Formation SQL5

Migration d'Oracle à PostgreSQL

17.12

Dalibo SCOP

https://dalibo.com/formations

Migration d'Oracle à PostgreSQL

Formation SQL5

TITRE : Migration d'Oracle à PostgreSQL
SOUS-TITRE : Formation SQL5

REVISION: 17.12
DATE: 8 janvier 2018
ISBN: 979-10-97371-09-8

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, Sharon Bonan, Damien Clochard, Christrophe Courtois, Marc Cousin, Gilles Darold, Jehan-Guillaume de Rorthais, Ronan Dunklau, Vik Fearing, Stefan Fercot, Pierre Giraud, Nicolas Gollet, Dimitri Fontaine, Virginie Jourdan, Guillaume Lelarge, Jean-Louis Louër, Thibaut Madelaine, Adrien Nayrat, Flavie Perette, Thomas Reiss, Maël Rimbault, Julien Rouhaud, Stéphane Schildknecht, Julien Tachoires, Nicolas Thauvin, Cédric Villemain, Thibaud Walkowiak

---

À propos de DALIBO :

DALIBO est le spécialiste français de PostgreSQL. Nous proposons du support, de la formation et du conseil depuis 2005.

Retrouvez toutes nos formations sur https://dalibo.com/formations

LICENCE CREATIVE COMMONS BY-NC-SA 2.0 FR

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

Vous êtes autorisé :

- 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.

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

Chers lectrices & lecteurs,

Nos formations PostgreSQL sont issues de plus de 12 ans 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 !

Contents

Licence Creative Commons BY-NC-SA 2.0 FR **5**

1 Licence Creative Commons CC-BY-NC-SA **11**

2 Plan de migration **12**
2.1 Introduction 12
2.2 Méthodologie de migration 13
2.3 Recommandations et pièges à éviter 18
2.4 Migration du schéma et des données 29
2.5 Fonctionnalités problématiques 34
2.6 Conclusion 40
2.7 Travaux pratiques 41

3 Schéma et Données **50**
3.1 Introduction 50
3.2 Installation d'Ora2Pg 51
3.3 Configuration d'Ora2Pg 59
3.4 Validation de la configuration 64
3.5 Configuration générique 75
3.6 Migration du schéma 80
3.7 Migration des données 106
3.8 Conclusion 120
3.9 Travaux pratiques 121

4 Procédures stockées **138**
4.1 Introduction 138
4.2 Outils et méthodes 138
4.3 Différences de syntaxes 142
4.4 Conversion automatique du code 156
4.5 Migration des procédures stockées 165
4.6 Tests et validation 168
4.7 Conclusion 171
4.8 Travaux pratiques 172

5 Portage des requêtes SQL **179**
5.1 Introduction 179
5.2 Compatibilité avec Oracle 179
5.3 Types de données 180
5.4 Données temporelles 183

5.5 Expressions conditionnelles 184
5.6 ROWNUM 187
5.7 Jointures 189
5.8 HAVING et GROUP BY 191
5.9 Opérateurs ensemblistes 192
5.10 Transactions 192
5.11 Hiérarchies 197
5.12 Incompatibilités 202

1 LICENCE CREATIVE COMMONS CC-BY-NC-SA

Vous êtes libres de redistribuer et/ou modifier cette création selon les conditions suivantes :

- Paternité
- Pas d'utilisation commerciale
- Partage des conditions initiales à l'identique

Cette formation (diapositives, manuels et travaux pratiques) est sous licence **CC-BY-NC-SA**.

Vous êtes libres de redistribuer et/ou modifier cette création selon les conditions suivantes :

- Paternité
- Pas d'utilisation commerciale
- Partage des conditions initiales à l'identique

Vous devez citer le nom de l'auteur original de la manière indiquée par l'auteur de l'œuvre ou le titulaire des droits qui vous confère cette autorisation (mais pas d'une manière qui suggérerait qu'ils vous soutiennent ou approuvent votre utilisation de l'œuvre).

Vous n'avez pas le droit d'utiliser cette création à des fins commerciales.

Si vous modifiez, transformez ou adaptez cette création, vous n'avez le droit de distribuer la création qui en résulte que sous un contrat identique à celui-ci.

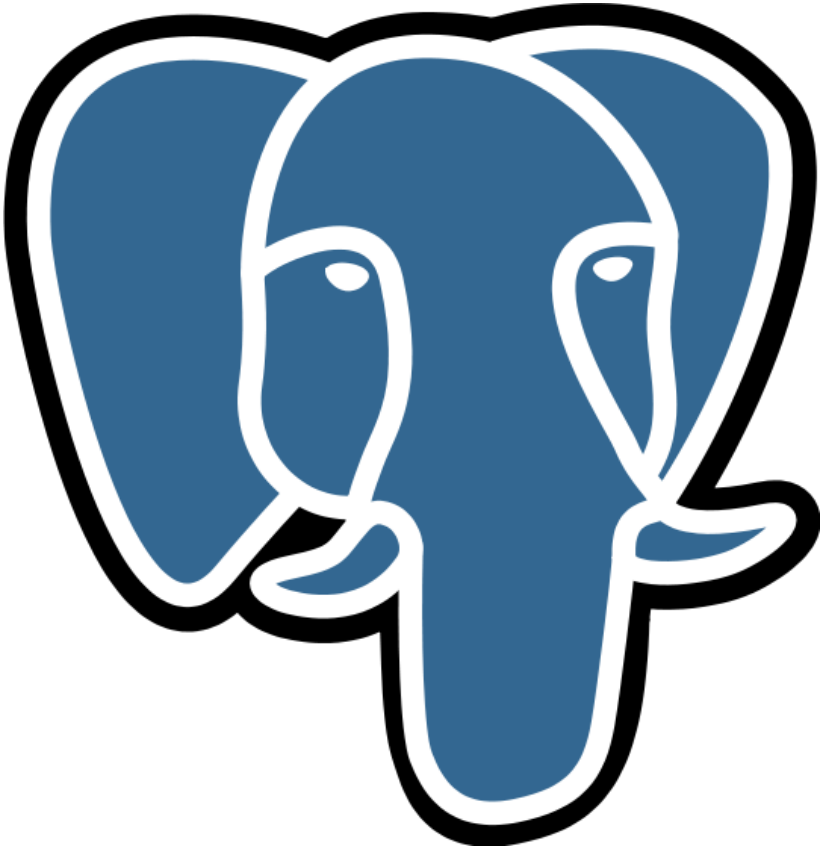
À chaque réutilisation ou distribution de cette création, vous devez faire apparaître clairement au public les conditions contractuelles de sa mise à disposition. La meilleure manière de les indiquer est un lien vers cette page web.

Chacune de ces conditions peut être levée si vous obtenez l'autorisation du titulaire des droits sur cette œuvre.

Rien dans ce contrat ne diminue ou ne restreint le droit moral de l'auteur ou des auteurs.

Le texte complet de la licence est disponible à cette adresse: http://creativecommons.org/licenses/by-nc-sa/2.0/fr/legalcode

2 PLAN DE MIGRATION



2.1 INTRODUCTION

Ce module est organisé en quatre parties :

- Méthodologie de la migration
- Recommandations et pièges à éviter
- Migration des données
- Fonctionnalités problématiques

Ce module est une introduction aux migrations de Oracle vers PostgreSQL. Nous y abordons comment gérer sa première migration (quelque soit le SGBD source et destination), des recommandations sur une migration d'Oracle vers PostgreSQL (où nous détaillons les pièges à éviter et les principales différences entre les deux SGBD), une réflexion sur le contenu de la migration et sur le choix de l'outil idoine, et nous finissons avec quelques détails sur des fonctionnalités manquantes ou implémentées différemment côté PostgreSQL qui rendront la migration plus difficile.

2.2 MÉTHODOLOGIE DE MIGRATION

La première migration est importante :

- Les méthodes employées seront réutilisées, améliorées…
- Un nouveau SGBD doit être supporté pendant de nombreuses années
- Elle influence la vision des utilisateurs vis-à-vis du SGBD
- Une migration ratée ou peu représentative est un argument pour les détracteurs du projet

La façon dont la première migration va se dérouler est essentielle. C'est sur cette expérience que les autres migrations seront abordées. Si l'expérience a été mauvaise, il est même probable que les migrations prévues après soient repoussées fortement, voire annulées.

Il est donc essentiel de réussir sa première migration. Réussir veut aussi dire bien la documenter car elle servira de base pour les prochaines migrations : les méthodes employées seront réutilisées, et certainement améliorées. Réussir veut aussi dire la publiciser, au moins en interne, pour que tout le monde sache que ce type de migration est réalisable et qu'une expérience est disponible en interne.

De plus, cette migration va influencer fortement la vision des développeurs et des utilisateurs vis-à-vis de ce SGBD. Réussir la migration veut donc aussi dire réussir à faire apprécier et accepter ce moteur de bases de données. Sans cela, il y a de fortes chances que les prochaines migrations ne soient pas demandées volontairement, ce qui rendra les migrations plus difficiles.

2.2.1 PROJET DE MIGRATION

Le projet doit être choisi avec soin :

- Ni trop gros (trop de risque)
- Ni trop petit (sans valeur)
- Transversal :
  - Implication maximale
  - Projet de groupe, pas individuel

Le premier projet de migration doit être sélectionné avec soin.

S'il est trop simple, il n'aura pas réellement de valeur. Par exemple, dans le cas d'une migration d'une base de 100 Mo, sans procédures stockées, sans fonctionnalités avancées, cela ne constituera pas une base qui permettra d' aborder tranquillement une migration d'une base de plusieurs centaines de Go et utilisant des fonctionnalités avancées.

L'inverse est aussi vrai. Un projet trop gros risque d'être un soucis. Prenez une base critique de plusieurs To, dotée d'un très grand nombre de procédures stockées. C'est un véritable challenge, y compris pour une personne expérimentée. Il y a de fortes chances que la migration soit longue, dure, mal vécue... et possiblement annulée à cause de sa complexité. Ceci aura un retentissement fort sur les prochaines migrations.

Il est préférable de choisir un projet un peu entre les deux : une base conséquence (plusieurs dizaines de Go), avec quelques procédures stockées, de la réplication, etc. Cela aura une vraie valeur, tout en étant à portée de main pour une première migration.

Une fois une telle migration réussie, il sera plus simple d'aborder correctement et sans crainte la migration de bases plus volumineuses ou plus complexes.

Il faut aussi ne pas oublier que la migration doit impliquer un groupe entier, pas seulement une personne. Les développeurs, les administrateurs, les équipes de support doivent tous être impliqués dans ce projet, pour qu'ils puissent intégrer les changements quant à l'utilisation de ce nouveau SGBD.

2.2.2 ÉQUIPE DU PROJET DE MIGRATION

- Chef de projet
- Équipe hétérogène (pas que des profils techniques)
- Recetteurs et utilisateurs nombreux (validation du projet la plus continue possible)

L'équipe du projet de migration doit être interne, même si une aide externe peut être sollicitée. Un chef de projet doit être nommé au sein d'une équipe hétérogène, composée de développeurs, d'administrateurs, de testeurs et d'utilisateurs. Il est à noter que les testeurs sont une partie essentielle de l'équipe.

2.2.3 EXPERTISE EXTÉRIEURE

- Société de service
- Contrat de support
- Expert PostgreSQL

Même si l'essentiel du projet est porté en interne, il est toujours possible de faire appel à une société externe spécialisée dans ce genre de migrations. Cela permet de gagner du temps sur certaines étapes de la migration pour éviter certains pièges, ou mettre en place l'outil de migration.

2.2.4 GESTION DE PROJET

- Réunions de lancement, de suivi
- Reporting
- Serveurs de projet
- …
- Pas un projet au rabais, ou un travail de stagiaire

Cette migration doit être gérée comme tout autre projet :

- une réunion de lancement ;
- des réunions de suivi ;
- des rapports d'avancements.

De même, ce projet a besoin de ressources, et notamment des serveurs de tests : par exemple un serveur Oracle contenant la base à migrer (mais qui ne soit pas le serveur de production), et un serveur PostgreSQL contenant la base à migrer. Ces deux serveurs doivent avoir la volumétrie réelle de la base de production, sinon les tests de performances n'auront pas vraiment de valeur.

En fait, il faut vraiment que cette migration soit considérée comme un vrai projet, et pas comme un projet au rabais, ce qui arrive malheureusement assez fréquemment. C'est une opération essentielle, et des ressources compétentes et suffisantes doivent être offertes pour la mener à bien.

2.2.5 PASSER À POSTGRESQL

- Ce n'est pas une révolution

- Le but est de faire des économies …
- … sans chamboulement

En soi, passer à PostgreSQL n'est pas une révolution. C'est un moteur de bases de données comme les autres, avec un support du SQL (et quelques extensions) et ses fonctionnalités propres. Ce qui change est plutôt l'implémentation mais, comme nous le verrons dans cette formation, si une fonctionnalité identique n' existe pas, une solution de contournement est généralement disponible.

La majorité des utilisateurs de PostgreSQL vienne à PostgreSQL pour faire des économies (sur les coûts de licence). Si jamais une telle migration demandait énormément de changements, ils ne viendraient pas à PostgreSQL. Or la majorité des migrations se passe bien, et les utilisateurs restent ensuite sur PostgreSQL. Les migrations qui échouent sont généralement celles qui n'ont pas été correctement gérées dès le départ (pas de ressources pour le projet, un projet trop gros dès le départ, etc.).

2.2.6 MOTIVER

- Formation indispensable
- Divers cursus
  - du chef de projet au développeur
- Adoption grandissante de PostgreSQL
  - pérennité

Le passage à un nouveau SGBD est un peu un saut dans l'inconnu pour la majorité des personnes impliquées. Elles connaissent bien un moteur de bases de données et souvent ne comprennent pas pourquoi on veut les faire passer à un autre moteur. C'est pour cela qu'il est nécessaire de les impliquer dès le début du projet et, le cas échéant, de les former. Il est possible d'avoir de nombreuses formations autour de PostgreSQL pour les différents acteurs : chefs de projet, administrateurs de bases de données, développeurs, etc.

2.2.7 VALORISER

- Concepts PostgreSQL très proches des SGBD propriétaires
  - Adapter les compétences
  - Ne pas tout reprendre à zéro

De toute façon, les concepts utilisés par PostgreSQL sont très proches des concepts des moteurs SGBD propriétaires. La majorité du temps, il suffit d' adapter les compétences.

Il n'est jamais nécessaire de reprendre tout à zéro. La connaissance d'un autre moteur de bases de données permet de passer très facilement à PostgreSQL, ce qui valorise l'équipe.

2.2.8 GESTION DES DÉLAIS

Souvent moins important:

- Le service existe déjà
- Donner du temps aux acteurs

Contrairement à d'autres projets, le service existe déjà. Les délais sont donc généralement moins importants, ce qui permet de donner du temps aux personnes impliquées dans le projet pour fournir une migration de qualité (et surtout documenter cette opération).

2.2.9 COÛTS

- Bugdet ?
- Open source <> gratuit
  - Coûts humains
  - Coûts matériels

Une migration aura un coût important. Ce n'est pas parce que PostgreSQL est un logiciel libre que tout est gratuit. La mise à disposition de ressources humaines et matérielles aura un coût. La formation du personnel aura un coût. Mais ce coût sera amoindri par le fait que, une fois cette migration réalisée, les prochaines migrations n'auront un coût qu'au niveau matériel principalement.

2.2.10 QUALITÉ

- Cruciale
  - La réussite est obligatoire
- Le travail effectué doit être réutilisable
- Ou tout du moins l'expérience et les méthodologies

La qualité de la première migration est cruciale. Si le but est de migrer les autres bases de données de l'entreprise, il est essentiel que la première migration soit une réussite totale. Il est essentielle qu'elle soit documentée, discutée, pour que le travail effectué

soit réutilisable (soit complètement, soit uniquement l'expérience et les méthodes) et que la prochaine migration soit ainsi moins coûteuse.

2.2.11 BUT DE LA PREMIÈRE MIGRATION

- Privilégier la qualité
- Contrôler les coûts
- N'est souvent pas contrainte par des délais stricts

Pour résumer, la première migration doit être suffisamment simple pour ne pas être un échec et suffisamment complexe pour être en confiance pour les prochaines migrations. Il est donc essentiel de bien choisir la base de sa première migration.

Il est aussi essentiel d'avoir des ressources humaines et matérielles suffisantes, tout en contrôlant les coûts.

Enfin, il est important de ne pas stresser les acteurs de cette migration avec des délais difficiles à tenir. Le service est déjà présent et fonctionnel, la première migration doit être un succès si on veut continuer, autant donner du temps aux équipes responsables de la migration.

2.3 RECOMMANDATIONS ET PIÈGES À ÉVITER

Oracle et PostgreSQL sont assez proches :

- Tous deux des SGBDR
- Le langage d'accès aux données est SQL
- Les deux ont des connecteurs pour la majorité des langages (Java, C, .Net…)
- Les langages embarqués sont différents
- C'est dans les détails que se trouvent les problèmes
- Ce document ne peut pas être exhaustif !

Les SGBD Oracle et PostgreSQL partagent beaucoup de fonctionnalités. Même si l'implémentation est différente, les fonctionnalités se ressemblent beaucoup.

Tous les deux sont des systèmes de gestion de bases de données relationnelles. Tous les deux utilisent le langage SQL (leur support de la norme diffère évidemment). Tous les deux ont des connecteurs pour la majorité des langages actuels (l'efficacité et le support des fonctionnalités du moteur dépendent de l'implémentation des connecteurs).

Par contre, les langages autorisés pour les procédures stockées sont différents, y compris ceux disponibles par défaut.

Même si les fonctionnalités majeures sont présentes dans les deux moteurs, les détails d'implémentation et de mise en place sont le cœur du problème. Par exemple, les tablespaces sont disponibles dans les deux SGBD mais l' implémentation est différente. Pour Oracle, il s'agit d'un fichier qu'il faut dimensionner. Pour PostgreSQL, il s'agit d'un répertoire dont la taille est libre. Il y a des avantages et des inconvénients à chaque implémentation. Au niveau d'Oracle, il suffit de surveiller l'espace disque au niveau du tablespace. Avec PostgreSQL, la supervision se fait au niveau du système d'exploitation. C'est plutôt un défaut de PostgreSQL car, de nos jours, les équipes système et base de données sont séparées. Par contre, le problème au niveau Oracle vient du dimensionnement des tablespaces. Il faut surveiller plus fortement au niveau Oracle car il est nécessaire d'agrandir le tablespace si jamais ce dernier vient à être utilisé complètement. Il n'y a pas ce problème avec PostgreSQL, tout du moins tant qu'il reste de l'espace disque.

Cette partie est donc consacrée à la revue des pièges à éviter et à montrer les différences d'implémentation pouvant susciter des problèmes ou des incompréhensions lors d'une migration.

2.3.1 POINTS COMMUNS

PostgreSQL et Oracle :

- Ont le même langage d'accès aux données (SQL)
  - mais des « variantes » différentes (extensions au standard)
- De nombreux concepts en commun:
  - Transactions et *savepoints*
  - MVCC et verrouillage
- Conservation
  - des logiques applicative et algorithmique
  - de l'architecture applicative

PostgreSQL et Oracle partagent le même langage d'accès et de définition des données. La norme SQL est plutôt bien suivie par ces deux SGBD. Néanmoins, certains moteurs se permettent des écarts par rapport à la norme, parfois pour gagner en performances, mais surtout pour faciliter la vie des développeurs. Cela fait que beaucoup de développeurs utilisent ces écarts à la norme, parfois sans le savoir. Lors d'une migration, cela pose beaucoup de problèmes si de tels écarts sont utilisés car les autres moteurs de bases de données ne les implémentent pas tous (si tant est qu'ils en aient le droit). PostgreSQL

essaie, quand cela est possible, de supporter les extensions à la norme réalisés par les autres moteurs. Les développeurs de PostgreSQL s'assurent que si une telle extension est ajoutée, la version proposée par la norme soit elle-aussi possible.

Ils partagent aussi certains concepts, comme les transactions et les points de retournements (savepoint), MVCC et la gestion des verrous. Cela permet de conserver les logiques applicative et algorithmique, au moins jusqu'à une certaine mesure.

2.3.2 DIFFÉRENCES POUR L'ARCHITECTURE APPLICATIVE

- Les transactions ne sont pas démarrées implicitement sous PostgreSQL
  - `BEGIN`
  - Sauf avec JDBC (`BEGIN` caché)
- Toute erreur non gérée dans une transaction entraîne son annulation
  - Oracle revient à l'état précédent de l'ordre en échec
  - PostgreSQL plus strict de ce point de vue

Pour PostgreSQL, si vous souhaitez pouvoir annuler des modifications, vous devez utiliser `BEGIN` avant d'exécuter les requêtes de modification. Toute transaction qui commence par un `BEGIN` doit être validée avec COMMIT ou annulée avec `ROLLBACK`. Si jamais la connexion est perdue entre le serveur et le client, le `ROLLBACK` est automatique.

Par exemple, si on insère une donnée dans une table, sans faire de `BEGIN` avant, et qu'on essaie d'annuler cette insertion, cela ne fonctionnera pas :

```
dev2=# CREATE TABLE t1(id integer);
CREATE TABLE
dev2=# INSERT INTO t1 VALUES (1);
INSERT 0 1
dev2=# ROLLBACK;
NOTICE:  there is no transaction in progress
ROLLBACK
dev2=# SELECT * FROM t1;
 id
----
  1
(1 row)
```

Par contre, si j'intègre un `BEGIN` avant, l'annulation se fait bien :

```
dev2=# BEGIN;
BEGIN
dev2=# INSERT INTO t1 VALUES (2);
INSERT 0 1
```

```
dev2=# ROLLBACK;
ROLLBACK
dev2=# SELECT * FROM t1;
 id
----
  1
(1 row)
```

Autre différence au niveau transactionnel : il est possible d'intégrer des ordres DDL dans des transactions. Par exemple :

```
dev2=# BEGIN;
BEGIN
dev2=# CREATE TABLE t2(id integer);
CREATE TABLE
dev2=# INSERT INTO t2 VALUES (1);
INSERT 0 1
dev2=# ROLLBACK;
ROLLBACK
dev2=# INSERT INTO t2 VALUES (2);
ERROR:  relation "t2" does not exist
LINE 1: INSERT INTO t2 VALUES (2);
                    ^
```

Enfin, quand une transaction est en erreur, vous ne sortez pas de la transaction. Vous devez absolument exécuter un ordre de fin de transaction (`COMMIT` ou `ROLLBACK`, peu importe, un `ROLLBACK` sera exécuté) :

```
dev2=# BEGIN;
BEGIN
dev2=# INSERT INTO t2 VALUES (2);
ERROR:  relation "t2" does not exist
LINE 1: INSERT INTO t2 VALUES (2);
                    ^
dev2=# INSERT INTO t1 VALUES (2);
ERROR:  current transaction is aborted, commands ignored until
        end of transaction block
dev2=# SELECT * FROM t1;
ERROR:  current transaction is aborted, commands ignored until
        end of transaction block
dev2=# ROLLBACK;
ROLLBACK
dev2=# SELECT * FROM t1;
 id
----
  1
(1 row)
```

2.3.3 DIFFÉRENCES SUR LES TYPES NUMÉRIQUES

- Oracle ne gère pas les types numériques « natifs » SQL :
  - `smallint`, `integer`, `bigint`
- Le type numeric du standard SQL est appelé `number` sous Oracle

Les types `smallin`, `integer`, `bigint`, `float`, `real`, `double precision` sont plus rapides que le type `numeric` sous PostgreSQL : ils utilisent directement les fonctions câblées des processeurs. Il faut donc les privilégier.

2.3.4 DIFFÉRENCES SUR LE TYPE BOOLÉEN

- Oracle n'a pas de type `boolean`.
- Attention aux ORM (Hibernate) suite à la migration de données
  - ils chercheront un `boolean` sous PostgreSQL alors que vous aurez migré un `int`

Oracle ne dispose pas du type booléen. Du coup, les développeurs utilisent fréquemment un entier qu'ils mettront à 0 pour `FALSE` et à 1 pour `TRUE`. Un système de migration ne saura pas dire si cette colonne de type `numeric` est, pour le développeur, un booléen ou une valeur entière. Du coup, le système de migration utilisera le typage de la colonne, à savoir entier. Or, les ORM chercheront un booléen parce que le code applicatif indique un booléen. Cela provoquera une erreur sur PostgreSQL, comme le montre l'exemple suivant :

```
dev2=# INSERT INTO t1 VALUES (true);
ERROR:  column "id" is of type integer but expression is of type boolean
LINE 1: insert into t1 values (true);
                               ^
HINT:  You will need to rewrite or cast the expression.
dev2=# INSERT INTO t1 VALUES ('t');
ERROR:  invalid input syntax for integer: "t"
LINE 1: insert into t1 values ('t');
                               ^
dev2=# CREATE TABLE t3 (c1 boolean);
CREATE TABLE
dev2=# INSERT INTO t3 VALUES (true);
INSERT 0 1
dev2=# INSERT INTO t3 VALUES ('f');
INSERT 0 1
```

```
dev2=# SELECT * FROM t3;
 c1
----
 t
 f
(2 rows)
```

2.3.5 DIFFÉRENCES SUR LES TYPES CHAÎNES

- Pas de `varchar2` dans PostgreSQL
  - le type est `varchar`
- Attention, sous Oracle, `''` = `IS NULL`
  - sous PostgreSQL, `''`et `NULL` sont distincts
- `varchar` peut ne pas prendre de taille sous PostgreSQL
  - 1 Go maximum dans ce cas
- Il existe aussi un type `text` équivalent à `varchar` sans taille
- Un seul encodage par base
- Collationnements
  - Par instance (avant la 8.4), par base de données (depuis la 8.4), par colonne (depuis la 9.1)

Au niveau de PostgreSQL, il existe trois types de données pour les chaînes de caractères : `char`, `varchar` et `text`. Le type `varchar2` d'Oracle est l'équivalent du type varchar de PostgreSQL. Il est possible de ne pas donner de taille à une colonne de type varchar, ce qui revient à la déclarer de type text. Dans ce cas, la taille maximale est de 1 Go. Suivant l'encodage, le nombre de caractères intégrables dans la colonne diffère.

