec PostgreSQL, il suffit de retourner le pseudo-type `RECORD`. Par exemple :

```
CREATE FUNCTION getRows(text) RETURNS SETOF RECORD
AS $$
DECLARE
    r RECORD;
BEGIN
    FOR r IN EXECUTE 'select * from ' || $1 LOOP
        RETURN NEXT r;
    END LOOP;
    RETURN;
END
$$
LANGUAGE 'plpgsql';
```

1.3.5 PACKAGES

- Paquet de variables et de procédures stockées
  - pas d'équivalent sous PostgreSQL
- Utilisation d'un schéma pour émuler les appels aux fonctions
  - nom_paquet.nom_fonction
- Variables globales non supportées
  - utiliser des tables ou des variables `custom`
- Les définitions de fonctions à l'intérieur du code d'une fonction ne sont pas supportées

Les « packages » ou paquets de procédures stockées sous Oracle permettent de grouper la définition de variables, fonctions et procédures. Il n'existe pas d'équivalent sous PostgreSQL.

Pour ne pas avoir à réécrire tous les appels vers les routines de ces paquets (`nom_paquet.nom_routine`), la solution est de créer sous PostgreSQL un schéma portant le même nom que le paquet. L'appel aux routines se fera alors de façon identique :

```
CALL nom_schema.nom_routine(...);
```

De même, la notion de variable globale n'existe pas sous PostgreSQL. Pour pouvoir émuler le comportement des variables globales, on peut utiliser les variables utilisateurs définies dans le fichier de configuration `postgresql.conf`.

Par exemple :

```
nom_paquet.ma_variable = '12'
```

ou être utilisée sans déclaration préalable dans le fichier de configuration comme suit.

Par exemple, pour créer une variable globale nommée `id_region`, il suffit d'utiliser la commande `SET` :

```
SET nom_paquet.ma_variable = '13';
```

ou la fonction `current_setting()` :

```
select set_config('nom_paquet.ma_variable', '13', false);
```

et pour utiliser sa valeur :

```
SELECT current_setting('nom_paquet.ma_variable') AS ma_variable;
```

Il est aussi possible d'utiliser une table pour définir ces variables et leurs valeurs.

Oracle permet de définir des fonctions à l'intérieur d'autres fonctions, PostgreSQL ne le permet pas. Elles devront être extraites du corps de leur fonction parente et déclarées comme les autres fonctions.

Certains langages, comme PL/Perl par exemple, disposent quant à eux, de variables globales.

1.4 CONVERSION AUTOMATIQUE DU CODE

- Paquets de procédure stockées
- En-têtes et paramètres des triggers, fonctions etc.
- Types des données
- Fonctions
- Modification de syntaxe

L'une des fonctionnalités les plus puissantes d'Ora2Pg est sa conversion automatique du code Oracle PL/SQL en code PL/pgSQL pour PostgreSQL. Même s'il y a eu beaucoup d'effort de développement au niveau de PostgreSQL pour faciliter la compatibilité avec Oracle, il reste certaines parties qui nécessitent une réécriture :

- les paquets (*packages*) de procédures stockées n'existent pas ;

- les en-têtes de fonctions ou de triggers et le passage de paramètres sont différents ;
- les déclarations de variables utilisent des types de données différents ;
- certaines fonctions n'existent pas mais ont un équivalent ;
- la syntaxe n'est pas la même sur beaucoup de points.

Cette partie va s'appliquer à décrire succinctement l'ensemble des conversions automatiques réalisées par Ora2Pg.

1.4.1 CONVERSIONS GLOBALES

- Les `PACKAGES` ou paquets de procédures stockées
- Les déclarations de triggers et routines
- Les paramètres des routines
- La conversion des types de variable

Les paquets de procédures stockées n'existent pas sous PostgreSQL. Pour éviter la réécriture complète des appels à ces routines, Ora2Pg crée un schéma portant le même nom que le paquet, permettant ainsi de convertir implicitement les appels à `PACKAGE.FONCTION` en `SCHEMA.FONCTION`.

L'autre apport d'Ora2Pg permettant de gagner beaucoup de temps dans le portage de code est la transformation des déclarations de triggers et routines de la syntaxe Oracle à la syntaxe PostgreSQL.

Pour les triggers par exemple, sous Oracle, ils sont déclarés de la façon suivante :

```
CREATE TRIGGER trigger_name
    BEFORE
    DELETE OR INSERT OR UPDATE
    ON table_name
       -- pl/sql block
```

alors que, sous PostgreSQL, le code PL/pgSQL doit être dans une fonction. Ora2Pg le convertira alors de la sorte :

```
CREATE OR REPLACE FUNCTION trigger_fct_trigger_name () RETURNS trigger AS
$BODY$
    DECLARE
    BEGIN
        -- plpgsql block
    END;
$BODY$
  LANGUAGE 'plpgsql';
```

```
CREATE TRIGGER trigger_name
    BEFORE
    DELETE OR INSERT OR UPDATE
    ON table_name
    FOR EACH ROW
        EXECUTE PROCEDURE trigger_fct_trigger_name ();
```

Pour les routines, les en-têtes sont entièrement réécrites. Par exemple :

```
CREATE FUNCTION simple_fct RETURN VARCHAR2 IS
BEGIN
    RETURN 'Simple Function';
END simple_fct;
```

deviendra :

```
CREATE OR REPLACE FUNCTION simple () RETURNS varchar AS $body$
BEGIN
    RETURN 'Simple Function';
END simple;
$body$
LANGUAGE PLPGSQL;
```

Pour les fonctions, les choses se compliquent avec le passage de paramètres. Là encore, Ora2Pg fait automatiquement la conversion. Par exemple, avec le code Oracle :

```
CREATE FUNCTION simple2_fct (string_in IN VARCHAR2 := 'No entry')
RETURN VARCHAR2 IS
BEGIN
    RETURN string_in;
END simple2_fct;
```

on obtient :

```
CREATE OR REPLACE FUNCTION simple2_fct (string_in IN text DEFAULT 'No entry')
RETURNS varchar AS
$body$
BEGIN
    RETURN string_in;
END simple2_fct;
$body$
LANGUAGE PLPGSQL;
```

Comme pour les paramètres de fonctions, les types de toutes les variables déclarées dans une routine sont automatiquement convertis dans leurs correspondances sous PostgreSQL et déplacés dans une section `DECLARE`.

Par exemple :

```
CREATE PROCEDURE load_file (pdname VARCHAR2, psname VARCHAR2, pfname VARCHAR2)
IS
    src_file BFILE;
    dst_file BLOB;
    lgh_file BINARY_INTEGER;
BEGIN
    -- pl/sql block
END load_file;
```

sera converti de la sorte dans PostgreSQL :

```
CREATE OR REPLACE PROCEDURE load_file (pdname text, psname text, pfname text)
AS $body$
DECLARE
    src_file bytea;
    dst_file bytea;
    lgh_file integer;
BEGIN
    -- plpgsql block
END
$body$
LANGUAGE PLPGSQL;
```

1.4.2 CORRESPONDANCE DES FONCTIONS - 1

Les noms diffèrent :

- `NVL()` => `coalesce()`
- `SYSDATE` => `LOCALTIMESTAMP`
  - équivalent de `CURRENT_TIMESTAMP` sans le fuseau horaire
- `NLSSORT(colname, 'nls_sort=GERMAN')` => `colname COLLATE "de_DE"`

Renommage des fonctions par Ora2Pg

Ora2Pg remplace les fonctions exclusives à Oracle qui ont un équivalent direct dans PostgreSQL. C'est le cas des fonctions usuelles comme `NVL` qui sera remplacée par `coalesce`, ou `SYSDATE` par `LOCALTIMESTAMP`.

1.4.3 CORRESPONDANCE DES FONCTIONS - 2

Les paramètres changent :

- `to_number(num)`
  - => `to_number(num, '99...99D99...99')`
- `to_date( string1, format_mask,  nls_language)`
  - => `to_date(text, text)`
- `replace(a, b)`
  - => `replace(a, b, ' ')`

Remplacement des paramètres par Ora2Pg

Si certaines fonctions Oracle peuvent être remplacées directement par leur équivalent sous PostgreSQL, d'autres doivent être réécrites.

Il suffit parfois simplement de modifier les paramètres. C'est le cas pour :

- `TO_NUMBER(num)` et `to_number(num, '999999999999999999999D9999999999999999999')` où PostgreSQL nécessite un second paramètre pour préciser le format.
- `TO_DATE(string1, format_mask,  nls_language)` et `to_date(text, text)` où le troisième paramètre n'existe pas sous PostgreSQL.
- `REPLACE(string, pattern)` et `REPLACE(string, pattern, ' ')`, PostgreSQL nécessite la présence du troisième paramètre même si la chaîne de substitution est vide.

