Formation MIGORPG

Migrer d’Oracle à PostgreSQL

24.09

Table des matières

Sur ce document

Formation	Formation MIGORPG
Titre	Migrer d'Oracle à PostgreSQL
Révision	24.09
ISBN	978-2-38168-121-4
PDF	https://dali.bo/migorpg_pdf
EPUB	https://dali.bo/migorpg_epub
HTML	https://dali.bo/migorpg_html
Slides	https://dali.bo/migorpg_slides

Vous trouverez en ligne les différentes versions complètes de ce document. La version imprimée ne contient pas les travaux pratiques. Ils sont présents dans la version numérique (PDF ou HTML).

Chers lectrices & lecteurs,

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

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

Nous mettons un point d'honneur à maintenir nos manuels à jour, avec des informations précises et des exemples détaillés. Toutefois malgré nos efforts et nos multiples relectures, il est probable que ce document contienne des oublis, des coquilles, des imprécisions ou des erreurs. Si vous constatez un souci, n'hésitez pas à le signaler via l'adresse formation@dalibo.com[1] !

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

[1] mailto:formation@dalibo.com

Remerciements

Ce manuel de formation est une aventure collective qui se transmet au sein de notre société depuis des années. Nous remercions chaleureusement ici toutes les personnes qui ont contribué directement ou indirectement à cet ouvrage, notamment :

Jean-Paul Argudo, Alexandre Anriot, Carole Arnaud, Alexandre Baron, David Bidoc, Sharon Bonan, Franck Boudehen, Arnaud Bruniquel, Pierrick Chovelon, Damien Clochard, Christophe Courtois, Marc Cousin, Gilles Darold, Jehan-Guillaume de Rorthais, Ronan Dunklau, Vik Fearing, Stefan Fercot, Pierre Giraud, Nicolas Gollet, Dimitri Fontaine, Florent Jardin, Virginie Jourdan, Luc Lamarle, Denis Laxalde, Guillaume Lelarge, Alain Lesage, Benoit Lobréau, Jean-Louis Louër, Thibaut Madelaine, Adrien Nayrat, Alexandre Pereira, Flavie Perette, Robin Portigliatti, Thomas Reiss, Maël Rimbault, Julien Rouhaud, Stéphane Schildknecht, Julien Tachoires, Nicolas Thauvin, Be Hai Tran, Christophe Truffier, Cédric Villemain, Thibaud Walkowiak, Frédéric Yhuel.

Forme de ce manuel

Les versions PDF, EPUB ou HTML de ce document sont structurées autour des slides de nos formations. Le texte suivant chaque slide contient le cours et de nombreux détails qui ne peuvent être données à l'oral.

Licence Creative Commons CC-BY-NC-SA

Cela inclut les diapositives, les manuels eux-mêmes et les travaux pratiques. Cette formation peut également contenir quelques images et schémas dont la redistribution est soumise à des licences différentes qui sont alors précisées.

Marques déposées

PostgreSQL® Postgres® et le logo Slonik sont des marques déposées[3] par PostgreSQL Community Association of Canada.

Versions de PostgreSQL couvertes

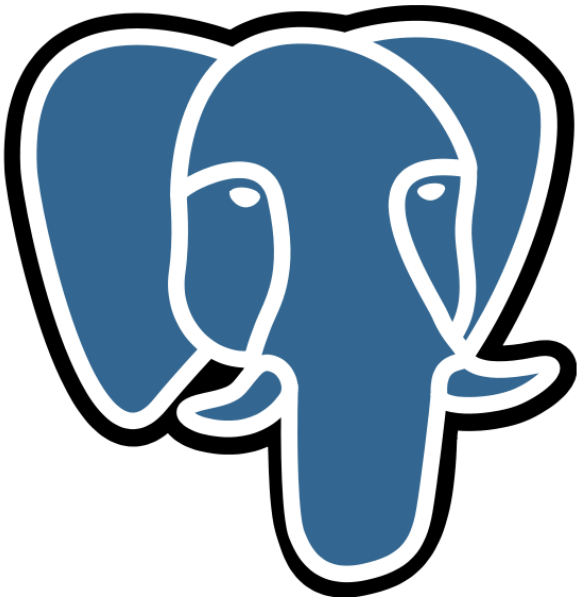
Ce document ne couvre que les versions supportées de PostgreSQL au moment de sa rédaction, soit les versions 12 à 16.

Sur les versions précédentes susceptibles d'être encore rencontrées en production, seuls quelques points très importants sont évoqués, en plus éventuellement de quelques éléments historiques.

Sauf précision contraire, le système d'exploitation utilisé est Linux.

[3] https://www.postgresql.org/about/policies/trademarks/

1/ Plan de migration



1.1 INTRODUCTION

Ce module est organisé en quatre parties :

- Méthodologie de la migration
- Choix de l'outil de migration
- Installation d'Ora2Pg
- Rapport d'évaluation

Ce module est une introduction aux migrations de Oracle vers PostgreSQL. Nous y abordons comment gérer sa première migration (quel que soit le SGBD source et destination), puis nous réfléchirons sur le contenu de la migration et sur le choix de l'outil idoine.

À l'issue de ce module, l'outil Ora2Pg sera installé sur l'environnement Linux de formation pour permettre la génération d'un premier rapport d'évaluation.

1.2 AVERTISSEMENT

- Ceci est écrit par des spécialistes de PostgreSQL
  - pas d'Oracle

Notre expertise est sur PostgreSQL et nous nous reposons sur notre expérience de plusieurs années de migrations Oracle vers PostgreSQL pour écrire cette formation.

Malgré tous nos efforts, certaines informations sur Oracle pourraient être erronées, ou ne plus être vraies avec les dernières versions. Nos clients n'utilisent pas forcément les mêmes fonctionnalités, avec le même niveau d'expertise, et nous n'avons pas forcément correctement restitué leur retour.

Si vous constatez des erreurs ou des manques, n'hésitez pas à nous en faire part[1] pour que nous puissions améliorer ce support de formation.

[1] mailto:formation@dalibo.com

1.3 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.

1.3.1 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 routines 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 souci. Prenez une base critique de plusieurs To, dotée d'un très grand nombre de routines 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 conséquente (plusieurs dizaines de Go), avec quelques routines stockées, de la réplication, etc. Cela aura une vraie valeur, tout en étant accessible 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 impliquées dans ce projet, pour qu'ils puissent intégrer les changements quant à l'utilisation de ce nouveau SGBD.

1.3.2 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.

1.3.3 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.

1.3.4 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 performance 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.

1.3.5 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 base 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 vient 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.).

1.3.6 Motiver

- Formations indispensables
- 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.

1.3.7 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.

1.3.8 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).

1.3.9 Coûts

- Budget ?
- 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.

1.3.10 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 essentiel qu'elle soit documentée, discutée, pour que le travail effectué soit réutilisable (soit complètement, soit uniquement l'expérience et les méthodes) afin que la prochaine migration soit moins coûteuse.

1.3.11 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 l'on veut continuer, autant donner du temps aux équipes responsables de la migration.

1.4 CHOIX DE L'OUTIL DE MIGRATION

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 routines stockées, il faut procéder à la migration du schéma et des données. C'est donc ce dont nous allons parler dans cette partie.

1.4.1 Besoins de la migration : schéma

- Veut-on migrer le schéma tel quel ?
- Utiliser les fonctionnalités de PostgreSQL ?
  - n'est plus vraiment à isofonctionnalité
- 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 isofonctionnelle.

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

1.4.2 Besoins de la migration : types

- On rencontre souvent les types suivants 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.

1.4.3 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.

1.4.4 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.

1.4.5 Migration des données

Veut-on :

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

Toujours dans les décisions à prendre avant la migration, il est important de savoir si l'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.

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.

1.4.6 Fonctionnalités problématiques

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

- Vues matérialisées (mise à jour)
- Partitionnement (différences)
- Synonymes
- Conversion de type implicite
- Absence de *hint* (tag de requête)
- Accès par ROWID

Certaines fonctionnalités Oracle n'ont pas d'équivalents natifs dans PostgreSQL. Il faut avoir conscience que l'outil de migration ne pourra pas convertir intégralement les objets avancés disponibles sur Oracle.

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[2]
- Materialized Views that Really Work[3]

Partitionnement

Les partitions telles que gérées sous Oracle existent sous PostgreSQL depuis la version 10. Avec une version inférieure, il faut utiliser l'héritage et définir des triggers et contraintes CHECK. Pour plus de détails, consultez le document 5.9. Partitionnement de tables[4] ainsi que le document Partitionnement (module DBA2)[5].

Les types de partitions supportées sont `LIST` et `RANGE`. Les partitions de type `HASH` sont supportées par PostgreSQL 11. Le partitionnement par référence n'est pas supporté.

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.

Absence de *hint*

[2] http://tech.jonathangardner.net/wiki/PostgreSQL/Materialized_Views
[3] https://www.pgcon.org/2008/schedule/events/69.en.html
[4] https://docs.postgresql.fr/current/ddl-partitioning.html
[5] https://dali.bo/v1_html

L'optimiseur Oracle supporte des *hints*, qui permettent au DBA de tromper l'optimiseur pour lui faire prendre des chemins que l'optimiseur a jugés 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 mises 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 vérifié 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.

Accès par ROWID

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.

1.4.7 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.

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 et MySQL, alors que les ETL autorisent un plus grand nombre de sources de données et de destinations, si bien que l'expérience acquise pour la migration d'Oracle vers PostgreSQL peut être réutilisée 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, avec des règles avancées de reprise d'étapes en erreur ou de conditions à remplir pour déclencher d'autres actions (comme la création des index ou l'activation des contraintes).

1.4.8 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 système 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`.

1.4.9 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.

1.4.10 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 -> PL/pgSQL
  - au partitionnement (par héritage, ou déclaratif)

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 réalise aussi automatiquement la conversion des types de données. Enfin, il s'occupe de la déclaration et du code des routines stockées (uniquement PL/SQL vers PL/pgSQL). Il propose aussi une aide au partitionnement, dont l'implémentation est vraiment différente entre Oracle et PostgreSQL.

1.4.11 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, quelques fois plus qu'Ora2Pg.

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

1.4.12 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 donc complexes. Cela demandera un apprentissage bien plus long que pour Ora2Pg. Cependant, ils sont utilisables dans bien plus de cas qu'Ora2Pg.

1.5 INSTALLATION D'ORA2PG

Étapes :

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

Nous allons aborder dans cette 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 ?