La grosse différence entre PostgreSQL et Oracle pour les chaînes de caractères tient dans la façon dont les chaînes vides sont gérées. Oracle ne fait pas de différence entre une chaîne vide et une chaîne `NULL`. PostgreSQL fait cette différence. Du coup, tous les tests de chaînes vides effectuées avec un `IS NULL` et tous les tests de chaînes `NULL` effectués avec une comparaison avec une chaîne vide ne donneront pas le même résultat avec PostgreSQL. Ces tests doivent être vérifiés systématiquement par les développeurs d'applications et de procédures stockées.

```
dev2=# SELECT cast('' AS varchar) IS NULL;
 ?column?
----------
 f
(1 row)
```

17.12

Au niveau encodage, PostgreSQL n'accepte qu'un encodage par base de données (l'encodage par défaut est UTF-8). Il accepte par contre plusieurs collationnements depuis la 8.4. Depuis la 9.1, il est même possible d'indiquer le collationnement dans les requêtes (au niveau d'un `ORDER BY` ou d'un `CREATE INDEX`).

2.3.6 DIFFÉRENCES SUR LES TYPES BINAIRES

- 2 implémentations différentes sous PostgreSQL
  - large objects et fonctions `lo_*`
  - `bytea`

L'implémentation des types binaires sur PostgreSQL est très particulière. De plus, elle est double, dans le sens où vous avez deux moyens d'importer et d' exporter des données binaires dans PostgreSQL.

La première, et plus ancienne, implémentation concerne les Large Objects. Cette implémentation dispose d'une API spécifique. Il ne s'agit pas à proprement parlé d'un type de données. Il faut passer par des procédures stockées internes qui permettent d'importer, d'exporter, de supprimer, de lister les Large Objects. Après l'import d'un Large Object, vous récupérez un identifiant que vous pouvez stocker dans une table utilisateur (généralement dans une colonne de type `OID`). Vous devez utiliser cet identifiant pour traiter l'objet en question (export, suppression, etc). Cette implémentation a de nombreux défauts, qui fait qu'elle est rarement utilisée. Parmi les défauts, notons que la suppression d'une ligne d'une table utilisateur référençant un Large Object ne supprime pas le Large Object référencé. Notons aussi qu'il est bien plus difficile d'interagir et de maintenir une table système. Notons enfin que la sauvegarde avec `pg_dump` est plus complexe et plus longue si des Larges Objects sont dans la base à sauvegarder. Son principal avantage sur la deuxième implémentation est la taille maximale d'un Large Object : 4 To depuis la 9.3 (2 Go avant).

La deuxième implémentation est un type de données appelé `bytea`. Comme toutes les colonnes dans PostgreSQL, sa taille maximale est 1 Go, ce qui est inférieur à la taille maximale d'un Large Object. Cependant, c'est son seul défaut.

Bien que l'implémentation des Large Objects est en perte de vitesse à cause des nombreux inconvénients inhérents à son implémentation, elle a été l' objet d'améliorations sur les dernières versions de PostgreSQL : gestion des droits de lecture ou écriture des Large Objects, notion de propriétaire d'un Large Object, limite de taille relevée à 4 To. Elle n'est donc pas obsolète.

2.3.7 DIFFÉRENCES SUR LES TYPES SPÉCIALISÉS

PostgreSQL fournit aussi de nombreux types de données spécialisés :

- Gestion des timestamps et intervals avec opérations arithmétiques
- Plans d'adressage IP (CIDR) et opérateurs de masquage
- Grande extensibilité des types: il est très facile d'en rajouter un nouveau
  - `PERIOD`
  - `ip4r`
  - etc.

L'un des gros avantages de PostgreSQL est son extensibilité. Mais même sans cela, PostgreSQL propose de nombreux types natifs qui vont bien au-delà des types habituels. Ce sont des types métiers, pour le réseau, la géométrie, la géographie, la gestion du temps, la gestion des intervalles de valeurs, etc.

Il est donc tout à fait possible d'améliorer une application en passant sur des types spécialisés de PostgreSQL.

2.3.8 DIFFÉRENCES ENTRE LES TYPES DATES

- Date
  - sous Oracle : `YYYY/MM/DD HH:MM:SS`
  - sous PostgreSQL: `YYYY-MM-DD` (conforme SQL)
- Time
  - sous Oracle : `YYYY/MM/DD HH:MM:SS`
  - sous PostgreSQL : `HH:MM:SS.mmmmmmm` (µs)
- Gestion des fuseaux horaires
  - sous PostgreSQL, par défaut
  - `timestamp` sous PostgreSQL : Date+Time (+TZ)
- Format de sortie conforme SQL sous PostgreSQL :

```
YYYY-MM-DD HH24:MI:SS.mmmmmmm+TZ
```

- Type interval
  - Date1-Date2 => `Interval`

Oracle a tendance à mélanger un peu tous les types dates. Ce n'est pas le cas au niveau de PostgreSQL. Une colonne de type `date` au niveau de PostgreSQL contient seulement une date, il n'y a pas d'heure ajoutée. Une colonne de type `time` au niveau de PostgreSQL

contient seulement un horodatage (heure, minute, seconde, milliseconde), mais pas de date.

Par défaut, PostgreSQL intègre le fuseau horaire dans les types `timestamp` (date et heure). Le stockage est fait en UTC, mais la restitution dépend du fuseau horaire indiqué par le client.

2.3.9 DIFFÉRENCES DU LANGAGE SQL - 1

Oracle : nombreuses extensions incompatibles avec PostgreSQL :

- jointure (+)
  - réécrite en `LEFT JOIN` (standard SQL)
- `CONNECT BY`
  - réécrite avec `WITH RECURSIVE` (standard SQL)
- Nombreuses fonctions telles que NVL
  - `COALESCE`, `CASE` (standard SQL)

Oracle contient de nombreuses extensions (lisez incompatibilité) avec la norme SQL.

Une ancienne écriture des jointures utilisait le signe +. Ces requêtes doivent être réécrites en utilisant la notation standard (`LEFT JOIN`) du standard SQL.

Oracle utilise le `CONNECT BY` pour permettre l'écriture de requêtes récursives. PostgreSQL supporte les requêtes récursives depuis la 8.3, mais avec la syntaxe de la norme SQL, à savoir `WITH RECURSIVE`. Cette syntaxe est décrite dans le manuel de PostgreSQL (http://docs.postgresql.fr/9.3/queries-with.html).

De nombreuses fonctions ont des noms différents entre Oracle et PostgreSQL. Par exemple, la fonction `NVL` sous Oracle s'appelle `coalesce` sous PostgreSQL.

2.3.10 DIFFÉRENCES DU LANGAGE SQL - 2

- Casse par défaut du nom des objets différente entre Oracle et PostgreSQL :
- Si casse non spécifiée :
  - majuscule sous Oracle
  - minuscule sous PostgreSQL
- Forcer la casse
  - `" "` autour des identifiants.

La casse par défaut des objets est différente entre Oracle et PostgreSQL. C'est d'ailleurs un exemple où Oracle respecte mieux le standard SQL que PostgreSQL. Si la casse n'est pas spécifiée, le nom de l'objet sera en majuscule sous Oracle et en minuscule sous PostgreSQL. Pour forcer la casse, il faudra utiliser des guillemets doubles, comme le montre cet exemple :

```
dev2=# CREATE TABLE toto(id integer);
CREATE TABLE
dev2=# CREATE TABLE TitI(id integer);
CREATE TABLE
dev2=# \d
         List of relations
 Schema | Name | Type  |   Owner
--------+------+-------+-----------
 public | t1   | table | guillaume
 public | t3   | table | guillaume
 public | titi | table | guillaume
 public | toto | table | guillaume
(4 rows)

dev2=# CREATE TABLE "TitI"(id integer);
CREATE TABLE
dev2=# \d
         List of relations
 Schema | Name | Type  |   Owner
--------+------+-------+-----------
 public | TitI | table | guillaume
 public | t1   | table | guillaume
 public | t3   | table | guillaume
 public | titi | table | guillaume
 public | toto | table | guillaume
(5 rows)
```

2.3.11 DIFFÉRENCES DU LANGAGE SQL - 3

- Il est très difficile de détecter tous les problèmes de langage SQL
- Le plus simple est :
  - de faire fonctionner l'application
  - de repérer tous les ordres SQL en erreur (ils sont tracés dans les journaux applicatifs)
  - de les corriger
- Attention aux mots réservés

17.12

Une simple lecture du code source est facilement source d'erreurs. Bien qu'il soit toujours intéressant de faire une première passe pour corriger les cas les plus flagrants, il est nécessaire ensuite de tester l'application sur PostgreSQL et de vérifier dans les journaux applicatifs les messages d'erreurs qui surviennent. Un outil comme pgbadger permet de récupérer les erreurs, leur fréquence et les requêtes qui ont causé les erreurs.

Pour récupérer la liste des mots réservés :

```
SELECT pg_get_keywords();
```

2.3.12 DIFFÉRENCES DU LANGAGE PL - 1

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 procédures stockées sont différents entre Oracle et PostgreSQL. Même si PL/pgSQL est un langage assez proche de PL/SQL, cela demandera une revue des procédures stockées et un recodage (automatique ou manuel) des procédures.

2.3.13 DIFFÉRENCES DU LANGAGE PL - 2

- PL/PgSQL est conçu pour être ressemblant à PL/SQL
  - Pas de package
  - Compilé à la première exécution, pas de façon globale
  - Pas de transaction autonome
  - Pas de fonctionnalités comme les directories : PL/PgSQL ne manipule pas de fichiers
  - Les autres langages PL comblent ce manque (et plus)
- Le gros du travail de portage !

PostgreSQL dispose de plusieurs langages de procédures stockées. Le langage PL/pgSQL est livré et activé par défaut depuis la version 9.0. Il est très proche du langage PL/SQL. Cela n'empêche qu'un travail d'adaptation sera nécessaire. Certaines fonctionnalités ou objects manquent : PostgreSQL ne dispose pas de packages, il ne dispose pas des transactions autonomes, il ne dispose pas de certaines fonctionnalités comme les directories (PL/pgSQL est un langage dit sûr dans le sens où il n'a accès qu'à la base de données via le langage SQL). Il existe évidemment des contournements à ces différents manques

(respectivement les schémas, `dblink`, et les langages du types PL/perl ou PL/python). Cependant, cela demande un travail d'adaptation plus important qu'il faut prendre en compte dans le cadre d'une migration. C'est en fait là que se situe le plus gros du travail de portage.

2.4 MIGRATION DU SCHÉMA ET DES DONNÉES

Avant de pouvoir porter l'application et le PL :

- Migrer le schéma
- Migrer les données

Avant de pouvoir traiter le code, qu'il soit applicatif ou issu des procédures stockées, il faut procéder à la migration du schéma et des données. C'est donc ce dont nous allons parler dans cette partie.

2.4.1 BESOINS DE LA MIGRATION : SCHÉMA

- Veut-on migrer le schéma tel quel ?
- Utiliser les fonctionnalités de PostgreSQL ?
  - N'est plus vraiment iso-fonctionnalité
- Créer un nouveau schéma :
  - D'un coup
  - Les tables d'abord, les index et contraintes ensuite ?

La première question à se poser concerne le schéma : veut-on le migrer tel quel ? Le changer peut permettre d'utiliser des fonctionnalités plus avancées de PostgreSQL. Cela peut être intéressant pour des raisons de performances, mais a comme inconvénient de ne plus être une migration iso-fonctionnelle.

Généralement, il faudra créer un nouveau schéma, et intégrer les objets par étapes : tables, index, puis contraintes.

2.4.2 BESOINS DE LA MIGRATION : TYPES

- On rencontre souvent les types suivant sous Oracle :
  - `number(18,0)`, `number(4,0)` ...

  - `int` : `-2147483648` à `+2147483647` (4 octets, `number(9,0)`)
  - `bigint` : `-9223372036854775808` à `9223372036854775807` (8 octets, `number(18,0)`)
- Type natifs bien plus performants (gérés par le processeur, taille fixe)
- Certains outils migrent en `numeric(x,0)`, d'autres en `int`/`bigint`
  - Peut être un critère de choix

Oracle utilise généralement number pour les types entiers. L'équivalent strict au niveau PostgreSQL est `numeric` mais il est préférable de passer à d'autres types de données comme `int` (un entier sur quatre octets) ou `bigint` (un entier sur huit octets) qui sont bien plus performants.

L'outil pour la migration devra être sélectionné suivant ses possibilités au niveau de la transformation de certains types en d'autres types, si jamais il est décidé de procéder ainsi.

2.4.3 BESOINS DE LA MIGRATION : AUTRES TYPES

- Types plein texte ?
- Blob ?
- GIS ?
- ...
- Un développement peut être nécessaire pour des types spéciaux

L'outil de migration doit pouvoir aussi gérer des types particuliers, comme les types spécifiques à la recherche plein texte, ceux spécifiques aux objets binaires, ceux spécifiques à la couche spatiale, etc. Il est possible qu'un développement soit nécessaire pour faciliter la migration. Un outil libre est préférable dans ce cas.

2.4.4 BESOINS DE LA MIGRATION

- Déclarer les tables
- Les remplir
- Puis seulement déclarer les index, PK, FK, contraintes...
- Performances...

Pour des raisons de performances, il est toujours préférable de ne déclarer les index et les contraintes qu'une fois les tables remplies. L'outil de migration doit aussi prendre cela

en compte : création des tables, remplissage des tables et enfin création des index et contraintes.

2.4.5 MIGRATION DES DONNÉES

Veut-on :

- Migrer en une seule fois les données ? (« Big Bang »)
- Pouvoir réaliser des incréments ?
- Paralléliser sur plusieurs sessions/theads ?
- Modifier des données « au passage » ?

Toujours dans les décisions à prendre avant la migration, il est important de savoir si on veut tout migrer d'un coup ou le faire en plusieurs étapes. Les deux possibilités ont leurs avantages et inconvénients.

De même, souhaite-t-on paralléliser l'import et l'export ? de ce choix dépend principalement l'outil que l'on va sélectionner pour la migration. Jusqu'à peu, Ora2Pg ne proposait pas la parallélisation, et seuls les ETL le permettaient.

Enfin, souhaite-t-on modifier des données lors de l'opération de migration ? là-aussi, cela peut se concevoir, notamment si certains types de données sont modifiés. Mais c'est une décision à prendre lors des premières étapes du projet.

2.4.6 CHOIX DE L'OUTIL

Suivant les réponses aux questions précédentes, vous choisirez :

- Ora2Pg
- Un ETL :
  - Kettle (Pentaho Data Integrator)
  - Talend
- De développer votre propre programme
- De mixer les solutions

Après avoir répondu aux questions précédentes et évalué la complexité de la migration, il sera possible de sélectionner le bon outil de migration. Il en existe différents, qui répondront différemment aux besoins.

17.12

Ora2Pg est un outil libre développé par Gilles Darold. Le rythme de développement est rapide. De nouvelles fonctionnalités sont proposées rapidement, suivant les demandes des utilisateurs, les nouveautés dans PostgreSQL et les découvertes réalisées par son auteur.

Les ETL sont intéressants pour les possibilités plus importantes. Ora2Pg ne fait que de la conversion Oracle vers PostgreSQL, MySQL aussi maintenant, alors que les ETL autorisent un plus grand nombre de source de données et de destination, si bien que l'expérience acquise pour la migration d'Oracle vers PostgreSQL peut être réutilisé pour réaliser la migration d'un autre moteur vers un autre moteur ou pour l'import ou l'export de données.

Il est aussi possible de développer sa propre solution si les besoins sont vraiment spécifiques au métier, voire de mixer différentes solutions. Par exemple, il était intéressant d'utiliser Ora2Pg pour la transformation du schéma et un ETL pour un export et import des données parallélisés.

2.4.7 ORA2PG - INTRODUCTION

- En Perl
- Se connecte à Oracle
- Génère un fichier SQL compatible avec PostgreSQL, en optimisant les types
- Conversion automatique d'une partie du code PL/SQL en PL/pgSQL
- Simple de mise en œuvre
- Rapide au chargement (utilise `COPY`)

Ora2Pg est un outil écrit en Perl. Il se connecte à Oracle via le connecteur Perl pour Oracle. Après analyse des catalogues systèmes Oracle et lecture de son fichier de configuration, il est capable de générer un fichier SQL compatible avec PostgreSQL ou de se connecter à une base PostgreSQL pour y exécuter ce script. Dans les dernières versions, il est même capable de convertir automatiquement une partie du code PL/SQL d'Oracle vers du PL/pgSQL sur PostgreSQL.

L'outil est plutôt simple de mise en œuvre et de prise en main. Il est rapide au chargement, notamment grâce à sa gestion de la commande `COPY`.

2.4.8 ORA2PG - DÉFAUTS

- Big-Bang

  - pas d'incrémental

Pour aborder immédiatement les inconvénients de Ora2Pg, il ne propose pas de solution incrémentale : c'est tout ou partie d'une table ou rien.

2.4.9 ORA2PG - FONCTIONNALITÉS

- Exporte tout le schéma Oracle :
  - tables, vues, séquences, contraintes d'intégrité, trigger, etc.
  - utilisateurs et droits
- Gère la conversion des types
  - `blob` et `clob` -> `bytea` et `text`
  - `number` -> `int`, `bigint`, `real`, `double`, `decimal`
- Réécrit les entêtes de fonction correspondant aux fonctions Oracle
- Aide à :
  - la conversion PL/SQL -> PLpgSQL
  - au partitionnement (par héritage, dans PostgreSQL)

Ora2Pg dispose néanmoins de nombreuses fonctionnalités. Il est capable d' exporter tout le schéma de données Oracle. Il est capable de convertir utilisateurs et droits sur les objets. Il convertit aussi automatiquement la conversion des types de données. Enfin, il s'occupe de la déclaration et du code des procédures stockées (uniquement PL/SQL vers PL/pgSQL). Il propose aussi une aide au partitionnement, dont l'implémentation est vraiment différente entre Oracle et PostgreSQL.

2.4.10 LES ETL - AVANTAGES

- Spécialisés dans la transformation et le chargement de données
- Rapides (cœur de métier)
- Parallélisables
- Très souples sur la transformation
- Migration incrémentale possible (fusion, *slow changing dimensions*, etc...)

Les ETL sont spécialisées dans la transformation et le chargement des données. Ils permettent la parallélisation pour leur traitement, ils sont très souples au niveau de la transformation de données. Tout cela leur permet d'être très rapide, certaines fois plus que ne le peut Ora2Pg.

De plus, ils permettent de faire de la migration incrémentale.

2.4.11 LES ETL - INCONVÉNIENTS

- Migration sommaire du schéma
  - quand c'est supporté
- Beaucoup de travail de paramétrage
  - peut-être 200 jobs à créer si 200 tables…
- Apprentissage long
  - outil complexe et riche fonctionnellement

La migration du schéma est au mieux sommaire, voire inexistante. Ce n'est clairement pas la fonctionnalité visée par les ETL.

Le paramétrage d'un ETL est souvent très long. Si vous devez migrer les données de 200 tables, vous aurez 200 jobs à créer. Dans ce cas, Ora2Pg est bien plus intéressant, vu que la migration de la totalité des tables est l' option par défaut.

Ce sont des outils riches et, du coup, complexes. Cela demandera un apprentissage bien plus long que pour Ora2Pg. Cependant, ils sont utilisables dans bien plus de cas que Ora2Pg.

2.5 FONCTIONNALITÉS PROBLÉMATIQUES

Lors de la migration, certaines fonctionnalités d'Oracle auront peu ou pas d'équivalent :

- Vues matérialisées
- Partitionnement
- Synonyme
- Conversion de type implicite
- Absence de *hint* (tag de requête)

Certaines fonctionnalités Oracle n'ont pas d'équivalents natifs dans PostgreSQL. Nous allons étudier les deux cas les plus fréquents : les vues matérialisées et le partitionnement.

2.5.1 VUES MATÉRIALISÉES - 1

- Sous Oracle :
  - Stocke le résultat d'une vue physiquement (table)

  - Permet la création d'index sur cette table
  - Est mise à jour au fil de l'eau ou à intervalle régulier
  - Réécriture transparente de requêtes
- Sous PostgreSQL :
  - N'existe qu'à partir de la version 9.3
  - Mise à jour uniquement sur demande
  - Pas de réécriture

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 réécriture de requête. La situation devrait cependant s'arranger au fil des versions. La version 9.4 apporte la mise à jour non bloquante des vues matérialisées.

2.5.2 VUES MATÉRIALISÉES - 2

Certaines choses peuvent être émulées :

- Vues matérialisées à mise à jour synchrone
  - utilisation d'un trigger
- Vues matérialisées à mise à jour asynchrone
  - tracer les changements, et appliquer à intervalle régulier (trigger + fonction)
- Vues matérialisées reconstruites à intervalle régulier
  - réexécuter un `CREATE TABLE AS SELECT` régulièrement
- Réécriture automatique des requêtes
  - pas de solution
  - réécriture manuelle des requêtes

PostgreSQL étant particulièrement extensible, il est possible de contourner ces manques, y compris pour les versions antérieures à la 9.3.

Avant la 9.3, il fallait créer une vraie table pour la vue matérialisée, et un trigger sur la table source pour alimenter la table destination.

```
dev2=# CREATE TABLE t1 (id integer);
CREATE TABLE
dev2=# INSERT INTO t1 VALUES (1);
INSERT 0 20
```

```
dev2=# CREATE TABLE v1 (id integer);
CREATE TABLE
dev2=# INSERT INTO v1 SELECT * FROM t1 WHERE id<10;
INSERT 0 1
dev2=# CREATE FUNCTION ins_v1() RETURNS trigger LANGUAGE plpgsql
AS $$
BEGIN
  IF new.id < 10 THEN
     INSERT INTO v1 VALUES (new.*);
  END IF;
  RETURN new;
END
$$;
CREATE FUNCTION
dev2=# CREATE TRIGGER tr1 AFTER INSERT ON t1 FOR EACH ROW
       EXECUTE PROCEDURE ins_v1();
CREATE TRIGGER
dev2=# SELECT * FROM t1;
 id
----
  1
(1 row)

dev2=# SELECT * FROM v1;
 id
----
  1
(1 row)

dev2=# INSERT INTO t1 VALUES (2);
INSERT 0 1
dev2=# SELECT * FROM v1;
 id
----
  1
  2
(2 rows)

dev2=# INSERT INTO t1 VALUES (30);
INSERT 0 1
dev2=# SELECT * FROM v1;
 id
----
  1
  2
(2 rows)
```

```
dev2=# SELECT * FROM t1;
 id
----
  1
  2
 30
(3 rows)
```

En 9.3, on utilisera plutôt la vue matérialisée car, avec ce nouveau type d'objet, on n'est plus obligé de passer par un trigger :

```
dev2=# CREATE MATERIALIZED VIEW v2 as SELECT * FROM t1 WHERE id<10;
SELECT 2
dev2=# SELECT * FROM v2;
 id
----
  1
  2
(2 rows)

dev2=# INSERT INTO t1 VALUES (3);
INSERT 0 1
dev2=# SELECT * FROM v2;
 id
----
  1
  2
(2 rows)

dev2=# SELECT * FROM v1;
 id
----
  1
  2
  3
(3 rows)

dev2=# REFRESH MATERIALIZED VIEW v2;
REFRESH MATERIALIZED VIEW
dev2=# SELECT * FROM v2;
 id
----
  1
  2
  3
(3 rows)
```

```
dev2=# INSERT INTO t1 VALUES (40);
INSERT 0 1
dev2=# REFRESH MATERIALIZED VIEW v2;
REFRESH MATERIALIZED VIEW
dev2=# SELECT * FROM v2;
 id
----
  1
  2
  3
(3 rows)
```

Si une mise à jour au fil de l'eau est requise, il faudra forcément passer par des triggers.

```
dev2=# CREATE FUNCTION maj_v2() RETURNS trigger LANGUAGE plpgsql
AS $$
  BEGIN
    REFRESH MATERIALIZED VIEW v2;
    RETURN new;
END
$$;
CREATE FUNCTION
dev2=# CREATE TRIGGER tr2 AFTER INSERT ON t1 FOR EACH ROW
       EXECUTE PROCEDURE maj_v2();
CREATE TRIGGER
dev2=# INSERT INTO t1 VALUES (40);
INSERT 0 1
dev2=# SELECT * FROM v2;
 id
----
  1
  2
  3
(3 rows)

dev2=# INSERT INTO t1 VALUES (5);
INSERT 0 1
dev2=# SELECT * FROM v2;
 id
----
  1
  2
  3
  5
(4 rows)
```

Par contre, ce genre de rafraîchissement au fil de l'eau de vues matérialisées est très coûteux. En effet, une vue matérialisée recalcule l'ensemble des données. Par ailleurs, avant la version 9.4, la vue est inaccessible pendant la durée du rafraîchissement.

2.5.3 PARTITIONNEMENT - ORACLE

```
create table sales (year number(4),
                    product varchar2(10),
                   amt number(10,2))
     partition by range (year)
     partition p1 values less than (1992) tablespace u1,
     partition p2 values less than (1993) tablespace u2,
     partition p3 values less than (1994) tablespace u3,
     partition p4 values less than (1995) tablespace u4,
   partition p5 values less than (MAXVALUE) tablespace u5;
```

Oracle dispose de clauses dans la commande `CREATE TABLE` permettant de définir simplement les partitions et la méthode de partitionnement.