1.4.4 CORRESPONDANCE DES FONCTIONS - 3

Les noms et les paramètres changent :

```
trunc(.*date.*)
 => date_trunc('day', ...date...)
substr( string, start_position, length )
 => substring(string from start_position for length)
```

Réécriture complète des fonctions et paramètres par Ora2Pg

Il est aussi possible que le nom de la fonction et les paramètres doivent être réécrits :

- `TRUNC(.*date.*)` et `date_trunc('day', ...)`, cas particulier d'un `TRUNC` sans format sur un champ avec le mot `date` dans le nom, laissant supposer qu'il s'agit d'un champ de type date.
- `SUBSTR( champ_text, 1, 255)` sera réécrit de la façon suivante : `substring(champ_text from 1 for 255)`

1.4.5 CORRESPONDANCE DES FONCTIONS - 4

La réécriture est complète :

```
add_months
 => ”+ 'N months'::interval”
add_years
 => ”+ 'N year'::interval”
TO_NUMBER(TO_CHAR(…))
 => to_char(…)::integer
decode("user_status",'active',"username",null)
 => (CASE WHEN user_status='active' THEN username ELSE NULL END)
```

Autres astuces employées par Ora2Pg

Il y a aussi les fonctions qui n'ont pas d'équivalent direct mais peuvent être écrites autrement :

- `ADD_MONTH(champ_date, 3)` est reformulée en utilisant l'ajout d'un interval : `champ_date + '3 months'::interval`
- `ADD_YEAR(champ_date, -5)` est remplacée par l'ajout d'un interval : `champ_date - '5 years'::interval`
- `TO_NUMBER(TO_CHAR(…))` nécessiterait l'emploi d'un format, mais plus simplement réécrite avec un cast : `to_char(…)::integer`
- `DECODE("user_status",'active',"username",null)`... cette fonction n'existe pas et sa réécriture est plus complexe :

```
(CASE WHEN user_status='active' THEN username ELSE NULL END)
```

1.4.6 RÉÉCRITURE DE PARTIES DE CODE - 1

- Réécrit les appels aux séquences
  - `nom.nextval` => `nextval('nom')`
  - `nom.currval` => `currval('nom')`
- Remplace les appels `:new.` en `NEW.` et `:old.` en `OLD.` dans les triggers
- Remplace `INSERTING|DELETING|UPDATING` en `TG_OP='INSERT|DELETE|UPDATE'` dans les fonctions de trigger

1.4.7 RÉÉCRITURE DE PARTIES DE CODE - 2

- Supprime le caractère `:` devant les nom de variable Oracle
- Remplace les sorties Oracle `DBMS_OUTPUT.(put_line|put|new_line)(...)` en `RAISE NOTICE '...'`
- Inversement des bornes min et max dans les boucles `FOR … IN … REVERSE min .. max`
- Réécrit les `RAISE EXCEPTION`avec concaténation `||` par le format à la `sprintf` utilisé par PostgreSQL

Au-delà de la réécriture des fonctions, il est parfois nécessaire de restructurer et modifier le code lui-même.

1.4.8 RÉÉCRITURE DE PARTIES DE CODE - 3

- Remplacement des `ROWNUM` dans la clause where par des clauses `LIMIT` et/ou `OFFSET`
- Réécrit la clause `HAVING … GROUP BY` (variante acceptée par Oracle mais pas PostgreSQL) en `GROUP BY … HAVING`
- Remplace les appels à `MINUS` par `EXCEPT`

Oracle utilise la notation suivante pour limiter le nombre d'enregistrement retournés :

```
SELECT * FROM table WHERE ROWNUM <= 10;
```

Avec PostgreSQL, la notation équivalent est la suivante :

```
SELECT * FROM table LIMIT 10;
```

Ces notations sont presque équivalentes, à la différence près qu'Oracle opère les tris `ORDER BY` après la limitation du nombre de ligne. Dans l'exemple précédent le tri se fera sur les 10 lignes retournées, alors que coté PostgreSQL, le tri est opéré avant.

Il faut donc faire très attention au résultat attendu, pour avoir le même résultat sous Oracle que le LIMIT, il faudrait utiliser la requête suivante :

```
SELECT * FROM (SELECT * FROM A ORDER BY id) WHERE
ROWNUM <= 10;
```

Ora2Pg va remplacer automatiquement les ROWNUM de la clause WHERE avec LIMIT :

- ROWNUM < ou <= N sont réécrit en LIMIT N
- ROWNUM = N est réécrit en LIMIT 1 OFFSET N
- ROWNUM > or >= N sont réécrit en LIMIT ALL OFFSET N

La conversion des ROWNUM utilisés pour énumérer les lignes dans les requêtes n'est pas couverte par Ora2Pg, par exemple :

```
SELECT * FROM (
    SELECT  t.*, ROWNUM AS rn
    FROM mytable t
    ORDER BY paginator, id
)
WHERE rn BETWEEN :start AND :end
```

devra être réécrit manuellement en fonction fenêtrée (`Window Function`) et l'utilisation de `ROW_NUMBER()` :

```
SELECT * FROM (
    SELECT  t.*, ROW_NUMBER() OVER (ORDER BY paginator, id) AS rn
    FROM    mytable t
)
WHERE rn BETWEEN :start AND :end
```

1.4.9 RÉÉCRITURE DE PARTIES DE CODE - 4

- Supprime les appels à `FROM DUAL`
- Supprime les `DEFAULT NULL` qui est la valeur par défaut sous PostgreSQL lorsqu'aucune valeur par défaut n'est précisée
- Suppression des noms d'objets répétés après les END, exemple : `END fct_name;` est réécrit en `END;`

1.4.10 RÉÉCRITURE DE PARTIES DE CODE - 5

- Déplacement des commentaires dans les `CASE` entre le `WHEN` et le `THEN`, non supporté par PostgreSQL
- Remplacement des conditions `IS NULL` et `IS NOT NULL` par des instructions à base de `coalesce` (pour Oracle, une chaîne vide est équivalente à NULL)
- Inverse les déclarations de curseur `CURSOR moncurseur;` pour les rendre compatibles avec PostgreSQL : `moncurseur CURSOR;`

Empty string vs NULL

Une chaîne vide est égale à `NULL` dans Oracle :

```
'' = NULL
```

Dans PostgreSQL et dans le SQL standard :

```
'' <> NULL
```

Du coup l'insertion d'une chaîne vide dans un champ avec une contrainte NOT NULL va remonter une exception sous Oracle, mais pas dans PostgreSQL :

```
CREATE TABLE tempt (
    id NUMBER NOT NULL,
    descr VARCHAR2(255) NOT NULL
);
INSERT INTO temp_table (id, descr) VALUES (2, '');
-- ORA-01400: cannot insert NULL into ("HR"."TEMPT"."DESCR")
```

Si la directive `NULL_EQUAL_EMPTY` est activée, Ora2Pg remplace toutes les conditions avec un test sur `NULL` par une appel à la fonction `coalesce()`.

```
(field1 IS NULL)
```

est remplacé par

```
(coalesce(field1::text, '') = '')
```

et

```
(field2 IS NOT NULL)
```

est remplacé par

```
(field2 IS NOT NULL AND field2::text <> '')
```

Le remplacement est réalisé par défaut pour être sur que vous aurez le même comportement. Ce mécanisme a ses limites car il n'est pas possible d'insérer une chaîne vide dans un champ numérique. La substitution n'est donc pas nécessaire, mais Ora2Pg ne sait pas le détecter. De même si vous êtes assuré de ne pas avoir ce genre de problème alors le remplacement des tests n'est pas nécessaire.

Pour désactiver ce fonctionnement d'Ora2Pg, positionner `NULL_EQUAL_EMPTY` à 0.

1.4.11 RÉÉCRITURE DE PARTIES DE CODE - 6

- Supprime le mot clé `IN` de la déclaration des curseurs.
- Remplacement des sorties de curseur `EXIT WHEN ...%NOTFOUND` par `IF NOT FOUND THEN EXIT; END IF;`
- Ajout du mot clé `STRICT` aux `SELECT … INTO` lorsqu'il y a `EXCEPTION … NO_DATA_FOUND` ou `TOO_MANY_ROWS`

1.4.12 RÉÉCRITURE DE PARTIES DE CODE - 7

- Remplacement des `REGEX_LIKE( string, pattern )` en syntaxe avec l'opérateur PostgreSQL de recherche regex `string ~ pattern`.
- Remplacement des appels aux variables d'environnement `SYS_CONTECT('USERENV', ...)` en équivalent PostgreSQL.
- Remplacement des fonctions spatiales `SDO_GEOM.*` en appels aux fonction PostGis équivalentes.
- Remplacement des opérateurs géométriques `SDO_*` en opérateurs correspondants PostGis.