1.5.1 Téléchargement

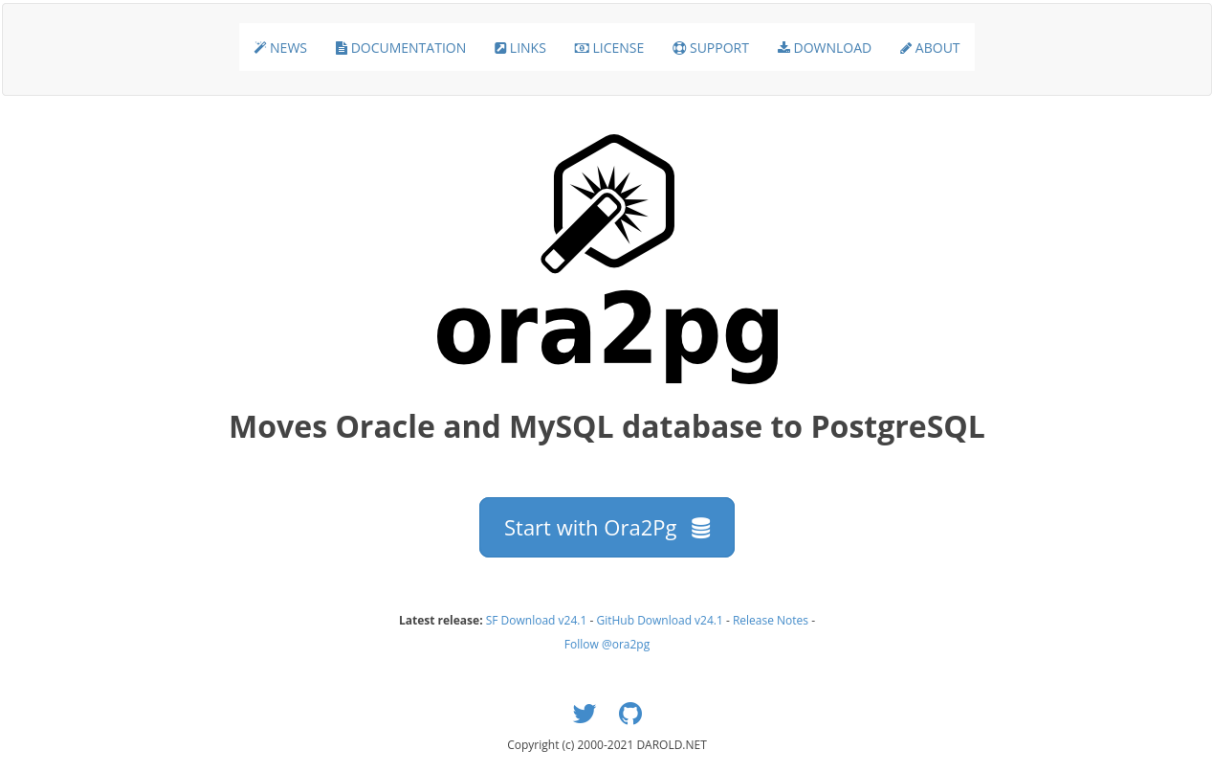
- Disponible via :
  - Web : https://ora2pg.darold.net/
  - Git : https://github.com/darold/ora2pg/
  - SourceForge : https://sourceforge.net/projects/ora2pg
- Dernière version : 24.3 (28 mars 2024)

Les fichiers sources et les instructions de compilation sont accessibles depuis le site officiel du projet[6].

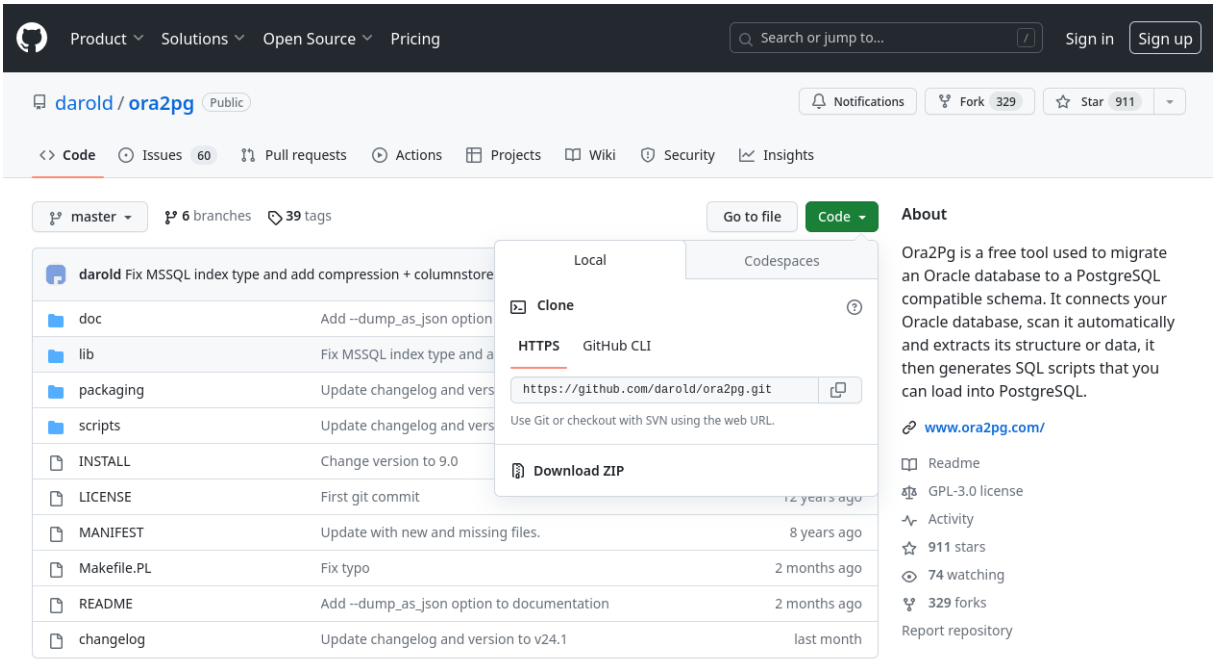
Il est très important de toujours télécharger la dernière version, car l'ajout de fonctionnalités et les corrections de bogues sont permanentes. En effet, ce projet bénéficie d'améliorations et de corrections au fur et à mesure des retours d'expérience des utilisateurs. Il est constamment mis à jour.

La dernière distribution officielle peut être téléchargée directement depuis GitHub.com[7], où la dernière version est disponible aux formats **zip** ou **tar.gz**.

[6] https://ora2pg.darold.net/
[7] https://github.com/darold/ora2pg/releases/



Pour obtenir le dernier code en développement, il faut aller sur la page du dépôt GitHub (https://github.com/darold/ora2pg) et cliquer sur le bouton *Download ZIP* de la branche de développement. Le fichier téléchargé sera nommé `ora2pg-master.zip`.



1.5.2 Dépendances requises

- Oracle >= 8i client ou serveur
- Environnement Oracle correct (`ORACLE_HOME` et `PATH` comprenant `sqlplus`)
- `libaio` (Redhat) ou `libaio1` (Debian)
- Unix : Perl 5.10+
- Windows : Strawberry Perl 5.10+ ou ActiveStep Perl 5.10+
- Modules Perl
  - `Time::HiRes`
  - `Perl DBI > v1.614` 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[8]. 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 12c 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 `libaio` . 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/x.x/client64
```

Pour une installation sous Windows, l'utilisation de Strawberry Perl[9] nécessitera les outils de compilation pour l'installation de `DBD::Oracle` alors que l'utilisation de la distribution libre d'ActiveState[10]

[8] http://search.cpan.org/
[9] http://strawberryperl.com/
[10] https://www.activestate.com/activeperl/downloads

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`.

1.5.3 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
- `DBD::ODBC` pour migrer les bases Microsoft SQL Server

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 permet de migrer les bases de données MySQL depuis la version 16.0 et les bases de données SQL Server depuis la version 24.0. Tout comme pour Oracle, Ora2Pg a besoin de se connecter à l'instance distante au travers d'un pilote Perl. C'est le rôle des modules Perl `DBD::MySQL` ou `DBD::ODBC`.

1.5.4 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-21.0.tar.bz2
cd ora2pg-21.0/
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.

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 le modifier 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.

1.6 ÉVALER UNE MIGRATION

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  : Comma separated list of objects to allow from export.
                    Can be used with SHOW_COLUMN too.
-b | --basedir dir: Set the default output directory, where files
                    resulting from exports will be stored.
-c | --conf file  : Set an alternate configuration file other than the
                    default /etc/ora2pg/ora2pg.conf.
-C | --cdc_file file: File used to store/read SCN per table during export.
                      default: TABLES_SCN.log in the current directory. This
                      is the file written by the --cdc_ready option.
-d | --debug      : Enable verbose output.
-D | --data_type str : Allow custom type replacement at command line.
-e | --exclude str: Comma separated list of objects to exclude from export.
                    Can be used with SHOW_COLUMN too.
-h | --help       : Print this short help.
-g | --grant_object type : Extract privilege from the given object type.
                    See possible values with GRANT_OBJECT configuration.
-i | --input file : File containing Oracle PL/SQL code to convert with
                    no Oracle database connection initiated.
-j | --jobs num   : Number of parallel process to send data to PostgreSQL.
-J | --copies num : Number of parallel connections to extract data from Oracle.
-l | --log file   : Set a log file. Default is stdout.
-L | --limit num  : Number of tuples extracted from Oracle and stored in
                    memory before writing, default: 10000.
-m | --mysql      : Export a MySQL database instead of an Oracle schema.
-M | --mssql      : Export a Microsoft SQL Server database.
-n | --namespace schema : Set the Oracle schema to extract from.
-N | --pg_schema schema : Set PostgreSQL's search_path.
-o | --out file   : Set the path to the output file where SQL will
                    be written. Default: output.sql in running directory.
-p | --plsql      : Enable PLSQL to PLPGSQL code conversion.
-P | --parallel num: Number of parallel tables to extract at the same time.
-q | --quiet      : Disable progress bar.
```

```
  -r | --relative   : use \ir instead of \i in the psql scripts generated.
  -s | --source DSN : Allow to set the Oracle DBI datasource.
  -S | --scn    SCN : Allow to set the Oracle System Change Number (SCN) to
                      use to export data. It will be used in the WHERE clause
                      to get the data. It is used with action COPY or INSERT.
  -t | --type export: Set the export type. It will override the one
                      given in the configuration file (TYPE).
  -T | --temp_dir dir: Set a distinct temporary directory when two
                      or more ora2pg are run in parallel.
  -u | --user name  : Set the Oracle database connection user.
                      ORA2PG_USER environment variable can be used instead.
  -v | --version    : Show Ora2Pg Version and exit.
  -w | --password pwd : Set the password of the Oracle database user.
                      ORA2PG_PASSWD environment variable can be used instead.
  -W | --where clause : Set the WHERE clause to apply to the Oracle query to
                      retrieve data. Can be used multiple time.
  --forceowner      : Force ora2pg to set tables and sequences owner like in
                Oracle database. If the value is set to a username this one
                will be used as the objects owner. By default it's the user
                used to connect to the Pg database that will be the owner.
  --nls_lang code: Set the Oracle NLS_LANG client encoding.
  --client_encoding code: Set the PostgreSQL client encoding.
  --view_as_table str: Comma separated list of views to export as table.
  --estimate_cost   : Activate the migration cost evaluation with SHOW_REPORT
  --cost_unit_value minutes: Number of minutes for a cost evaluation unit.
                default: 5 minutes, corresponds to a migration conducted by a
                PostgreSQL expert. Set it to 10 if this is your first migration.
--dump_as_html     : Force ora2pg to dump report in HTML, used only with
                     SHOW_REPORT. Default is to dump report as simple text.
--dump_as_csv      : As above but force ora2pg to dump report in CSV.
--dump_as_json     : As above but force ora2pg to dump report in JSON.
--dump_as_sheet    : Report migration assessment with one CSV line per database.
--init_project name: Initialise a typical ora2pg project tree. Top directory
                     will be created under project base dir.
--project_base dir : Define the base dir for ora2pg project trees. Default
                     is current directory.
--print_header     : Used with --dump_as_sheet to print the CSV header
                     especially for the first run of ora2pg.
--human_days_limit num : Set the number of human-days limit where the migration
                     assessment level switch from B to C. Default is set to
                     5 human-days.
--audit_user list  : Comma separated list of usernames to filter queries in
                     the DBA_AUDIT_TRAIL table. Used only with SHOW_REPORT
                     and QUERY export type.
--pg_dsn DSN       : Set the datasource to PostgreSQL for direct import.
--pg_user name     : Set the PostgreSQL user to use.
--pg_pwd password  : Set the PostgreSQL password to use.
--count_rows       : Force ora2pg to perform a real row count in TEST,
                     TEST_COUNT and SHOW_TABLE actions.
--no_header        : Do not append Ora2Pg header to output file
--oracle_speed     : Use to know at which speed Oracle is able to send
                     data. No data will be processed or written.
--ora2pg_speed     : Use to know at which speed Ora2Pg is able to send
                     transformed data. Nothing will be written.
--blob_to_lo       : export BLOB as large objects, can only be used with
                     action SHOW_COLUMN, TABLE and INSERT.
```

```
    --cdc_ready         : use current SCN per table to export data and register
                          them into a file named TABLES_SCN.log per default. It
                          can be changed using -C | --cdc_file.
    --lo_import         : use psql \lo_import command to import BLOB as large
                          object. Can be use to import data with COPY and import
                          large object manually in a second pass. It is recquired
                          for BLOB > 1GB. See documentation for more explanation.
    --mview_as_table str: Comma separated list of materialized views to export
                          as regular table.
    --drop_if_exists    : Drop the object before creation if it exists.
    --delete clause     : Set the DELETE clause to apply to the Oracle query to
                          be applied before importing data. Can be used multiple
             time.
```