2.5.4 PARTITIONNEMENT - POSTGRESQL

Pas d'équivalent simple sous PostgreSQL :

```
CREATE TABLE sales (year numeric(4),
                    product varchar(10),
                   amt numeric(10,2));
CREATE TABLE sales_y1992 (
    CHECK ( year = 1992 )
) INHERITS (sales);
CREATE TABLE sales_y1993 (
    CHECK ( year = 1993 )
) INHERITS (sales);
…
```

PostgreSQL ne dispose pas de ces clauses. Il est nécessaire de passer par l'héritage pour la création des partitions, par l'ajout de contraintes `CHECK` pour que le planificateur sache qu'il peut ne parcourir que certaines partitions, et par l'ajout de triggers pour que les insertions, mises à jour et suppressions se passent correctement.

Autrement dit, la mise en place du partitionnement sous PostgreSQL est lourde et peu intuitive.

17.12

Benchmarks sur le partitionnement sous PostgreSQL : http://www.mkyong.com/database/performance-testing-on-partition-table-in-postgresql-part-3/

2.5.5 PARTITIONNEMENT - PROBLÈMES

- Triggers pour envoyer les enregistrements dans la bonne partition
- Pas de contrainte d'unicité globale
- Donc pas de clé primaire
- Donc pas de clé étrangère pointant sur la table partitionnée

En plus d'une mise en place difficile et d'une administration tout autant complexe, un bon nombre d'autres inconvénients sont présents.

L'unicité globale ne peut pas être assurée. PostgreSQL dépend d'un index pour assurer ou forcer cette contrainte, et il n'est pas possible de créer un index sur plusieurs tables. Du coup, une table partitionnée ne peut pas avoir de contrainte unique ou de clé primaire. De ce fait, il n'est pas non plus possible d'utiliser des clés étrangères vers cette table.

2.6 CONCLUSION

Points essentiels :

- Grande importance de la première migration.
- Même si Oracle et PostgreSQL sont assez similaire il y a de nombreuses différences.
- Étude de la migration, ce qui doit ou pas être migré et comment.
- Choix des outils de migration.
- La majorité du temps de migration est imputable à la conversion du PL/SQL

2.6.1 QUESTIONS

N'hésitez pas, c'est le moment !

2.7 TRAVAUX PRATIQUES

2.7.1 ÉNONCÉS

TP1.1

Correspondance des types de données

Donner les correspondances pour PostgreSQL des types de données Oracle suivants :

- `NUMBER(4)`
- `NUMBER(10)`
- `NUMBER(9,3)`
- `NUMBER`
- `VARCHAR(25)`
- `VARCHAR2`
- `BLOB`
- `CLOB`

Aidez-vous de la documentation PostgreSQL sur les types de données : http://docs.postgresql.fr/9.4/datatype.html

---

TP1.2

Champs NULL

La requête `SELECT` suivante ne retourne pas le même résultat selon qu'elle est exécutée sur Oracle ou sur PostgreSQL (2 lignes sous Oracle et 1 sous PostgreSQL), pourquoi ?

```
CREATE TABLE label (id NUMBER, lbl VARCHAR(25));
INSERT INTO label VALUES (1, 'Label 1');
INSERT INTO label VALUES (2, NULL);
INSERT INTO label VALUES (3, 'Label 3');
INSERT INTO label VALUES (4, '');

SELECT count(id) FROM label WHERE lbl IS NULL;
```

Réécrire la requête `SELECT` pour qu'elle renvoie le même résultat sur PostgreSQL que sur Oracle.

TP1.3

Concaténation de chaine NULL

Réécrire la requête suivante pour obtenir le même résultat sur PostgreSQL :

17.12

```
SELECT 'Label'||NULL FROM DUAL;

LABEL
-----
Label
```

Note : Exécutez la requête sur PostgreSQL pour voir la valeur retournée.

TP1.4

Les traitements sur les dates

Réécrire pour PostgreSQL la requête suivante (NB : seule la date du jour nous intéresse, pas les heures) :

```
SELECT SYSDATE + 1 FROM DUAL;
```

Exemple de valeur retournée sur Oracle (affichage par défaut sans les heures !) :

```
SYSDATE+1
---------
14-MAR-14
```

Faites de même avec la requête :

```
SELECT add_months(to_date('14-MAR-2014'), 2) FROM DUAL;
```

Valeur retournée sur Oracle :

```
ADD_MONTH
---------
14-MAY-14
```

TP1.5

Jointures (+)

Même si la notation `(+)` n'est pas recommandée, il peut rester de nombreux codes utilisant cette notation.

Réécrire le code suivant dans le respect de la norme :

```
SELECT * FROM employees e, departments d
WHERE e.employee_id = d.manager_id (+)
```

Puis ce code :

```
SELECT * FROM appellation a, region r
WHERE r.id (+) = a.region_id;
```

TP1.6

ROWNUM

Réécrire la requête suivante utilisant ROWNUM pour numéroter les lignes retournées :

```
SELECT ROWNUM, country_name, region_name
  FROM countries c
  JOIN regions r ON (c.region_id = r.region_id);
```

et cete requête utilisant ROWNUM pour limiter le nombre de lignes ramenées :

```
SELECT country_name, region_name
  FROM countries c
  JOIN regions r ON (c.region_id = r.region_id)
WHERE ROWNUM < 21;
```

TP1.7

Portage de DECODE

La construction suivante utilise la fonction DECODE :

```
SELECT LAST_NAME, JOB_ID, SALARY,
       DECODE(JOB_ID,
              'PU_CLERK', SALARY * 1.05,
              'SH_CLERK', SALARY * 1.10,
              'ST_CLERK', SALARY * 1.15,
              SALARY) "Proposed Salary"
  FROM EMPLOYEES
 WHERE JOB_ID LIKE '%_CLERK'
   AND LAST_NAME < 'E'
 ORDER BY LAST_NAME;
```

Réécrire cette requête pour son exécution sous PostgreSQL.

Autre exemple à convertir :

```
DECODE("user_status",'active',"username",NULL)
```

TP1.8

FUNCTION

Porter sur PostgreSQL la fonction Oracle suivante :

```
CREATE OR REPLACE FUNCTION text_length(a CLOB)
   RETURN NUMBER DETERMINISTIC IS
```

```
BEGIN
  RETURN DBMS_LOB.GETLENGTH(a);
END;
```

Conseil : les fonctions sur les chaînes de caractères sont listées ici : http://docs.postgresql.fr/9.6/functions-string.html

---

TP1.9

CONNECT BY

Réécrire la requête suivante à base de `CONNECT BY` sous Oracle :

```
SELECT numero, nom, fonction, manager
  FROM employes
  START WITH manager IS NULL
  CONNECT BY PRIOR numero = manager;
```

Cette requête explore la hiérarchie de la table `EMPLOYES`. La colonne `manager` de cette table désigne le responsable hiérarchique d'un employé. Si elle vaut `NULL`, alors la personne est au sommet de la hiérarchie comme exprimé par la partie `START WITH manager IS NULL` de la requête. Le lien avec l'employé et son responsable hiérarchique est construit avec la clause `CONNECT BY PRIOR numero = manager` qui indique que la valeur de la colonne `manager` correspond à l'identifiant `numero` du niveau de hiérarchie précédent.

Voici le retour de cette requête sous Oracle :

```
1 | A. Boss        | DIRECTEUR       |
4 | C. Second      | SOUS DIRECTEUR | 1
5 | F. Dsi         | DSI             | 1
3 | C. Secretaire | ASSISTANTE      | 1
2 | J.P Devel      | DEVELOPPEUR     | 5
```

pour vous aider, le portage de ce type de requête se fait à l'aide d'une requête récursive (`WITH RECURSIVE`) sous PostgreSQL. Pour plus d'information voir la documentation : http://docs.postgresql.fr/9.6/queries-with.html

Voici les ordres SQL permettant de créer cette table et d'y insérer quelques données pour faciliter les tests de réécriture.

```
CREATE TABLE employes (
        numero integer,
        nom text,
        fonction text,
        manager integer
);
```

```
INSERT INTO employes VALUES (1, 'A. Boss', 'DIRECTEUR', NULL);
INSERT INTO employes VALUES (4, 'C. Second', 'SOUS DIRECTEUR', 1);
INSERT INTO employes VALUES (5, 'F. Dsi', 'DSI', 1);
INSERT INTO employes VALUES (2, 'J.P Devel', 'DEVELOPPEUR', 5);
INSERT INTO employes VALUES (3, 'C. Secretaire', 'ASSISTANTE', 1);
```

2.7.2 SOLUTIONS

TP1.1

Correspondance des types de données

- À `NUMBER(4)` correspond `SMALLINT`
- À `NUMBER(10)` correspond `BIGINT`
- À `NUMBER(9,3)` correspond `NUMERIC(9,3)`
- À `NUMBER` correspond `NUMERIC`
- `VARCHAR(25)` est identique sur les deux SGBDR
- `VARCHAR2` peut être traduit en `VARCHAR()` ou `TEXT`
- À `BLOB` correspond le type `BYTEA`
- et `CLOB` est un champ `TEXT` sous PostgreSQL.

Ici il y a une particularité avec le type `NUMBER` sans paramètres qui peut aussi bien être un entier qu'un décimal. Il convient dans ce cas de vérifier les données pour utiliser le type adéquat. S'il s'agit de données de type monétaire, comme des tarifs ou montants de facture, ou tout autre donnée pour laquelle une grande précision est demandée, il est impératif d'utiliser le type numérique pour éviter les effets de bord des arrondis.

TP1.2

Champs NULL

La requête ne retourne pas le même résultat selon si elle est exécutée sur Oracle ou sur PostgreSQL car sous Oracle, le type de données `VARCHAR` ou `VARCHAR2` assimile la chaîne vide à la valeur `NULL`. Ce comportement fait que les id 2 et 4 sont retournés par Oracle, alors que PostgreSQl ne retournera que l'id 2.

Pour émuler le comportement non standard d'Oracle, voici la requête qui peut être utilisée à cet effet sur PostgreSQL :

```
SELECT count(id) FROM label WHERE lbl IS NULL OR (lbl = '');
```

TP1.3

Concaténation de champ NULL

De même que dans l'exercice précédent, Oracle permet de concaténer une chaîne avec une valeur `NULL` sans problèmes. Avec PostgreSQL, la valeur `NULL` est propagée dans les opérations : une valeur `NULL` concaténée à une chaîne de caractère donne `NULL`.

Pour émuler le fonctionnement d'Oracle, il faudra réécrire la requête comme suit :

```
SELECT coalesce('Label'||NULL, 'Label');
```

TP1.4

Les traitements sur les dates

PostgreSQL connaît les fonctions `CURRENT_DATE` (date sans heure) et `CURRENT_TIMESTAMP` (type `TIMESTAMP WITH TIME ZONE`, soit date avec heure).

Si on ne s'intéresse qu'à la date, la première requête peut simplement être remplacée par :

```
SELECT CURRENT_DATE + 1;
```

Ce qui est équivalent à :

```
SELECT CURRENT_DATE + '1 days'::interval;
```

Pour la seconde il y a un peu plus de travail de conversion :

```
SELECT '2014-03-14'::date + '2 months'::interval;
```

```
SELECT * FROM appellation a
LEFT OUTER JOIN region r ON a.region_id = r.id;
```

renvoie :

```
      ?column?
---------------------
 2014-05-14 00:00:00
(1 ligne)
```

Pour éliminer la partie heure, il faut forcer le type retourné :

```
SELECT cast('2014-03-14'::date + '2 months'::interval as date);
```

ou, ce qui revient au même :

```
SELECT ('2014-03-14'::date + '2 months'::interval)::date;
```

TP1.5

Jointures (+)

La première requête correspond à une jointure de type `LEFT OUTER JOIN` :

```
SELECT *
  FROM employees e
  LEFT OUTER JOIN departments d ON e.employee_id = d.manager_id;
```

et la seconde à une jointure de type `RIGHT OUTER JOIN` :

```
SELECT *
  FROM employees e
  RIGHT OUTER JOIN departments d ON e.employee_id = d.manager_id;
```

TP1.6

ROWNUM

La requête permettant la numérotation des lignes sera réécrite de la façon suivante :

```
SELECT row_number() OVER () AS rownum, country_name, region_name
  FROM countries c
  JOIN regions r ON (c.region_id = r.region_id);
```

La clause `OVER ()` devrait comporter à minima un `ORDER BY` pour spécifier l'ordre dans lequel on souhaite avoir les résultats. Cela dit, la requête telle qu'elle est écrite ci-dessus est une transposition fidèle de son équivalent Oracle.

La seconde requête ramène les 20 premiers éléments (arbitrairement, sans tri) :

```
SELECT country_name, region_name
  FROM countries c
  JOIN regions r ON (c.region_id = r.region_id)
  LIMIT 20 OFFSET 0
```

TP1.7

Portage de DECODE

La fonction `DECODE` d'Oracle est un équivalent propriétaire de la clause `CASE`, qui est normalisée.

Cette construction doit être réécrite de cette façon :

```
SELECT last_name, job_id, salary,
       CASE job_id
            WHEN 'PU_CLERK' THEN salary * 1.05
```

```
            WHEN 'SH_CLERK' THEN salary * 1.10
            WHEN 'ST_CLERK' THEN salary * 1.15
            ELSE salary
       END AS "Proposed salary"
  FROM employees
 WHERE job_id LIKE '%_CLERK'
   AND last_name < 'E'
 ORDER BY last_name;
```

Réécriture du second exemple :

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

TP1.8

FUNCTION

La fonction peut être réécrite de la façon suivante en langage PL/pgSQL :

```
CREATE OR REPLACE FUNCTION text_length (a text) RETURNS integer AS
$$
BEGIN
        RETURN char_length(a);
END
$$
LANGUAGE PLPGSQL
IMMUTABLE;
```

ou simplement en SQL :

```
CREATE OR REPLACE FUNCTION text_length (a text) RETURNS integer AS
$$
SELECT char_length(a);
$$
LANGUAGE SQL
IMMUTABLE;
```

TP1.9

CONNECT BY

La récursion est initialisée dans une première requête qui récupère les lignes qui correspondent à la condition de la clause `START WITH` de la requête précédente : `mgr IS NULL`.

La récursion continue ensuite avec la requête suivante qui réalise une jointure entre la table `emp` et la vue virtuelle `emp_hierarchy` qui est définie par la clause `WITH RECURSIVE`. La

condition de jointure correspond à la clause `CONNECT BY`. La vue virtuelle `emp_hierarchy` a pour alias `prior` pour mieux représenter la transposition de la clause `CONNECT BY`.

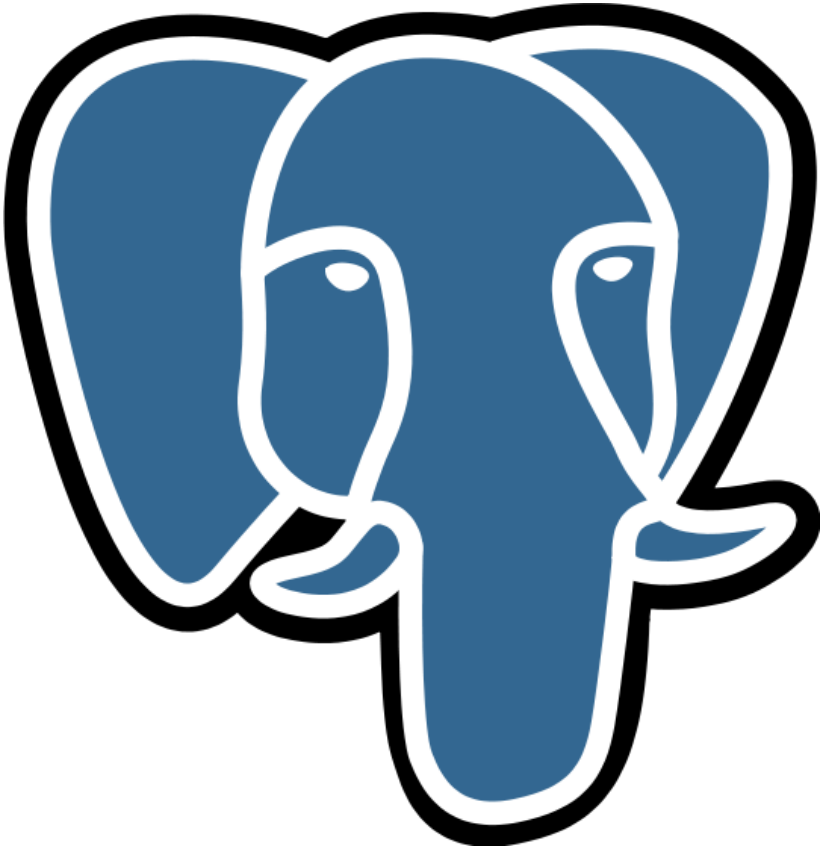
La requête récursive pour PostgreSQL serait alors écrite de la façon suivante :

```
WITH RECURSIVE employes_hierarchie (numero, nom, fonction, manager) AS
(
    SELECT numero, nom, fonction, manager
    FROM employes
      WHERE manager IS NULL
    UNION ALL
    SELECT employes.numero, employes.nom, employes.fonction, employes.manager
    FROM employes
    JOIN employes_hierarchie prior ON (employes.manager = prior.numero)
)
SELECT * FROM employes_hierarchie;
```

Résultat :

```
 numero |      nom       |    fonction     | manager
--------+----------------+-----------------+---------
      1 | A. Boss        | DIRECTEUR       |
      4 | C. Second      | SOUS DIRECTEUR  |       1
      5 | F. Dsi         | DSI             |       1
      3 | C. Secretaire  | ASSISTANTE      |       1
      2 | J.P Devel      | DEVELOPPEUR     |       3
(5 lignes)
```

3 SCHÉMA ET DONNÉES



3.1 INTRODUCTION

Ce module est organisé en quatre parties :

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

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

3.2 INSTALLATION D'ORA2PG

Étapes :

- Téléchargement
- Pré-requis
- Compilation
- Installation
- Utilisation

Nous allons aborder dans cette première partie les différentes étapes à réaliser pour installer Ora2Pg à partir des sources :

- Où trouver les fichiers sources ?
- Quelle version choisir ?
- Comment préparer le serveur pour accueillir PostgreSQL ?
- Quelle procédure de compilation suivre ?
- Comment installer les fichiers compilés ?

3.2.1 TÉLÉCHARGEMENT

- Disponible via :
  - HTTP : http://ora2pg.darold.net/
  - Git : https://github.com/darold/ora2pg
- Télécharger le fichier `ora2pg-X.Y.tar.bz2`
- Version au 13/01/2016 : 16.2

Cet outil est développé par Gilles Darold, aujourd'hui consultant PostgreSQL chez Dalibo.

Les fichiers sources et les instructions de compilation sont accessibles depuis le site officiel du projet[1] .

Il est très important de toujours télécharger la dernière version car l'ajout de fonctionnalités et les corrections de bogues sont permanentes. En effet, ce projet bénéficie au fur et à mesure des retours d'expérience des utilisateurs d'améliorations et de corrections. Il est constamment mis à jour.

[1] http://ora2pg.darold.net/

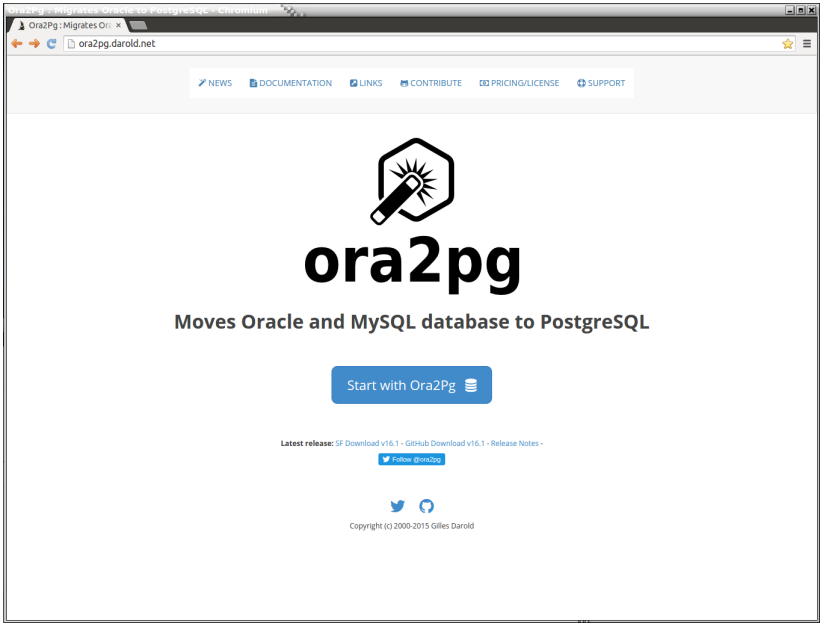
17.12

Les distributions officielles peuvent être téléchargées directement depuis SourceForge.net[2] , par exemple :

```
wget http://downloads.sourceforge.net/project/ora2pg/16.2/ora2pg-16.2.tar.bz2
```

Voici comment récupérer la version 16.2 des sources de Ora2Pg :

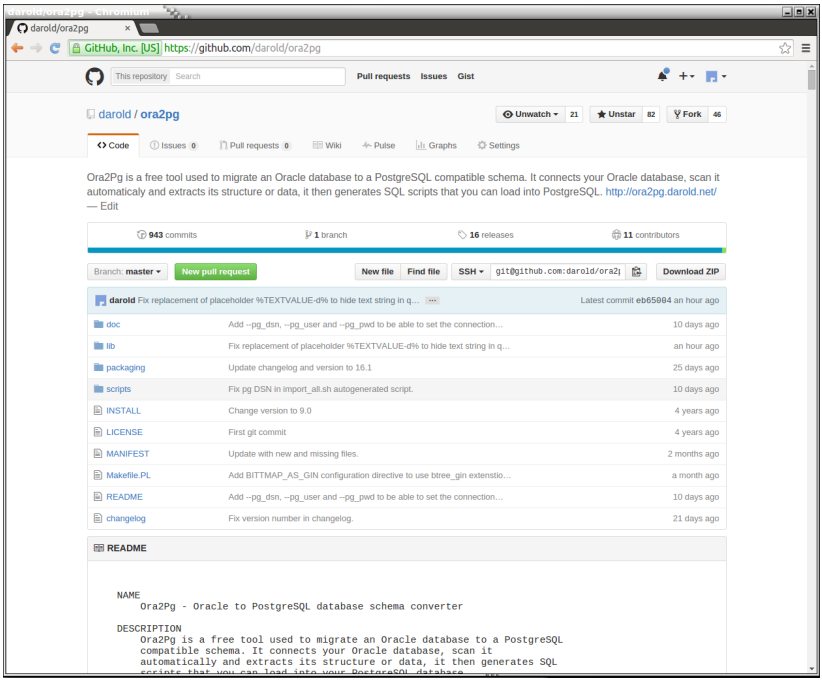
- Aller sur la page d'accueil du projet Ora2Pg et dans la section *Latest release* cliquer sur le lien *SF Download v16.2*.



- Vous arriverez sur la page sourceforge du projet, cliquez sur le bouton *Download*.

Pour obtenir le dernier code en développement, il faut aller sur le dépôt GitHub (https://github.com/darold/ora2pg) du projet et cliquer sur le bouton *Download ZIP*.

[2] http://sourceforge.net/projects/ora2pg/



Le fichier téléchargé sera nommé `ora2pg-master.zip`.

3.2.2 DÉPENDANCES REQUISES

- Oracle >= 8i client ou serveur
- Environnement Oracle correct (`ORACLE_HOME` et `PATH` comprenant `sqlplus`)
- `libaio1`
- Unix : Perl 5.8+
- Windows : Strawberry Perl 5.10+ ou ActiveStep Perl 5.10+
- Modules Perl
  - `Time::HiRes`
  - `Perl DBI` et `DBD::Oracle`

Ora2Pg est entièrement codé en `Perl`.

Ora2Pg se connecte à Oracle grâce à l'interface de bases de données pour `Perl`, appelée `DBI`.

Dans cette interface, Ora2Pg va utiliser le connecteur Oracle, appelé `DBD::Oracle` (DBD, acronyme de *DataBase Driver*). Tous les modules Perl, s'ils ne sont pas disponibles en paquet pour votre distribution, peuvent toujours être téléchargés à partir du site CPAN (http://search.cpan.org/). Il suffit de saisir le nom du module (ex : `DBD::Oracle`) dans la case de recherche et la page de téléchargement du module vous sera proposée.

Le client lourd d'Oracle est nécessaire pour utiliser la couche `OCI`. Cependant, les nouvelles versions *Instant Client* (à partir de la version 10g) suffisent amplement à Ora2Pg. Attention toutefois, s'il est possible d'utiliser un client Oracle 11g pour se connecter à des bases Oracle de versions inférieures, l'inverse n'est pas vrai.

Ainsi il convient d'installer au minimum un client Oracle, comme `instantclient-basic`, `instantclient-sdk` ou `instantclient-sqlplus`. Pour que `sqlplus` puisse fonctionner correctement il faut au préalable installer la librairie `libaio1`. Cette bibliothèque permet aux applications en espace utilisateur d'utiliser les appels système asynchrones d'E/S du noyau Linux.

`DBD::Oracle` va s'appuyer sur les variables d'environnement pour déterminer où se trouvent les bibliothèques d'Oracle.

Dans le monde Oracle, ces variables d'environnement sont très connues (`ORACLE_BASE`, `ORACLE_HOME`, `NLS_LANG`, etc). Pour `DBD::Oracle`, le positionnement de la variable `ORACLE_HOME` suffit.

```
export ORACLE_HOME=/usr/lib/oracle/10.2.0.4/client64
```

Pour une installation sous Windows, l'utilisation de Strawberry Perl[3] nécessitera les outils de compilation pour l'installation de `DBD::Oracle` alors que l'utilisation de la distribution libre d'ActiveState[4] permet d'installer directement une version binaire de la bibliothèque.

Pour les anciennes versions de Perl ou certaines distributions il peut être nécessaire d'installer le module Perl, `Time::HiRes`.