1.4.13 REMPLACEMENT CONCERNANT LES EXCEPTIONS

Remplacement de :

- `STORAGE_ERROR` par `OUT_OF_MEMORY`
- `ZERO_DIVIDE` par `DIVISION_BY_ZERO`
- `INVALID_CURSOR` par `INVALID_CURSOR_STATE`
- `SQLCODE` par le presque équivalent `SQLSTATE` sous PostgreSQL
- `raise_application_error` en `RAISE EXCEPTION`

Un certain nombre d'exceptions ont leur équivalence sous PostgreSQL.

1.4.14 REMPLACEMENT AUTRES MOTS CLÉS

Remplacement de :

- `SYS_REFCURSOR` par `REFCURSOR`
- `SQL%NOTFOUND` par `NOT FOUND`
- `SYS_EXTRACT_UTC` par `AT TIME ZONE 'UTC'`
- `dup_val_on_index` en `unique_violation`

La liste des conversions est assez limitée, et il ne faut pas hésiter à faire des retours à l'auteur d'Ora2Pg pour qu'il inclue celles que vous détectez.

1.5 MIGRATION DES PROCÉDURES STOCKÉES

Étapes :

- Cas des procédures avec transactions autonomes
- Import des fonctions et paquets de fonctions
- Absence de fonctions ou paquets

C'est la partie la plus importante en terme de complexité et de temps dans la migration. Voici les étapes de la migration abordées dans cette partie :

- Comment importer les fonctions et les *packages* définis dans Oracle ?
- Pourquoi certaines fonctions sont-elles absentes de l'export ?

1.5.1 CAS DES TRANSACTIONS AUTONOMES

Non supportées nativement par PostgreSQL, Ora2Pg utilise une fonction de substitution :

- La fonction d'origine est renommée avec le suffix `_atx`
- La fonction de substitution prend le nom originel de la fonction
- La fonction de substitution appelle la fonction `_atx` au travers d'un dblink
- Utilisation possible de l'extension `pg_background` à partir de PostgreSQL 9.5 et Ora2Pg 17.5

Voici un exemple de procédure Oracle utilisant une transaction autonome pour tracer toutes les actions réalisées indépendamment et peu importe le résultat de la transaction.

Code Oracle :

```
CREATE PROCEDURE LOG_ACTION (username VARCHAR2, msg VARCHAR2)
IS
```

```
  PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
  INSERT INTO table_tracking VALUES (username, msg);
  COMMIT;
END log_action;
```

Ora2Pg va donc d'abord transformer cette routine et la renommer avec le suffix `_atx` comme suit :

```
CREATE OR REPLACE PROCEDURE log_action_atx (username text, msg text) AS $body$
BEGIN
  INSERT INTO table_tracking VALUES (username, msg);
END;
$body$ LANGUAGE plpgsql SECURITY DEFINER;
```

puis créer la routine de substitution qui sera appelée par l'applicatif :

```
CREATE OR REPLACE PROCEDURE log_action (username text, msg text) AS $body$
  -- Change this to reflect the dblink connection string
  v_conn_str  text := format('port=%s dbname=%s user=%s',
                      current_setting('port'), current_database(), current_user
                     );
  v_query     text;
BEGIN
  v_query := 'SELECT true FROM log_action_atx ( ' || quote_nullable(username)
          || ',' || quote_nullable(msg) || ' )';
  PERFORM * FROM dblink(v_conn_str, v_query) AS p (ret boolean);
END;
$body$ LANGUAGE plpgsql SECURITY DEFINER;
```

Dans le cas où la fonction est fortement utilisée, il est préférable de passer par un pooler de connexion comme pgbouncer sur les connexions dblink pour éviter les pertes de performances aux reconnexions incessantes.

À partir de la version Ora2Pg 17.5, il est possible de changer la réécriture des transactions autonomes avec l'extension `pg_background`. Il est nécessaire d'installer l'extension (sources sur Github : https://github.com/vibhorkum/pg_background) et d'activer le paramètre de configuration Ora2Pg :

```
# Use pg_background extension to create an autonomous transaction instead
# of using a dblink wrapper. With pg >= 9.5 only, default is to use dblink.
PG_BACKGROUND       1
```

Voici un exemple de la fonction de substitution générée par Ora2Pg pour `pg_background` :

```
CREATE OR REPLACE PROCEDURE log_action (username text, msg text) AS $body$
DECLARE
  v_query     text;
```

```
BEGIN
  v_query := 'SELECT true FROM log_action_atx ( ' || quote_nullable(username)
             || ',' || quote_nullable(msg) || ' )';
  PERFORM * FROM pg_background_result(pg_background_launch(v_query)) AS p (ret boolean);
END;
$body$ LANGUAGE plpgsql SECURITY DEFINER;
```

1.5.2 IMPORT DES PROCÉDURES ET PAQUETS AVEC ORA2PG

Chargement des fonctions et procédures :

```
psql --single-transaction -U myuser -f schema/functions/functions.sql mydb
psql --single-transaction -U myuser -f schema/procedures/procedures.sql mydb
```

Chargement des paquets de procédures stockées :

```
psql --single-transaction -U myuser -f schema/packages/packages.sql mydb
```

Le chargement du code PL/SQL transformé en PL/pgSQL par Ora2Pg se fait de la même manière que le code de création du schéma ou l'import des données, à savoir par la commande `psql`. Cependant, il y a une différence dans l'emploi de l'option `--single-transaction`. Comme le portage du code PL/SQL peut ne pas être complet et peut nécessiter des modifications manuelles, il y a de grande chance que le chargement génère des erreurs. Dans ce cas, l'inclusion dans une transaction provoque l'annulation de tout ce qui a été exécuté avant l'erreur évitant d'avoir du code obsolète créé dans la base.

C'est la même chose pour les paquets de fonctions. Pour simplifier le portage, comme les `packages` n'existent pas sous PostgreSQL, Ora2Pg va créer un schéma portant le nom du paquet et importer les fonctions dans ce schéma. Ceci permet de garder la notation Oracle : `PACKAGE.PROCEDURE` qui sera en fait sous PostgreSQL : `SCHEMA.FONCTION`.

Pour faciliter l'import et l'édition manuelle du code des procédures stockées, l'activation de la variable `FILE_PER_FUNCTION` permet d'exporter chaque fonction, procédure et trigger dans un fichier dédié, nommé par exemple `NOM_FONCTION_functions.sql`, pour les fonctions. Bien sûr, Ora2Pg crée aussi un fichier de chargement global permettant de charger tous les fichiers en un seul appel. Ce fichier sera ici nommé `functions.sql` ou `procedures.sql`.

Pour les paquets de procédures stockées, toujours si cette variable est activée, Ora2Pg va créer un sous répertoire portant le nom du paquet ou schéma. Les fonctions ou procédures du paquet seront exportées dans leurs fichiers respectifs tel qu'au dessus.

Pour permettre la prise en compte immédiate des erreurs et leur traitement au fil de l'import, les fichiers sont préfixés par l'appel à la commande suivante :

```
\set ON_ERROR_STOP ON
```

provoquant l'arrêt immédiat de l'import dès qu'une erreur est rencontrée.

En cas de doute et d'erreur sur le code converti automatiquement par Ora2Pg, vous pouvez comparer avec le code source du PL/SQL d'Oracle exporté dans les sous-répertoires du dossier `sources` du projet.

1.5.3 CODE NON EXPORTÉ

Absence de certaines fonctions ou paquets de fonctions dans l'export
- Le code a été invalidé par Oracle
- Activer `COMPILE_SCHEMA`
- Activer `EXPORT_INVALID`

Certains commentaires des paquets de fonctions ne sont pas importés

Si la variable de configuration `EXPORT_INVALID` n'était pas activée lors de l'export du schéma, le code marqué comme invalide par Oracle ne sera pas exporté. Ora2Pg n'extrait par défaut que le code valide. Si on ne veut pas exporter tout le code invalide, en activant la variable `COMPILE_SCHEMA`, Ora2Pg demandera à Oracle de vérifier à nouveau le code afin de valider ce qui peut l'être. Si la valeur de la directive `COMPILE_SCHEMA` vaut `1` c'est l'intégralité du code qui sera revalidé. Si sa valeur est un nom de schéma Oracle, seuls les objets appartenant à ce schéma le seront.