1.6.1 Rapport d'évaluation - 1

- Rapport exhaustif du contenu de la base Oracle

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

Par défaut, le rapport se présente au format texte sur la sortie standard. Pour proposer un format plus lisible, il est **recommandé** d'activer l'option `--dump_as_html` qui crée un rapport au format HTML. D'autres formats existent, comme le CSV (`--dump_as_csv`) ou le JSON (`--dump_as_json`), si la centralisation et la consolidation de rapports sont nécessaires.

Voici un exemple de rapport obtenu avec cette commande :

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

-------------------------------------------------------------------------------
Object  Number  Invalid Comments    Details
-------------------------------------------------------------------------------
```

```
DATABASE LINK   2   0   Database links will be exported as SQL/MED PostgreSQL's
                Foreign Data Wrapper (FDW) extensions using oracle_fdw.
GLOBAL TEMPORARY TABLE  0   0   Global temporary table are not supported by
                PostgreSQL and will not be exported. You will have to
                rewrite some application code to match the PostgreSQL
                temporary table behavior.
INDEX   28  0   19 index(es) are concerned by the export, others are automatically
                generated and will do so on PostgreSQL. Bitmap will be
                exported as btree_gin index(es) and hash index(es) will be
                exported as b-tree index(es) if any. Domain index are exported
                as b-tree but commented to be edited to mainly use FTS.
                Cluster, bitmap join and IOT indexes will not be exported at all.
                Reverse indexes are not exported too, you may use a trigram-based
                index (see pg_trgm) or a reverse() function based index and search.
               Use 'varchar_pattern_ops', 'text_pattern_ops' or 'bpchar_pattern_ops'
                operators in your indexes to improve search with the LIKE operator
                respectively into varchar, text or char columns.
                3 function based b-tree index(es).
                13 b-tree index(es).
                3 spatial index index(es).
INDEX PARTITION 2   0   Only local indexes partition are exported, they are
                build on the column used for the partitioning.
JOB 0   0   Job are not exported. You may set external cron job with them.
PROCEDURE   3   1   Total size of procedure code: 1870 bytes.
SEQUENCE    3   0   Sequences are fully supported, but all call to
                sequence_name.NEXTVAL or sequence_name.CURRVAL will be
                transformed into NEXTVAL('sequence_name') or
                CURRVAL('sequence_name').
SYNONYM 0   0   SYNONYMs will be exported as views. SYNONYMs do not exists with
                PostgreSQL but a common workaround is to use views or set the
                PostgreSQL search_path in
                your session to access object outside the current schema.
TABLE   22  0    2 check constraint(s). 1 unknown types. Total number of rows: 421.
                Top 10 of tables sorted by number of rows:.
                sg_infrastructure_route has 188 rows.
                employees has 107 rows. departments has 27 rows. countries has
                25 rows.
                locations has 23 rows. jobs has 19 rows. job_history has 10 rows.
                error_log_sample has 6 rows. emptyclob has 4 rows. regions has
                4 rows.
                Top 10 of largest tables:...
TABLE PARTITION 5   0   Partitions are exported using table inheritance and
                check constraint.
                Hash and Key partitions are not supported by PostgreSQL and will not
                be exported.
                5 RANGE partitions..
TRIGGER 3   1   Total size of trigger code: 736 bytes.
VIEW    1   0   Views are fully supported but can use specific functions.
--------------------------------------------------------------------------------
Total   69  2
--------------------------------------------------------------------------------
```

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

1.6.2 Rapport d'évaluation - 2

- Estimation du coût de migration

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

Évaluer la charge de migration d'une base Oracle

Pour déterminer le coût en jours/personne de la migration, Ora2Pg dispose d'une directive de configuration nommée `ESTIMATE_COST`. Celle-ci peut aussi être activée en ligne de commande : `--estimate_cost`. Cette fonctionnalité n'est disponible qu'avec le type d'export `SHOW_REPORT`.

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

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

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

-------------------------------------------------------------------------------
Object  Number  Invalid Estimated cost  Comments    Details
-------------------------------------------------------------------------------
DATABASE LINK   2   0   6   Database links will be exported as SQL/MED PostgreSQL's
                    Foreign Data Wrapper (FDW) extensions using oracle_fdw.
GLOBAL TEMPORARY TABLE  0   0   0   Global temporary table are not supported by
                    PostgreSQL and will not be exported. You will have to
                    rewrite some application code to match the PostgreSQL
                    temporary table behavior.
INDEX   28  0   5.3 19 index(es) are concerned by the export, others are
                    automatically
                    generated and will do so on PostgreSQL. Bitmap will be
                    exported as btree_gin index(es) and hash index(es) will be
                    exported as b-tree index(es) if any. Domain index are exported
                    as b-tree but commented to be edited to mainly use FTS.
                   Cluster, bitmap join and IOT indexes will not be exported at all.
                    Reverse indexes are not exported too, you may use a
                    trigram-based index (see pg_trgm) or a reverse() function
                    based index and search.  Use 'varchar_pattern_ops',
                    'text_pattern_ops' or 'bpchar_pattern_ops' operators in
```

```
                    your indexes to improve search with the LIKE operator
                    respectively into varchar, text or char columns.
                    3 function based b-tree index(es). 13 b-tree index(es).
                    3 spatial index index(es).
INDEX PARTITION 2   0   0   Only local indexes partition are exported, they are
                    build on the column used for the partitioning.
JOB 0   0   0   Job are not exported. You may set external cron job with them.
PROCEDURE   3   1   16  Total size of procedure code: 1870 bytes.
                    add_job_history: 3. test_dupl_vazba: 7. secure_dml: 3.
SEQUENCE    3   0   1   Sequences are fully supported, but all call to
                    sequence_name.NEXTVAL or sequence_name.CURRVAL will be
                    transformed into NEXTVAL('sequence_name')
                    or CURRVAL('sequence_name').
SYNONYM 0   0   0   SYNONYMs will be exported as views. SYNONYMs do not exists
                    with PostgreSQL but a common workaround is to use views
                    or set the PostgreSQL search_path in
                    your session to access object outside the current schema.
TABLE   22  0   2.4  2 check constraint(s). 1 unknown types. Total number of
                    rows: 421.
                    Top 10 of tables sorted by number of rows:.
                    sg_infrastructure_route
                    has 188 rows. employees has 107 rows. departments has 27 rows.
                    countries has 25 rows. locations has 23 rows. jobs has 19 rows.
                    job_history has 10 rows. error_log_sample has 6 rows.
                    regions has 4 rows.
                    emptyclob has 4 rows. Top 10 of largest tables:.
TABLE PARTITION 5   0   1   Partitions are exported using table inheritance
                    and check constraint.
                    Hash and Key partitions are not supported by PostgreSQL
                    and will not be exported.
                    5 RANGE partitions..
TRIGGER 3   1   9.3 Total size of trigger code: 736 bytes.  cisvpolpre_bi:
                    3.3. update_job_history: 3.
VIEW    1   0   1   Views are fully supported but can use specific functions.
-------------------------------------------------------------------------------
Total   69  2   42.0042.00 cost migration units means approximatively 1 man-day(s).
                    The migration unit was set to 5 minute(s)

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

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

Details of cost assessment per function
Function test_dupl_vazba total estimated cost: 7
```

```
    CONCAT => 9 (cost: 0.1)
    TEST => 2
    SIZE => 1
    TO_CHAR => 1 (cost: 0.1)
    PRAGMA => 1 (cost: 3)
Function add_job_history total estimated cost: 3
    TEST => 2
    SIZE => 1
Function secure_dml total estimated cost: 3
    TEST => 2
    SIZE => 1
-------------------------------------------------------------------------------

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

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

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

```
COST_UNIT_VALUE       10
```

ou en ligne de commande :

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

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

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

```
DUMP_AS_HTML       1
```

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

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

Ora2Pg propose un exemple de rapport en HTML sur son site : Ora2Pg - Database Migration Report[11]

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

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

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

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

1.7 CONCLUSION

Points essentiels :

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

1.7.1 Questions

N'hésitez pas, c'est le moment !

1.8 QUIZ

https://dali.bo/n1_quiz

1.9 INSTALLATION DE POSTGRESQL DEPUIS LES PAQUETS COMMUNAUTAIRES

L'installation est détaillée ici pour Rocky Linux 8 et 9 (similaire à Red Hat et à d'autres variantes comem Oracle Linux et Fedora), et Debian/Ubuntu.

Elle ne dure que quelques minutes.

1.9.1 Sur Rocky Linux 8 ou 9

ATTENTION : Red Hat, CentOS, Rocky Linux fournissent souvent par défaut des versions de PostgreSQL qui ne sont plus supportées. Ne jamais installer les packages `postgresql`, `postgresql-client` et `postgresql-server` ! L'utilisation des dépôts du PGDG est fortement conseillée.

Installation du dépôt communautaire :

Les dépôts de la communauté sont sur https://yum.postgresql.org/. Les commandes qui suivent sont inspirées de celles générées par l'assistant sur https://www.postgresql.org/download/linux/redhat/, en précisant :

- la version majeure de PostgreSQL (ici la 16) ;
- la distribution (ici Rocky Linux 8) ;
- l'architecture (ici x86_64, la plus courante).

Les commandes sont à lancer sous **root** :

```
# dnf install -y https://download.postgresql.org/pub/repos/yum/reporpms\
/EL-8-x86_64/pgdg-redhat-repo-latest.noarch.rpm

# dnf -qy module disable postgresql
```

Installation de PostgreSQL 16 (client, serveur, librairies, extensions) :

```
# dnf install -y postgresql16-server postgresql16-contrib
```

Les outils clients et les librairies nécessaires seront automatiquement installés.

Une fonctionnalité avancée optionnelle, le JIT (*Just In Time compilation*), nécessite un paquet séparé.