3.2.3 DÉPENDANCES OPTIONNELLES

En option :

- PostgreSQL >= 8.4 client ou serveur
- `DBD::Pg` pour l'import direct dans PostgreSQL
- `Compress::Zlib` : compression des fichiers en sortie
- `DBD::MySQL` pour migrer les bases MySQL

[3] http://strawberryperl.com/
[4] http://www.activestate.com/activeperl/downloads

Le connecteur PostgreSQL pour DBI, DBD::Pg est nécessaire uniquement si l'on veut migrer directement les données depuis Oracle vers PostgreSQL sans avoir à passer par des fichiers intermédiaires. DBD::Pg nécessite au minimum les bibliothèques du client PostgreSQL.

On peut se passer de ce module dans la mesure où, par défaut, Ora2Pg va écrire les objets et données à migrer dans des fichiers. Ces fichiers peuvent alors être chargés à l'aide de la commande psql ou être transférés sur une autre machine disposant de cet outil.

La bibliothèque Perl Compress::Zlib est nécessaire si vous souhaitez que les fichiers de sortie soient compressés avec gzip. C'est notamment très utile pour les fichiers de données volumineux par exemple.

Fort heureusement, on peut aussi utiliser le binaire bzip2 pour compresser le fichier de sortie. Dans ce cas, il suffit d'indiquer, dans le fichier de configuration d'Ora2Pg, l'emplacement du binaire bzip2 sur le système si celui-ci n'est pas dans le PATH.

Ora2Pg, depuis la version 16.0, permet de migrer les bases de données MySQL. Tout comme pour Oracle, Ora2Pg a besoin de se connecter à l'instance MySQL au travers d'un driver Perl. C'est le rôle du module Perl DBD::MySQL.

3.2.4 COMPILATION ET INSTALLATION

- Décompresser l'archive téléchargée
- Générer les fichiers de compilation
- Compiler et installer
- Ora2Pg est prêt à être configuré !

Voici les lignes de commande à saisir pour compiler et installer Ora2Pg :

```
tar xjf ora2pg-16.2.tar.bz2

cd ora2pg-16.2/
perl Makefile.PL

make && sudo make install
```

Sous Windows, il faut remplacer la dernière ligne par la ligne suivante :

```
dmake && dmake install
```

dmake est l'équivalent de make pour Windows, il peut être téléchargé depuis cette URL : http://search.cpan.org/dist/dmake/. Téléchargez-le et installez dmake quelque part dans le PATH Windows.

17.12

Le fichier de configuration d'Ora2Pg par défaut est `/etc/ora2pg/ora2pg.conf` sous Unix et `C:\ora2pg\ora2pg.conf` sous Windows.

L'installation provoquera la création d'un modèle de fichier de configuration : `ora2pg.conf.dist`, avec toutes les variables définies par défaut. Il suffira de le renommer en `ora2pg.conf` et de paramétrer pour obtenir le comportement souhaité.

```
cp /etc/ora2pg/ora2pg.conf.dist /etc/ora2pg/ora2pg.conf
```

Les paramètres de ce fichier sont explicités de manière exhaustive plus loin dans la formation.

3.2.5 USAGE DE LA COMMANDE ORA2PG

Le script `ora2pg` s'utilise de la façon suivante :

```
ora2pg [-dhpqv --estimate_cost --dump_as_html] [--option value]
```

Utilisation basique :

```
ora2pg -t ACTION [-c fichier_de_configuration]
```

Certaines options ne nécessitent pas de valeurs et ne sont introduites dans la ligne de commande que pour activer certains comportements. C'est le cas notamment de l'option `-d` ou `--debug` permettant d'activer le mode trace.

Toutes les options courtes ont une version longue, par exemple `-q` et `--quiet`.

Voici l'intégralité des options disponibles en ligne de commande pour le script Perl `ora2pg` et leur explication :

```
-a | --allow str  : liste de nom d'objets séparés par une virgule autorisés lors
                    de l'export.
                    Peut notamment être utilisé avec SHOW_COLUMN pour
                    n'afficher qu'une table.
-b | --basedir dir: utilisé pour spécifier le répertoire de base où le(s)
                    fichier(s) résultant
                    de l'export seront stockés. Par défault il s'agit du
                    répertoire courant.
-c | --conf file  : permet de spécifier un fichier de configuration différent
                    de celui par
                    défaut : /etc/ora2pg/ora2pg.conf.
-d | --debug      : active le mode trace.
-e | --exclude str: liste de nom d'objets séparés par une virgule à exclure de
```

```
                    l'export.
-h | --help       : affiche le message d'aide.
-i | --input_file file: fichier donné en entrée à la place d'une base de
                    données et contenant le code Oracle PL/SQL à convertir.
                    Aucune connexion à une base de données
                    n'est initialisé.
-j | --jobs number : nombre de connexions en parallèle utilisé pour envoyer les
                     données à PostgreSQL.
-J | --ora_copies n: nombre de connexions en parallèle utilisé pour extraire
                     les données d'Oracle.
-l | --log file   : permet de spécifier un fichier ou seront inscrit les
                    traces. Par défaut : stdout.
-L | --data_limit n : nombre d'enregistrements d'Oracle stocké en mémoire et
                      passé à un processus d'écriture dans PostgreSQL ou dans
                      un fichier.
-m | --mysql      : signale que l'export se fait d'une base de données MySQL et
                    non Oracle.
-n | --namespace schema : utilisé pour spécifier le schéma Oracle à partir
                          duquel les exports auront lieu.
-o | --out file   : utilisé pour définir le fichier de sortie ou le code SQL
                    généré sera écrit.  par défaut le fichier est nommé :
                    output.sql
-p | --plsql      : active la conversion automatique du code PL/SQL en PL/PGPSQL
-P | --parallel num: Nombre de tables à extraire en parallèle au même instant.
-q | --quiet      : désactive l'affichage de la barre de progression.
-s | --source DSN  : permet de définir le datasource (DSN) pour DBI Oracle.
-t | --type export: utilisé pour spécifier le type export à réaliser.
-T | --temp_dir DIR: permet de changer le répertoire utilisé pour les fichiers
                     temporaires lorsque deux ou plus scripts ora2pg sont
                     exécutés en parallèle.
-u | --user  user : permet de définir l'utilisateur à utiliser pour la
                    connexion à Oracle.
-v | --version    : affiche la version d'Ora2Pg.
-w | --password pwd: permet de définir le mot de passe à utiliser pour la
                     connexion à Oracle.
--forceowner value: si elle est activé force ora2pg à définir le propriétaire
                    des objets tel qu'il est défini dans Oracle. Si la valeur
                    est un nom d'utilisateur c'est celui ci qui sera utilisé.
                    Par défaut il s'agit de l'utilisateur avec lequel la
```

```
                    connexion à PostgreSQL se fait lors de l'import.
--nls_lang code   : permet de forcer l'encodage à utiliser par le client Oracle
                    (positionne NLS_LANG).
--client_encoding code: permet de forcer l'encodage à utiliser par le client
                        PostgreSQL.
--view_as_table str: liste de nom de vues séparés par une virgule qui doivent
                     être exportées comme des tables.
--estimate_cost   : active l'évaluation du coût de migration avec SHOW_REPORT
--cost_unit_value minutes: nombre de minutes attribuées à l'unité de coût de
                           migration. Cinq par défaut, cela correspond à une
                           migration conduite par un expert PostgreSQL.
                           Utilisez 10 ou 15 minutes s'il s'agit d'une première
                           migration.
--dump_as_html    : force ora2pg à générer le rapport au format HTML, ne
                    fonctionne qu'avec SHOW_REPORT.  Le format par défaut des
                    rapports est le texte simple.
--dump_as_csv     : comme l'option précédente mais force ora2pg à générer un
                    rapport en CSV.
--dump_as_sheet   : génère un rapport d'estimation de migration en CSV et une
                    ligne par base.
--init_project NAME: initialise un arbre de projet typique ora2pg. Un
                     répertoire portant le nom du projet sera créé dans le
                     répertoire de base positionné avec l'option suivante.
--project_base DIR : définie le répertoire de base pour l'arbre de projet
                     ora2pg. Par défaut il s'agit du répertoire courant.
--print_header    : utilisé avec --dump_as_sheet pour afficher l'entête des
                    colonnes CSV notamment lors de la première exécution
                    d'ora2pg.
--human_days_limit n : défini la limite en jour-homme où l'estimation du niveau
                       de difficulté de migration passe de B à C. Par défaut 5
                       jours-homme.
--audit_user LIST  : liste d'utilisateurs séparés par une virgule pourfiltrer
                     les requêtes de la table DBA_AUDIT_TRAIL. Utilisé
                     uniquement avec SHOW_REPORT et QUERY.
--pg_dsn DSN      : permet de définir la chaine de connexion vers PostgreSQL
                    pour l'import direct des données.
--pg_user name    : positionne l'utilisateur PostgreSQL à utiliser pour se
                    connecter.
--pg_pwd password  : positionne le mot de passe à utiliser pour se connecter.
```

3.3 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 ?

3.3.1 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
```

17.12

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.

3.3.2 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 10g (10.2.0) Express Edition, le `ORACLE_HOME` ressemble à cela :

```
ORACLE_HOME /usr/lib/oracle/xe/app/oracle/product/10.2.0/server
```

Pour un client de la même version on peut avoir :

```
ORACLE_HOME /usr/lib/oracle/xe/app/oracle/product/10.2.0/client
```

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 sur la console tous les messages, y compris d'erreurs, qu'il a à envoyer.

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.

3.3.3 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_DNS

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 le déclarer 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 = DB_SID)  )
          )
 EOF
```

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

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 :

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

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
```

3.4 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

3.4.1 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 connexions :

Par exemple, pour l'instance d'exemple fournie par Oracle XE et le schéma HR, 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 (25 rows)
[2] TABLE DEPARTMENTS (27 rows)
[3] TABLE EMPLOYEES (107 rows)
[4] TABLE JOBS (19 rows)
[5] TABLE JOB_HISTORY (10 rows)
[6] TABLE LOCATIONS (23 rows)
[7] TABLE REGIONS (4 rows)
```

Si on ne connaît pas le nom du schéma ou que l'on ne s'en souvient pas, on peut utiliser la commande suivante pour lister tous les schémas de l'instance Oracle :

```
ora2pg -t SHOW_SCHEMA
```

Cela permettra de trouver la bonne valeur à donner à la directive SCHEMA dans le fichier de configuration.

L'utilisation de l'action SHOW_COLUMN :

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

renvoit 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 (25 rows)
  COUNTRY_ID : CHAR(2) => char(2)
  COUNTRY_NAME : VARCHAR2(40) => varchar(40)
  REGION_ID : NUMBER(22) => bigint
```

3.4.2 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...

3.4.3 GESTION DE L'ENCODAGE - 2

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

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:
    NLS_LANG AMERICAN_AMERICA.AL32UTF8
    NLS_NCHAR AL32UTF8
    CLIENT_ENCODING UTF8
    NLS_TIMESTAMP_FORMAT YYYY-MM-DD HH24:MI:SS.FF
    NLS_DATE_FORMAT YYYY-MM-DD HH24:MI:SS
Showing current Oracle encoding and possible PostgreSQL client encoding:
    NLS_LANG AMERICAN_AMERICA.WE8MSWIN1252
    NLS_NCHAR WE8MSWIN1252
    CLIENT_ENCODING WIN1252
    NLS_TIMESTAMP_FORMAT YYYY-MM-DD HH24:MI:SS.FF
    NLS_DATE_FORMAT YYYY-MM-DD HH24:MI:SS
```

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 à par défaut la valeur AL32UTF8.

Il est fortement conseillé de conserver le comportement par défaut d'Ora2Pg pour éviter les erreurs liées à l'encodage, mais si l'on veut éviter le coût de l'encodage, 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`.

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` à 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[5] , 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

[5] http://www.postgresql.org/docs/9.6/static/multibyte.html

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`.

3.4.4 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 v15.2 - Database Migration Report
-------------------------------------------------------------------------------
Version Oracle Database 12c Enterprise Edition Release 12.1.0.2.0
Schema  HR
Size    35.00 MB

-------------------------------------------------------------------------------
Object          Number  Invalid Comments            Details
-------------------------------------------------------------------------------
DATABASE LINK   4       0       Database links will be exported as SQL/MED
                                PostgreSQL's Foreign Data Wrapper (FDW)
                                extensions using oracle_fdw.
FUNCTION        3       1       Total size of function code: 791 bytes.
INDEX           32      0       20 index(es) are concerned by the export,
```


			others are automatically generated and will do so on PostgreSQL. Bitmap index(es) will be exported as b-tree index(es) if any. Cluster, domain, bitmap join and IOT indexes will not be exported at all. Reverse indexes are not exported too, you may use a trigram-based index (see pg_trgm) or a reverse() function based index and search. 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. 5 domain index(es). 4 function based b-tree index(es). 11 b-tree index(es).
JOB	0	0	Job are not exported. You may set external cron job with them.
MATERIALIZED VIEW	2	0	All materialized view will be exported as snapshot materialized views, they are only updated when fully refreshed.
PACKAGE BODY	2	0	Total size of package code: 2992 bytes. Number of procedures and functions found inside those packages: 6.
PROCEDURE	2	0	Total size of procedure code: 772 bytes.
SEQUENCE	4	0	Sequences are fully supported, but all call to sequence_name.NEXTVAL or sequence_name.CURRVAL will be transformed into NEXTVAL('sequence_name') or CURRVAL('sequence_name').
SYNONYM	3	0	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. emp_details_view_v is an alias to HR.EMP_DETAILS_VIEW. public.emp_table is a link to hr.employees@curr_user. offices is an alias to HR.LOCATIONS.
TABLE	47	0	1 external table(s) will be exported as file_fdw foreign table. See EXTERNAL_TO_FDW

```
                                configuration directive to export as standard
                                table or use COPY in your code if you just want
                                to load data from external files. 6 check
                                constraint(s).  1 binary columns. 5 unknown
                                types. Total number of rows:
                                1573. Top 10 of tables sorted by number of
                                rows:. customer_summary has 1154 rows.
                                employees has 107 rows. user_role has 55 rows.
                                t1 has 28 rows.  departments has 27 rows.
                                countries has 25 rows. locations has 23 rows.
                                crsts_for_dta_p0 has 20 rows. jobs has 19 rows.
                                user_permission has 18 rows.
TABLE PARTITION  10      0      Partitions are exported using table inheritance
                                and check constraint. Hash partitions are not
                                supported by PostgreSQL and will not be
                                exported.  4 list partitions. 6 range
                                partitions.
TABLE SUBPARTITION  2    0
TRIGGER          6       1      Total size of trigger code: 2120 bytes.
TYPE             3       0      2 type(s) are concerned by the export, others
                                are not supported. Note that Type inherited and
                                Subtype are converted as table, type
                                inheritance is not supported.2 nested tables. 1
                                object type.
VIEW             6       0      Views are fully supported.
--------------------------------------------------------------------------------
Total            126     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/homme 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 procédures stockées (fonctions, procédures et paquets de fonctions), puis donne un score à chaque objet et à chaque fonction suivant le volume de code et la complexité de réécriture manuelle de ce code. En effet, la réécriture d'une fonction comportant un CONNECT BY ne prend pas le même temps que la réécriture d'une fonction comportant des appels à GOTO.

```
-------------------------------------------------------------------------------
Ora2Pg v15.2 - Database Migration Report
-------------------------------------------------------------------------------
Version Oracle Database 12c Enterprise Edition Release 12.1.0.2.0
Schema  HR
Size    35.00 MB

-------------------------------------------------------------------------------
Object          Number  Invalid Estimated cost  Comments  Details
-------------------------------------------------------------------------------
DATABASE LINK   4       0       12      Database links will be exported as
                                        SQL/MED PostgreSQL's Foreign Data
                                        Wrapper (FDW) extensions using
                                        oracle_fdw.
FUNCTION        3       1       13      Total size of function code: 791 bytes.
                                        get_tab_ptf: 4. get_tab_tf: 3. get_bal:
                                        3.
INDEX           32      0       6       20 index(es) are concerned by the
                                        export, others are automatically
                                        generated and will do so on PostgreSQL.
                                        Bitmap index(es) will be exported as
                                        b-tree index(es) if any. Cluster,
                                        domain, bitmap join and IOT indexes
                                        will not be exported at all. Reverse
                                        indexes are not exported too, you may
                                        use a trigram-based index (see pg_trgm)
                                        or a reverse() function based index and
                                        search. 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. 5 domain index(es). 4 function based b-tree index(es). 11 b-tree index(es).
JOB	0	0	0	Job are not exported. You may set external cron job with them.
MATERIALIZED VIEW	2	0	6	All materialized view will be exported as snapshot materialized views, they are only updated when fully refreshed.
PACKAGE BODY	2	0	44	Total size of package code: 2992 bytes. Number of procedures and functions found inside those packages: 6. emp_mgmt.create_dept: 3. emp_mgmt.hire: 11. emp_mgmt.increase_comm: 3. emp_mgmt.increase_sal: 3. emp_mgmt.remove_dept: 3. emp_mgmt.remove_emp: 3.
PROCEDURE	2	0	8	Total size of procedure code: 772 bytes. secure_dml: 3. add_job_history: 3.
SEQUENCE	4	0	0.4	Sequences are fully supported, but all call to sequence_name.NEXTVAL or sequence_name.CURRVAL will be transformed into NEXTVAL('sequence_name') or CURRVAL('sequence_name').
SYNONYM	3	0	6	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. emp_details_view_v is an alias to HR.EMP_DETAILS_VIEW. public.emp_table is a link to hr.employees@curr_user. offices is an alias to HR.LOCATIONS.
TABLE	47	0	23.9	1 external table(s) will be exported as file_fdw foreign table. See EXTERNAL_TO_FDW configuration directive

```
                                                to export as standard table or use COPY
                                                in your code if you just want to load
                                                data from external files. 4 check
                                                constraint(s). 1 binary columns. 5
                                                unknown types. Total number of rows:
                                                1573.  Top 10 of tables sorted by
                                                number of rows:.  customer_summary has
                                                1154 rows.  employees has 107 rows.
                                                user_role has 55 rows. t1 has 28 rows.
                                                departments has 27 rows.  countries has
                                                25 rows. locations has 23 rows.
                                                crsts_for_dta_p0 has 20 rows. jobs has
                                                19 rows.  user_permission has 18 rows.
TABLE PARTITION 10      0       1       Partitions are exported using table
                                        inheritance and check constraint. Hash
                                        partitions are not supported by
                                        PostgreSQL and will not be exported. 4
                                        list partitions. 6 range partitions.
TABLE SUBPARTITION  2   0       0.4
TRIGGER         6       1       36      Total size of trigger code: 2120 bytes.
                                        check_raise_on_avg: 18.
                                        cgh_alteration_createur_i: 3.
                                        update_job_history: 3. ioft_emp_perm:
                                        3.  ioft_insert_role_perm: 3.
TYPE            3       0       2       2 type(s) are concerned by the export,
                                        others are not supported. Note that
                                        Type inherited and Subtype are
                                        converted as table, type inheritance is
                                        not supported. 2 nested tables. 1
                                        object type.
VIEW            6       0       6       Views are fully supported.
--------------------------------------------------------------------------------
Total           126     2       164.7   164.7 cost migration units means
                                        approximatively 2 man-day(s).  The
                                        migration unit was set to 5 minute(s)
--------------------------------------------------------------------------------
```

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.homme.

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 fonction.

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[6]

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 les 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
```

[6] http://ora2pg.darold.net/report.html

3.5 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`.

3.5.1 FICHIERS DE SORTIE

Utilisation de fichiers de sortie dédiés

- `FILE_PER_CONSTRAINT` 1
- `FILE_PER_INDEX` 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`) et les index (`FILE_PER_INDEX`). 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 fonction (`FILE_PER_FUNCTION`) pour permettre un traitement individualisé.

3.5.2 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 importé 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 suivantes permettent de déterminer le comportement lors de l'export des données, à savoir 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 procédures 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.

3.5.3 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.

3.5.4 VERSIONS DE POSTGRESQL

- PostgreSQL >= 9.1
  - `PG_SUPPORTS_INSTEADOF` 1
  - `STANDARD_CONFORMING_STRINGS` 1
  - `PG_SUPPORTS_IFEXISTS` 1
- PostgreSQL >= 9.3
  - `PG_SUPPORTS_MVIEW` 1
- PostgreSQL >= 9.4
  - `PG_SUPPORTS_CHECKOPTION` 1
  - `BITMAP_AS_GIN` 1

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 existe six variables de configuration à désactiver suivant votre version de PostgreSQL et les objets exportés.

- `PG_SUPPORTS_INSTEADOF` pour le support de la clause `INSTEAD OF` dans les définitions de triggers ;
- `STANDARD_CONFORMING_STRINGS` pour l'échappement dans les chaines de caractères ;
- `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 ;
- `BITMAP_AS_GIN` pour autoriser l'export des index bitmap dans leur équivalent avec l'extension `btree_gin`.

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.

3.5.5 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

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 ont 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.

3.5.6 CONFIGURATION LIÉE AUX LOB

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

- LONGREADLEN 8192
- 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.

17.12

```
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.

3.6 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 ?

3.6.1 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 procédures 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 16.1
#
#------------------------------------------------------------------------------
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.

3.6.2 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`, `GRANT`, `SEQUENCE`, `TRIGGER`, `PACKAGE`, `FUNCTION`, `PROCEDURE`, `PARTITION`, `TYPE`, `INSERT`, `COPY`, `TABLESPACE`, `SHOW_SCHEMA`, `SHOW_TABLE`, `SHOW_COLUMN`, `SHOW_ENCODING`, `KETTLE`, `QUERY` 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.

3.6.3 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` et `FILE_PER_CONSTRAINT`, la commande d'extraction des définitions de tables, contraintes et index va créer trois fichiers dans le répertoire de sortie `schema/tables` :

- `table.sql`
- `CONSTRAINTS_table.sql`
- `INDEXES_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.

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.

3.6.4 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).

3.6.5 EXPORT DES PROCÉDURES 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 déclencheurs, les fonctions et procédures stockées. 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.

3.6.6 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 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.

3.6.7 EXPORT DES PARTITIONS

Export des partitions :

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

Si la base de données Oracle possède des partitions, Ora2Pg va convertir toutes les partitions à base d'héritage et de contraintes CHECK. 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 en code compatible PostgreSQL va donner cela :

- Partition par range

```
CREATE TABLE sales_range
(
    salesman_id  integer,
    salesman_name varchar(30),
    sales_amount  bigint,
    sales_date    timestamp
);

CREATE TABLE sales_range_jan2000 (
```

```
    CHECK ( sales_date < DATE '2000-02-01' AND sales_date >= DATE '2000-01-01' )
) INHERITS (sales_range);
CREATE TABLE sales_range_feb2000 (
    CHECK ( sales_date < DATE '2000-03-01' AND sales_date >= DATE '2000-02-01' )
) INHERITS (sales_range);

CREATE INDEX sales_range_feb2000_sales_date ON sales_range_feb2000 (sales_date);
CREATE INDEX sales_range_feb2000_sales_date ON sales_range_feb2000 (sales_date);

CREATE OR REPLACE FUNCTION sales_range_insert_trigger()
RETURNS TRIGGER AS $$
BEGIN
    IF ( NEW.sales_date < DATE '2000-02-01' AND
         NEW.sales_date >= DATE '2000-01-01' ) THEN
        INSERT INTO sales_range_jan2000 VALUES (NEW.*);
    ELSIF ( NEW.sales_date < DATE '2000-03-01' AND
            NEW.sales_date >= DATE '2000-02-01' ) THEN
        INSERT INTO sales_range_feb2000 VALUES (NEW.*);
    ELSE
        RAISE EXCEPTION 'Value out of range.  '
                        'Fix the sales_range_insert_trigger() function!';
    END IF;
    RETURN NULL;
END;
$$
LANGUAGE plpgsql;

CREATE TRIGGER insert_sales_range_trigger
    BEFORE INSERT ON sales_range
    FOR EACH ROW EXECUTE PROCEDURE sales_range_insert_trigger();
```

- Partition par liste

```
CREATE TABLE sales_list
(
    salesman_id  integer,
    salesman_name varchar(30),
    sales_state   varchar(20),
    sales_amount  bigint,
    sales_date    timestamp
);

CREATE TABLE sales_west (
    CHECK ( sales_state IN ('California', 'Hawaii') )
) INHERITS (sales_list);
CREATE TABLE sales_east (
    CHECK ( sales_state IN ('New York', 'Virginia', 'Florida'))
```

```
) INHERITS (sales_list);
CREATE TABLE sales_other () INHERITS (sales_list);

CREATE INDEX sales_list_west_sales_state ON sales_west (sales_state);
CREATE INDEX sales_list_east_sales_state ON sales_east (sales_state);
CREATE INDEX sales_list_other_sales_state ON sales_other (sales_state);

CREATE OR REPLACE FUNCTION sales_list_insert_trigger()
RETURNS TRIGGER AS $$
BEGIN
    IF ( NEW.sales_state IN ('California', 'Hawaii') ) THEN
        INSERT INTO sales_west VALUES (NEW.*);
    ELSIF ( NEW.sales_state IN ('New York', 'Virginia', 'Florida') ) THEN
        INSERT INTO sales_west VALUES (NEW.*);
    ELSE
        INSERT INTO sales_other VALUES (NEW.*);
    END IF;
    RETURN NULL;
END;
$$
LANGUAGE plpgsql;

CREATE TRIGGER insert_sales_list_trigger
    BEFORE INSERT ON sales_list
    FOR EACH ROW EXECUTE PROCEDURE sales_list_insert_trigger();
```

Les partitions de type HASH (partitionnement automatique) et de type COMPOSITE (plusieurs types créant des sous-partitions) ne sont pas ou mal supportées.

3.6.8 EXPORT DES VUES MATÉRIALISÉES

Depuis PostgreSQL 9.3 :

- PG_SUPPORT_MVIEW 1

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

  Avant la 9.3 :

- utilisation de fonctions, table et vues dédiées

Depuis PostgreSQL 9.3

Une nouvelle directive, PG_SUPPORT_MVIEW, permet d'utiliser le support natif des vues matérialisées dans PostgreSQL 9.3. Elle est maintenant activée par défaut et permet d'exporter les vues matérialisées directement avec la syntaxe SQL.