Ora2Pg préserve les commentaires définis dans le corps et à l'extérieur des fonctions d'Oracle. Par contre, lors du chargement dans PostgreSQL, les commentaires définis en dehors de ces fonctions ne seront pas intégrés.

1.6 TESTS ET VALIDATION

Valider le portage du code :
- Fonctionnement à l'identique
- Possibilité de résultats différents
- Déboguer le code PL/pgSQL et comparer avec le code source
- Ne pas oublier le test des scripts ou jobs externes

L'étape des tests unitaires est indispensable pour détecter les erreurs avant la mise en production et être sûr, en dehors de quelques différences acceptables, d'avoir le même comportement et les mêmes résultats que ce soit avec Oracle ou PostgreSQL.

Les tests doivent être réalisés unitairement, fonction par fonction lors de la conversion du code, puis fonctionnalité par fonctionnalité au niveau de l'application.

Il est possible que les résultats diffèrent soit légèrement, par exemple avec le nombre de décimales après la virgule, soit fortement, bien que le code PL/pgSQL ait été importé sans erreurs.

Pour vous aider, vous pouvez utiliser le débogueur `pldebugger` qui vous indiquera la ligne problématique dans le code et `plpgsql_check` qui vous remontera des problèmes de référence à des tables, colonnes ou variables inexistantes.

En cas de doute sur le code converti automatiquement par Ora2Pg, vous pouvez comparer avec le code source du PL/SQL d'Oracle exporté dans les sous-répertoires du dossier `sources` du projet.

1.6.1 ORA2PG : TESTS INTÉGRÉS

Deux actions permettent de tester à minima :

- `TEST` : compare le nombre d'objets et de lignes des deux bases.
  - `ora2pg -c config/ora2pg.conf -t TEST`
- `TEST_VIEW` : compare le nombre de lignes retounées par les vues.
  - `ora2pg -c config/ora2pg.conf -t TEST_VIEW`
- Dans les deux cas `PG_DSN` doit être positionné.

Dénombrement des objets migrés

Ora2Pg dispose d'une action permettant de réaliser une série de tests sur les objets ayant été migrés.

Cette action nommée `TEST` permet de savoir si tous les objets de la base Oracle ont été créés sous PostgreSQL. Pour que cette fonctionnalité puisse être utilisée, il est nécessaire de configurer les paramètres de connexion à la base PostgreSQL, à savoir `PG_DSN`, `PG_USER` et `PG_PWD`. Puis, une fois cette connexion définie, exécuter la commande :

```
$ ora2pg -t TEST -c config/ora2pg.conf > check_migration_diff.txt
```

Lors de ce test, Ora2Pg va dénombrer les informations suivantes des deux cotés, base source et base de destination :

- les index par table ;
- les contraintes d'unicité par table ;
- les contraintes check par table ;
- les contraintes NOT NULL par table ;

- les clés primaires par table ;
- les colonnes avec valeurs par défaut par table ;
- les clés étrangères par table ;
- les triggers par table ;
- les partitions par table partitionnée ;
- les tables dans la base ;
- les triggers dans la base ;
- les vues dans la base ;
- les vues matérialisées dans la base ;
- les séquences dans la base ;
- les types utilisateurs dans la base ;
- les tables distantes (FDW) dans la base.

Pour chaque objet dénombré, une section affichant les erreurs rencontrées permet d'identifier la source du problème. Voici un exemple de rapport généré :

```
[TEST INDEXES COUNT]
ORACLEDB:COUNTRIES:1
POSTGRES:countries:1
ORACLEDB:C50_LEX_CARTES:1
POSTGRES:c50_lex_cartes:1
ORACLEDB:SUPPLIER:1
POSTGRES:supplier:1
ORACLEDB:DEPARTMENTS:2
POSTGRES:departments:2
ORACLEDB:JOB_HISTORY:4
POSTGRES:job_history:4
ORACLEDB:REGIONS:1
POSTGRES:regions:1
ORACLEDB:MYTABLE:1
POSTGRES:mytable:1
ORACLEDB:LOCATIONS:4
POSTGRES:locations:4
ORACLEDB:EMPLOYEES:6
POSTGRES:employees:6
ORACLEDB:JOBS:1
POSTGRES:jobs:1
[ERRORS INDEXES COUNT]
OK, Oracle and PostgreSQL have the same number of indexes.

[TEST UNIQUE CONSTRAINTS COUNT]
ORACLEDB:COUNTRIES:1
POSTGRES:countries:1
ORACLEDB:SUPPLIER:1
POSTGRES:supplier:1
```

```
ORACLEDB:DEPARTMENTS:1
POSTGRES:departments:1
ORACLEDB:JOB_HISTORY:1
POSTGRES:job_history:1
ORACLEDB:REGIONS:1
POSTGRES:regions:1
ORACLEDB:MYTABLE:1
POSTGRES:mytable:1
ORACLEDB:LOCATIONS:1
POSTGRES:locations:1
ORACLEDB:EMPLOYEES:2
POSTGRES:employees:2
ORACLEDB:JOBS:1
POSTGRES:jobs:1
[ERRORS UNIQUE CONSTRAINTS COUNT]
OK, Oracle and PostgreSQL have the same number of unique constraints.

[TEST PRIMARY KEYS COUNT]
ORACLEDB:COUNTRIES:1
POSTGRES:countries:1
ORACLEDB:SUPPLIER:1
POSTGRES:supplier:1
ORACLEDB:DEPARTMENTS:1
POSTGRES:departments:1
ORACLEDB:JOB_HISTORY:1
POSTGRES:job_history:1
ORACLEDB:REGIONS:1
POSTGRES:regions:1
ORACLEDB:MYTABLE:1
POSTGRES:mytable:1
ORACLEDB:LOCATIONS:1
POSTGRES:locations:1
ORACLEDB:EMPLOYEES:1
POSTGRES:employees:1
ORACLEDB:JOBS:1
POSTGRES:jobs:1
[ERRORS PRIMARY KEYS COUNT]
OK, Oracle and PostgreSQL have the same number of primary keys.

[TEST CHECK CONSTRAINTS COUNT]
ORACLEDB:COUNTRIES:0
POSTGRES:countries:0
ORACLEDB:C50_LEX_CARTES:0
POSTGRES:c50_lex_cartes:0
ORACLEDB:SUPPLIER:0
POSTGRES:supplier:0
```

```
ORACLEDB:DEPARTMENTS:0
POSTGRES:departments:0
ORACLEDB:EMPLOYEES:1
POSTGRES:employees:1
ORACLEDB:JOBS:0
POSTGRES:jobs:0
ORACLEDB:JOB_HISTORY:1
POSTGRES:job_history:1
ORACLEDB:REGIONS:0
POSTGRES:regions:0
ORACLEDB:MESURE:0
POSTGRES:mesure:0
ORACLEDB:FICHIER_DONNEE:0
POSTGRES:fichier_donnee:0
ORACLEDB:LOCATIONS:0
POSTGRES:locations:0
[ERRORS CHECK CONSTRAINTS COUNT]
OK, Oracle and PostgreSQL have the same number of check constraints.

[TEST NOT NULL CONSTRAINTS COUNT]
ORACLEDB:TIME_TZ2:0
POSTGRES:time_tz2:0
ORACLEDB:COUNTRIES:1
POSTGRES:countries:1
ORACLEDB:C50_LEX_CARTES:1
POSTGRES:c50_lex_cartes:1
ORACLEDB:SUPPLIER:2
POSTGRES:supplier:2
ORACLEDB:DEPARTMENTS:2
POSTGRES:departments:2
ORACLEDB:MYTABLE:1
POSTGRES:mytable:1
ORACLEDB:EMPLOYEES:5
POSTGRES:employees:5
ORACLEDB:JOBS:2
POSTGRES:jobs:2
ORACLEDB:VAL_RESULTS:0
POSTGRES:val_results:0
ORACLEDB:JOB_HISTORY:4
POSTGRES:job_history:4
ORACLEDB:TESTA:0
POSTGRES:testa:0
ORACLEDB:REGIONS:1
POSTGRES:regions:1
ORACLEDB:TEST_TZ:0
POSTGRES:test_tz:0
```

```
ORACLEDB:MESURE:1
POSTGRES:mesure:1
ORACLEDB:FICHIER_DONNEE:1
POSTGRES:fichier_donnee:1
ORACLEDB:TEST_NUM:0
POSTGRES:test_num:0
ORACLEDB:LOCATIONS:2
POSTGRES:locations:2
[ERRORS NOT NULL CONSTRAINTS COUNT]
OK, Oracle and PostgreSQL have the same number of null constraints.

[TEST COLUMN DEFAULT VALUE COUNT]
ORACLEDB:TIME_TZ2:0
POSTGRES:time_tz2:0
ORACLEDB:COUNTRIES:0
POSTGRES:countries:0
ORACLEDB:C50_LEX_CARTES:0
POSTGRES:c50_lex_cartes:0
ORACLEDB:SUPPLIER:0
POSTGRES:supplier:0
ORACLEDB:DEPARTMENTS:0
POSTGRES:departments:0
ORACLEDB:MYTABLE:0
POSTGRES:mytable:0
ORACLEDB:EMPLOYEES:0
POSTGRES:employees:0
ORACLEDB:JOBS:0
POSTGRES:jobs:0
ORACLEDB:VAL_RESULTS:0
POSTGRES:val_results:0
ORACLEDB:JOB_HISTORY:0
POSTGRES:job_history:0
ORACLEDB:TESTA:0
POSTGRES:testa:0
ORACLEDB:REGIONS:0
POSTGRES:regions:0
ORACLEDB:TEST_TZ:0
POSTGRES:test_tz:0
ORACLEDB:MESURE:0
POSTGRES:mesure:0
ORACLEDB:FICHIER_DONNEE:0
POSTGRES:fichier_donnee:0
ORACLEDB:TEST_NUM:0
POSTGRES:test_num:0
ORACLEDB:LOCATIONS:0
POSTGRES:locations:0
```

```
[ERRORS COLUMN DEFAULT VALUE COUNT]
OK, Oracle and PostgreSQL have the same number of column default value.

[TEST FOREIGN KEYS COUNT]
ORACLEDB:COUNTRIES:1
POSTGRES:countries:1
ORACLEDB:DEPARTMENTS:2
POSTGRES:departments:2
ORACLEDB:LOCATIONS:1
POSTGRES:locations:1
ORACLEDB:JOB_HISTORY:3
POSTGRES:job_history:3
ORACLEDB:EMPLOYEES:3
POSTGRES:employees:3
[ERRORS FOREIGN KEYS COUNT]
OK, Oracle and PostgreSQL have the same number of foreign keys.

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

[TEST PARTITION COUNT]
[ERRORS PARTITION COUNT]
OK, Oracle and PostgreSQL have the same number of PARTITION.

[TEST TABLE COUNT]
ORACLEDB:TABLE:21
POSTGRES:TABLE:20
[ERRORS TABLE COUNT]
TABLE does not have the same count in source database (21) and in PostgreSQL (20).

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

[TEST VIEW COUNT]
ORACLEDB:VIEW:1
POSTGRES:VIEW:5
[ERRORS VIEW COUNT]
VIEW does not have the same count in source database (1) and in PostgreSQL (5).

[TEST MVIEW COUNT]
```

```
ORACLEDB:MVIEW:1

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

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

[TEST TYPE COUNT]
ORACLEDB:TYPE:1
POSTGRES:TYPE:21
[ERRORS TYPE COUNT]
TYPE does not have the same count in source database (1) and in PostgreSQL (21).

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

Il est aussi possible de demander à Ora2Pg de dénombrer et de comparer le nombre de lignes de chaque table avec l'option `--count_rows` :