```
# dnf install postgresql16-llvmjit
```

Création d'une première instance :

Il est conseillé de déclarer `PG_SETUP_INITDB_OPTIONS`, notamment pour mettre en place les sommes de contrôle et forcer les traces en anglais :

```
# export PGSETUP_INITDB_OPTIONS='--data-checksums --lc-messages=C'
# /usr/pgsql-16/bin/postgresql-16-setup initdb
# cat /var/lib/pgsql/16/initdb.log
```

Ce dernier fichier permet de vérifier que tout s'est bien passé et doit finir par :

```
Success. You can now start the database server using:

    /usr/pgsql-16/bin/pg_ctl -D /var/lib/pgsql/16/data/ -l logfile start
```

Chemins :

Objet	Chemin
Binaires	`/usr/pgsql-16/bin`
Répertoire de l'utilisateur **postgres**	`/var/lib/pgsql`
`PGDATA` par défaut	`/var/lib/pgsql/16/data`
Fichiers de configuration	dans `PGDATA/`
Traces	dans `PGDATA/log`

Configuration :

Modifier `postgresql.conf` est facultatif pour un premier lancement.

Commandes d'administration habituelles :

Démarrage, arrêt, statut, rechargement à chaud de la configuration, redémarrage :

```
# systemctl start postgresql-16
# systemctl stop postgresql-16
# systemctl status postgresql-16
# systemctl reload postgresql-16
# systemctl restart postgresql-16
```

Test rapide de bon fonctionnement et connexion à psql :

```
# systemctl --all |grep postgres
# sudo -iu postgres psql
```

Démarrage de l'instance au lancement du système d'exploitation :

```
# systemctl enable postgresql-16
```

Ouverture du *firewall* pour le port 5432 :

Voir si le *firewall* est actif :

```
# systemctl status firewalld
```

Si c'est le cas, autoriser un accès extérieur :

```
# firewall-cmd --zone=public --add-port=5432/tcp --permanent
# firewall-cmd --reload
# firewall-cmd --list-all
```

(Rappelons que `listen_addresses` doit être également modifié dans `postgresql.conf`.)

Création d'autres instances :

Si des instances de *versions majeures différentes* doivent être installées, il faut d'abord installer les binaires pour chacune (adapter le numéro dans `dnf install …`) et appeler le script d'installation de chaque version. l'instance par défaut de chaque version vivra dans un sous-répertoire numéroté de `/var/lib/pgsql` automatiquement créé à l'installation. Il faudra juste modifier les ports dans les `postgresql.conf` pour que les instances puissent tourner simultanément.

Si plusieurs instances d'une *même version majeure* (forcément de la même version mineure) doivent cohabiter sur le même serveur, il faut les installer dans des `PGDATA` différents.

- Ne pas utiliser de tiret dans le nom d'une instance (problèmes potentiels avec systemd).
- Respecter les normes et conventions de l'OS : placer les instances dans un nouveau sous-répertoire de `/var/lib/pgsqsl/16/` (ou l'équivalent pour d'autres versions majeures).

Pour créer une seconde instance, nommée par exemple **infocentre** :

- Création du fichier service de la deuxième instance :

```
# cp /lib/systemd/system/postgresql-16.service \
        /etc/systemd/system/postgresql-16-infocentre.service
```

- Modification de ce dernier fichier avec le nouveau chemin :

```
Environment=PGDATA=/var/lib/pgsql/16/infocentre
```

- Option 1 : création d'une nouvelle instance vierge :

```
# export PGSETUP_INITDB_OPTIONS='--data-checksums --lc-messages=C'
# /usr/pgsql-16/bin/postgresql-16-setup initdb postgresql-16-infocentre
```

- Option 2 : restauration d'une sauvegarde : la procédure dépend de votre outil.
- Adaptation de `/var/lib/pgsql/16/infocentre/postgresql.conf` (port surtout).
- Commandes de maintenance de cette instance :

```
# systemctl [start|stop|reload|status] postgresql-16-infocentre
# systemctl [enable|disable] postgresql-16-infocentre
```

- Ouvrir le nouveau port dans le firewall au besoin.

1.9.2 Sur Debian / Ubuntu

Sauf précision, tout est à effectuer en tant qu'utilisateur **root**.

Référence : https://apt.postgresql.org/

Installation du dépôt communautaire :

L'installation des dépôts du PGDG est prévue dans le paquet Debian :

```
# apt update
# apt install -y  gnupg2  postgresql-common
# /usr/share/postgresql-common/pgdg/apt.postgresql.org.sh
```

Ce dernier ordre créera le fichier du dépôt `/etc/apt/sources.list.d/pgdg.list` adapté à la distribution en place.

Installation de PostgreSQL 16 :

La méthode la plus propre consiste à modifier la configuration par défaut avant l'installation :

Dans `/etc/postgresql-common/createcluster.conf`, paramétrer au moins les sommes de contrôle et les traces en anglais :

```
initdb_options = '--data-checksums --lc-messages=C'
```

Puis installer les paquets serveur et clients et leurs dépendances :

```
# apt install postgresql-16 postgresql-client-16
```

La première instance est automatiquement créée, démarrée et déclarée comme service à lancer au démarrage du système. Elle porte un nom (par défaut `main`).

Elle est immédiatement accessible par l'utilisateur système **postgres**.

Chemins :

Objet	Chemin
Binaires	`/usr/lib/postgresql/16/bin/`
Répertoire de l'utilisateur **postgres**	`/var/lib/postgresql`
PGDATA de l'instance par défaut	`/var/lib/postgresql/16/main`
Fichiers de configuration	dans `/etc/postgresql/16/main/`
Traces	dans `/var/log/postgresql/`

Configuration

Modifier `postgresql.conf` est facultatif pour un premier essai.

Démarrage/arrêt de l'instance, rechargement de configuration :

Debian fournit ses propres outils, qui demandent en paramètre la version et le nom de l'instance :

```
# pg_ctlcluster 16 main [start|stop|reload|status|restart]
```

Démarrage de l'instance avec le serveur :

C'est en place par défaut, et modifiable dans `/etc/postgresql/16/main/start.conf`.

Ouverture du firewall :

Debian et Ubuntu n'installent pas de firewall par défaut.

Statut des instances du serveur :

```
# pg_lsclusters
```

Test rapide de bon fonctionnement et connexion à psql :

```
# systemctl --all |grep postgres
# sudo -iu postgres psql
```

Destruction d'une instance :

```
# pg_dropcluster 16 main
```

Création d'autres instances :

Ce qui suit est valable pour remplacer l'instance par défaut par une autre, par exemple pour mettre les *checksums* en place :

- optionnellement, `/etc/postgresql-common/createcluster.conf` permet de mettre en place tout d'entrée les *checksums*, les messages en anglais, le format des traces ou un emplacement séparé pour les journaux :

```
initdb_options = '--data-checksums --lc-messages=C'
log_line_prefix = '%t [%p]: [%l-1] user=%u,db=%d,app=%a,client=%h '
waldir = '/var/lib/postgresql/wal/%v/%c/pg_wal'
```

- créer une instance :

```
# pg_createcluster 16 infocentre
```

Il est également possible de préciser certains paramètres du fichier `postgresql.conf`, voire les chemins des fichiers (il est conseillé de conserver les chemins par défaut) :

```
# pg_createcluster 16 infocentre \
  --port=12345 \
  --datadir=/PGDATA/16/infocentre \
  --pgoption shared_buffers='8GB' --pgoption work_mem='50MB' \
  --  --data-checksums --waldir=/ssd/postgresql/16/infocentre/journaux
```

- adapter au besoin `/etc/postgresql/16/infocentre/postgresql.conf` ;
- démarrage :

```
# pg_ctlcluster 16 infocentre start
```

1.9.3 Accès à l'instance depuis le serveur même (toutes distributions)

Par défaut, l'instance n'est accessible que par l'utilisateur système **postgres**, qui n'a pas de mot de passe. Un détour par `sudo` est nécessaire :

```
$ sudo -iu postgres psql
psql (16.0)
Type "help" for help.
postgres=#
```

Ce qui suit permet la connexion directement depuis un utilisateur du système :

Pour des tests (pas en production !), il suffit de passer à `trust` le type de la connexion en local dans le `pg_hba.conf` :

```
local   all             postgres                                trust
```

La connexion en tant qu'utilisateur `postgres` (ou tout autre) n'est alors plus sécurisée :

```
dalibo:~$ psql -U postgres
psql (16.0)
Type "help" for help.
postgres=#
```

Une authentification par mot de passe est plus sécurisée :

- dans `pg_hba.conf`, paramétrer une authentification par mot de passe pour les accès depuis `localhost` (déjà en place sous Debian) :

```
# IPv4 local connections:
host    all             all             127.0.0.1/32            scram-sha-256
# IPv6 local connections:
host    all             all             ::1/128                 scram-sha-256
```

(Ne pas oublier de recharger la configuration en cas de modification.)

- ajouter un mot de passe à l'utilisateur `postgres` de l'instance :

```
dalibo:~$ sudo -iu postgres psql
psql (16.0)
Type "help" for help.
postgres=# \password
Enter new password for user "postgres":
Enter it again:
postgres=# quit

dalibo:~$ psql -h localhost -U postgres
Password for user postgres:
psql (16.0)
Type "help" for help.
postgres=#
```

- Pour se connecter sans taper le mot de passe à une instance, un fichier `.pgpass` dans le répertoire personnel doit contenir les informations sur cette connexion :

```
localhost:5432:*:postgres:motdepassetrèslong
```

Ce fichier doit être protégé des autres utilisateurs :

```
$ chmod 600 ~/.pgpass
```

- Pour n'avoir à taper que `psql`, on peut définir ces variables d'environnement dans la session voire dans `~/.bashrc` :

```
export PGUSER=postgres
export PGDATABASE=postgres
export PGHOST=localhost
```

Rappels :

- en cas de problème, consulter les traces (dans `/var/lib/pgsql/16/data/log` ou `/var/log/postgresql/`) ;
- toute modification de `pg_hba.conf` ou `postgresql.conf` impliquant de recharger la configuration peut être réalisée par une de ces trois méthodes en fonction du système :

```
root:~# systemctl reload postgresql-16

root:~# pg_ctlcluster 16 main reload

postgres:~$ psql -c 'SELECT pg_reload_conf()'
```

2/ Schéma et données

2.1 INTRODUCTION

Ce module est organisé en trois parties :

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

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

2.2 CONFIGURATION D'ORA2PG

Étapes de la configuration :

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

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

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

2.2.1 Structure du fichier

Structure

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

Chaque ligne non commentée doit commencer par l'une des clés de configuration. Il y en a environ 180 différentes.

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

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

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

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

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

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

2.2.2 Configuration locale

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

IMPORT

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

Par exemple :

```
IMPORT common.conf
```

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

ORACLE_HOME

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

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

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

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

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

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

DEBUG

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

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

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

LOGFILE

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

2.2.3 Connexion à Oracle

- `ORACLE_DSN`
  - `dbi:Oracle:host=serveur;sid=INSTANCE`
  - `dbi:Oracle:TNSNAME`
  - `dbi:Oracle://serveur:1521/service`
- `ORACLE_USER` (`system` par défaut)
- `ORACLE_PWD` (`manager` par défaut)
- `SCHEMA`
  - `NOM_SCHEMA` versus `SYSUSERS`
- `USER_GRANTS [0|1]`
  - l'utilisateur Oracle a-t-il les droits DBA ?

ORACLE_DSN

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

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

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

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

Il est possible aussi d'utiliser la notation *Easy Connect* ou simplement l'alias de connexion renseigné dans le fichier `tnsnames.ora` :