```
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;
```

Si une mise à jour au fil de l'eau est requise, il faudra forcément passer par des triggers, ceci n'est pas encore implémenté nativement dans PostgreSQL.

Avant PostgreSQL 9.3

Si PG_SUPPORT_MVIEW est désactivé, Ora2Pg générera le nécessaire pour gérer des vues matérialisées de type *snapshot*, c'est-à-dire qui sont mises à jour uniquement lorsqu'elles sont rafraîchies. Les autres types nécessitent l'écriture de code spécifique différent à chaque vue.

Voici un exemple simple de déclaration de vue matérialisée sous Oracle :

```
CREATE MATERIALIZED VIEW LOG ON employees
   WITH PRIMARY KEY
   INCLUDING NEW VALUES;

CREATE MATERIALIZED VIEW emp_data
   PCTFREE 5 PCTUSED 60
   STORAGE (INITIAL 50K NEXT 50K)
   REFRESH FAST NEXT sysdate + 7
   AS SELECT * FROM employees;
```

Voici maintenant la conversion automatique du code réalisée par Ora2Pg lorsque PG_SUPPORT_MVIEW est désactivé :

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

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

CREATE OR REPLACE FUNCTION create_materialized_view(text, text, text)
RETURNS VOID
AS $$
```

```
DECLARE
    mview ALIAS FOR $1; -- name of the materialized view to create
    vname ALIAS FOR $2; -- name of the related view
    iname ALIAS FOR $3; -- name of the colum of mview to used as unique key
    entry materialized_views%ROWTYPE;
BEGIN
    EXECUTE 'SELECT * FROM materialized_views
             WHERE mview_name = ' || quote_literal(mview) INTO entry;
    IF entry.iname IS NOT NULL THEN
        RAISE EXCEPTION 'Materialized view % already exist.', mview;
    END IF;

    EXECUTE 'REVOKE ALL ON ' || quote_ident(vname) || ' FROM PUBLIC';
    EXECUTE 'GRANT SELECT ON ' || quote_ident(vname) || ' TO PUBLIC';
    EXECUTE 'CREATE TABLE ' || quote_ident(mview) || '
             AS SELECT * FROM ' || quote_ident(vname);
    EXECUTE 'REVOKE ALL ON ' || quote_ident(mview) || ' FROM PUBLIC';
    EXECUTE 'GRANT SELECT ON ' || quote_ident(mview) || ' TO PUBLIC';
    INSERT INTO materialized_views (mview_name, view_name, iname, last_refresh)
    VALUES (
        quote_literal(mview),
        quote_literal(vname),
        quote_literal(iname),
        CURRENT_TIMESTAMP
    );
    IF iname IS NOT NULL THEN
        EXECUTE 'CREATE INDEX ' ||
                 quote_ident(mview) || '_' || quote_ident(iname)
                 || '_idx ON ' ||
             quote_ident(mview) || '(' || quote_ident(iname) || ')';
    END IF;

    RETURN;
END
$$
SECURITY DEFINER
LANGUAGE plpgsql;

CREATE OR REPLACE FUNCTION drop_materialized_view(text) RETURNS VOID
AS
$$
DECLARE
    mview ALIAS FOR $1;
    entry materialized_views%ROWTYPE;
BEGIN
    EXECUTE 'SELECT * FROM materialized_views
```

```
            WHERE mview_name = ''' || quote_literal(mview) || ''''
            INTO entry;
    IF entry.iname IS NULL THEN
        RAISE EXCEPTION 'Materialized view % does not exist.', mview;
    END IF;

    IF entry.iname IS NOT NULL THEN
        EXECUTE 'DROP INDEX ' ||
                quote_ident(mview) || '_' || entry.iname  || '_idx';
    END IF;
    EXECUTE 'DROP TABLE ' || quote_ident(mview);
    EXECUTE 'DELETE FROM materialized_views
            WHERE mview_name=''' || quote_literal(mview) || '''';

    RETURN;
END
$$
SECURITY DEFINER
LANGUAGE plpgsql ;

CREATE OR REPLACE FUNCTION refresh_full_materialized_view(text) RETURNS VOID
AS $$
DECLARE
    mview ALIAS FOR $1;
    entry materialized_views%ROWTYPE;
BEGIN
    EXECUTE 'SELECT * FROM materialized_views
             WHERE mview_name = ''' || quote_literal(mview) || ''''
             INTO entry;
    IF entry.iname IS NULL THEN
        RAISE EXCEPTION 'Materialized view % does not exist.', mview;
    END IF;

    IF entry.iname IS NOT NULL THEN
        EXECUTE 'DROP INDEX ' ||
                quote_ident(mview) || '_' || entry.iname  || '_idx';
    END IF;
    EXECUTE 'TRUNCATE ' || quote_ident(mview);
    EXECUTE 'INSERT INTO ' || quote_ident(mview) ||
            ' SELECT * FROM ' || entry.view_name;
    EXECUTE 'UPDATE materialized_views SET last_refresh=CURRENT_TIMESTAMP
            WHERE mview_name=''' ||
        quote_literal(mview) || '''';

    IF entry.iname IS NOT NULL THEN
        EXECUTE 'CREATE INDEX ' || quote_ident(mview) || '_' ||
```

```
                entry.iname  || '_idx ON ' || quote_ident(mview) ||
            '(' || entry.iname || ')';
    END IF;

    RETURN;
END
$$
SECURITY DEFINER
LANGUAGE plpgsql ;

CREATE 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;

SELECT create_materialized_view('emp_data','emp_data_mview',
       change with the name of the colum to used for the index);
```

On voit que quels que soient le type et la complexité de la vue matérialisée, Ora2Pg la transforme en vue rafraîchie à la demande. Dans ce cas, obtenir le code source n'a donc pas un grand intérêt, c'est pourquoi ce n'est pas réalisé par le script d'export.

La table `materialized_views` et les différentes fonctions (`create_materialized_view`, `drop_materialized_view`, `refresh_full_materialized_view`) ne sont générées qu'une seule fois quelque soit le nombre de vues matérialisées créées.

Vient ensuite la création de la vue `emp_data_mview` qui servira à la vue matérialisée qui est créée par l'appel à la fonction `create_materialized_view` et qui doit remplacer le commentaire *change with the name of the colum to used for the index* dans le code de l'appel de la fonction.

Dans l'exemple, la table `EMPLOYEE` possède une clé primaire sur le champ `EMPLOYEE_ID`, c'est ce champ qui va aussi être utilisé pour créer un index sur la table de la vue matérialisée.

```
SELECT create_materialized_view('emp_data','emp_data_mview', 'employee_id');
```

Par la suite pour rafraîchir la vue matérialisée il suffit d'appeler la fonction de rafraîchissement :

```
SELECT refresh_full_materialized_view('emp_data');
```

Pour plus d'information à ce sujet, reportez-vous à l'excellent article de Benjamin Arai

Materialized Views in PostgreSQL[7] .

3.6.9 EXPORT DES SYNONYMS

PostgreSQL ne possède pas d'objet de type SYNONYM :

- Ce sont des alias vers des objets d'autres schéma 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 SYNONYM 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 SYNONYM sous Oracle :

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

Les SYNONYM 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 générera Ora2Pg lors de l'export des synonymes.

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

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

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

[7] http://www.benjaminarai.com/benjamin_arai/index.php?display=/postgresql_materialized_views.php

17.12

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 crée 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;
```

3.6.10 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 utilisé 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 créer 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 ',');
```

3.6.11 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
```

17.12

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[8]

3.6.12 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

[8] http://pgxn.org/dist/oracle_fdw/

- Si le type BFILE est redéfini en TEXT, stockage du chemin du fichier externe

---

3.6.13 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)

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,...

Il existe aussi une extension PostgreSQL nommée external_file[9] qui permet d'émuler les DIRECTORY et BFILE d'Oracle. Si le type PostgreSQL associé au type Oracle dans la directive de configuration DATA_TYPE est positionné à EFILE (...,BFILE:EFILE,...), Ora2Pg fera les conversions nécessaires pour utiliser ce type.

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

17.12

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).

3.6.14 RECHERCHE PLEIN TEXTE

Oracle Index Texte

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

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 indexes GiST et GIN permettant de créer un index sur une colonne texte pour les recherches rapide 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+anglais 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,english_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.

- Creation 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.

- 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 stats 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 : http://docs.postgresql.fr/9.4/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.

3.6.15 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
```

17.12

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.

3.6.16 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).

3.6.17 IMPORT DIFFÉRÉ

Chargements 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.

3.6.18 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 type 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 forcement 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.

3.6.19 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
```

3.7 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 ?

3.7.1 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'instruction `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 suppression/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.

3.7.2 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 processus. 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 20000, 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.

3.7.3 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` et dans ce cas Ora2Pg renvoi 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 provoque 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`.

3.7.4 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.

3.7.5 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.

3.7.6 RESTAURATION PARALLÉLISÉE DES CONTRAINTES

Action :
110

- 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 le 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.

3.7.7 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*.

3.7.8 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 » :

17.12

```
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
```

3.7.9 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
```

17.12

```
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 64GB 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.

3.7.10 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 utilisateur de l'export ou limité 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

17.12

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

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

Quelquefois 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`.

3.8 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'amélioration ou de correction, 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 procédures stockées.

3.8.1 POUR ALLER PLUS LOIN

- Documentation officielle

- Autres sources d'informations

Vous pouvez retrouver la documentation en ligne en anglais sur le site officiel http://ora2pg.darold.net/.

Une série de documents concernant la migration Oracle vers PostgreSQL est disponible sur le wiki PostgreSQL : http://wiki.postgresql.org/wiki/Converting_from_other_Databases_to_PostgreSQL#Oracle

---

3.8.2 QUESTIONS

N'hésitez pas, c'est le moment !

3.9 TRAVAUX PRATIQUES

3.9.1 ÉNONCÉS

Installation

Environnement

Vérifier que les logiciels de compilation sont installés.

Créer l'utilisateur `postgres` avec comme répertoire personnel `/opt/pgsql/` .

Installation d'Oracle client

Installer le client Oracle dans `/opt/oracle/11.2/` et `DBD::Oracle`.

Installation de PostgreSQL

Installer le client et serveur PostgreSQL dans `/opt/pgsql/9.3/` et `DBD::Pg`.

Téléchargement d'Ora2Pg

Consulter le site officiel du projet et relevez la dernière version d'Ora2Pg.

Télécharger les fichiers sources de la dernière version et les placer dans `/opt/ora2pg/src`.

Compilation et installation d'Ora2Pg

Compiler et installer les sources.

Créer l'espace de travail

17.12

Créer une arborescence de travail destinée à recevoir les fichiers du projet de migration sous `/opt/ora2pg/tp_migration`.

Configuration

Configurer la connexion Oracle

Vérifier la connexion à la base 0racle avec `sqlplus`.

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.

3.9.2 SOLUTIONS

Installation

Environnement

Vérifier que les logiciels de compilation sont installés :

```
dpkg -l | grep -E 'gcc|make'
```

Sous Fedora, RedHat ou CentOS, il faudra utiliser `yum` :

```
yum list | grep -E 'gcc|make'
```

Créer l'utilisateur système `postgres` avec pour HOME `/opt/pgsql` :

```
useradd -d /opt/pgsql -m -r -s /bin/bash postgres
```

Installation d'Oracle

Télécharger les librairies client Oracle :

http://www.oracle.com/ -> *Download* -> *Oracle Instant Client* -> *Instant Client for Linux x86-64* ou *Instant Client for Linux x86*

Vous devez auparavant accepter la licence en cochant la case *Accept License Agreement* et posséder un compte Oracle. Voici les archives à télécharger :

```
instantclient-basic-linux-11.2.0.3.0.zip
instantclient-sqlplus-linux-11.2.0.3.0.zip
instantclient-sdk-linux-11.2.0.3.0.zip
```

Veillez à télécharger la version 64bit si vous possédez une telle architecture :

```
instantclient-basic-linux.x64-11.2.0.3.0.zip
instantclient-sqlplus-linux.x64-11.2.0.3.0.zip
instantclient-sdk-linux.x64-11.2.0.3.0.zip
```

Installer aussi la librairie `libaio1` depuis les paquets de la distribution :

```
apt-get install libaio1
```

Installez ensuite les paquets téléchargés dans le répertoire d'installation :

```
unzip -x instantclient-basic-linux-11.2.0.3.0.zip
mv instantclient_11_2/ /opt/oracle/11.2/
unzip -x instantclient-sdk-linux-11.2.0.3.0.zip
mv instantclient_11_2/* /opt/oracle/11.2/instantclient_11_2/
unzip -x instantclient-sqlplus-linux-11.2.0.3.0.zip
mv instantclient_11_2/* /opt/oracle/11.2/instantclient_11_2/
```

Installation du driver DBI

Le driver Oracle pour Perl DBI peut être trouvé à partir du module de recherche du site CPAN :

17.12

http://search.cpan.org/search?query=DBD::Oracle

Voici la procédure d'installation complète de la dernière version de `DBD::Oracle` :

```
wget http://search.cpan.org/CPAN/authors/id/P/PY/PYTHIAN/DBD-Oracle-1.74.tar.gz
tar xzf DBD-Oracle-1.74.tar.gz
cd DBD-Oracle-1.74/
export ORACLE_HOME=/opt/oracle/11.2/instantclient_11_2/
LD_LIBRARY_PATH=/opt/oracle/11.2/instantclient_11_2/ perl Makefile.PL
make
make install
```

Installation de PostgreSQL

Si PostgreSQL est déjà installé, passez directement à la section « Installation du driver DBI » de ce chapitre.

Se connecter en tant qu'utilisateur `postgres` :

```
su - postgres
```

Téléchargement

Sur le site officiel du projet, télécharger les fichiers sources de la dernière version.

```
wget http://ftp.postgresql.org/pub/source/v9.6.1/postgresql-9.6.1.tar.gz
# commande suivante est équivalente à :
# tar --gunzip --extract --file postgresql-9.6.1.tar.gz
tar zxf postgresql-9.6.1.tar.gz
cd postgresql-9.6.1
```

Compilation et installation

Configurer puis compiler les sources de PostgreSQL :

```
./configure --prefix /opt/pgsql/9.6
make
```

(cette opération peut prendre du temps)

Ensuite installer les fichiers obtenus :

```
make install
```

Dans ce TP, nous nous sommes attachés à changer le moins possible d'utilisateur système. Il se peut que vous ayez à installer les fichiers obtenus en tant qu'utilisateur `root` dans d'autres environnements en fonction de la politique de sécurité adoptée.

Configurer le système

Les binaires installés sont situés dans le répertoire `/opt/pgsql/9.6/bin`.

Éditer le fichier `~postgres/.profile` et ajouter la ligne suivante à la fin du fichier:

```
echo 'export PATH=/opt/pgsql/9.6/bin:$PATH' >> ~postgres/.profile
```

Lancement

Initialiser et démarrer le serveur. Toujours en tant qu'utilisateur `postgres` :

```
initdb -D /opt/pgsql/9.6/data
postmaster -D /opt/pgsql/9.6/data > /tmp/postgresql.log 2>&1
createdb test
psql test
```

Positionner la variable d'environnement `PGDATA`.

Éditer le fichier `~postgres/.profile` et ajouter la ligne suivante à la fin du fichier :

```
echo "export PGDATA=/opt/pgsql/9.6/data" >> ~postgres/.profile
```

Créer un script d'initialisation

Un script d'initialisation d'exemple pour linux est disponible dans les sources à l'emplacement suivant :

```
contrib/start-scripts/linux
```

Pour adapter ce script, le copier dans /etc/init.d/postgresql-tp" éditez le et modifiez les paramètres suivants :

```
prefix=/opt/pgsql/9.6
PGDATA="/opt/pgsql/9.6/data"
PGLOG="$PGDATA/serverlog"
```

Pour activer postgresql au démarrage, sous Fedora exécuter la commande suivante :

```
chkconfig --add postgresql-tp
```

Sous Debian :

```
update-rc.d postgresql-tp defaults
```

Installation du driver DBI

Le driver PostgreSQL pour Perl DBI peut être trouvé à partir du module de recherche du site CPAN :

```
<http://search.cpan.org/search?query=DBD::Pg>
```

Voici la procédure d'installation complète de la dernière version de `DBD::Pg`

```
wget http://search.cpan.org/CPAN/authors/id/T/TU/TURNSTEP/DBD-Pg-2.19.3.tar.gz
tar xzf DBD-Pg-2.19.3.tar.gz
cd DBD-Pg-2.19.3/
export POSTGRES_LIB="/opt/pgsql/9.6/lib -lssl -lcrypto"
perl Makefile.PL
make
make install
```

Téléchargement d'Ora2Pg

Consulter le site officiel du projet http://ora2pg.darold.net/, la page *News* indique la dernière version d'Ora2Pg. cpan Le téléchargement des fichiers sources de la dernière version peut se faire soit depuis le dépôt SourceForge à partir du navigateur, soit directement en ligne de commande :

```
mkdir -p /opt/ora2pg/src
cd /opt/ora2pg/src/
wget http://downloads.sourceforge.net/project/ora2pg/14.0/ora2pg-14.0.tar.bz2
```

Compilation et installation d'Ora2Pg

```
export ORACLE_HOME=/opt/oracle/11.2/instantclient_11_2/
tar xjf ora2pg-14.0.tar.bz2
cd ora2pg-14.0/
perl Makefile
make
make install
```

L'installation nécessite les droits administrateur.

Pour cela l'installation peut se faire à l'aide de `sudo` (`sudo make install`) soit directement sous l'utilisateur `root`.

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 /opt/ora2pg
```

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 nommé : `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 :

```
export ORACLE_HOME=/opt/oracle/11.2/instantclient_11_2/
sqlplus hr/phoenix@192.168.1.109:1521/xe
```

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/local/instantclient_11_2
```

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/XE
ORACLE_USER     hr
ORACLE_PWD      phoenix
```

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

```
USER_GRANTS     1
```

Test de la connexion Ora2Pg vers la base Oracle :

```
ora2pg -d -c config/ora2pg.conf -t SHOW_SCHEMA
    Using character set: NLS_LANG=AMERICAN_AMERICA.AL32UTF8, NLS_NCHAR=AL32UTF8.
    Using Perl output encoding :utf8.
    Using PostgreSQL client encoding UTF8.
Trying to connect to database: dbi:Oracle:host=192.168.1.150;sid=XE;port=1521
Isolation level: SET TRANSACTION ISOLATION LEVEL SERIALIZABLE
    Found Package: EMP_ACTIONS
    Found Package: EMP_MGMT
Looking at package emp_mgmt...
Looking at package emp_actions...
Showing all schema...
SCHEMA HR
```

La connexion est opérationnelle.

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

```
SCHEMA  CHANGE_THIS_SCHEMA_NAME
```

par

```
SCHEMA  HR
```

Pour lister les tables de l'instance :

```
[1] TABLE COUNTRIES (25 rows)
[2] TABLE DEPARTMENTS (27 rows)
[3] TABLE EMPLOYEES (107 rows)
[4] TABLE JOBS (19 rows)
[5] TABLE JOB_HISTORY (10 rows)
[6] TABLE LOCATIONS (23 rows)
[7] TABLE PHONE (1 rows)
[8] TABLE REGIONS (4 rows)
[9] TABLE SSN (1 rows)
[10] TABLE TEST (1 rows)
```

```
-------------------------------------------------------
Total number of rows: 218

Top 10 of tables sorted by number of rows:
    [1] TABLE EMPLOYEES has 107 rows
    [2] TABLE DEPARTMENTS has 27 rows
    [3] TABLE COUNTRIES has 25 rows
    [4] TABLE LOCATIONS has 23 rows
    [5] TABLE JOBS has 19 rows
    [6] TABLE JOB_HISTORY has 10 rows
    [7] TABLE REGIONS has 4 rows
    [8] TABLE SSN has 1 rows
    [9] TABLE TEST has 1 rows
    [10] TABLE PHONE has 1 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 :

```
cd /opt/ora2pg/tp_migration
sh export_schema.sh
```

Voici la listes des commandes exécutées par le script :

```
Running: ora2pg -p -t TABLE \
         -o table.sql -b ./schema/tables -c ./config/ora2pg.conf
Running: ora2pg -p -t PACKAGE \
         -o package.sql -b ./schema/packages -c ./config/ora2pg.conf
Running: ora2pg -p -t VIEW \
         -o view.sql -b ./schema/views -c ./config/ora2pg.conf
Running: ora2pg -p -t GRANT \
         -o grant.sql -b ./schema/grants -c ./config/ora2pg.conf
Running: ora2pg -p -t SEQUENCE \
         -o sequence.sql -b ./schema/sequences -c ./config/ora2pg.conf
Running: ora2pg -p -t TRIGGER \
         -o trigger.sql -b ./schema/triggers -c ./config/ora2pg.conf
Running: ora2pg -p -t FUNCTION \
```

```
        -o function.sql -b ./schema/functions -c ./config/ora2pg.conf
Running: ora2pg -p -t PROCEDURE \
        -o procedure.sql -b ./schema/procedures -c ./config/ora2pg.conf
Running: ora2pg -p -t TABLESPACE \
        -o tablespace.sql -b ./schema/tablespaces -c ./config/ora2pg.conf
Running: ora2pg -p -t PARTITION \
        -o partition.sql -b ./schema/partitions -c ./config/ora2pg.conf
Running: ora2pg -p -t TYPE \
        -o type.sql -b ./schema/types -c ./config/ora2pg.conf
Running: ora2pg -p -t MVIEW \
        -o mview.sql -b ./schema/mviews -c ./config/ora2pg.conf
Running: ora2pg -p -t DBLINK \
        -o dblink.sql -b ./schema/dblinks -c ./config/ora2pg.conf
Running: ora2pg -p -t SYNONYM \
        -o synonym.sql -b ./schema/synonyms -c ./config/ora2pg.conf
Running: ora2pg -p -t DIRECTORY \
        -o directorie.sql -b ./schema/directories -c ./config/ora2pg.conf
```

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

Voici l'arbre des fichiers générés :

```
tp_migration/
```

```
config
    ora2pg.conf
data
export_schema.sh
reports
    columns.txt
    report.html
    tables.txt
schema
    functions
        function.sql
    grants
        grant.sql
    mviews
        DEPARTMENTS_MV1_mview.sql
        MV_DEPARTMENTS_mview.sql
        mview.sql
    packages
        package.sql
    partitions
        partition.sql
    procedures
        ADD_JOB_HISTORY_procedure.sql
        procedure.sql
        SECURE_DML_procedure.sql
    sequences
        sequence.sql
    tables
        CONSTRAINTS_table.sql
        INDEXES_table.sql
        table.sql
    tablespaces
        tablespace.sql
        TBSP_INDEXES_tablespace.sql
    triggers
        CHECK_RAISE_ON_AVG_trigger.sql
        trigger.sql
        UPDATE_JOB_HISTORY_trigger.sql
    types
```

```
            type.sql
        views
            EMP_DETAILS_VIEW_view.sql
            view.sql
    sources
        functions
            function.sql
        mviews
            DEPARTMENTS_MV1_mview.sql
            MV_DEPARTMENTS_mview.sql
            mview.sql
        packages
            package.sql
        partitions
            partition.sql
        procedures
            ADD_JOB_HISTORY_procedure.sql
            procedure.sql
            SECURE_DML_procedure.sql
        triggers
            CHECK_RAISE_ON_AVG_trigger.sql
            trigger.sql
            UPDATE_JOB_HISTORY_trigger.sql
        types
            type.sql
        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/types/type.sql
```

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

Vérification de la bonne application du script d'import :

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

```
                    Liste des relations
Schéma|       Nom   |Type | Propriét.| Taille | Description
------+------------+-----+----------+--------+-----------------------------------
public| countries   |table| migration| 0 bytes| country table. Contains 25 rows.
      |             |     |          |        | References with locations table.
public| departments|table| migration| 0 bytes| Departments table that shows
      |             |     |          |        | details of departments where
      |             |     |          |        | employees work. Contains 27 rows;
      |             |     |          |        | references with locations,
      |             |     |          |        | employees, and job_history tables.
public| emp_details|vue   | migration| 0 bytes|
      | _view       |     |          |        |
public| employees   |table| migration| 0 bytes| employees table. Contains 107 rows.
      |             |     |          |        | References with departments,
      |             |     |          |        | jobs, job_history tables.
      |             |     |          |        | Contains a self reference.
public| job_history|table| migration| 0 bytes| Table that stores job history of
      |             |     |          |        | the employees. If an employee
      |             |     |          |        | changes departments within the job
      |             |     |          |        | or changes jobs within the department,
      |             |     |          |        | new rows get inserted into this
      |             |     |          |        | table with old job information of
      |             |     |          |        | the employee. Contains a complex
      |             |     |          |        | primary key: employee_id+start_date.
      |             |     |          |        | Contains 25 rows. References with
      |             |     |          |        | jobs, employees, and departments
      |             |     |          |        | tables.
public| jobs        |table| migration| 0 bytes| jobs table with job titles and
      |             |     |          |        | salary ranges. Contains 19 rows.
      |             |     |          |        | References with employees and
      |             |     |          |        | job_history table.
public| locations   |table| migration| 0 bytes| Locations table that contains
      |             |     |          |        | specific address of a specific
      |             |     |          |        | office, warehouse, and/or
```

```
      |            |     |          |        | production site of a company.
      |            |     |          |        | Does not store addresses /
      |            |     |          |        | locations of customers. Contains
      |            |     |          |        | 23 rows; references with the
      |            |     |          |        | departments and countries tables.
public| person_typ |table| migration| 0 bytes|
public| phone      |table| migration| 8192 b |
public| regions    |table| migration| 0 bytes|
public| ssn        |table| migration| 8192 b |
public| student_typ|table| migration| 0 bytes|
public| test       |table| migration| 8192 b |
(13 lignes)
```

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
```

```
[========================>] 10/10 tables (100.0%) end of scanning.
[========================>] 25/25 rows (100.0%) Table COUNTRIES (25.0 recs/sec)
[==>                      ]  25/219 rows (11.4%) on total data (avg: 25.0 recs/sec)
[========================>] 27/27 rows (100.0%) Table DEPARTMENTS (27.0 recs/sec)
[=====>                   ]  52/219 rows (23.7%) on total data (avg: 52.0 recs/sec)
[========================>] 107/107 rows (100.0%) Table EMPLOYEES (107.0 recs/sec)
[================>        ] 159/219 rows (72.6%) on total data (avg: 159.0 recs/sec)
[========================>] 19/19 rows (100.0%) Table JOBS (19.0 recs/sec)
[==================>      ] 178/219 rows (81.3%) on total data (avg: 178.0 recs/sec)
[========================>] 10/10 rows (100.0%) Table JOB_HISTORY (10.0 recs/sec)
[===================>     ] 188/219 rows (85.8%) on total data (avg: 188.0 recs/sec)
[========================>] 23/23 rows (100.0%) Table LOCATIONS (23.0 recs/sec)
[======================== ] 211/219 rows (96.3%) on total data (avg: 211.0 recs/sec)
[========================>] 2/2 rows (100.0%) Table PHONE (2.0 recs/sec)
[======================== ] 213/219 rows (97.3%) on total data (avg: 213.0 recs/sec)
[========================>] 4/4 rows (100.0%) Table REGIONS (4.0 recs/sec)
[======================== ] 217/219 rows (99.1%) on total data (avg: 217.0 recs/sec)
[========================>] 2/2 rows (100.0%) Table SSN (2.0 recs/sec)
[========================>] 219/219 rows (100.0%) on total data (avg: 219.0 recs/sec)
[========================>] 0/0 rows (100.0%) Table TEST (0.0 recs/sec)
```

```
[========================>] 219/219 rows (100.0%) on total data (avg: 219.0 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
   PHONE_data.sql
   REGIONS_data.sql
   SSN_data.sql
   TEST_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 :

```
SET client_encoding TO 'WIN1252';

\set ON_ERROR_STOP ON

BEGIN;

\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/PHONE_data.sql
\i data/REGIONS_data.sql
\i data/SSN_data.sql
\i data/TEST_data.sql


COMMIT;
```

17.12

Exécutons le chargement :

```
psql -U migration pghr -f ./data/data.sql
```

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, indexes, séquences et triggers.