```
$ ora2pg -t TEST -c config/ora2pg.conf --count_rows > check_migration_diff.txt
```

Évidement cela n'a de sens que si la base source n'a pas subi de modification du nombre de lignes entre temps.

Dénombrement des résultats des vues

En raison du formatage des données retournées par Oracle il n'est pas possible de comparer simplement les données entre les deux bases, cependant on peut déjà s'assurer que le nombre de lignes renvoyées par les vues est identique. Pour cela l'action `TEST_VIEW` peut être utilisée.

```
$ ora2pg -t TEST_VIEW -c config/ora2pg.conf > check_view_migration_diff.txt
```

1.6.2 OUTILS DE TESTS UNITAIRES POUR POSTGRESQL

- pgTap
  - http://www.pgtap.org/
- pgUnit
  - http://en.dklab.ru/lib/dklab_pgunit/
- Epic
  - http://www.epictest.org/

`pgTAP` est une bibliothèque de fonctions pour PostgreSQL développées par David E. Wheeler permettant d'écrire des tests unitaires au format `TAP` (`Test Anything Protocol`) dans des scripts exécutables par la commande `psql`.

`pgTAP` permet de vraiment tester la base de données, non seulement en vérifiant la structure du schéma, mais aussi en testant les vues, les procédures, les fonctions, les règles, ou triggers.

Voici un exemple de test avec la syntaxe `pgTap` :

```
-- Start a transaction.
BEGIN;
SELECT plan( 2 );
\set domain_id 1
\set src_id 1

-- Insert stuff.
SELECT ok(
    insert_stuff( 'www.foo.com', '{1,2,3}', :domain_id, :src_id ),
    'insert_stuff() should return true'
);

-- Check for domain stuff records.
SELECT is(
    ARRAY(
        SELECT stuff_id
          FROM domain_stuff
         WHERE domain_id = :domain_id
           AND src_id = :src_id
         ORDER BY stuff_id
    ),
    ARRAY[ 1, 2, 3 ],
    'The stuff should have been associated with the domain'
);

SELECT * FROM finish();
ROLLBACK;
```

Vous pouvez aussi écrire un scénario complet de validation de la structure de la base de données après export :

```
BEGIN;
SELECT plan( 18 );

SELECT has_table( 'domains' );
SELECT has_table( 'stuff' );
SELECT has_table( 'sources' );
SELECT has_table( 'domain_stuff' );

SELECT has_column( 'domains', 'id' );
SELECT col_is_pk(  'domains', 'id' );
SELECT has_column( 'domains', 'domain' );

SELECT has_column( 'stuff',   'id' );
SELECT col_is_pk(  'stuff', 'id' );
SELECT has_column( 'stuff',   'name' );

SELECT has_column( 'sources', 'id' );
SELECT col_is_pk(  'sources', 'id' );
SELECT has_column( 'sources', 'name' );

SELECT has_column( 'domain_stuff', 'domain_id' );
SELECT has_column( 'domain_stuff', 'source_id' );
SELECT has_column( 'domain_stuff', 'stuff_id' );
SELECT col_is_pk(
    'domain_stuff',
    ARRAY['domain_id', 'source_id', 'stuff_id']
);

SELECT can_ok(
    'insert_stuff',
    ARRAY[ 'text', 'integer[]', 'integer', 'integer' ]
);

SELECT * FROM finish();
ROLLBACK;
```

`pgUnit` et `Epic` sont deux autres bibliothèques de fonctions PL/pgSQL permettant de réaliser des tests unitaires, mais `pgTAP` est le plus intéressant car le format `TAP` trouve des implémentations en C, C++, Python, PHP, Perl, Java, JavaScript, et autres.

Pour plus d'informations sur le format `TAP`, consultez le site officiel[8] , vous trouverez un

[8] https://testanything.org/

exemple d'implémentation Java avec le projet tap4j[9] .

1.6.3 PLANS DE TESTS COMPLETS

- Tests sur la base données
- Tests sur l'application
- Tests sur les performances
- Stress test
- Tests des scripts de maintenance et job

Toutes les différentes composantes du projet de migration doivent être testées, pas seulement la base de données et l'application mais aussi les performances et les scripts de maintenance. Cela peut permettre par exemple de s'apercevoir qu'un index n'a pas été créé ou que le serveur PostgreSQL n'a pas été optimisé correctement.

1.7 CONCLUSION

- La conversion automatique fait gagner du temps
- Mais les réécritures manuelles peuvent s'avérer nombreuses
- La phase de tests est la plus importante de la migration

La conversion du code fait gagner du temps. Aussi étonnant que cela puisse paraître, elle est très fonctionnelle. Cependant, tout aussi excellente qu'elle soit, il faudra toujours vérifier les procédures stockées. Il faudra s'assurer que le résultat produit est le bon, et que les performances sont au moins tout aussi bonnes. Cela fait que cette partie de la migration est généralement la plus dure et la plus longue.

1.7.1 QUESTIONS

N'hésitez pas, c'est le moment !

---

[9] https://sourceforge.net/projects/tap4j/

1.8 QUIZ

https://dali.bo/n4_quiz

1.9 TRAVAUX PRATIQUES

1.9.1 IMPORTER LES ROUTINES

Créer les fonctions, procédures et les triggers dans la base `pghr` à partir des sources converties en PL/pgSQL.