```
ORACLE_DSN      Oracle://serveur:1521/service # Easy connect
ORACLE_DSN      dbi:Oracle:XE             # Alias TNS
```

L'alias ci-dessus dispose de la définition suivante dans le fichier `$ORACLE_HOME/network/admin/tnsnames.ora` :

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

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

Pour MySQL un datasource typique sera de la forme :

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

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

ORACLE_USER et ORACLE_PWD

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

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

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

SCHEMA

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

Par exemple, pour exporter les objets du schéma `HR` de la base de données de démonstration, la valeur de la directive doit être :

```
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_010600 FLOWS_FILES MDDATA ORACLE_OCM SPATIAL_CSW_ADMIN_USR
SPATIAL_WFS_ADMIN_USR XS$NULL PERFSTAT SQLTXPLAIN DMSYS TSMSYS WKSYS APEX_040000
APEX_040200 DVSYS OJVMSYS GSMADMIN_INTERNAL APPQOSSYS DVSYS DVF AUDSYS
APEX_030200 MGMT_VIEW ODM ODM_MTR TRACESRV MTMSYS OWBSYS_AUDIT WEBSYS WK_PROXY
OSE$HTTP$ADMIN AURORA$JIS$UTILITY$ AURORA$ORB$UNAUTHENTICATED
DBMS_PRIVILEGE_CAPTURE CSMIG MGDSYS SDE DBSFWUSER
```

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`, il est possible d'ajouter une nouvelle ligne qui enrichira la liste des schémas à exclure :

```
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
```

2.3 EXPLORATION DE LA BASE DISTANTE

Cette étape d'exploration permet d'obtenir des informations sur la base Oracle :

- Liste des tables et colonnes
- Recherche de l'encodage de la base

2.3.1 Découverte de la base

Certaines informations sont disponibles immédiatement, sans plus de configuration :

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

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

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

```
ora2pg -t SHOW_SCHEMA
```

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

La commande :

```
ora2pg -t SHOW_TABLE
```

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

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

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

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

L'utilisation de l'action `SHOW_COLUMN` :

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

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

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

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

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

2.3.2 Gestion de l'encodage - 1

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

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

SHOW_ENCODING

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

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

```
Current encoding settings that will be used by Ora2Pg:
    Oracle NLS_LANG AMERICAN_AMERICA.AL32UTF8
```

```
    Oracle NLS_NCHAR AL32UTF8
    Oracle NLS_TIMESTAMP_FORMAT YYYY-MM-DD HH24:MI:SS.FF6
    Oracle NLS_DATE_FORMAT YYYY-MM-DD HH24:MI:SS
    PostgreSQL CLIENT_ENCODING UTF8
    Perl output encoding ''
Showing current Oracle encoding and possible PostgreSQL client encoding:
    Oracle NLS_LANG AMERICAN_AMERICA.AL32UTF8
    Oracle NLS_NCHAR AL32UTF8
    Oracle NLS_TIMESTAMP_FORMAT YYYY-MM-DD HH24:MI:SS.FF6
    Oracle NLS_DATE_FORMAT YYYY-MM-DD HH24:MI:SS
    PostgreSQL CLIENT_ENCODING UTF8
```

NLS_LANG et NLS_CHAR

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

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

```
$ sqlplus hr/secret@xe

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

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

17 ligne(s) sélectionnée(s).
```

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

```
SELECT * FROM nls_instance_parameters ;
```

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

```
SELECT * FROM nls_database_parameters;
```

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

2.3.3 Gestion de l'encodage - 2

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

CLIENT_ENCODING

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

Si le `NLS_LANG` a été modifié dans le fichier de configuration et 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[1], qui fait partie de la documentation officielle de PostgreSQL.

BINMODE

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

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

2.4 CONFIGURATION GÉNÉRIQUE

Le but du fichier de configuration générique est multiple :

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

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

2.4.1 Fichiers de sortie

Utilisation de fichiers de sortie dédiés

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

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

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

2.4.2 Ordres SQL additionnels

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

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

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

`COMPILE_SCHEMA` permet de demander à Oracle de vérifier à nouveau le code PL/SQL et de valider ce qui doit l'être. Par exemple, un trigger a pu être ajouté et faire appel à une fonction avant qu'elle ne soit créée, et sera dans ce cas marqué invalide par Oracle. L'activation de cette variable permet de forcer Oracle à revalider le code. Par défaut ce comportement n'est pas activé, le code valide seul sera exporté.

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

2.4.3 Comportement côté PostgreSQL

Utilisation d'un schéma sous PostgreSQL ?

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

Il faut ensuite se poser la question de savoir si l'on souhaite recréer le schéma ou l'utilisateur Oracle sous lequel seront créés tous les objets dans PostgreSQL. Si la réponse est oui, il faut activer la di-

rective `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.

2.4.4 Versions de PostgreSQL

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

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

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

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

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

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

Par défaut donc, tous ces paramètres sont activés.

2.4.5 Bases spatiales

La base contient des champs de type `SDO_GEOMETRY`.

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

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

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

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

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

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

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

Le système de référence spatial (SRID) utilisé va être la valeur retournée depuis la table des métadonnées spatiales 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 préfixés par le nom du schéma. Pour éviter cela, il est préférable de positionner le nom du schéma PostGIS dans la directive `POSTGIS_SCHEMA` et celui-ci sera ajouté au `search_path` lors de la création des objets.

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

2.4.6 Configuration liée aux LOB

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

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

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

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

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

Il est conseillé de laisser Ora2Pg gérer l'export des LOB en utilisant des pointeurs sur les enregistrements (*LOB Locator*) lui permettant de récupérer les données de ces champs en plusieurs fois. Cela évite la contrainte de recherche de la bonne valeur à attribuer à `LONGREADLEN`.

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

```
BLOB_LIMIT      500
```

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

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

2.5 MIGRATION DU SCHÉMA

Étapes :

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

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

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

2.5.1 Organisation de l'espace de travail

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

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

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

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

À minima, il est conseillé d'avoir :

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

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

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

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

Voici l'arborescence générée par Ora2Pg :

```
/opt/ora2pg/mydb_project/
├── config
│   └── ora2pg.conf
├── data
├── export_schema.sh
├── import_all.sh
├── reports
├── schema
│   ├── dblinks
│   ├── directories
│   ├── functions
│   ├── grants
│   ├── mviews
│   ├── packages
│   ├── partitions
│   ├── procedures
│   ├── sequences
│   ├── synonyms
│   ├── tables
│   ├── tablespaces
│   ├── triggers
│   ├── types
│   └── views
└── sources
    ├── functions
    ├── mviews
    ├── packages
    ├── partitions
    ├── procedures
    ├── triggers
    ├── types
    └── views
```

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

```
#!/bin/sh
#------------------------------------------------------------------------------
```

```
#
# Generated by Ora2Pg, the Oracle database Schema converter, version 23.2
#
#------------------------------------------------------------------------------
EXPORT_TYPE="SEQUENCE TABLE PACKAGE VIEW GRANT TRIGGER FUNCTION PROCEDURE
             TABLESPACE PARTITION TYPE MVIEW DBLINK SYNONYM DIRECTORY"
SOURCE_TYPE="PACKAGE VIEW TRIGGER FUNCTION PROCEDURE PARTITION TYPE MVIEW"
namespace="."
unit_cost=5

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 \
  --cost_unit_value  $unit_cost --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 exports 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.

2.5.2 Utilisation de la configuration générique

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

Lors de la création par Ora2Pg du répertoire de travail, le fichier de configuration générique est créé à partir du fichier `/etc/ora2pg/ora2pg.conf.dist` et enregistré dans le répertoire `mydb_project/config/`. Les modifications appliquées à ce fichier sont celles exposées 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 --init_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.

Type d'export

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`, à savoir :

- `TABLE` : Extrait l'ensemble des tables, avec leurs index et leurs contraintes d'intégrité.
- `VIEW` : Extrait les vues et leur définition uniquement.
- `GRANT` : Extrait les rôles compatibles avec PostgreSQL et réaffecte les permissions aux différents objets de la base.
- `SEQUENCE` : Extrait l'ensemble des séquences et leur dernière valeur connue.
- `TABLESPACE` : Extrait les noms de tablespaces pour le stockage des tables et des index.
- `TRIGGER` : Extrait la définition des triggers.
- `FUNCTION` : Extrait les fonctions.
- `PROCEDURE` : Extrait les procédures.
- `PACKAGE` : Extrait les paquets et leur définition (*package bodies*).
- `INSERT` : Extrait les données sous forme de requêtes `INSERT`.
- `COPY` : Extrait les données sous forme d'instruction `COPY`.
- `PARTITION` : Extrait les partitions de type `RANGE` et `LIST`, ainsi que les sous-partitions.
- `TYPE` : Extrait les types définis par l'utilisateur.
- `FDW` : Exporte les tables sous forme de tables externes avec `oracle_fdw`.
- `MVIEW` : Exporte les vues matérialisées.
- `QUERY` : Tente de convertir automatiquement les requêtes SQL présents dans les tables d'audit d'Oracle (`DBA_AUDIT_TRAIL`).
- `KETTLE` : Génére les fichiers modèles XML utilisés par Kettle, un ETL qui dispose d'une version communautaire et dont le nom moderne est Pentaho Data Integration (PDI). Cet export nécessite que les chaînes de connexions Oracle et PostgreSQL soient définies (DSN).
- `DBLINK` : Génére la définition d'un objet serveur reposant sur l'extension `oracle_fdw` pour émuler un DBLink Oracle.
- `SYNONYM` : Exporte les synonymes sous la forme de vues dans un autre schéma.
- `DIRECTORY` : Exporte la définition des *directories* Oracle en s'appuyant sur l'extension `external_file`.
- `LOAD` : Distribue une liste de requêtes à travers plusieurs connexions à l'instance PostgreSQL.
- `TEST` : Réalise une analyse des différences entre les bases Oracle et PostgreSQL à l'issue de la migration.
- `TEST_COUNT` : Réalise un décompte des lignes entre les bases Oracle et PostgreSQL à l'issue de la copie des données.
- `TEST_VIEW` : Réalise le décompte des lignes retournées par les vues en les bases Oracle et PostgreSQL.
- `TEST_DATA` : Vérifie une à une les valeurs retournées par les deux systèmes.
- `SEQUENCE_VALUES` : Exporte les instructions de mise à jour des séquences à leur dernière value connue.
- `SHOW_VERSION` : Affiche la version de la base Oracle.
- `SHOW_SCHEMA` : Affiche la liste des schémas disponibles depuis la base Oracle.
- `SHOW_TABLE` : Affiche la liste des tables disponibles.
- `SHOW_COLUMN` : Affiche la liste des colonnes de tables disponible ainsi que les transformations qui seront réalisées par Ora2Pg lors de la conversion automatique. Remonte un avertissement

si un mot réservé dans PostgreSQL est présent dans le nom des objets Oracle.
- `SHOW_ENCODING` : Affiche les valeurs `NLS_LANG` et `CLIENT_ENCODING` qu'Ora2Pg prévoit d'utiliser ainsi que l'encodage de la base Oracle avec les correspondances possibles avec une connexion client PostgreSQL.
- `SHOW_REPORT` : Retourne un rapport détaillé du contenu de la base Oracle.

L'auteur d'Ora2Pg a présenté les différents types de test lors de sa conférence « La validation de migration facilitée par Ora2Pg[2] », à la PostgreSQL Session 14 qui s'est tenue à Paris le 17 novembre 2021.