```
psql -U migration pghr -f schema/tables/CONSTRAINTS_table.sql
psql -U migration pghr -f schema/tables/INDEXES_table.sql
psql -U migration pghr -f schema/triggers/trigger.sql --single-transaction
```

Seul la dernière commande génère une erreur :

```
psql:./schema/triggers/UPDATE_JOB_HISTORY_trigger.sql:17:
    ERROR:  syntax error at or near "add_job_history"
LIGNE 3 :   add_job_history(OLD.employee_id, OLD.hire_date, LOCALTIMES...
```

Ceci est dû au fait que le trigger utilise la fonction `add_job_history()` qui n'a pas encore été créé, il faut donc l'importer avant de créer le trigger

4 PROCÉDURES STOCKÉES

4.1 INTRODUCTION

Ce module est organisé en cinq parties :

1. Outils et méthodes
2. Différences de syntaxes
3. Conversion automatique
4. Migration des procédures stockées
5. 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.

4.2 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.

4.2.1 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[10] . 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

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()`

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

- `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.

4.2.2 LES OUTILS DE CONVERSION

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

Ora2Pg[11] 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.

4.2.3 LES OUTILS DE DÉBOGAGE

- pldebugger (ex edb-debugger)

[11] http://ora2pg.darold.net/

- plpgsql_lint
- 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_lint`, un module pour PostgreSQL 9.0 et plus 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[12] ;
- `plpgsql_lint` de Pavel Stehule peut être téléchargé ici[13] .

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

Il existe aussi SQLMaestro[14] , un outil propriétaire qui permet l'exécution pas à pas du code PL/pgSQL.

4.3 DIFFÉRENCES DE SYNTAXES

- Différences au niveau du schéma
- Différences de type de données
- Différences dans le code

Cette partie dresse une liste exhaustive des différences majeures entre Oracle et PostgreSQL

4.3.1 DIFFÉRENCES DE SCHÉMA - 1

- Le schéma sous Oracle : USER.OBJECT
  - sous PostgreSQL, véritable espace de nommage
- Oracle convertit les noms d'objet en majuscule : NOM_TABLE
  - PostgreSQL les convertit en minuscule : nom_table
- Les types de données doivent être redéfinis (NUMBER(p,s), CLOB, etc.)
- Les synonymes n'existent pas
  - utiliser des vues si nécessaire.

[12] https://git.postgresql.org/gitweb/?p=pldebugger.git
[13] https://github.com/okbob/plpgsql_lint
[14] http://www.sqlmaestro.com/products/postgresql/maestro/tour/pgsql_debugger/

- Les vues matérialisées existent depuis PosgreSQL 9.3
  - mais manquent encore de fonctionnalités

Les schémas

Sous PostgreSQL, les schémas sont de véritables espaces de nommage dont on peut changer le propriétaire, alors qu'un schéma Oracle n'est ni plus ni moins qu'un utilisateur auquel des objets seront associés.

Sensibilité à la casse

Lorsque les noms des objets ne sont pas écrits entre guillemets doubles, Oracle les transforme en majuscule alors que PostgreSQL les transforme toujours en minuscule. S'ils sont écrits entre guillemets doubles, les deux ont le même comportement : le nom est pris tel qu'écrit.

Si vous avez créé vos objets avec des guillemets doubles sous Oracle et que vous les exportez aussi avec des guillemets doubles, vous devrez toujours inclure ces guillemets doubles dans le code de vos requêtes lorsque vous ferez appel à un objet. C'est donc déconseillé, sous Oracle comme sous PostgreSQL.

Types de données

Les types de données sont différents entre les deux SGBD.

Synonymes

Les synonymes d'Oracle n'ont pas d'équivalent sous PostgreSQL. Il doit être possible d'utiliser des vues pour tenter d'émuler cette fonctionnalité dans la mesure où il s'agit d'accéder à des objets d'autres schémas.

Vues matérialisées

Concernant les vues matérialisées, elles existent sous PostgreSQL depuis la version 9.3. Cependant, elles ne disposent pas de toutes les fonctionnalités accessibles sous Oracle. Il est possible de les implémenter de toutes pièces en utilisant des fonctions et triggers. En attendant leur implémentation complète au cœur du code de PostgreSQL, voici des documents expliquant de manière détaillée comment implémenter des vues matérialisées :

- PostgreSQL/Materialized Views[15]
- Materialized Views that Really Work[16]

[15] http://tech.jonathangardner.net/wiki/PostgreSQL/Materialized_Views
[16] http://www.pgcon.org/2008/schedule/events/69.en.html

4.3.2 DIFFERENCES DE SCHÉMA - 2

- La création des tables est entièrement compatible mais :
  - les tables temporaires globales n'existent pas sous PostgreSQL
  - INITTRANS, MAXEXTENTS sont inutiles (et n'existent pas)
  - PCTFREE correspond au paramètre `fillfactor`
  - PCTUSED est inutile (et n'existe pas)
- Les partitions n'existent pas directement
  - héritage, trigger et contraintes check
- Les colonnes virtuelles non plus
  - utilisation de vues

Création de table

La définition des tables est quasiment identique pour les deux SGBD à la différence près que PostgreSQL n'a pas de table temporaire dont les données insérées ne persistent que le temps d'une transaction ou d'une session. Sous PostgreSQL c'est la table elle-même qui est supprimée à la fin de la session.

Il n'y a pas non plus de notion de réservation de nombre de transactions allouées à chaque bloc ou d'extents.

`PCTFREE` qui indique (en pourcentage) l'espace que l'on souhaite conserver dans le bloc pour les mises à jour, correspond au `fillfactor` sous PostgreSQL. PCTUSED n'existe pas (il n'a pas de sens dans l'implémentation de PostgreSQL).

```
CREATE TABLE distributors (
    did     integer,
    name    varchar(40),
    UNIQUE(name) WITH (fillfactor=70)
)
WITH (fillfactor=70);
```

Partitionnement

Les partitions telles que gérées sous Oracle n'existent pas sous PostgreSQL. Il faut utiliser l'héritage et définir des triggers et contraintes CHECK. Pour plus de détails, consultez le document 5.9. Partitioning[17] ainsi que le document Partitionnement (dans la KB Dalibo)[18] . Les types de partitions supportées sont `List` et `Range`. Les partitions de type `Hash` ne sont pas supportées par PostgreSQL.

Colonnes virtuelles

[17] http://www.postgresql.org/docs/9.1/static/ddl-partitioning.html
[18] https://kb.dalibo.com/formations/partitionnement/presentation

Pour remplacer les colonnes virtuelles, les vues sont idéales. Voici un exemple de définition de colonne virtuelle sous Oracle :

```
CREATE TABLE employees (
  id          NUMBER,
  first_name  VARCHAR2(10),
  salary      NUMBER(9,2),
  commission  NUMBER(3),
  salary2     NUMBER GENERATED ALWAYS AS
              (ROUND(salary*(1+commission/100),2)) VIRTUAL,
);
```

Et voici la version à base d'une vue dans PostgreSQL :

```
CREATE TABLE employees (
  id          bigint,
  first_name  varchar(10),
  salary      double precision,
  commission  integer
);

CREATE VIEW virt_employees AS SELECT id, first_name, salary, commission,
    (ROUND((salary*(1+commission/100))::numeric,2)) salary2
FROM employees;
```

Vous pouvez aussi utiliser les triggers pour mettre à jour les colonnes et permettre l'utilisation d'un index sur cette colonne, mais, dans ce cas, la colonne ne sera plus vraiment "virtuelle".

4.3.3 DIFFÉRENCES DE SCHÉMA - 3

- Les contraintes sont identiques (clés primaires, étrangères et uniques, ...).
- Les index : btree uniquement, les autres n'existent pas (bitmap principalement).
- Les tablespaces : la même chose dans sa fonctionnalité principale.
- Les types utilisateurs (CREATE TYPE) nécessitent une réécriture.
- Les liens inter bases (`DBLINK`) n'existent pas sauf sous forme d'extension (dblink ou fdw).

Les contraintes

L'ensemble des contraintes fonctionne exactement de la même manière, que ce soit pour les clés primaires, les clés étrangères et les clés uniques ou pour les contraintes `CHECK` et `NOT NULL`.

Les index

Pour les index, seule la forme BTREE correspond, les autres ne sont pas implémentées mais PostgreSQL dispose lui-aussi d'autres types d'index. Quoiqu'il en soit, la plupart des index utilisés sont des index de type BTREE.

Les index BITMAP sur disque n'existent pas sous PostgreSQL. Ils sont créés en mémoire si nécessaire à partir des index de type BTREE.

Les index IOT ne sont pas non plus supportés et peuvent être simulés à l'aide de la commande CLUSTER qui trie une table en fonction de l'index.

Tablespaces

Les tablespaces correspondent, dans leur fonctionnalité principale, à ce qui est fait sur Oracle, à savoir à définir un espace du système de fichiers où un plusieurs objets de la base pourront être stockés. Il n'y a pas de notion de taille initiale ni d'extension du tablespace sous PostgreSQL si ce n'est les limites imposées par le système de fichiers.

Types utilisateur

L'ensemble des types pouvant être défini par un utilisateur sont supportés avec plus ou moins d'adaptation. Il peut notamment être nécessaire de redéfinir des fonctions d'entrée / sortie définissant le comportement lors d'une insertion / lecture sur les données du type. Dans la plupart des cas, il s'agit de types composites ou de tableaux parfaitement supportés par PostgreSQL.

Exemple de type composite version Oracle :

```
CREATE OR REPLACE TYPE phone_t AS OBJECT (
    a_code   CHAR(3),
    p_number CHAR(8)
);
```

et la version PostgreSQL :

```
CREATE TYPE phone_t AS (
    a_code   char(3),
    p_number char(8)
);
```

Exemple d'un tableau de type :

```
CREATE OR REPLACE TYPE phonelist AS VARRAY(50) OF phone_t;
```

qui sera traduit en :

```
CREATE TYPE phonelist AS (phonelist phone_t[50]);
```

dblink

PostgreSQL ne permet pas d'accéder nativement à une autre base de données à l'intérieur d'une requête SQL. Il est cependant possible d'utiliser les extensions `dblink` ou `Foreign Data Wrapper` pour accéder à des données à distance mais sans pour autant pouvoir utiliser une notation à base de `@` dans la requête.

4.3.4 DIFFÉRENCES SUR LES TYPES DE DONNÉES

Opérations sur les dates :

- `DATE + NUMBER`
  - => `DATE + interval '1 jours'`
- `TIMESTAMP - TIMESTAMP` = `NUMBER`
  - Sous PostgreSQL => `interval`
- `NLS_DATE_FORMAT` (`TO_CHAR` et `TO_DATE`)
  - => `DateStyle`

Pas de conversion implicite vers et depuis les types chaînes de caractères :

```
SELECT * FROM depts WHERE numero BETWEEN 0 AND 42;
```

Opérations sur les dates

Oracle autorise l'ajout ou la soustraction d'un nombre entier à une date. Par exemple :

```
SELECT SYSDATE + 1 FROM DUAL;
```

retournera la date de demain. Pour obtenir le même résultat avec PostgreSQL, il faut utiliser un intervalle :

```
SELECT now() + interval '1 day';
```

De même, la soustraction d'un timestamp à un autre retourne un nombre correspondant au nombre de jour entre ces deux dates, alors que, sous PostgreSQL, cette opération retourne un intervalle.

Pour Oracle, le format défini par `NLS_DATE_FORMAT` détermine le format des dates qui sera utilisé pour la sortie des fonctions `TO_CHAR()` et `TO_DATE()`. Avec PostgreSQL, cela dépend du format défini par la variable de configuration `DateStyle` (par défaut `ISO, DMY`).

Conversion implicite

Les conversions implicites de et vers un champ de type texte ont été supprimées sous PostgreSQL depuis la version 8.3. Par exemple, il n'est pas possible de faire ce type de requête :

17.12

```
create table depts ( numero char(2), nom varchar(25) );
pghr=# select * from depts where numero between 0 AND 42;
ERROR:  operator does not exist: character >= integer
LIGNE 1 : select * from depts where numero between 0 AND 42;
```

Si l'on veut pouvoir faire fonctionner cette requête, il faut réaliser une conversion :

```
select * from depts where numero::integer between 0 AND 42;
```

Avec Oracle, ce type de conversion est implicite.

4.3.5 DIFFÉRENCES DANS LE CODE - GÉNÉRAL 1

- nom_sequence.nextval => nextval('nom_sequence')
- Pas de transaction autonome à moins de passer par dblink
- 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 le module `dblink`.

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`.

4.3.6 DIFFÉRENCES DANS LE CODE - GÉNÉRAL 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 :

17.12

```
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 FUNCTION tousLesAuteurs() RETURNS VOID
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 vide et NULL

Oracle traite les chaînes vide 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.

4.3.7 DIFFÉRENCES DANS LE CODE - TRIGGER

- 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
$$
DECLARE
    noitem integer;
BEGIN
    select into noitem max(no_produit) from produit;
    IF noitem ISNULL THEN
      noitem:=0;
    END IF;
    NEW.no_produit:=noitem+1;
    RETURN NEW;
  END;
$$
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;
```

4.3.8 DIFFÉRENCES DE CODE - FONCTIONS

- PostgreSQL n'a que des fonctions
  - une procédure retourne VOID
- 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

PostgreSQL ne connaît que les déclarations de fonctions. Pour PostgreSQL, une procédure n'est ni plus ni moins qu'une fonction qui retourne `VOID`.

À la déclaration d'une fonction, s'il n'y a pas de paramètre 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 VOID AS ...
```

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';
```

4.3.9 DIFFÉRENCES DANS LE CODE - PACKAGES

- Paquet de variables et de procédures stockées
- 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 paquet 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 fonctions de ces paquets (`nom_paquet.nom_fonction`), la solution est de créer sous PostgreSQL un schéma portant le même nom que le paquet. L'appel aux fonctions se fera alors de façon identique : `nom_schema.nom_fonction`.

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, on peut définir un nouveau paramètre dans le fichier `postgresql.conf` comme `custom_variable_classes` pour les versions de PostgreSQL avant la 9.2 :

```
custom_variable_classes = 'globalvar'
```

Nous avons maintenant à disposition un espace de déclaration de variables utilisables dans le code SQL et accessibles depuis toutes les bases de données du cluster PostgreSQL.

Pour les versions depuis la 9.2, la variable peut être déclarée directement dans le fichier de configuration avec une valeur initiale :

```
globalvar.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 globalvar.id_region = '38';
```

ou la fonction `current_setting()` :

```
select set_config('globalvar.ma_variable','12',false);
```

et pour utiliser sa valeur :

```
SELECT current_setting('globalvar.id_region') AS id_region;
```

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.

4.3.10 OUTER JOIN ORACLE (+) VERSUS JOINTURES ANSI - 1/2

- Left outer join :

```
SELECT * FROM a, b WHERE a.id = b.id (+)
=> SELECT * FROM a LEFT OUTER JOIN b ON (a.id = b.id)
```

- Right outer join :

```
SELECT * FROM  a, c WHERE a.id (+) = c.id
=> SELECT * FROM a RIGHT OUTER JOIN c ON (a.id = c.id)
```

4.3.11 OUTER JOIN ORACLE (+) VERSUS JOINTURES ANSI - 2/2

Full outer join :

```
SELECT * FROM a, b WHERE a.id = b.id (+)
UNION ALL SELECT * FROM a, b WHERE a.id (+) = b.id AND a.id = NULL
=> SELECT * FROM a FULL OUTER JOIN b ON (a.id = b.id)
```

Jointure externe à gauche et à droite

Le SGBD Oracle utilise la notation (+) pour décrire le côté où se trouvent les valeurs NULL.

Pour une jointure à gauche, l'annotation (+) serait placée du côté droit (et inversement pour une jointure à droite). Cette forme n'est pas supportée par PostgreSQL. Il faut donc réécrire les jointures avec la notation normalisée : `LEFT OUTER JOIN` ou `LEFT JOIN` pour une jointure à gauche et `RIGHT OUTER JOIN` ou `RIGHT JOIN` pour une jointure à droite.

Par exemple :

17.12

```
SELECT nom,prenom,titre
FROM auteurs a , auteurs_livres al, livres l
WHERE a.id_auteur = al.ref_auteur
   AND al.ref_livre = l.id_livre(+);

SELECT nom,prenom,titre
FROM auteurs a , auteurs_livres al
    LEFT JOIN livres l ON l.id_livre = a.ref_livre
WHERE a.id_auteur = l.red_auteur;
```

4.4 CONVERSION AUTOMATIQUE DU CODE

- Paquets de procédure stockées
- Entêtes 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 entêtes 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 sixième partie va s'appliquer à décrire succinctement l'ensemble des conversions automatiques réalisées par Ora2Pg.

4.4.1 CONVERSIONS GLOBALES

- Les PACKAGES ou paquets de procédures stockées.
- Les déclarations de triggers et fonctions.
- Les paramètres des fonctions.
- 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 fonctions, 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 fonctions 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 `plpgsql` 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 fonctions, les entêtes 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 fonction 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 FUNCTION load_file (pdname text, psname text, pfname text)
RETURNS VOID AS
$body$
DECLARE
    src_file bytea;
    dst_file bytea;
    lgh_file integer;
BEGIN
    plpgsql block
END;
```

Dans ce cas, la procédure est transformée en fonction car, sous PostgreSQL, une procédure n'est qu'une fonction retournant VOID.

4.4.2 CORRESPONDANCE DES FONCTIONS - 1

Les noms diffèrent :

- `NVL()` => `coalesce()`
- `SYSDATE` => `LOCALTIMESTAMP`
  - équivalent de `CURRENT_TIMESTAMP` sans le fuseau horaire

4.4.3 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, ' ')`

Les astuces employés par Ora2Pg

Si certaines fonctions Oracle peuvent être remplacées directement par leur équivalent sous PostgreSQL, comme par exemple `NVL` par `coalesce` ou `SYSDATE` par `LOCALTIMESTAMP`, 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, '99999999999999999999D99999999999999999999')` 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.

4.4.4 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)
```

4.4.5 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 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)`

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)
```

4.4.6 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 fonction de trigger

4.4.7 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.

4.4.8 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`
- Ajout du mot clé `STRICT` aux `SELECT … INTO` lorsqu'il y a `EXCEPTION … NO_DATA_FOUND` ou `TOO_MANY_ROWS`
- Remplace les appels à `MINUS` par `EXCEPT`

4.4.9 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;`

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
```

4.4.10 RÉÉCRITURE DE PARTIES DE CODE - 6

- 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;`

4.4.11 RÉÉCRITURE DE PARTIES DE CODE - 7

- 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` dans les requêtes `SELECT ... INTO ...` si une exception sur `NO_DATA_FOUND` ou `TOO_MANY_ROW` est levée

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.

4.4.12 RÉÉCRITURE DE PARTIES DE CODE - 8

- 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.

4.4.13 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.

4.4.14 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.

4.5 MIGRATION DES PROCÉDURES STOCKÉES

Étapes :

- Cas des procédures avec transaction 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 ?

4.5.1 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.

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

Code Oracle :

17.12

```
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 fonction et la renommer avec le suffix `_atx` comme suit :

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

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

```
CREATE OR REPLACE FUNCTION log_action (username text, msg text) RETURNS VOID AS
$body$
DECLARE
    -- Change this to reflect the dblink connection string
    v_conn_str text := 'port=5432 dbname=testdb host=localhost '
                       'user=pguser password=pgpass';
    v_query text;
BEGIN
    v_query := 'SELECT true FROM log_action_atx ( ' || quote_literal(username)
               || ', ' || quote_literal(msg) || ' )';
    PERFORM * FROM dblink(v_conn_str, v_query) AS p (ret boolean);
END;
$body$
LANGUAGE plpgsql STRICT 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.

4.5.2 IMPORT DES PROCÉDURES ET PAQUETS AVEC ORA2PG

Chargement des fonctions et procédures :

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

Chargement des paquets de fonctions :

```
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`.

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.

4.5.3 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.

4.6 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 `plpgsql` ait été importé sans erreurs.

Pour vous aider, vous pouvez utiliser le débogueur `edb-debugger` qui vous indiquera la ligne problématique dans le code et `plpgsql_lint` 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.

4.6.1 OUTILS DE TESTS UNITAIRES POUR POSTGRESQL

- pgTap
  - http://www.pgtap.org/
- pgUnit
  - http://en.dklab.ru/lib/dklab_pgunit/
  - http://pgfoundry.org/projects/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[19] , vous trouverez un exemple d'implémentation Java avec le projet tap4j[20] .

4.6.2 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.

4.7 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.

4.7.1 QUESTIONS

N'hésitez pas, c'est le moment !

---

[19] http://testanything.org/
[20] http://sourceforge.net/projects/tap4j/

4.8 TRAVAUX PRATIQUES

4.8.1 ÉNONCÉS

Import des procédures stockées

Importer les fonctions

Importer les fonctions et procédures converties en PL/pgSQL.

Importer les procédures stockées

Créer le schéma associé au paquet de fonctions et charger les fonctions.

Tests unitaires

Créer un script de test simple des procédures stockées basé sur les commandes `psql` et `sqlplus`.

4.8.2 SOLUTIONS

Import des procédures stockées

Importer les procédures et fonctions

Commençons par importer la fonction manquante à notre trigger. Comme nous avons choisi d'exporter les fonctions dans des fichiers séparés, la fonction se trouve dans le fichier `schema/procedures/ADD_JOB_HISTORY_procedure.sql`.

```
psql -U migration pghr -f schema/procedures/ADD_JOB_HISTORY_procedure.sql
```

```
SET
psql:schema/procedures/ADD_JOB_HISTORY_procedure.sql:24: NOTICE:
    type reference job_history.employee_id%TYPE converted to integer
psql:schema/procedures/ADD_JOB_HISTORY_procedure.sql:24: NOTICE:
    type reference job_history.start_date%TYPE converted to timestamp without time zon
psql:schema/procedures/ADD_JOB_HISTORY_procedure.sql:24: NOTICE:
    type reference job_history.end_date%TYPE converted to timestamp without time zone
psql:schema/procedures/ADD_JOB_HISTORY_procedure.sql:24: NOTICE:
    type reference job_history.job_id%TYPE converted to character varying
psql:schema/procedures/ADD_JOB_HISTORY_procedure.sql:24: NOTICE:
    type reference job_history.department_id%TYPE converted to smallint
CREATE FUNCTION
```

Il n'y a eu aucune erreur à la création de la fonction, uniquement des indications sur les types réellement utilisés.

Voici le code de création de la fonction :

```
CREATE OR REPLACE FUNCTION
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
   )
 RETURNS VOID 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;
```

Le trigger peut maintenant être chargé.

```
psql -U migration pghr -f schema/triggers/trigger.sql
```

Mais ce chargement génère à nouveau une erreur :

```
psql:./schema/triggers/UPDATE_JOB_HISTORY_trigger.sql:17:
    ERROR:  syntax error at or near "add_job_history"
LIGNE 3 :   add_job_history(OLD.employee_id, OLD.hire_date, LOCALTIMES...
```

Le problème ici est que la fonction est appelée directement, ce qui n'est pas possible avec PostgreSQL : il faut l'appeler avec une instruction `SELECT`. Si l'on fait cette modification, le trigger est chargé sans erreur.