```
schema/
├── functions
│   ├── EMP_SAL_RANKING_function.sql
│   ├── function.sql
│   └── LAST_FIRST_NAME_function.sql
├── procedures
│   ├── ADD_JOB_HISTORY_procedure.sql
│   ├── procedure.sql
│   └── SECURE_DML_procedure.sql
└── triggers
    ├── trigger.sql
    └── UPDATE_JOB_HISTORY_trigger.sql
```

1.9.2 IMPORTER LES PAQUETS DE PROCÉDURES STOCKÉES

Créer les paquets de procédures stockées en corrigeant les différentes erreurs.

```
schema/
└── 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
```

1.9.3 TESTS UNITAIRES

Installer l'extension pgTAP[10] .

Pour les fonctions et procédures stockées suivantes, écrire les tests unitaires SQL à l'aide de l'extension pgTAP pour garantir que les résultats en provenance de l'instance Oracle soient identiques sur l'instance PostgreSQL.

Les appels aux routines suivantes proviennent de l'instance Oracle.

1. Fonction `last_first_name(bigint)`

```
SELECT last_first_name(105) FROM dual;

LAST_FIRST_NAME(105)
------------------------------
Employee: 105 - AUSTIN, DAVID
```

2. Fonction `emp_sal_ranking(bigint)`

```
SELECT emp_sal_ranking(105) FROM dual;

EMP_SAL_RANKING(105)
--------------------
                .125
```

Procéder aux transformations nécessaires pour que les résultats soient conformes entre les deux systèmes.

3. Trigger `update_job_history`

```
UPDATE employees SET job_id = 'AC_MGR' WHERE employee_id = 105;

SELECT employee_id, job_id FROM job_history
 WHERE employee_id = 105 AND end_date > TRUNC(sysdate);

ROLLBACK;

EMPLOYEE_ID JOB_ID
----------- ----------
        105 IT_PROG
```

Contrôler à l'aide d'un test unitaire que le trigger se déclenche correctement.

4. Procédure stockée `emp_mgmt.increase_sal(bigint, bigint)`

```
CALL emp_mgmt.increase_sal(105, 500);

SELECT salary FROM employees WHERE employee_id = 105;
```

[10] https://pgtap.org/

```
ROLLBACK;

    SALARY
----------
      5300
```

Procéder aux transformations nécessaires pour que la procédure s'exécute correctement.

1.10 TRAVAUX PRATIQUES (SOLUTIONS)

1.10.1 IMPORTER LES ROUTINES

> Créer les fonctions, procédures et les triggers dans la base `pghr` à partir des sources converties en PL/pgSQL.

Pour faciliter la correction, les variables suivantes sont exportées pour l'utilisateur **postgres** :

```
export PGDATABASE=pghr
export PGUSER=migration
```

Commençons par importer la procédure manquante appelée dans le trigger `update_job_history`. Comme nous avons choisi d'exporter les fonctions dans des fichiers séparés, la procédure se trouve dans le fichier `ADD_JOB_HISTORY_procedure.sql` du répertoire de travail `schema/procedures`.

```
psql -f schema/procedures/ADD_JOB_HISTORY_procedure.sql
```

```
SET
NOTICE: job_history.employee_id%TYPE converted to integer
NOTICE: job_history.start_date%TYPE converted to timestamp without time zone
NOTICE: job_history.end_date%TYPE converted to timestamp without time zone
NOTICE: job_history.job_id%TYPE converted to character varying
NOTICE: job_history.department_id%TYPE converted to smallint
CREATE PROCEDURE
```

Il n'y a eu aucune erreur à la création de la procédure `add_job_history`, uniquement des indications sur les types réellement utilisés. Voici le code de création de la procédure :

```
CREATE OR REPLACE PROCEDURE add_job_history (
    p_emp_id job_history.employee_id%type ,
    p_start_date job_history.start_date%type ,
    p_end_date job_history.end_date%type ,
    p_job_id job_history.job_id%type ,
    p_department_id job_history.department_id%type )
AS $body$
BEGIN
  INSERT INTO job_history(employee_id, start_date, end_date,
                          job_id, department_id)
    VALUES (p_emp_id, p_start_date, p_end_date, p_job_id, p_department_id);
END;
$body$
LANGUAGE PLPGSQL SECURITY DEFINER;
```

Le trigger peut maintenant être rechargé.

```
psql -f schema/triggers/trigger.sql

SET
SET
NOTICE: trigger "update_job_history" for relation "employees"
        does not exist, skipping
DROP TRIGGER
CREATE FUNCTION
CREATE TRIGGER
```

Le chargement de la deuxième procédure est direct, sans erreur. Aucune retouche n'est à faire sur le code de la fonction :

```
psql -f schema/procedures/SECURE_DML_procedure.sql

SET
CREATE PROCEDURE
```

Pour l'import de routines en général, le mieux est d'utiliser le fichier d'import global. Par exemple pour les fonctions, il suffit de procéder de la façon suivante :

```
psql --single-transaction -f schema/functions/function.sql
```

et de voir s'il y a des erreurs.

```
SET
SET
CREATE FUNCTION
SET
CREATE FUNCTION
```

Il n'y a aucune erreur, la conversion par Ora2Pg semble complète.

ATTENTION, cela ne veut pas dire qu'il n'y a pas d'adaptation à faire. Nous verrons dans le dernier exercice que la routine `emp_sal_ranking(bigint)` ne retourne pas les mêmes résultats que la procédure équivalente sur Oracle.

1.10.2 IMPORTER LES PAQUETS DE PROCÉDURES STOCKÉES

> Créer les paquets de procédures stockées en corrigeant les différentes erreurs.

La base d'exemple contient deux paquets de procédures stockées (*packages*) `emp_actions` et `emp_mgmt` composés respectivement de 4 et 6 routines. Lançons un premier chargement :

```
psql -f schema/packages/package.sql --single-transaction
```

Création des extensions si nécessaire

Il faut s'attendre à l'une des erreurs suivantes :

```
ERROR: could not open extension control file "…/dblink.control"
No such file or directory
```

L'une des fonctions du paquet `emp_actions` est décrite comme étant une transaction autonome. Ora2Pg réalise la réécriture à l'aide de l'extension `dblink` par défaut. (Cette extension fait partie des contribs à installer avec PostgreSQL.)

Une nouvelle exécution remonte un nouveau message :

```
ERROR:  permission denied to create extension "dblink"
HINT:  Must be superuser to create this extension.
```

L'aide nous informe que l'extension ne peut être créée avec le compte `migration` s'il ne dispose pas des droits superutilisateur. Nous devons lancer la commande suivante avec le compte `postgres` du serveur pour progresser (ou tout autre superutilisateur à votre disposition) :

```
psql -U postgres -c "CREATE EXTENSION IF NOT EXISTS dblink"
```

Déclaration des variables globales

Si l'on relance l'import des paquets, une autre erreur survient :

```
psql -f schema/packages/package.sql --single-transaction

psql:schema/packages/emp_mgmt/remove_dept_package.sql:19:
  ERROR:  "current_setting" is not a known variable
  LINE 6:       SELECT COUNT(*) INTO STRICT current_setting('emp_mgmt....
                                            ^
```

La procédure `remove_dept` utilise une variable globale au sein du paquet `emp_mgmt`. Or, au moment de la récupération de la variable `emp_mgmt.tot_depts` avec la méthode `current_setting()`, cette dernière n'est pas initialisée. Les variables globales dans le code PL/pgSQL ne sont pas supportées sous PostgreSQL, tout au moins pas de cette manière. L'outil Ora2Pg génère automatique le fichier de configuration `schema/packages/global_variables.conf` pour gérer simplement les valeurs par défaut de ces variables.

```
emp_mgmt.tot_depts = ''
emp_mgmt.tot_emps = ''
```

Pour leur prise en compte, il est nécessaire de les intégrer dans la configuration de l'instance, au choix dans le fichier `postgresql.conf`, le fichier `postgresql.auto.conf` ou tout autre fichier prévu dans les inclusions `include_dir` ou `include_if_exists` de

l'instance. **Attention**, ce type de déclaration est statique et valable pour toute l'instance ; la méthode `set_config()` n'aura d'effet qu'à l'intérieur de la session qui l'invoque.

Dans cet exercice, l'usage de ces deux variables `tot_depts` et `tot_emps` sert à tenir à jour le nombre total de départements et d'employés, probablement pour éviter l'appel à `count(*)`. Ceci peut être fait d'une autre manière, notamment par l'usage d'une table de variables mise à jour par triggers.

Ici, la solution consiste à modifier tous les fichiers y faisant référence en les supprimant ou en les mettant en commentaire.