Répertoire de stockage des fichiers

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.

Nom du fichier de sortie

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

Conversion automatique du code

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

2.5.3 Export de la structure de la base

- Export des tables, contraintes et index

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

- Export des séquences

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

- Export des vues

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

- Préservation des tablespaces Oracle : `USE_TABLESPACE`

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

- `table.sql`

[2] https://www.pgsessions.com/assets/archives/pgs14_validation_migration_facilitee_avec_Ora2Pg.pdf

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

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

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

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

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

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

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

2.5.4 Modification de la structure des objets

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

REPLACE_TABLES et **REPLACE_COLS**

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

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

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

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

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

INDEXES_SUFFIX et **INDEXES_RENAMING**

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

```
INDEXES_SUFFIX      _idx
```

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

```
INDEXES_RENAMING    1
```

DATA_TYPE et **MODIFY_TYPE**

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

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

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

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

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

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

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

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

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

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

REPLACE_AS_BOOLEAN et **BOOLEAN_VALUES**

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

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

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

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

2.5.5 Export des objets globaux

- Les rôles et droits

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

- Les tablespaces

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

- Les types composites

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

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

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

2.5.6 Export des routines stockées

- Export des objets avec conversion de code

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

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

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

2.5.7 Export des sources PL/SQL

- Extraction du code brut d'Oracle

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

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

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

2.5.8 Export des partitions

- Partitions par `range`, `list` et `hash`
- Partitions et sous-partitions

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

- Paramètrage : `DISABLE_PARTITION` et `PG_SUPPORTS_PARTITION`

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

Toutes les lignes insérées dans la table partitionnée seront alors redirigées vers une des partitions en se basant sur la valeur de la clé de partitionnement. Les méthodes de partitionnement supportées par Ora2Pg sont le partitionnement par intervalles (`RANGE`), par liste (`LIST`) et par hachage (`HASH`).

Les partitions peuvent elles-mêmes être définies comme des tables partitionnées, en utilisant le sous-partitionnement. Les partitions peuvent avoir leurs propres index, contraintes et valeurs par défaut, différents de ceux des autres partitions.

Le paramètre `DISABLE_PARTITION` permet de ne pas reprendre le partitionnement d'un table alors que le paramètre `PG_SUPPORTS_PARTITION` désactive la conversion en partitionnement déclaratif au profit du partitionnement par héritage, soit l'ancien comportement avant PostgreSQL v10.

Voici deux exemples de conversion avec le partitionnement déclaratif réalisées avec Ora2Pg :

- 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')),
);
```

Deviendra :

```
CREATE TABLE sales_range
(
    salesman_id  integer,
    salesman_name varchar(30),
```

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

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

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

- Partition par liste

```
CREATE TABLE sales_list
(
    salesman_id  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)
);
```

Deviendra :

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

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

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

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

2.5.9 Export des vues matérialisées

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

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

Les vues matérialisées ne sont supportées qu'à partir de la version 9.3 pour PostgreSQL. Elles ne supportent pas toutes les fonctionnalités qu'offre Oracle : pas de mise à jour au fil de l'eau, pas de rafraîchissement incrémental à l'aide de journaux de vue matérialisée (`MATERIALIZED VIEW LOG` sous Oracle), pas de réécriture de requête (*query rewrite*).

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

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

```
REFRESH MATERIALIZED VIEW emp_data_mview;
```

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

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

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

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

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

Pour aller plus loin :

- Postgres 9.4 feature highlight - REFRESH CONCURRENTLY a materialized view[3]
- Conférence pgDay Paris 2019 : Don't forget materialized view[4] (vidéo[5])

[3] https://paquier.xyz/postgresql-2/postgres-9-4-feature-highlight-refresh-concurrently-a-materialized-view
[4] https://www.postgresql.eu/events/pgdayparis2019/schedule/session/2398-dont-forget-the-materialized-views
[5] https://www.youtube.com/watch?v=X8_ZY-XMM0E

2.5.10 Export des synonymes

PostgreSQL ne possède pas d'objet de type `SYNONYM`

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

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

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

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

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

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

Modification du search_path

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

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

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

Utilisation de vues

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

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

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

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

sera exporté par Ora2Pg de la façon suivante :

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

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

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

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

2.5.11 Export des tables externes

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

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

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

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

Voici un exemple de table externe sous Oracle :

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

CREATE TABLE ext_table (
id       NUMBER(6),
nom      VARCHAR2(20),
prenom   VARCHAR2(20),
activite CHAR(1))
ORGANIZATION EXTERNAL (
  TYPE oracle_loader
  DEFAULT DIRECTORY ext_directory
    ACCESS PARAMETERS (
    RECORDS DELIMITED BY NEWLINE
    FIELDS TERMINATED BY ','
    MISSING FIELD VALUES ARE NULL
```

```
    REJECT ROWS WITH ALL NULL FIELDS
    (id, nom, prenom, activite))
    LOCATION ('person.dat')
  )
PARALLEL
REJECT LIMIT 0
NOMONITORING;
```

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

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

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

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

2.5.12 Export des DATABASE LINK

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

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

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

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

```
CREATE PUBLIC DATABASE LINK remote_service_name CONNECT TO scott
IDENTIFIED BY tiger USING 'remote_db_name';
```

et s'utilisent ensuite de la façon suivante :

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

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

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

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

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

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

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

et le résultat de la commande ora2pg :

```
CREATE FOREIGN TABLE employees_fdw (...) SERVER remote_service_name
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_fdw;
```

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[6]

2.5.13 Export des BFILE et DIRECTORY - 1

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

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

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

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

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

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,...`

2.5.14 Export des BFILE et DIRECTORY - 2

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

Il existe aussi une extension PostgreSQL nommée external_file[7] 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.

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`.

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

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

2.5.15 Recherche Plein Texte

Oracle Index Texte

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

2.5.16 Recherche Plein Texte

PostgreSQL : *Full Text Search*/Recherche Plein Texte

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

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

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

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

Exemple :

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

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

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

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

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

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

Voici un exemple succinct de mise en place de FTS :

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

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

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

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

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

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

- Création de la fonction qui sera associée au trigger :

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

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

- Création du trigger :

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

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

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

- Création de l'index associé au vecteur :

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

L'index permet bien sûr une recherche plus rapide.

- Collecte des statistiques sur la table :

```
ANALYZE depeche ;
```

- Utilisation :

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

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

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

On peut trier par pertinence :

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

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

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

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

2.5.17 Préparation de l'import

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

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

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

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

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

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

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

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

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

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

2.5.18 Import du schéma

Création des objets du schéma :

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

La base étant créée, il ne reste plus qu'à charger les différents objets en commençant par les tables, puis les partitions, s'il y en a, les vues et pour finir les tablespaces pour déplacer les objets dans leurs 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).

2.5.19 Import différé

Chargement différé de certains objets :

- Séquences
- Contraintes
- Déclencheurs
- Index

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

2.5.20 Bilan de l'export/import

Bilan de l'export/import du schéma :

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

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

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

```
SET client_encoding TO autre_encodage;
```

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

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

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

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

2.5.21 Exemple d'erreurs

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

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

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

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

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

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

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

2.6 MIGRATION DES DONNÉES

Étapes :

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

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

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

2.6.1 Exporter les données

- Création des fichiers de données :

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

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

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

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

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

2.6.2 Cas des données CLOB/BLOB

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

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

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

Pour accélérer l'échappement des données `bytea`, il faut activer l'utilisation du parallélisme. Cela permet en général d'aller deux à trois fois plus vite. Pour cela, il faut utiliser l'option `-j` en ligne de

commande ou la variable `JOBS` du fichier de configuration. La valeur est le nombre de cœurs CPU que l'on veut utiliser.

Lorsque les options de parallélisation sont activées, il est important de s'assurer que la valeur de `DATA_LIMIT` corresponde à la vitesse moyenne maximal d'export d'un simple 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 20 000, alors `DATA_LIMIT` devra avoir cette valeur. Ceci vous permettra d'être sûr de tirer le meilleur parti de la parallélisation. Une valeur excessive par contre peut conduire à des dépassements de ressources, une valeur trop faible forcera Ora2Pg à créer des processus inutilement.

2.6.3 Cas des données spatiales

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

CONVERT_SRID

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

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

DEFAULT_SRID

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

GEOMETRY_EXTRACT_TYPE

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

- WKT

- WKB
- INTERNAL

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

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

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

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

La valeur par défaut est `INTERNAL`.

2.6.4 Import des données

- Import des fichiers de données :

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

- Chargement direct dans PostgreSQL lors de l'export

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

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

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

2.6.5 Restauration des contraintes

Restauration des contraintes, triggers, séquences et index

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

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

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

2.6.6 Restauration parallélisée des contraintes

Action :

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

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

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

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

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

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

2.6.7 Problèmes d'import des données

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

Si le type d'export `INSERT` a été choisi, il arrive très souvent que cela conduise à des erreurs de caractères invalides lors de l'insertion, car le caractère `backslash` n'est pas échappé si `STANDARD_CONFORMING_STRING` est activé, ce qui correspond au standard SQL. 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, ce qu'Ora2Pg fait par défaut.

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 pouvez donc ajuster le jeu 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, converti par défaut vers le type indiqué par la variable `DEFAULT_NUMERIC`, c'est-à-dire `bigint`. Comme le type `NUMBER` 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. 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*.

2.6.8 Performances de l'import des données

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

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

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

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

COPY ou INSERT

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

PG_DSN

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

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

- le connecteur DBI à utiliser ;
- le nom de la base de données PostgreSQL, `dbname=` ;
- le nom du serveur PostgreSQL à utiliser, `host=` ;

- et le port sur lequel le serveur PostgreSQL écoute, `port=`.

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

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

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

PG_USER

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

Exemple, pour l'utilisateur `prod` :

```
PG_USER prod
```

PG_PWD

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

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

```
PG_PWD secret
```

DATA_LIMIT

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

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

```
DATA_LIMIT 100000
```

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

```
DATA_LIMIT 1000
```

2.6.9 Utiliser le parallélisme

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

Ora2Pg de base n'utilise qu'un seul CPU ou cœur pour le chargement des données. Ceci est très limitant en termes 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 veut 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 supérieure à 1, la variable `FILE_PER_TABLE` est automatiquement activée.

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

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

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

```
DATA_LIMIT        60000
```

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

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

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

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

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

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

2.6.10 Limitation des données exportées

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

ALLOW

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

On peut cependant limiter l'export à certains 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 quel que 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 ont une clause `INSTEAD OF` (liés à des vues) :

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

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

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

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

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

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

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

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

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

ou bien

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

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

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

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

WHERE

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

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

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

- `WHERE  1=1`

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

Exemple:

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

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

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

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

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

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

Cela donne :

```
COPY "departments" ("department_id","department_name",[...])
FROM stdin;
10      Administration  200     1700
20      Marketing       201     1800
30      Purchasing      114     1700
40      Human Resources 203     2400
50      Shipping        121     1500
60      IT      103     1400
70      Public Relations        204     2700
80      Sales   145     2500
90      Executive       100     1700
\.
```

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

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

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

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

Ce qui donne comme résultat:

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

REPLACE_QUERY

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

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

Quelques fois cela n'est pas suffisant, par exemple si l'on souhaite faire une jointure sur une table d'identifiants à migrer ou tout autre requête plus complexe que ce que ne peut produire Ora2Pg. Dans ce cas il est possible de forcer Ora2Pg à 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`.