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

```
psql -U migration pghr -f schema/procedures/SECURE_DML_procedure.sql
SET
CREATE FUNCTION
```

Pour l'import de fonction le mieux est d'utiliser le fichier d'import global des fonctions de la façon suivante :

```
psql -U migration pghr -f tp_migration/schema/functions/function.sql \
     --single-transaction
```

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. Notamment, si on exécute la fonction `emp_sal_ranking`, on obtient ce message d'erreur :

```
pghr=> select emp_sal_ranking(105);
ERROR:  invalid input syntax for integer: "0.125"
CONTEXTE : PL/pgSQL function emp_sal_ranking(bigint) while casting return
           value to function's return type
```

Ceci est normal, la fonction est déclarée comme retournant un `bigint` alors que le résultat de l'opération de retour est potentiellement un nombre à virgule. Il est donc nécessaire ici de retourner un `float` ou un `double`.

```
pghr=> select emp_sal_ranking(105);
 emp_sal_ranking
-----------------
           0.125
(1 ligne)
```

L'autre fonction n'a aucun problème de conversion :

```
pghr=> select last_first_name(105);
        last_first_name
-------------------------------
 Employee: 105 - AUSTIN, DAVID
(1 ligne)
```

Importer les procédures stockées

La base d'exemple contient deux paquets de fonctions `EMP_ACTIONS` et `EMP_MGMT` composés de 3 et 6 fonctions. Premier chargement :

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

Il faut s'attendre à des erreurs et c'est le cas :

```
psql:schema/packages/package.sql:21: NOTICE:  schema "emp_actions" does not
                                              exist, skipping
DROP SCHEMA
CREATE SCHEMA
SET
CREATE FUNCTION
```

```
SET
CREATE FUNCTION
SET
CREATE FUNCTION
psql:schema/packages/package.sql:21: NOTICE:  schema "emp_mgmt" does not
                                              exist, skipping
DROP SCHEMA
CREATE SCHEMA
SET
psql:schema/packages/emp_mgmt/hire_package.sql:7: ERROR:  syntax error at or
                                                          near "tot_emps"
LIGNE 1 : tot_emps NUMBER;
          ^
```

À la lecture du fichier `schema/packages/emp_mgmt/hire_package.sql` on s'aperçoit tout de suite de la présence de deux variables globales, `tot_emps` et `tot_depts`. Les variables globales dans le code PL/pgSQL ne sont pas supportées sous PostgreSQL, tout au moins pas de cette manière. Pour simplifier, on modifie tous les fichiers de fonction y faisant référence en les supprimant ou en les mettant en commentaire.

Ici les variables `tot_depts` et `tot_emps` servent 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.

De manière générale, pour émuler les variables globales on peut utiliser un table où utiliser un langage procédural permettant ce type de stockage global, comme le PL/Perl.

Ce qui nous donne au chargement suivant :

```
psql -U migration pghr -f schema/package/package.sql
```

On rencontre une autre erreur :

```
psql:schema/packages/emp_mgmt/increase_sal_package.sql:26:
    ERROR:  unrecognized exception condition "no_sal"
CONTEXTE : compilation of PL/pgSQL function "increase_sal" near line 10
```

Après étude de la fonction, l'erreur est imputable à une variable contenant un message d'erreur non définie.

```
...sql
THEN RAISE no_sal;
...
```

Il nous suffit de le remplacer par :

```
...
THEN RAISE 'cannot have null salary';
...
```

Après avoir résolu un dernier problème identique dans le fichier `schema/ packages/ emp_mgmt/ increase_comm_package.sql`, le paquet de fonctions a été créé sans erreur.

Tests unitaires Voici un exemple de script utilisant les commandes `psql` et `sqlplus` pour réaliser les tests unitaires des procédures stockées importées:

```
#!/bin/bash
#-------------------------------------------------------------------------------
# script utilisé pour valider les réponses des fonctions
# entre Oracle et PostgreSQL
#-------------------------------------------------------------------------------

#-- Oracle --
export ORACLE_HOME="/opt/oracle/11.2/instantclient_11_2"
export PATH="$PATH:/opt/oracle/11.2/instantclient_11_2"

ORACLE_SID="xe"
ORACLE_HOST="192.168.1.109"
ORACLE_USER="hr"
ORACLE_PWD="phoenix"
SQLPLUS="sqlplus -S
        $ORACLE_USER/$ORACLE_PWD@$ORACLE_HOST/$ORACLE_SID @/tmp/oracle.sql"

#-- PostgreSQL --
PGSQL_HOST="localhost"
PGSQL_USER="migration"
PGSQL_DB=pghr
PSQL="psql -tA -U $PGSQL_USER -f /tmp/postgresql.sql $PGSQL_DB"

#PGSQL_PORT=5432
#PSQL="psql -h $PGSQL_HOST -p $PGSQL_PORT -tA -U $PGSQL_USER
            -f /tmp/postgresql.sql $PGSQL_DB"

exec_oracle () {
  echo "
set verify on
set feedback off
set termout on
set linesize 40
set verify off
```

```
set feedback off
set pagesize 0" > /tmp/oracle.sql
  echo $1 >> /tmp/oracle.sql
  echo "exit;" >> /tmp/oracle.sql
  echo -n "\tOracle : \t";
  $SQLPLUS
}

exec_postgresql () {
  echo $1 > /tmp/postgresql.sql
  echo -n "\tPostgreSQL : \t"
  $PSQL
}

#------------------ debut des tests unitaires
echo "----------------------------------------------------------------------"
echo "Test fonction : secure_dml()"
exec_oracle "exec secure_dm();";
exec_postgresql  "select secure_dml();"
echo "----------------------------------------------------------------------"
echo "Test fonction : emp_sal_ranking()"
exec_oracle "select emp_sal_ranking(105) from dual;";
exec_postgresql  "select emp_sal_ranking(105);"
echo "----------------------------------------------------------------------"
echo "Test fonction : last_first_name()"
exec_oracle "select last_first_name(105) from dual;";
exec_postgresql  "select last_first_name(105);"
exit 0
```

Une exécution du script donne le résultat suivant :

```
sh tests_unitaires.sh

----------------------------------------------------------------------
Test fonction : secure_dml()
    Oracle :    BEGIN secure_dm(); END;

      *
ERROR at line 1:
ORA-06550: line 1, column 7:
PLS-00201: identifier 'SECURE_DM' must
be declared
ORA-06550: line 1, column 7:
PL/SQL: Statement ignored


    PostgreSQL :    psql:/tmp/postgresql.sql:1:
```

```
            ERROR:  You may only make changes outside normal office hours
-----------------------------------------------------------------------
Test fonction : emp_sal_ranking()
    Oracle :             .125
    PostgreSQL :    0.125
-----------------------------------------------------------------------
Test fonction : last_first_name()
    Oracle :    Employee: 105 - AUSTIN, DAVID
    PostgreSQL :    Employee: 105 - AUSTIN, DAVID
```

5 PORTAGE DES REQUÊTES SQL

5.1 INTRODUCTION

- Portage des requêtes SQL
  - élément commun: SQL
  - dialecte Oracle

Après avoir migré les données, il faut également retravailler à minima les requêtes de façon à ce qu'elles puissent s'exécuter sur PostgreSQL. Le langage SQL étant issu d'une norme ISO qui évolue constamment, le travail n'est pas aussi important que s'il s'agissait d'une réécriture dans un nouveau langage. Mais certaines formes d'écritures peuvent poser problème. Elles sont héritées des temps où Oracle offrait ses propres extensions au langage SQL avant que les fonctionnalités ne soient disponibles dans la norme SQL. Bien qu' Oracle supporte maintenant les dernières avancées de la norme SQL, de nombreuses applications à migrer utilisent encore le dialecte SQL. Ce chapitre a pour objectif de présenter les principaux éléments qui nécessitent une réécriture.

5.2 COMPATIBILITÉ AVEC ORACLE

- Langage SQL
  - norme ISO
  - dernière version 2011
- La façon d'écrire les requêtes ne change pas
  - sauf certains détails

5.2.1 TABLE DUAL

Table `DUAL` pas nécessaire

```
SELECT fonction();
```

```
SELECT current_timestamp;
```

5.2.2 CONVERSIONS IMPLICITES

- Conversions implicites de et vers un type text
  - supporté par Oracle
  - plus supporté par PostgreSQL depuis la version 8.3

```
SELECT 1 = 'a'::text;
```

5.3 TYPES DE DONNÉES

- Plusieurs incompatibilités
  - Oracle ne supporte pas bien la norme SQL
  - types numériques, chaînes, binaires, dates
- PostgreSQL fournit également des types spécialisés

5.3.1 DIFFÉRENCES SUR LES TYPES NUMÉRIQUES

- Oracle ne gère pas les types numériques «natifs» SQL :
  - smallint, integer, bigint
- Le type numeric du standard SQL est appelé number sous Oracle

Les types smallint, integer, bigint, float, real, double precision sont plus rapides que le type numeric sous PostgreSQL: ils utilisent directement les fonctions câblées des processeurs. Il faut donc les privilégier.

5.3.2 DIFFÉRENCES SUR LES TYPES CHAÎNES

- Pas de varchar2 dans PostgreSQL
  - le type est varchar
- Attention, sous Oracle, `''  = IS NULL`
  - sous PostgreSQL, `''` et `NULL` sont distincts
- varchar peut ne pas prendre de taille sous PostgreSQL
  - 1 Go maximum dans ce cas
- Il existe aussi un type «text» équivalent à varchar sans taille
- Un seul encodage par base
- Collationnements

  - Par instance (avant la 8.4), par base de données (depuis la 8.4), par colonne (depuis la 9.1)

Au niveau de PostgreSQL, il existe trois types de données pour les chaînes de caractères : char, varchar et text. Le type varchar2 d'Oracle est l'équivalent du type varchar de PostgreSQL. Il est possible de ne pas donner de taille à une colonne de type varchar, ce qui revient à la déclarer de type text. Dans ce cas, la taille maximale est de 1 Go. Suivant l'encodage, le nombre de caractères intégrables dans la colonne diffère.

La grosse différence entre PostgreSQL et Oracle pour les chaînes de caractères tient dans la façon dont les chaînes vides sont gérées. Oracle ne fait pas de différence entre une chaîne vide et une chaîne NULL. PostgreSQL fait cette différence. Du coup, tous les tests de chaînes vides effectuées avec un `IS NULL` et tous les tests de chaînes NULL effectués avec une comparaison avec une chaîne vide ne donneront pas le même résultat avec PostgreSQL. Ces tests doivent être vérifiés systématiquement par les développeurs d'applications et de procédures stockées.

```
dev2=# SELECT cast('' AS varchar) IS NULL;
 ?column?
----------
 f
(1 row)
```

Au niveau encodage, PostgreSQL n'accepte qu'un encodage par base de données (l'encodage par défaut est UTF-8). Il accepte par contre plusieurs collationnements depuis la 8.4. Depuis la 9.1, il est même possible d'indiquer le collationnement dans les requêtes (au niveau d'un `ORDER BY` ou d'un `CREATE INDEX`).

5.3.3 DIFFÉRENCES SUR LES TYPES BINAIRES

- 2 implémentations différentes sous PostgreSQL
  - `large objects` et fonctions `lo_*`
  - `bytea`

L'implémentation des types binaires sur PostgreSQL est très particulière. De plus, elle est double, dans le sens où vous avez deux moyens d'importer et d'exporter des données binaires dans PostgreSQL.

La première, et plus ancienne, implémentation concerne les Large Objects. Cette implémentation dispose d'une API spécifique. Il ne s'agit pas à proprement parler d'un type de données. Il faut passer par des procédures stockées internes qui permettent d'importer, d'exporter, de supprimer, de lister les Large Objects. Après l'import d'un Large

Object, vous récupérez un identifiant que vous pouvez stocker dans une table utilisateur (généralement dans une colonne de type OID). Vous devez utiliser cet identifiant pour traiter l'objet en question (export, suppression, etc). Cette implémentation a de nombreux défauts, qui fait qu'elle est rarement utilisée. Parmi les défauts, notons que la suppression d'une ligne d'une table utilisateur référençant un Large Object ne supprime pas le Large Object référencé. Notons aussi qu'il est bien plus difficile d'interagir et de maintenir une table système. Notons enfin que la sauvegarde avec pg_dump est plus complexe et plus longue si des Larges Objects sont dans la base à sauvegarder. Son principal avantage sur la deuxième implémentation est la taille maximale d'un Large Object : 4 To depuis la 9.3 (2 Go avant).

La deuxième implémentation est un type de données appelé bytea. Comme toutes les colonnes dans PostgreSQL, sa taille maximale est 1 Go, ce qui est inférieur à la taille maximale d'un Large Object. Cependant, c'est son seul défaut.

Bien que l'implémentation des Large Objects est en perte de vitesse à cause des nombreux inconvénients inhérents à son implémentation, elle a été l' objet d'améliorations sur les dernières versions de PostgreSQL : gestion des droits de lecture ou écriture des Large Objects, notion de propriétaire d'un Large Object, limite de taille relevée à 4 To. Elle n'est donc pas obsolète.

5.3.4 DIFFÉRENCES SUR LES TYPES SPÉCIALISÉS

PostgreSQL fournit aussi de nombreux types de données spécialisés :

- Gestion des timestamps et intervals avec opérations arithmétiques
- Plans d'adressage IP (CIDR) et opérateurs de masquage
- Grande extensibilité des types: il est très facile d'en rajouter un nouveau
  - PERIOD
  - ip4r
  - etc…

L'un des gros avantages de PostgreSQL est son extensibilité. Mais même sans cela, PostgreSQL propose de nombreux types natifs qui vont bien au-delà des types habituels. Ce sont des types métiers, pour le réseau, la géométrie, la géographie, la gestion du temps, la gestion des intervalles de valeurs, etc.

Il est donc tout à fait possible d'améliorer une application en passant sur des types spécialisés de PostgreSQL.

5.4 DONNÉES TEMPORELLES

- Types différents
- Fonctions équivalentes à `SYSDATE`
- Fonctions de manipulation différentes

5.4.1 DIFFÉRENCES ENTRE LES TYPES DATES 1/2

- Date
  - sous Oracle: `YYYY/MM/DD HH:MM:SS`
  - sous PostgreSQL: `YYYY-MM-DD` (conforme SQL)
- Time
  - sous Oracle: `YYYY/MM/DD HH:MM:SS`
  - sous PostgreSQL: `HH:MM:SS.mmmmmmm` (µs)

5.4.2 DIFFÉRENCES ENTRE LES TYPES DATES 2/2

- Gestion des fuseaux horaires
  - sous PostgreSQL, par défaut
  - timestamp sous PostgreSQL: Date+Time (+TZ)
- Format de sortie conforme SQL sous PostgreSQL:

  ```
  YYYY-MM-DD HH24:MI:SS.mmmmmmm+TZ
  ```

- Type interval
  - Date1-Date2 => Interval

Oracle a tendance à mélanger un peu tous les types dates. Ce n'est pas le cas au niveau de PostgreSQL. Une colonne de type date au niveau de PostgreSQL contient seulement une date, il n'y a pas d'heure ajoutée. Une colonne de type time au niveau de PostgreSQL contient seulement un horodatage (heure, minute, seconde, milliseconde), mais pas de date.

Par défaut, PostgreSQL intègre le fuseau horaire dans les types timestamp (date et heure). Le stockage est fait en UTC, mais la restitution dépend du fuseau horaire indiqué par le client.

17.12

5.4.3 SYSDATE

- SYSDATE
  - retourne la date et l'heure courante, sans timezone
  - équivalent direct :
    ```
    SELECT localtimestamp;
    ```
- PostgreSQL implémente d'autres fonctions :
  - current_timestamp
  - current_date
  - current_time

5.4.4 MANIPULATIONS DES DONNÉES TEMPORELLES

- PostgreSQL ne propose pas de fonctions add_months, etc.

```
SELECT current_date + interval '3 days';

SELECT current_date + interval '1 days' * 3;

SELECT (now() - '2014-01-01') * 2 + now()
```

Quel est le premier jour de la première semaine de l'année :

```
SELECT date '2014-01-04' - interval '1 day' *
    (extract('dow' from date '2014-01-04') - 1);
```

Pour l'année courante :

```
SELECT (date_trunc('year', now()) + interval '3 days') - interval '1 day' *
    (extract('dow' from (date_trunc('year', now()) + interval '3 days')) - 1);
```

5.5 EXPRESSIONS CONDITIONNELLES

- DECODE
- NVL

5.5.1 DECODE

Équivalent de la clause CASE du standard

```
CASE expr
  WHEN valeur1 THEN valeur_retour1
  WHEN valeur2 THEN valeur_retour2
  ELSE valeur_retour3
END

CASE
  WHEN expr1 THEN valeur_retour1
  WHEN expr2 THEN valeur_retour2
  ELSE valeur_retour3
END
```

La fonction DECODE d'Oracle est un équivalent propriétaire de la clause CASE, qui est normalisée. Oracle supporte CASE mais DECODE est souvent utilisé par habitude.

La construction suivante utilise la fonction DECODE :

```
SELECT emp_name,
       decode(trunc (( yrs_of_service + 3) / 4), 0, 0.04,
                                                 1, 0.04,
                                                 0.06) as perc_value
  FROM employees;
```

Cette construction doit être réécrite de cette façon :

```
SELECT emp_name,
       CASE WHEN trunc(yrs_of_service + 3) / 4 = 0 THEN 0.04
            WHEN trunc(yrs_of_service + 3) / 4 = 1 THEN 0.04
            ELSE 0.06
       END
  FROM employees;
```

Cet autre exemple :

```
DECODE("user_status",'active',"username",NULL)
```

sera transposé de cette façon :

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

Attention aux commentaires entre le WHEN et le THEN qui ne sont pas supportés par PostgreSQL.

5.5.2 NVL

- Retourne le premier argument non NULL

```
SELECT NVL(description, description_courte, '(aucune)')FROM articles;
```

- Équivalent de la norme SQL : `COALESCE`

```
SELECT COALESCE(description, description_courte, '(aucune)') FROM articles;
```

La fonction `NVL` d'Oracle est encore souvent utilisée, bien que la fonction normalisée `COALESCE` soit également implémentée. Ces deux fonctions retournent le premier argument qui n'est pas `NULL`. Bien évidemment, PostgreSQL n'implémente que la fonction normalisée `COALESCE`. Un simple remplacement de l'appel de `NVL` par un appel à `COALESCE` est suffisant.

Ainsi, la requête suivante :

```
SELECT NVL(description, description_courte, '(aucune)')FROM articles;
```

se verra portée facilement de cette façon :

```
SELECT COALESCE(description, description_courte, '(aucune)') FROM articles;
```

5.5.3 COMMON TABLE EXPRESSIONS

- Syntaxe quasiment identique
- Attention à la recursion
  - `WITH RECURSIVE` obligatoire dans PostgreSQL

Un article écrit par Lucas Jellema montre les évolutions d'Oracle 11gR2 concernant les requêtes récursives. Les différents exemples montrent que les requêtes écrites utilisent les CTE au lieu du `CONNECT BY` qui fait partie seulement du dialecte SQL Oracle. L'article est disponible à cette adresse[21] .

Si l'on exécute la seconde requête donnée en exemple (la première employant `CONNECT BY` directement sur PostgreSQL, on obtient le message d'erreur suivant :

```
DÉTAIL : There is a WITH item named "employees", but it cannot be referenced
         from this part of the query.
ASTUCE : Use WITH RECURSIVE, or re-order the WITH items to remove forward
         references.
```

Pour corriger ce problème, il suffit simplement d'ajouter la clause `RECURSIVE`, comme l' indique tout simplement le message d'erreur et la requête pourra être exécutée sans difficulté.

---

[21] https://technology.amis.nl/2009/09/01/oracle-rdbms-11gr2-goodbye-connect-by-or-the-end-of-hierarchical-querying-as-we-know-

5.6 ROWNUM

- pseudo-colonne Oracle
- numérote les lignes du résultat
  - parfois utiliser pour limiter le résultat

Oracle propose une pseudo-colonne ROWNUM qui permet de numéroter les lignes du résultat d'une requête SQL. La clause ROWNUM peut être utilisée soit pour numéroter les lignes de l'ensemble retourné par la requête. Elle peut aussi être utilisée pour limiter l'ensemble retourné par une requête.

5.6.1 NUMÉROTER LES LIGNES

- `ROWNUM` n'existe pas dans PostgreSQL
  - `row_number() OVER ()`
  - attention si `ORDER BY`

Dans le premier cas, à savoir numéroter les lignes de l'ensemble retourné par la requête, il faut réécrire la requête pour utiliser la fonction de fenêtrage `row_number()`. Bien qu'Oracle préconise d'utiliser la fonction normalisée `row_number()`, il est fréquent de trouver ROWNUM dans une requête issue d'une application s'appuyant sur une ancienne version d'Oracle :

```
SELECT ROWNUM, * FROM employees;
```

La requête sera réécrite de la façon suivante :

```
SELECT ROW_NUMBER() OVER () AS rownum, * FROM employees;
```

Il faut toutefois faire attention à une clause `ORDER BY` dans une requête employant `ROWNUM` pour numéroter les lignes retournées par une requête. En effet, le tri commandé par `ORDER BY` est réalisé après l'ajout de la pseudo-colonne `ROWNUM`. Il faudra vérifier le résultat de la requête sous Oracle et PostgreSQL pour vérifier qu'elles retourneront des résultats identiques.

La clause `WITH ORDINALITY` de PostgreSQL 9.4 permet de numéroter les lignes de résultat d'un appel de fonction.

5.6.2 LIMITER LE RÉSULTAT

- Retourne les dix premières lignes de résultats :

  - `WHERE ROWNUM < 11`
- PostgreSQL propose l'ordre `LIMIT xx` :

```
SELECT *
  FROM employees
 LIMIT 10;
```

Pour limiter l'ensemble retourné par une requête, il faut supprimer les prédicats utilisant `ROWNUM` dans la clause et les transformer en couple `LIMIT`/`OFFSET`.

La requête suivante retourne les 10 premières lignes de la table `employees` sous Oracle :

```
SELECT *
  FROM employees
 WHERE ROWNUM < 11;
```

Elle sera réécrite de la façon suivante lors du portage de la requête pour PostgreSQL :

```
SELECT *
  FROM employees
 LIMIT 10;
```

5.6.3 ROWNUM ET ORDER BY

- Oracle effectue le tri après l'ajout de ROWNUM
- PostgreSQL applique le tri avant de limiter le résultat
- Résultats différents

De la même façon que précédemment, Oracle effectuera le tri commandé par `ORDER BY` après l'ajout de la pseudo-colonne `ROWNUM`, comme le montre le plan d'exécution d' une requête similaire à l'exemple donné plus haut :

```
Operation       Options    Filter Predicates
SELECT STATEMENT
SORT            ORDER BY
COUNT           STOPKEY    ROWNUM<5
TABLE ACCESS    FULL
```

Au contraire, PostgreSQL va appliquer le tri avant la limitation du résultat. Lorsque PostgreSQL rencontre une clause `LIMIT` et un tri avec `ORDER BY`, il appliquera d'abord le tri avant de limiter le résultat.

```
test=# EXPLAIN SELECT * FROM t1 ORDER BY col DESC LIMIT 10;
                          QUERY PLAN
-----------------------------------------------------------------
 Limit  (cost=4.16..4.19 rows=10 width=4)
```

```
   ->  Sort  (cost=4.16..4.41 rows=100 width=4)
         Sort Key: col
         ->  Seq Scan on t1  (cost=0.00..2.00 rows=100 width=4)
(4 rows)
```

Si une requête Oracle est écrite de manière aussi simple, il conviendra de la réécrire de la façon suivante :

```
SELECT r.*
  FROM (SELECT *
          FROM t1
         LIMIT 10) r
 ORDER BY col
```

Il faudra néanmoins se poser la question de la pertinence de cette requête car le résultat n'est pas nécessairement celui attendu :

Néanmoins, pour palier ce comportement de l'optimiseur Oracle, les développeurs ont souvent écrit ce genre de requête en utilisant une sous-requête, telle que la suivante :

```
SELECT ROWNUM, r.*
  FROM (SELECT *
          FROM t1
         ORDER BY col) r
 WHERE ROWNUM BETWEEN 1 AND 10;
```

Cette requête serait simplifiée de cette façon une fois migrée vers PostgreSQL :

```
SELECT *
  FROM t1
 ORDER BY col
 LIMIT 10;
```

5.7 JOINTURES

- Jointures internes
  - `FROM tab1, tab2 WHERE tab1.col = tab2.col`
  - `FROM tab1 JOIN tab2 ON (tab1.col = tab2.col`

Le SGBD Oracle supporte la syntaxe normalisée d'écriture des jointures seulement depuis la version 9i. Auparavant, les jointures étaient exprimées telle que le définissait la première version de la norme SQL, avec une notation propriétaire pour la gestion des jointures externes. PostgreSQL ne supporte pas cette notation propriétaire, mais supporte parfaitement la notation portée par la norme SQL.

La requête suivante peut être conservée telle qu'elle est écrite :

17.12

```
SELECT *
  FROM t1, t2
 WHERE t1.col1 = t2.col1
```

Cependant, cette syntaxe ne permet pas d'écrire de jointure externe. Il est donc recommandé d'utiliser systématiquement la nouvelle notation, qui est aussi bien plus lisible dans le cas où des jointures simples et externes sont mélangées :

```
SELECT *
  FROM t1
  JOIN t2 ON (t1.col1 = t2.col1)
```

5.7.1 JOINTURES EXTERNES

- Syntaxe (+) d'Oracle historique
- `LEFT JOIN`
- `RIGHT JOIN`
- `FULL OUTER JOIN`

Le SGBD Oracle utilise la notation `(+)` pour décrire le côté où se trouvent les valeurs NULL. Pour une jointure à gauche, l'annotation `(+)` serait placée du côté droit (et inversement pour une jointure à droite). Cette forme n'est pas supportée par PostgreSQL. Il faut donc réécrire les jointures avec la notation normalisée : LEFT OUTER JOIN ou LEFT JOIN pour une jointure à gauche et RIGHT OUTER JOIN ou RIGHT JOIN pour une jointure à droite.