```
find schema/packages -type f -name *.sql -exec grep -nH "setting('emp" {} \;

-- schema/packages/emp_mgmt/hire_package.sql:23:
PERFORM set_config('emp_mgmt.tot_emps',
   current_setting('emp_mgmt.tot_emps')::bigint + 1, false);

-- schema/packages/emp_mgmt/create_dept_package.sql:21:
PERFORM set_config('emp_mgmt.tot_depts',
   current_setting('emp_mgmt.tot_depts')::bigint + 1, false);

-- schema/packages/emp_mgmt/remove_emp_package.sql:14:
PERFORM set_config('emp_mgmt.tot_emps',
   current_setting('emp_mgmt.tot_emps')::bigint - 1, false);

-- schema/packages/emp_mgmt/remove_dept_package.sql:14:
PERFORM set_config('emp_mgmt.tot_depts',
   current_setting('emp_mgmt.tot_depts')::bigint - 1, false);

-- schema/packages/emp_mgmt/remove_dept_package.sql:15:
SELECT COUNT(*) INTO STRICT
   current_setting('emp_mgmt.tot_emps')::bigint FROM employees;
```

De manière générale, pour émuler les variables globales, on peut utiliser une table de variables mise à jour par triggers, ou utiliser un langage procédural permettant ce type de stockage global, comme le PL/Perl.

Création des paquets sans erreur

À présent, l'exécution du script `package.sql` sur la base aboutit sans erreur :

```
export PGOPTIONS='-c client_min_messages=warning'
psql -f schema/packages/package.sql --single-transaction

SET
DROP SCHEMA
CREATE SCHEMA
SET
```

Procédures stockées

```
CREATE PROCEDURE
SET
CREATE PROCEDURE
SET
CREATE FUNCTION
SET
CREATE EXTENSION
CREATE FUNCTION
CREATE FUNCTION
DROP SCHEMA
CREATE SCHEMA
SET
CREATE FUNCTION
SET
CREATE FUNCTION
SET
CREATE PROCEDURE
SET
CREATE PROCEDURE
SET
CREATE PROCEDURE
SET
CREATE PROCEDURE
```

1.10.3 TESTS UNITAIRES

> Installer l'extension pgTAP[a] .
>
> [a] https://pgtap.org/

Le paquet est disponible à partir du dépôt PGDG.

Par exemple :

```
sudo yum install -y pgtap_14    # CentOS 7 / Red Hat 7
sudo dnf install -y pgtap_14    # Rocky Linux 8, Red Hat 8
sudo apt install postgresql-14-pgtap      # Debian, Ubuntu...
```

La paquet `pgtap` fournit deux composants :

- un client `pg_prove` pour exécuter les tests unitaires ;
- une extension `pgtap` à installer dans une base de données pour bénéficier d'un ensemble de fonctions de tests unitaires.

Dans le cas où la version du paquet pgTAP correspondant à la version de votre PostgreSQL n'existe pas, il est possible d'installer les fichiers de l'extension à l'aide de l'outil `pgxn` contenu dans le paquet système `pgxnclient`. Son installation nécessite plusieurs étapes de dépendances et de compilation, à adapter selon la version de PostgreSQL employée :

- Installation des dépendances sous Centos 7 :

```
# en root
yum update
yum groupinstall "Development Tools"

yum install -y centos-release-scl epel-release
yum install -y postgresql13-devel
yum install -y python36-six pgxnclient patch
```

- Installation des fichiers de l'extension pgTAP

```
# en root
PATH=/usr/pgsql-14/bin:$PATH pgxn install pgtap
```

- Compilation de l'outil `pg_prove` à l'aide de l'utilitaire `cpan`

```
# en root
yum install -y perl-CPAN

PERL_MB_OPT="--install_path lib=/usr/share/perl5"
PERL_MB_OPT="$PERL_MB_OPT --install_path bin=/usr/local/bin"
PERL_MB_OPT="$PERL_MB_OPT --install_path bindoc=/usr/share/man/man1"
PERL_MB_OPT="$PERL_MB_OPT --install_path libdoc=/usr/share/man/man3"
export PERL_MB_OPT

cpan install Module::Build
cpan TAP::Parser::SourceHandler::pgTAP
```

> Pour les fonctions et procédures stockées suivantes, écrire les tests unitaires SQL à l'aide de l'extension pgTAP pour garantir que les résultats en provenance de l'instance Oracle soient identiques sur l'instance PostgreSQL.

Chaque composant de la base `pghr` sera testé dans un fichier distinct, à créer dans un nouveau répertoire `tests` du projet Ora2Pg.

```
cd $HOME/tp_migration
mkdir -p tests/{functions,triggers,packages}
```

L'extension `pgtap` doit être installée dans la base `pghr`, idéalement dans un schéma dédié pour isoler ses fonctions de la logique métier de l'application HR.

```
\c pghr
CREATE SCHEMA IF NOT EXISTS pgtap;
```

```
CREATE EXTENSION pgtap WITH SCHEMA pgtap;

GRANT USAGE ON SCHEMA pgtap TO public;
GRANT EXECUTE ON ALL ROUTINES IN SCHEMA pgtap TO public;
```

1. Fonction `last_first_name(bigint)`

Un fichier de tests unitaires se découpe en trois parties, à savoir : l'ouverture d'une transaction `BEGIN` et l'appel de la méthode `plan(int)` pour annoncer le nombre de tests dans le fichier, la définition des tests unitaires (ici, la méthode sera `is(any, any, text)` pour comparer le résultat de l'appel de fonction avec un résultat prédéterminé) et la fermeture de la transaction avec l'appel de la méthode `finish()` pour indiquer que les tests sont terminés.

```
-- tests/functions/last_first_name_test.sql
SET search_path = pgtap,public;

BEGIN;
SELECT plan(1);

SELECT is(
  last_first_name(105),
  'Employee: 105 - AUSTIN, DAVID',
  'Function last_first_name should return same result as Oracle'
);

SELECT * FROM finish();
ROLLBACK;
```

Ce script peut-être exécuté avec le client `psql` comme suit, même s'il faut préférer l'outil `pg_prove` pour la suite des corrections.

```
psql -f tests/functions/last_first_name_test.sql

SET
BEGIN
 plan
------
 1..1
(1 row)

                              is
----------------------------------------------------------------
 ok 1 - Function last_first_name should return same result as Oracle
(1 row)

 finish
```

```
--------
(0 rows)

ROLLBACK
```

Et avec l'outil `pg_prove` :

```
pg_prove tests/functions/last_first_name_test.sql

tests/functions/last_first_name_test.sql .. ok
All tests successful.
Files=1, Tests=1,  0 wallclock secs
  ( 0.02 usr +  0.00 sys =  0.02 CPU)
Result: PASS
```

Pour cette fonction, les résultats sont conformes à ceux attendus, il n'y aura pas de réécriture du code PL/pgSQL.

---

2. Fonction `emp_sal_ranking(bigint)`

Cette fonction présente une anomalie de conversion par Ora2Pg. En effet, la signature suivante en provenance d'Oracle :

```
FUNCTION emp_sal_ranking (empid NUMBER)
  RETURN NUMBER
```

a été convertie implicitement lors de l'export par :

```
FUNCTION emp_sal_ranking (empid bigint)
  RETURNS bigint
```

Le type `NUMBER` n'existe pas avec PostgreSQL. Pour avoir un équivalent, il faut savoir si les données manipulées sont entières, décimales ou rationnelles. Les types disponibles sont respectivement `integer`, `numeric` et `float` (ou `double precision`).

La méthode `is(any, any, text)` de l'extension requiert que les deux premiers arguments soient du même typage afin de réaliser une comparaison fiable. Avant toute réécriture de la fonction `emp_sal_ranking`, nous pouvons forcer le type de retour avec l'opérateur `::numeric` dans le test pgTAP comme suit pour comparer les deux résultats sans erreur :

```
-- tests/functions/emp_sal_ranking_test.sql
SET search_path = pgtap,public;

BEGIN;
SELECT plan(1);
```

```
SELECT is(
  emp_sal_ranking(105)::numeric,
  .125,
  'Function emp_sal_ranking should return a decimal value as Oracle'
);

SELECT * FROM finish();
ROLLBACK;
```

L'outil `pg_prove` peut être exécuté avec les options `--recurse` et `--ext` pour parcourir les sous-répertoires contenant l'ensemble des scripts SQL.