2.6.11 Après la migration

- Supervision
- Optimisation
- `VACUUM FREEZE` préventif

Quand votre migration est terminée, et la nouvelle base en production, le travail n'est pas terminé.

Supervision & optimisation :

Il faut superviser cette nouvelle instance pour vérifier que les performances sont celles prévues. Il faut s'attendre à devoir optimiser quelques requêtes inattendues. Les outils pour relever les requêtes consommatrices sont classiques : pgBadger, pg_stat_statements…

Le gel massif des lignes :

Il existe un piège peu connu lié à toute migration logique d'une grande base : le gel massif des lignes. Pour des raisons techniques de recyclage des numéros de transaction, PostgreSQL doit « geler » les lignes anciennes et jamais modifiées, ce qui implique de réécrire le bloc. Or, toutes les lignes insérées

par une migration ont le même « âge ». Si elles ne sont pas modifiées, ces lignes risquent d'être toutes gelées et réécrites en même temps : ce peut être très brutal en terme de saturation disque, de journaux générés, etc. si la base est grosse. Le délai avant le déclenchement du gel automatique dépend de la consommation des numéros de transaction sur l'instance migrée, et varie de quelques semaines à des années.

Des ordres `VACUUM FREEZE` sur les plus grosses tables à des moments calmes permettent d'étaler ces écritures. Si ces ordre sont interrompus, l'essentiel de qu'il a pu geler ne sera plus à re-geler plus tard.

Pour les détails, voir https://dali.bo/m4_html#le-wraparound-1 et https://dali.bo/m5_html#paramétrage-du-freeze-1.

2.7 CONCLUSION

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

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

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

2.7.1 Pour aller plus loin

- Documentation officielle[8]
- Conférence sur Data2Pg[9]
- Wiki PostgreSQL[10]

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

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

- Migration vers PostgreSQL : mener de gros volumes de données à bon port[11]

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

2.7.2 Questions

N'hésitez pas, c'est le moment !

[11] https://www.youtube.com/watch?v=CR67iLHTocY

2.8 QUIZ

https://dali.bo/n2_quiz

3/ Requêtes SQL

3.1 INTRODUCTION

Ce module est organisé en quatre parties :

- Compatibilité avec Oracle
- Types de données
- Différences de syntaxes
- Transactions

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.

3.2 COMPATIBILITÉ AVEC ORACLE

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

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 routines stockées sont différents, y compris ceux qui sont 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. Cette partie dresse une liste non exhaustive des différences majeures entre Oracle et PosgreSQL pour mieux appréhender les problèmes ou des incompréhensions lors d'une migration.

3.2.1 Points communs

PostgreSQL et Oracle :

- Ont le même langage d'accès aux données (SQL)
  - mais des « variantes » différentes (extensions au standard)
- 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, tous les moteurs se permettent des écarts par

rapport à la norme, parfois pour gagner en performances, mais surtout pour faciliter la vie des développeurs. Puis ces écarts persistent par la nécessaire compatibilité descendante.

Beaucoup de développeurs utilisent donc ces écarts à la norme, souvent 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 de la norme réalisées 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.

PostgreSQL et Oracle partagent aussi certains concepts, comme les transactions et les *savepoints*, MVCC (même si l'implémentation diffère) et la gestion des verrous. Cela permet de conserver les logiques applicative et algorithmique, au moins jusqu'à une certaine mesure.

3.2.2 Différences de schéma - 1

- Le schéma sous Oracle : `USER.OBJECT`
  - sous PostgreSQL, véritable espace de nommage
- 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 colonnes générées sont disponibles depuis PostgreSQL 12
  - uniquement pour les colonnes stockées
  - utilisation de vues pour simuler les colonnes virtuelles

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.

Création de table

La définition des tables est quasiment identique pour les deux SGBD à la différence près que PostgreSQL ne supporte pas les tables temporaires globales, 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.

L'extension `pgtt` de Gilles Darold permet d'émuler le comportement de tables temporaires globales : https://github.com/darold/pgtt

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);
```

Colonnes virtuelles

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;
```

Depuis PostgreSQL 12, il est possible de créer une colonne générée, dite stockée, pour permettre l'utilisation d'un index sur cette colonne. La colonne n'est pas vraiment « virtuelle ».

Voici un exemple de la syntaxe dans PostgreSQL :

```
CREATE TABLE employees (
  id           bigint,
  first_name   varchar(10),
  salary       double precision,
  commission   integer,
  salary2      double precision generated always as
               (ROUND((salary*(1+commission/100))::numeric,2)) STORED
);
```

En savoir plus : Colonnes générées[1]

3.2.3 Différences de schéma - 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 rare exemple où PostgreSQL s'écarte du standard SQL.

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.

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
```

[1] https://docs.postgresql.fr/current/ddl-generated-columns.html

```
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)
```

3.2.4 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'extensions (`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. Quoi qu'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.

3.2.5 Autres différences anecdotiques

- `HAVING` et `GROUP BY`
  - Oracle permet `GROUP BY` **après** `HAVING`
  - PostgreSQL impose `GROUP BY` **avant** `HAVING`
- Table `DUAL` n'est pas nécessaire
- Conversions implicites de et vers un type `text`
  - supporté par Oracle
  - plus supporté par PostgreSQL depuis la version 8.3
  - convertir explicitement :

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

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;
```

De nombreuses requêtes SQL avec Oracle utilisent la pseudo-table `DUAL` pour manipuler des valeurs issues de fonctions ou de variables, sans besoin de les extraire d'une table particulière. Or avec Oracle, les clauses `SELECT` et `FROM` sont inséparables.

Avec PostgreSQL, les syntaxes suivantes sont correctes :

```
SELECT fonction();
```

```
SELECT current_timestamp;
```

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 :

```
CREATE TABLE depts ( numero CHAR(2), nom VARCHAR(25) );

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 faire fonctionner cette requête, il faut préciser explicitement la conversion à réaliser :

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

Avec Oracle, ce type de conversion est implicite.

3.3 TYPES DE DONNÉES

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

3.3.1 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.

3.3.2 Différences sur les types chaînes

- Pas de `varchar2` dans PostgreSQL
  - mais `varchar`
  - `varchar (n)` : taille en nombre de caractères
- Le type `text` équivalent à `varchar` sans taille (1 Go maximum)
- Attention, sous Oracle, `''` équivaut à `NULL`
  - sous PostgreSQL, `''` et `NULL` sont distincts
- Un seul encodage par base
- Collationnement par colonne

Au niveau de PostgreSQL, il existe trois types de données pour les chaînes de caractères :

- `char` (alias de `character`) ;
- `varchar` (alias de `character varying`) ;
- `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 théorique est de 1 Go. Suivant l'encodage, le nombre de caractères intégrables dans la colonne diffère.

En pratique, il n'y a pas de différence à l'utilisation, en vitesse ou en taille, entre ces différents types de chaînes. Noter que la taille de chaîne indiquée (par exemple dans `varchar(5)`) est bien exprimée en caractères (même s'il faut plusieurs octets pour stocker chacun).

Une grosse différence entre Oracle et PostgreSQL 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 routines 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. Le collationnement se gère ensuite colonne par colonne et peut être modifié au sein d'une requête (au niveau d'un `ORDER BY` ou d'un `CREATE INDEX`).

3.3.3 Différences sur le type booléen

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

Comme Oracle ne dispose pas d'un type booléen, les développeurs (ou leurs ORM) l'émulent fréquemment avec un entier qu'ils mettront à 0 pour `FALSE` et à 1 pour `TRUE` (alternativement, on rencontre aussi des chaînes avec `Y` ou `N`). Un système de migration ne saura pas détecter 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, un ORM, cherchera un booléen parce que le code applicatif indiquera un booléen avant comme après la migration. Cela provoquera une erreur sur PostgreSQL, comme le montre l'exemple suivant :

```
dev2=# CREATE TABLE t1 (c1 int);
CREATE TABLE
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 t2 (c1 boolean);
CREATE TABLE
dev2=# INSERT INTO t2 VALUES (true);
INSERT 0 1
dev2=# INSERT INTO t2 VALUES ('f');
INSERT 0 1
dev2=# SELECT * FROM t2;
 c1
 t
 f
(2 rows)
```

3.3.4 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 routines 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.

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.

La facilité d'insertion et de lecture d'un champ binaire dépend du client et du langage, et peut nécessiter encodage et décodage. À part cela un bytea est un champ comme un autre.

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.

3.3.5 Différences sur les types dates

- PostgreSQL :
  - `date` : jour
  - `time` : heure seule
  - `timestamp` : date + heure (alias `timestamptz`)
- Fuseaux horaires
  - `YYYY-MM-DD HH24:MI:SS.mmmmmmm+TZ` (conforme SQL)
- Type `interval` sous PostgreSQL

Oracle a tendance à mélanger un peu tous les types dates. Ce n'est pas le cas au niveau de PostgreSQL. Pour lui, une colonne de type `date` 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 le type `timestamp with time zone` (alias `timestamptz`). Le stockage est fait en UTC, mais la restitution dépend du fuseau horaire indiqué par le client.

Bien que le type `timestamp without time zone` soit aussi disponible (et malheureusement alias de `timestamp`), il est chaudement conseillé de ne travailler qu'avec le type `timestamptz` pour faciliter

les conversions de fuseaux horaires. Il n'y a même pas de pénalité en taille du champ.

La conversion se fait automatiquement à l'affichage dans le fuseau horaire courant (selon le paramètre `timezone` dans la session). Par exemple :

```
postgres=# SHOW timezone;
   TimeZone
--------------
 Europe/Paris
(1 ligne)

postgres=# CREATE TABLE moments (t timestamptz) ;
CREATE TABLE

postgres=# SELECT now() ;
              now
-------------------------------
 2021-05-10 18:41:25.497356+02
(1 ligne)

postgres=# INSERT INTO moments SELECT now() ;
INSERT 0 1

postgres=# SET timezone TO 'America/New_York' ;
SET

postgres=# SELECT * FROM moments ;
               t
-------------------------------
 2021-05-10 12:41:21.914092-04
(1 ligne)

postgres=# SET timezone TO 'Asia/Katmandu' ;
SET

postgres=# SELECT * FROM moments ;
                t
---------------------------------
 2021-05-10 22:26:21.914092+05:45
(1 ligne)
```

PostgreSQL fournit un type `interval` très pratique pour les calculs temporels :

```
postgres=# SELECT t + interval '1h' FROM moments ;
             ?column?
---------------------------------
 2021-05-10 23:26:21.914092+05:45
```

3.3.6 Différences sur les fonctions temporelles

- `SYSDATE`
  - retourne la date et l'heure courante, sans *timezone*
  - équivalent avec PostgreSQL : `localtimestamp`
- PostgreSQL ne propose pas de fonctions `add_months`, etc.
- `NLS_DATE_FORMAT` (`TO_CHAR` et `TO_DATE`)
  - configuration `DateStyle`

Le mot clé `SYSDATE` est très fréquent pour générer une valeur horodatée avec Oracle. Son équivalent direct avec PostgreSQL est `localtimestamp`, qui correspond à la date et l'heure de la *timezone* du **client** mais sans que cette *timezone* n'y soit renseignée. Il s'agit d'une de type `timestamp without timezone`.

PostgreSQL implémente d'autres fonctions pour générer les valeurs pour chaque type de date à l'horloge actuelle. Il existe par ailleurs la fonction `now()` qui remplace fréquemment la valeur `current_timestamp`.