La requête suivante, écrite pour Oracle et qui comporte une jointure à gauche :

```
SELECT *
  FROM t1, t2
 WHERE t1.col1 = t2.col3 (+);
```

nécessite d'être réécrite de la manière suivante :

```
SELECT *
  FROM t1
  LEFT JOIN t2 ON (t1.col1 = t2.col3);
```

De la même façon, la requête suivante comporte une jointure à droite :

```
SELECT *
  FROM t1, t2
 WHERE t1.col1 (+) = t2.col3 ;
```

et nécessite d'être réécrite de la manière suivante :

```
SELECT *
  FROM t1
  RIGHT JOIN t2 ON (t1.col1 = t2.col3);
```

Dans les versions précédant la version 9i d'Oracle, une jointure externe complète (`FULL OUTER JOIN`) devait être exprimée à l'aide d 'un `UNION` entre une jointure à gauche et une jointure à droite. L'exemple suivant implémente une jointure externe complète :

```
SELECT *
  FROM t1, t2
 WHERE t1.col1 = t2.col3 (+)
UNION ALL
SELECT *
  FROM t1, t2
 WHERE t1.col1 (+) = t2.col3
   AND t1.col IS NULL
```

Cette requête doit être réécrite et sera par ailleurs simplifiée de la façon suivante :

```
SELECT *
  FROM t1
  FULL OUTER JOIN t2 ON (t1.col1 = t2.col3);
```

5.7.2 PRODUIT CARTÉSIEN

- `FROM t1, t2;`
- `FROM t1 CROSS JOIN t2`

Un produit cartésien peut être exprimé de la façon suivante dans Oracle et PostgreSQL :

```
SELECT *
  FROM t1, t2;
```

Néanmoins, la notation normalisée est moins ambigüe et montre clairement l'intention de faire un produit cartésien :

```
SELECT *
  FROM t1
  CROSS JOIN t2;
```

5.8 HAVING ET GROUP BY

- Oracle permet `GROUP BY` **après** `HAVING`
- PostgreSQL impose `GROUP BY` **avant** `HAVING`

17.12

Bien que la documentation Oracle indique que la clause `GROUP BY` précède la clause `HAVING` , la grammaire Oracle autorise l'inverse. Il faut donc corriger les requêtes écrites de la façon `HAVING … GROUP BY`.

Les requêtes de la forme suivante :

```
SELECT * FROM test HAVING count(*) > 3 GROUP BY i;
```

seront transposées de la façon suivante pour pouvoir s'exécuter sous PostgreSQL :

```
SELECT * FROM test GROUP BY i HAVING count(*) > 3;
```

5.9 OPÉRATEURS ENSEMBLISTES

- `UNION` / `UNION ALL`
- `INTERSECT`
- `EXCEPT`
  - équivalent de `MINUS`

L'opérateur ensembliste `MINUS` est à transposer en `EXCEPT` pour PostgreSQL. Les autres opérateurs ensemblistes `UNION`, `UNION ALL` et `INSERSECT` ne nécessitent pas de transposition.

Ainsi, la requête suivante retourne les produits de l'inventaire qui n'ont pas fait l'objet d'une commande. Elle est exprimée ainsi pour Oracle :

```
SELECT product_id FROM inventories
MINUS
SELECT product_id FROM order_items
ORDER BY product_id;
```

La requête sera transposé de la façon suivante pour PostgreSQL :

```
SELECT product_id FROM inventories
EXCEPT
SELECT product_id FROM order_items
ORDER BY product_id;
```

5.10 TRANSACTIONS

- Les transactions ne sont pas démarrées automatiquement
  - `BEGIN`
  - sauf avec JDBC (`BEGIN` caché)

- Toute erreur non gérée dans une transaction entraîne son annulation
  - Oracle revient à l'état précédent de l'ordre en échec
  - PostgreSQL plus strict de ce point de vue
- DDL transactionnels

Pour PostgreSQL, si vous souhaitez pouvoir annuler des modifications, vous devez utiliser BEGIN avant d'exécuter les requêtes de modification. Toute transaction qui commence par un BEGIN doit être validée avec COMMIT ou annulée avec ROLLBACK. Si jamais la connexion est perdue entre le serveur et le client, le ROLLBACK est automatique.

Par exemple, si on insère une donnée dans une table, sans faire de BEGIN avant, et qu'on essaie d'annuler cette insertion, cela ne fonctionnera pas :

```
dev2=# CREATE TABLE t1(id integer);
CREATE TABLE
dev2=# INSERT INTO t1 VALUES (1);
INSERT 0 1
dev2=# ROLLBACK;
NOTICE:  there is no transaction in progress
ROLLBACK
dev2=# SELECT * FROM t1;
 id
----
  1
(1 row)
```

Par contre, si on intègre un BEGIN avant, l'annulation se fait bien :

```
dev2=# BEGIN;
BEGIN
dev2=# INSERT INTO t1 VALUES (2);
INSERT 0 1
dev2=# ROLLBACK;
ROLLBACK
dev2=# SELECT * FROM t1;
 id
----
  1
(1 row)
```

Autre différence au niveau transactionnel : il est possible d'intégrer des ordres DDL dans des transactions. Par exemple :

```
dev2=# BEGIN;
BEGIN
dev2=# CREATE TABLE t2(id integer);
CREATE TABLE
dev2=# INSERT INTO t2 VALUES (1);
```

```
INSERT 0 1
dev2=# ROLLBACK;
ROLLBACK
dev2=# INSERT INTO t2 VALUES (2);
ERROR:  relation "t2" does not exist
LINE 1: INSERT INTO t2 VALUES (2);
                    ^
```

Enfin, quand une transaction est en erreur, vous ne sortez pas de la transaction. Vous devez absolument exécuter un ordre de fin de transaction (COMMIT ou ROLLBACK, peu importe, un ROLLBACK sera exécuté) :

```
dev2=# BEGIN;
BEGIN
dev2=# INSERT INTO t2 VALUES (2);
ERROR:  relation "t2" does not exist
LINE 1: INSERT INTO t2 VALUES (2);
                    ^
dev2=# INSERT INTO t1 VALUES (2);
ERROR:  current transaction is aborted, commands ignored until
        end of transaction block
dev2=# SELECT * FROM t1;
ERROR:  current transaction is aborted, commands ignored until
        end of transaction block
dev2=# ROLLBACK;
ROLLBACK
dev2=# SELECT * FROM t1;
 id
----
  1
(1 row)
```

5.10.1 NIVEAUX D'ISOLATION

- `BEGIN TRANSACTION ISOLATION LEVEL xxxx`
  - `READ COMMITTED`
  - `REPEATABLE READ`
  - `SERIALIZABLE`

Il est possible d'indiquer le niveau d'isolation d'une transaction en l' indiquant dans l'ordre d'ouverture d'une transaction :

```
BEGIN [ WORK | TRANSACTION ] [ mode_transaction [, ...] ]
```

où `mode_transaction` est :

```
ISOLATION LEVEL
    { SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ UNCOMMITTED }
    READ WRITE | READ ONLY
    [ NOT ] DEFERRABLE
```

`READ UNCOMMITTED` est un synonyme de `READ COMMITTED` sous PostgreSQL, tout comme sous Oracle : les moteurs étant MVCC, le mode `READ UNCOMMITTED` n'a pas d'intérêt (les écrivains ne bloquent pas les lecteurs, les lecteurs ne bloquent pas les écrivains).

Par ailleurs, Oracle et PostgreSQL implémentent un niveau d'isolation `SERIALIZABLE`. PostgreSQL implémente le niveau d'isolation `SERIALIZABLE` avec des verrous optimistes afin de garantir un meilleur débit transactionnel. La plupart des SGBD implémentent ce niveau d'isolation par le biais de verrous pessimistes, grevant ainsi les performances. Les versions plus anciennes d'Oracle possédaient d'ailleurs un paramètre non-documenté `SERIALIZABLE` pour activer l'emploi de verrous pessimistes, mais il n'est plus supporté depuis Oracle 8.1.6. Ce paramètre permet donc d'activer ce mode d'isolation de façon à ce qu'il soit respectueux de la norme, au prix de performances dégradées. Dans les versions actuelles, Oracle n'utilise pas de verrou et de ce fait, son implémentation du niveau d'isolation `SERIALIZABLE` n'est pas respectueuse de la norme, à la différence de PostgreSQL. Il faut noter également que depuis la version 9.1, PostgreSQL est le premier SGBD qui implémente un mode d'isolation `SERIALIZABLE` parfaitement respectueux de la norme SQL. Cette fonctionnalité, issue de travaux de recherches universitaires[22] , est appelée *Serializable Snapshot Isolation* et corrige les défauts des implémentations[23] précédentes du niveau `SERIALIZABLE`.

Oracle permet de positionner le niveau d'isolation des transactions pour une session donnée, c'est-à-dire pour toutes les transactions réalisées dans la même session.

L'ordre SQL suivant permet de positionner le niveau d'isolation au niveau de la session pour Oracle :

```
ALTER SESSION SET ISOLATION LEVEL ...;
```

L'ordre `SET SESSION ...` permet de réaliser la même chose pour PostgreSQL :

```
SET SESSION TRANSACTION ISOLATION LEVEL ...;
```

Pour plus de détails sur les niveaux d'isolation, consulter la documentation de PostgreSQL sur l'isolation des transactions[24] .

[22] http://cs.nyu.edu/courses/fall11/CSCI-GA.2434-001/p729-cahill.pdf
[23] http://docs.postgresql.fr/current/transaction-iso.html#MVCC-SERIALIZABILITY
[24] http://docs.postgresql.fr/current/transaction-iso.html

5.10.2 SAVEPOINT

- SAVEPOINT
- RELEASE SAVEPOINT
- ROLLBACK TO SAVEPOINT

Les SAVEPOINT fonctionnent sans régression par rapport au SGBD Oracle. Les verrous acquis avant la mise en place d'un SAVEPOINT ne sont pas relâchés si un SAVEPOINT est relâché par un RELEASE SAVEPOINT ou un ROLLBACK TO SAVEPOINT

La documentation de PostgreSQL met néanmoins en garde contre la modification de lignes après le positionnement d'un SAVEPOINT alors que ces lignes ont été verrouillées par un SELECT .. FOR UPDATE avant le positionnement du SAVEPOINT. En effet, le verrou acquis par le SELECT ... FOR UPDATE peut être relâché au moment du ROLLBACK TO SAVEPOINT. La séquence suivante d' ordres SQL est donc à éviter :

```
BEGIN;
SELECT * FROM ma_table WHERE cle = 1 FOR UPDATE;
SAVEPOINT s;
UPDATE ma_table SET ... WHERE cle = 1;
ROLLBACK TO SAVEPOINT s;
```

5.10.3 VERROUS IMPLICITES

- PostgreSQL pose un verrou sur les objets accédés
  - y compris en SELECT

Bien que PostgreSQL et Oracle partagent de nombreuses similitudes au niveau du verrouillage, il faut prendre en compte certaines différences subtiles.

Les ordres DML acquièrent des verrous implicites. La différence notable entre Oracle et PostgreSQL concerne l'ordre SELECT : Oracle n'acquiert aucun verrou, tandis que PostgreSQL pose un verrou de type ACCESS SHARE. De ce fait, Oracle ne protège en aucun cas les lecteurs de modifications telles que la suppression d'une table. Une lecture peut être interrompue suite à un DROP TABLE concurrent. L' acquisition par PostgreSQL d'un verrou ACCESS SHARE pour la lecture protège de ce genre de problèmes.

Les ordres INSERT, UPDATE et DELETE verrouillent les lignes modifiées.

5.10.4 VERROUS EXPLICITES

- `SELECT FOR SHARE/UPDATE`
  - quelques subtilités
- `LOCK TABLE`

Les ordres `SELECT FOR UPDATE` peuvent nécessiter des adaptations. La syntaxe Oracle est en effet un peu plus riche que celle de PostgreSQL pour ce qui concerne cet ordre SQL.

Oracle propose une syntaxe `WAIT` et `NOWAIT`. PostgreSQL ne propose que la clause `NOWAIT`. La clause `WAIT` est implicite si `NOWAIT` n'est pas spécifié, il faudra donc la supprimer. La requête `SELECT ... FOR UPDATE WAIT;` devient `SELECT ... FOR UPDATE;`.

En l'état, la clause `OF` Oracle est incompatible avec le clause `OF` de PostgreSQL. Cette clause permet d'indiquer la table verrouillée pour une mise à jour ultérieure. Seulement, la clause `OF` d'Oracle désigne une colonne d'une table, tandis que la clause `OF` de PostgreSQL désigne une table.

La clause `SKIP LOCKED` existe dans PostgreSQL depuis la version 9.5.

Concernant la syntaxe de l'ordre `LOCK TABLE` d'Oracle est compatible avec celle de PostgreSQL pour les cas généraux. L'ensemble des modes de verrouillage proposés par Oracle existent tous dans PostgreSQL. On peut noter que PostgreSQL propose plus de type de verrous.

Tout comme pour l'ordre `SELECT FOR UPDATE`, Oracle propose une syntaxe `WAIT` et `NOWAIT`. PostgreSQL ne propose aussi que la clause`NOWAIT`. La clause `WAIT` est implicite si `NOWAIT` n'est pas spécifié, il faudra donc la supprimer. La requête `LOCK TABLE ... WAIT;` devient `LOCK TABLE ...;`.

Les clauses `PARTITION` et `SUBPARTITION` ne peuvent cependant pas être reprises. Dans le cas de la mise en œuvre du partitionnement dans PostgreSQL, il faut désigner la table correspondant à la partition ciblée par l'acquisition d'un verrou.

5.11 HIÉRARCHIES

- Explorer un arbre hiérarchique
  - `CONNECT BY` Oracle
  - `WITH RECURSIVE` PostgreSQL

17.12

Oracle propose historiquement la fonction `CONNECT BY` qui permet d'explorer un arbre hiérarchique. Cette fonction spécifique à Oracle possède des fonctionnalités avancées comme la détection de cycle et propose des pseudos-colonnes comme le niveau de la hiérarchie et la construction d'un chemin.

Il n'existe pas de clause directement équivalente dans PostgreSQL, aussi un travail important de portage doit être réalisé pour porter les requêtes utilisant cette clause.

5.11.1 SYNTAXE "CONNECT BY"

- `START WITH`
  - condition de départ
- `CONNECT BY PRIOR`
  - lien hiérarchique

```
SELECT empno, ename, job, mgr
  FROM emp
  START WITH mgr IS NULL
  CONNECT BY PRIOR empno = mgr
```

Soit la requête SQL suivante qui explore la hiérarchie de la table `emp`. La colonne `mgr` de cette table désigne le responsable hiérarchique d'un employé. Si elle vaut NULL, alors la personne est au sommet de la hiérarchie (`START WITH mgr IS NULL`). Le lien avec l'employé et son responsable hiérarchique est construit avec la clause `CONNECT BY PRIOR empno = mgr` qui indique que la valeur de la colonne `mgr` correspond à l'identifiant `empno` du niveau de hiérarchie précédent.

```
SELECT empno, ename, job, mgr
  FROM emp
  START WITH mgr IS NULL
  CONNECT BY PRIOR empno = mgr
```

Le portage de cette requête est réalisé à l'aide d'une requête récursive (`WITH RECURSIVE`). La récursion est initialisée dans une première requête qui récupère les lignes qui correspondent à la condition de la clause `START WITH` de la requête précédente : `mgr IS NULL`. La récursion continue ensuite avec la requête suivante qui réalise une jointure entre la table `emp` et la vue virtuelle `emp_hierarchy` qui est définie par la clause `WITH RECURSIVE`. La condition de jointure correspond à la clause `CONNECT BY`. La vue virtuelle `emp_hierarchy` a pour alias `prior` pour mieux représenter la transposition de la clause `CONNECT BY`.

La requête récursive pour PostgreSQL serait alors écrite de la façon suivante :

```
WITH RECURSIVE emp_hierarchy (empno, ename, job, mgr) AS (
SELECT empno, ename, job, mgr
  FROM emp
 WHERE mgr IS NULL
UNION ALL
SELECT emp.empno, emp.ename, emp.job, emp.mgr
  FROM emp
  JOIN emp_hierarchy prior ON (emp.mgr = prior.empno)
)
SELECT * FROM emp_hierarchy;
```

Il faudra néanmoins faire attention à l'ordre des lignes qui sera différent avec la requête `WITH RECURSIVE`. En effet, Oracle utilise un algorithme *depth-first* dans son implémentation du `CONNECT BY`. Ainsi, il explorera d'abord chaque branche avant de passer à la suivante. L'implémentation `WITH RECURSIVE` est de type *breadth-first* qui explore chaque niveau de hiérarchie avant de descendre.

Il est possible de retrouver l'ordre de tri d'une requête `CONNECT BY` pour une version antérieure à la 11g d'Oracle en triant sur une colonne `path`, telle qu'elle est construite pour émuler la clause `SYS_CONNECT_BY_PATH` :

```
WITH RECURSIVE emp_hierarchy (empno, ename, job, mgr, path) AS (
SELECT empno, ename, job, mgr, ARRAY[ename::text] AS path
  FROM emp
 WHERE mgr IS NULL
UNION ALL
SELECT emp.empno, emp.ename, emp.job, emp.mgr, prior.path
       || emp.ename::text AS path
  FROM emp
  JOIN emp_hierarchy prior ON (emp.mgr = prior.empno)
)
SELECT empno, ename, job FROM emp_hierarchy AS emp
ORDER BY path
```

Si vous utilisez Oracle 11g, la requête retournera quoi qu'il en soit les résultats dans un ordre différent.

5.11.2 WITH RECURSIVE

```
WITH RECURSIVE hierarchie AS (
condition de départ
UNION ALL
clause de récursion
```

```
)
SELECT * FROM hierarchie
```

5.11.3 NIVEAU DE HIÉRARCHIE

- `LEVEL` donne le niveau de hiérarchie
- condition de départ

  ```
  1 AS level
  ```

- clause de récursion

  ```
  prior.level + 1
  ```

La clause `LEVEL` permet d'obtenir le niveau de hiérarchie d'un élément.

```
SELECT empno, ename, job, mgr, level
  FROM emp
  START WITH mgr IS NULL
  CONNECT BY PRIOR empno = mgr
```

Le portage de la clause `LEVEL` est facile. La requête d'initialisation de la récursion initialise la colonne `level` à 1. La requête de récursion effectue ensuite une incrémentation de cette colonne pour chaque niveau de hiérarchie exploré :

```
WITH RECURSIVE emp_hierarchy (empno, ename, job, mgr, level) AS (
SELECT empno, ename, job, mgr, 1 AS level
  FROM emp
 WHERE mgr IS NULL
UNION ALL
SELECT emp.empno, emp.ename, emp.job, emp.mgr, prior.level + 1
  FROM emp
  JOIN emp_hierarchy prior ON (emp.mgr = prior.empno)
)
SELECT * FROM emp_hierarchy;
```

5.11.4 CHEMIN DE HIÉRARCHIE

- `niveau 1/niveau 2/niveau 3`
- condition de départ
  - `niveau initial AS path`
- clause de récursion
  - concatène le niveau précédent avec le path

```
      - prior.path || niveau courant
```

La clause `SYS_CONNECT_BY_PATH` permet d'obtenir un chemin où chaque élément est séparé de l' autre par un caractère donné. Par exemple, la requête suivante indique qui sont les différents responsables d'un employé de cette façon :

```
SELECT empno, ename, job, mgr, SYS_CONNECT_BY_PATH(ename, '/') AS path
  FROM emp
  START WITH mgr IS NULL
  CONNECT BY PRIOR empno = mgr
```

Le portage de la clause `SYS_CONNECT_BY_PATH` est également assez facile. La requête d'initialisation de la récursion construit l'élément racine : `'/' || ename AS path`. La requête de récursion réalise quant à elle une concaténation entre le `path` récupéré de la précédente itération et l'élément à concaténé : `prior.path || '/' || emp.ename` :

```
WITH RECURSIVE emp_hierarchy (empno, ename, job, mgr, path) AS (
SELECT empno, ename, job, mgr, '/' || ename AS path
  FROM emp
 WHERE mgr IS NULL
UNION ALL
SELECT emp.empno, emp.ename, emp.job, emp.mgr, prior.path || '/' || emp.ename
  FROM emp
  JOIN emp_hierarchy prior ON (emp.mgr = prior.empno)
)
SELECT * FROM emp_hierarchy
```

Une autre façon de faire est d' utiliser un tableau pour stocker le chemin le temps de la récursion, puis de construire la représentation textuelle de ces chemins au moment de la sortie des résultats. À noter la conversion de la valeur de `ename` en type `text` pour chaque élément ajouté dans le tableau `path`. Cette variante peut être utile pour l' émulation de la clause `NOCYCLE` comme vu plus bas :

```
WITH RECURSIVE emp_hierarchy (empno, ename, job, mgr, path) AS (
SELECT empno, ename, job, mgr, ARRAY[ename::text] AS path
  FROM emp
 WHERE mgr IS NULL
UNION ALL
SELECT emp.empno, emp.ename, emp.job, emp.mgr, prior.path ||
       emp.ename::text AS path
  FROM emp
  JOIN emp_hierarchy prior ON (emp.mgr = prior.empno)
)
SELECT empno, ename, job, array_to_string(path, '/') AS path
FROM emp_hierarchy AS emp
```

5.11.5 DÉTECTION DES CYCLES

- équivalent de `NOCYCLE`
- tableau contenant les éléments
  - pseudo colonne `cycle`
  - `element = ANY (tableau) AS cycle`
  - `WHERE cycle = false`

La requête Oracle suivante :

```
SELECT empno, ename, job, mgr
  FROM emp
  START WITH mgr IS NULL
  CONNECT BY NOCYCLE PRIOR empno = mgr
```

sera transposée pour PostgreSQL de la façon suivante :

```
WITH RECURSIVE emp_hierarchy (empno, ename, job, mgr, path, cycle) AS (
SELECT empno, ename, job, mgr, ARRAY[ename::text] AS path, false AS cycle
  FROM emp
 WHERE mgr IS NULL
UNION ALL
SELECT emp.empno, emp.ename, emp.job, emp.mgr, prior.path ||
       emp.ename::text AS path, emp.ename = ANY(prior.path) AS cycle
  FROM emp
  JOIN emp_hierarchy prior ON (emp.mgr = prior.empno)
 WHERE cycle = false
)
SELECT empno, ename, job, mgr
  FROM emp_hierarchy AS emp
 WHERE cycle = false;
```

5.12 INCOMPATIBILITÉS

- Certaines fonctionnalités Oracle sont sans équivalent.
  - hints
  - accès par ROWID

5.12.1 HINTS

- Forcer un plan d'exécution

- N'existent pas dans PostgreSQL

L'optimiseur Oracle supporte des *hints*, qui permettent au DBA de tromper l' optimiseur pour lui faire prendre des chemins que l'optimiseur a jugé trop coûteux. Ces *hints* sont exprimés sous la forme de commentaires et ne seront donc pas pris en compte par PostgreSQL, qui ne gère pas ces hints.

Néanmoins, une requête comportant un *hint* pour contrôler l'optimiseur Oracle doit faire l'objet d'une attention particulière, et l'analyse de son plan d' exécution devra être faite minutieusement, pour s'assurer que, sous PostgreSQL, la requête n'a pas de problème particulier, et agir en conséquence le cas échéant. C'est notamment vrai lorsque l'une des tables mise en œuvre est particulièrement volumineuse. Mais, de manière générale, l'ensemble des requêtes portées devront voir leur plan d'exécution vérifié.

Le plan d'exécution de la requête sera vérifiée avec l'ordre `EXPLAIN ANALYZE` qui fournit non seulement le plan d'exécution en précisant les estimations de sélectivité réalisées par l'optimiseur, mais va également exécuter la requête et fournir la sélectivité réelle de chaque nœud du plan d' exécution. Une forte divergence entre la sélectivité estimée et réelle permet de détecter un problème. Souvent, il s'agit d'un problème de précision des statistiques. Il est possible d'agir sur cette précision de plusieurs manières.

Tout d'abord, il est possible d'augmenter le nombre d'échantillons collectés, pour construire notamment les histogrammes. Le paramètre `default_statistics_target` contrôle la précision de cet échantillon. Pour une base de forte volumétrie, ce paramètre sera augmenté systématiquement dans une proportion raisonnable. Pour une base de volumétrie normale, ce paramètre sera plutôt augmenté en ciblant une colonne particulière avec l'ordre SQL `ALTER TABLE … ALTER COLUMN … SET STATISTICS …;`. De plus, il est possible de forcer artificiellement le nombre de valeurs distinctes d'une colonne avec l' ordre SQL `ALTER TABLE … SET COLUMN … SET n_distinct = …;`. Il est aussi souvent utile d'envisager une réécriture de la requête : si l'optimiseur, sous Oracle comme sous PostgreSQL, n'arrive pas à trouver un bon plan, c'est probablement qu'elle est écrite d'une façon qui empêche ce dernier de travailler correctement.

5.12.2 ACCÈS PAR ROWID

- localisation physique d'une ligne dans une table.
  - il existe un équivalent dans PostgreSQL
  - mais c'est à proscrire

Dans de très rares cas, des requêtes SQL utilisent la colonne `ROWID` d'Oracle, par exemple pour dédoublonner des enregistrements. Le `ROWID` est la localisation physique d'une ligne

dans une table. L'équivalent dans PostgreSQL est le `ctid`.

Plus précisément, le `ROWID` Oracle représente une *adresse* logique d'une ligne, encodée sous la forme `OOOOOO.FFF.BBBBBB.RRR` où O représente le numéro d'objet, F le fichier, B le numéro de bloc et R la ligne dans le bloc. Le format est différent dans le cas d'une table stockée dans un BIG FILE TABLESPACE, mais le principe reste identique.

Quant au `ctid` de PostgreSQL, il ne représente qu'un couple *(numéro du bloc, numéro de l'enregistrement)*, aucune autre information de localisation physique n'est disponible. Le `ctid` n'est donc unique qu'au sein d'une table. De part ce fait, une requête ramenant le `ctid` des lignes d'une table partitionnée peut présenter des `ctid` en doublons. On peut dans ce cas utiliser le champ caché `tableoid` (l'identifiant unique de la table dans le catalogue) de chaque table pour différencier les doublons par partition.

Cette méthode d'accès est donc à proscrire, sauf opération particulière et cadrée.