```
pg_prove --recurse --ext .sql tests

tests/functions/emp_sal_ranking_test.sql .. 1/1
# Failed test 1: "Function emp_sal_ranking should return a float value as Oracle"
#   have: 0
#   want: 0.125
# Looks like you failed 1 test of 1
tests/functions/emp_sal_ranking_test.sql .. Failed 1/1 subtests
tests/functions/last_first_name_test.sql .. ok

Test Summary Report
-------------------
tests/functions/emp_sal_ranking_test.sql (Wstat: 0 Tests: 1 Failed: 1)
  Failed test:  1
Files=2, Tests=2,  0 wallclock secs
  ( 0.02 usr +  0.00 sys =  0.02 CPU)
Result: FAIL
```

L'anomalie devient évidente avec le test en erreur. La valeur de retour de la fonction `emp_sal_ranking` diffère du résultat proposé par Oracle. Comme exposé plus haut, le typage est la principale source d'erreur, avec la perte de précision lors d'une division de deux `double precision` dans PostgreSQL.

> Procéder aux transformations nécessaires pour que les résultats soient conformes entre les deux systèmes.

D'après le code de la fonction `emp_sal_ranking`, le quotient produit une valeur décimale comprise entre `0` et `1` à partir d'une opération sur les données de la colonne `employees.salary`. Le type `double precision` est donc adapté pour cette fonction.

Corriger le fichier `EMP_SAL_RANKING_function.sql` dans le répertoire `schema/functions` du projet `tp_migration` pour changer le type de retour.

```
) RETURNS bigint AS $body$
```

Devient

```
) RETURNS double precision AS $body$
```

Recréer la fonction `emp_sal_ranking`.

```
psql -c 'DROP FUNCTION emp_sal_ranking(bigint)'
psql -f schema/functions/EMP_SAL_RANKING_function.sql
```

```
SET
CREATE FUNCTION
```

Le nouveau test est à présent positif.

```
pg_prove --recurse --ext .sql tests
```

```
tests/functions/emp_sal_ranking_test.sql .. ok
tests/functions/last_first_name_test.sql .. ok
All tests successful.
Files=2, Tests=2,  0 wallclock secs
  ( 0.02 usr  0.00 sys +  0.00 cusr  0.01 csys =  0.03 CPU)
Result: PASS
```

---

3. Trigger `update_job_history`

> Contrôler à l'aide d'un test unitaire que le trigger se déclenche correctement.

Ce trigger déclenche la fonction `trigger_fct_update_job_history`, qui elle-même fait appel à la procédure `add_job_history`. Le but de ce test est de s'assurer que l'ensemble des règles logiques est bien respecté au moment du déclenchement du trigger, à savoir l'événement `AFTER UPDATE OF job_id, department_id ON employees`.

Le test suivant se déroule en deux étapes :

- Mise à jour d'une ligne `employees` ;
- Comparaison d'un résultat de requête avec un jeu de données.

```
-- tests/triggers/update_job_history_test.sql
SET search_path = pgtap,public;

BEGIN;
SELECT plan(2);
```

```
SELECT lives_ok(
  $$UPDATE employees SET job_id = 'AC_MGR' WHERE employee_id = 105$$,
  'An update on employees should be sucessful'
);

SELECT results_eq(
  'SELECT employee_id, job_id FROM job_history
    WHERE employee_id = 105 AND end_date > current_date',
  $$VALUES (105, 'IT_PROG'::varchar)$$,
  'Trigger update_job_history should result on a new insert into job_history'
);

SELECT * FROM finish();
ROLLBACK;
```

La colonne `job_id` étant du type `varchar(10)`, il est nécessaire de convertir la chaîne de texte `IT_PROG` à l'aide de l'opérateur `::varchar` dans le but de réussir la comparaison.

Lancer la série de tests unitaires avec `pg_prove`.

```
pg_prove --recurse --ext .sql tests
```

```
tests/functions/emp_sal_ranking_test.sql .... ok
tests/functions/last_first_name_test.sql .... ok
tests/triggers/update_job_history_test.sql .. ok
All tests successful.
Files=3, Tests=4,  1 wallclock secs
  ( 0.03 usr  0.00 sys +  0.00 cusr  0.01 csys =  0.04 CPU)
Result: PASS
```

4. Procédure stockée `emp_mgmt.increase_sal(bigint, bigint)`

Ce dernier test est similaire au précédent, avec une instruction `CALL` sur la procédure `emp_mgmt.increase_sal`, suivie d'une comparaison du nouveau salaire avec une valeur prédéterminée.

```
-- tests/packages/emp_mgmt.increase_sal_test.sql
SET search_path = pgtap,public;

BEGIN;
SELECT plan(2);

SELECT lives_ok(
  'CALL emp_mgmt.increase_sal(105, 500)',
  'Procedure increase_sal should be successful'
);

SELECT row_eq(
```

```
  'SELECT salary FROM employees WHERE employee_id = 105',
  ROW(5300::float),
  'Procedure increase_sal should increase salary'
);

SELECT * FROM finish();
ROLLBACK;
```

La méthode `row_eq` est similaire à la précédente méthode `results_eq`, si ce n'est qu'elle n'attend qu'une seule ligne de résultat.

Lancer la série de tests unitaires avec `pg_prove`.

```
pg_prove --recurse --ext .sql tests

tests/functions/emp_sal_ranking_test.sql ....... ok
tests/functions/last_first_name_test.sql ....... ok
tests/packages/emp_mgmt.increase_sal_test.sql .. 1/2
# Failed test 1: "Procedure increase_sal should be successful"
#   died: 42702:
#   column reference "employee_id" is ambiguous
#     DETAIL: It could refer to either a PL/pgSQL variable or a table column.
#     CONTEXT:
#       PL/pgSQL function emp_mgmt.increase_sal(bigint,bigint) line 12
#       SQL statement "CALL emp_mgmt.increase_sal(105, 500)"
#       PL/pgSQL function lives_ok(text,text) line 14 at EXECUTE
# Failed test 2: "Procedure increase_sal should increase salary"
#   have: (4800)
#   want: (5300)
# Looks like you failed 2 tests of 2
tests/packages/emp_mgmt.increase_sal_test.sql .. Failed 2/2 subtests
tests/triggers/update_job_history_test.sql ..... ok

Test Summary Report
-------------------
tests/packages/emp_mgmt.increase_sal_test.sql (Wstat: 0 Tests: 2 Failed: 2)
  Failed tests:  1-2
Files=4, Tests=6,  0 wallclock secs
  ( 0.03 usr  0.00 sys +  0.01 cusr  0.01 csys =  0.05 CPU)
Result: FAIL
```

Pour ce nouveau cas de test, une ambiguïté sur un nom de colonne `employee_id` est levée lors de l'appel de la procédure `increase_sal` du paquet `emp_mgmt`. Puisque la mise à jour du salaire n'a pas eu lieu, le second test est également en erreur avec deux résultats qui diffèrent l'un de l'autre.

> Procéder aux transformations nécessaires pour que la procédure s'exécute correctement.

En consultant le code procédural dans le fichier `increase_sal_package.sql`, une condition de jointure présente une anomalie avec l'absence d'alias de colonne. Bien que cette syntaxe soit supportée avec Oracle, PostgreSQL se montre moins permissif et requiert une correction.

Corriger le fichier `increase_sal_package.sql` présent dans le répertoire `schema/packages/emp_mgmt`.

```
UPDATE employees
SET salary = salary + salary_incr
WHERE employee_id = employee_id;
```

Devient

```
UPDATE employees
SET salary = salary + salary_incr
WHERE employees.employee_id = increase_sal.employee_id;
```

Recréer la procédure `emp_mgmt.increase_sal`.

```
psql -f schema/packages/emp_mgmt/increase_sal_package.sql
```

```
SET
CREATE PROCEDURE
```

Le nouveau test est à présent positif.

```
pg_prove --recurse --ext .sql tests
```

```
tests/functions/emp_sal_ranking_test.sql ....... ok
tests/functions/last_first_name_test.sql ....... ok
tests/packages/emp_mgmt.increase_sal_test.sql .. ok
tests/triggers/update_job_history_test.sql ..... ok
All tests successful.
Files=4, Tests=6,  0 wallclock secs
  ( 0.02 usr  0.01 sys +  0.01 cusr  0.01 csys =  0.05 CPU)
Result: PASS
```

NOTES

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.