```
select pg_typeof(localtimestamp) AS localtimestamp,
       pg_typeof(localtime) AS localtime,
       pg_typeof(current_timestamp) AS current_timestamp,
       pg_typeof(current_time) AS current_time,
       pg_typeof(current_date) AS current_date \gx
```

```
-[ RECORD 1 ]-----+----------------------------
localtimestamp    | timestamp without time zone
localtime         | time without time zone
current_timestamp | timestamp with time zone
current_time      | time with time zone
current_date      | date
```

PostgreSQL ne dispose pas de fonctions pour l'ajout ou la soustraction d'une date à une autre. Le type `interval` y remédie et permet d'aller aussi loin, voire plus, que ne le propose Oracle.

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

SELECT current_date + interval '1 days' * 3;

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

Exemples :

> 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);
```

> Quel est le premier jour de 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);
```

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`).

3.3.7 Différences sur les types spécialisés

PostgreSQL fournit de nombreux types de données spécialisés :

- timestamps et intervalles, avec opérations arithmétiques
- Adressage IP (CIDR), avec opérateurs de masquage
- Grande extensibilité des types : il est très facile d'en rajouter un nouveau
  - `PERIOD`
  - `ip4r`…
- Nombreuses extensions
  - PostGIS

L'un des gros avantages de PostgreSQL est son extensibilité. Le mécanisme des extensions permet de rajouter des types spécialisés dans un domaine. Le plus bel exemple est PostGIS[2], une extension dédiée aux objets géographiques.

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] https://postgis.net/

3.4 DIFFÉRENCES DE SYNTAXES

- Expressions conditionnelles `DECODE` et `NVL`
- Concaténation de chaînes avec `NULL`
- Pseudo-colonne `ROWNUM`
- Jointures
- Opérateurs ensemblistes
- Hiérarchies

Cette partie s'attardera sur les différences notables entre Oracle et PostgreSQL dans la rédaction de requêtes complexes avec le langage SQL.

3.4.1 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.

3.4.2 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;
```

3.4.3 Concaténation avec NULL

Changement de comportement de l'opérateur `||`

- Rappel : pour Oracle, `NULL` équivaut à une chaîne vide `''`
- Pour PostgreSQL : `SELECT 'chaîne' || NULL` équivaut à `NULL`
- Réécritures possibles :

```
SELECT 'chaîne' || COALESCE(null, '');

SELECT CONCAT('chaîne', null);
```

Du fait de la spécificité du type `VARCHAR2` d'Oracle, la concaténation d'une valeur `NULL` dans une chaîne de caractères ne pose pas de problème, contrairement à PostgreSQL. Le standard SQL définit en effet que pour la plupart des fonctions, un paramètre `NULL` entraîne un résultat `NULL` (on parle de fonctions *STRICT* dans PostgreSQL).

Dans le cas présent, une valeur `NULL` dans une opération de concaténation sera propagée au résultat : le résultat sera une chaîne `NULL` et non la chaîne de caractères attendue.

```
-- avec Oracle, une valeur NULL ne pose pas de problème
SELECT 'nom de l''employé: ' || last_name FROM employees;

-- avec PostgreSQL, la même opération retourne NULL, il faut forcer
-- le remplacement d'une valeur nulle par un espace vide avec COALESCE
SELECT $$nom de l'employé: $$ || COALESCE(last_name, '') FROM employees;
```

Une autre solution consiste à employer la méthode `CONCAT` pour s'affranchir de l'opérateur `||` et de la fonction `COALESCE` :

```
-- avec PostgreSQL, le CONCAT retourne la valeur attendue
SELECT CONCAT($$nom de l'employé: $$, last_name) FROM employees;
```

3.4.4 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.

3.4.5 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.

3.4.6 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;
```

3.4.7 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)
```

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;
```

3.4.8 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 :

```
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;
```

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

3.4.9 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 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(+);
```

Deviendra :

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

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

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

et nécessite d'être réécrite de la manière suivante :

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

La norme ANSI apporte la syntaxe `FULL OUTER JOIN` pour renvoyer toutes les lignes jointes entre deux tables, ainsi que les lignes sans correspondances à gauche comme à droite.

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 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(+)
UNION ALL
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
   AND a.id_auteur IS NULL;
```

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

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

3.4.10 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;
```

3.4.11 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;
```

3.4.12 Hiérarchies

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

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.

3.4.13 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.

3.4.14 WITH RECURSIVE

```
WITH RECURSIVE hierarchie AS (
condition de départ
UNION ALL
clause de récursion
)
SELECT * FROM hierarchie
```

3.4.15 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;
```

3.4.16 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 concaténer : `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
```

3.4.17 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;
```

3.4.18 Common Table Expressions

- Syntaxe quasiment identique
- Attention à la récursion
  - `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[3].

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

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

3.5 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)
```

Dans PostgreSQL, l'*autocommit* est un paramétrage du client. Il est possible de le désactiver dans `psql` avec le paramètrage `\set AUTOCOMMIT off`. Le `BEGIN` deviendra automatique et implicite,

et il faudra entrer `COMMIT` ou `ROLLBACK` pour terminer la transaction et en ouvrir une nouvelle automatiquement.

De même, le pilote JDBC de la communauté PostgreSQL[4] permet de désactiver l'*autocommit*, et rajoutera silencieusement les `BEGIN`.

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)
```

[4] https://jdbc.postgresql.org/

3.5.1 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[5], est appelée *Serializable Snapshot Isolation* et corrige les défauts des implémentations[6] 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 :

[5] https://people.eecs.berkeley.edu/~kubitron/courses/cs262a-F13/handouts/papers/p729-cahill.pdf
[6] https://docs.postgresql.fr/current/transaction-iso.html#MVCC-SERIALIZABILITY

```
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[7].

3.5.2 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;
```

3.5.3 Verrous explicites

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

[7] https://docs.postgresql.fr/current/transaction-iso.html

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.

3.6 CONCLUSION

- Oracle et PostgreSQL répondent à la même norme ISO SQL
  - … mais la conversion des requêtes SQL sera nécessaire
  - le typage des données est bien plus fourni avec PostgreSQL
- Pour détecter les erreurs de syntaxe :
  - faire fonctionner l'application
  - repérer tous les ordres SQL en erreur (traces applicatives)
  - les corriger
- Attention aux mots réservés

Le langage SQL est commun entre Oracle et PostgreSQL. La norme SQL veille à ce qu'aucun éditeur logiciel ne propose un langage propriétaire. Cependant, le respect strict de cette norme est bien plus important pour PostgreSQL, alors que de son côté, Oracle propose des mots clés spécifiques tels que `NVL`, `ROWNUM` ou `CONNECT BY`, pour ne citer qu'eux.

Le plus grand soin doit être apporté dans la conversion des types lors d'une migration. Alors qu'un type temporel sera stocké en `date` sur Oracle, il sera décliné en plusieurs sous-ensembles définis par la norme avec PostgreSQL. Les types `date`, `time` et `timestamp` sont bien distincts et auront leur propre comportement vis-à-vis des opérateurs arithmétiques ou de comparaison.

Les types numériques sont plus nombreux avec PostgreSQL, dans le but de gérer plus finement le stockage de chaque valeur (2, 4 ou 8 octets selon les déclinaisons). Il peut y avoir des surprises lors d'une migration où une colonne de type `NUMBER` sera convertie en `numeric` dans PostgreSQL alors que les valeurs devraient être traitées en `integer` dans la logique applicative.

Le portage des requêtes SQL vers PostgreSQL est donc indispensable pour garantir la meilleure utilisation des fonctionnalités de ce moteur. Bien qu'il soit toujours intéressant de faire une première passe dans le code source de votre application pour corriger les cas les plus flagrants, il est nécessaire ensuite de l'exécuter sur PostgreSQL et de vérifier dans les journaux applicatifs les messages d'erreurs qui surviennent.

Un outil comme pgBadger[8] permet de récupérer les erreurs, leur fréquence et les requêtes qui ont causé les erreurs.

Pour récupérer la liste des mots réservés :

```
SELECT * FROM pg_get_keywords();
```

[8] https://pgbadger.darold.net/

3.6.1 Questions

N’hésitez pas, c’est le moment !

3.7 QUIZ

https://dali.bo/n3_quiz

4/ Procédures stockées

4.1 INTRODUCTION

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

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

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

4.1.1 Sommaire

Ce module est organisé en cinq parties :

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

C'est la partie la plus importante en termes 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[1]. 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)`

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

- `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()`
- `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[2] 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)
- plpgsql_check
- SQLMaestro

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

- `pldebugger` (anciennement edb-debugger) peut être téléchargé sur github[3] ;
- `plpgsql_check` de Pavel Stehule peut être téléchargé ici[4].

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

Il existe aussi SQLMaestro[5], un outil propriétaire qui permet l'exécution pas à pas du code PL/pgSQL.

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

4.3 DIFFÉRENCES DANS LE CODE

- Généralité
- Triggers
- Routines
- Packages

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

4.3.1 Généralité - 1

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

Les séquences

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

Transactions autonomes

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

Article Dalibo : Support des transactions autonomes dans PostgreSQL[6]

Différences de syntaxe

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

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

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`.

4.3.2 Généralité - 2

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

Boucle inversée

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

Sous Oracle, on écrit :

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

et avec PostgreSQL, on écrira :

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

Langage d'une fonction

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

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

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

CONNECT BY

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

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

Voici une requête `CONNECT BY` Oracle :

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

et voici sa traduction pour PostgreSQL :

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

Les curseurs

Au niveau des curseurs, leurs références sont de type `REFCURSOR` au lieu de `REF CURSOR`. Par exemple la déclaration d'une référence sur un curseur se fait de la façon suivante sous Oracle :

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

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

```
return_cur REFCURSOR;
```

Le type retourné lors de la manipulation des curseurs est un enregistrement `RECORD` et non pas `nom_curseur%ROWTYPE` sous Oracle. Avec PostgreSQL, il est possible à la lecture du curseur de placer cet enregistrement dans une cible qui peut être une variable ligne, une variable `record` ou une liste de variables simples séparées par des virgules.

BULK COLLECT

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

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

peut être traduit sous PostgreSQL de la façon suivante :

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

Chaines vides et NULL

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

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

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

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

4.3.3 Triggers

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

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

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

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

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

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

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

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

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

Sous Oracle, nous aurions cela :

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

4.3.4 Routines

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

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

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

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

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

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

CALL hello_world();
-- WARNING:  hello world

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

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

```
CREATE FUNCTION getRows(text) RETURNS SETOF RECORD
AS $$
DECLARE
    r RECORD;
```

```
BEGIN
    FOR r IN EXECUTE 'select * from ' || $1 LOOP
        RETURN NEXT r;
    END LOOP;
    RETURN;
END
$$
LANGUAGE 'plpgsql';
```

4.3.5 Packages

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

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

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

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

De même, la notion de variable globale n'existe pas sous PostgreSQL. Pour pouvoir émuler le comportement des variables globales, on peut utiliser les paramètres de configuration définis par l'utilisateur à renseigner dans le fichier de configuration `postgresql.conf` ou à l'aide des commandes `ALTER DATABASE` et `ALTER ROLE`.

Par exemple :

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

Il est possible de les initialiser sans déclaration préalable dans le fichier de configuration. Par exemple, pour créer une variable globale nommée `id_region`, il suffit d'utiliser la commande `SET` :

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

ou la fonction `set_config()` :

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

et `current_setting()` pour utiliser sa valeur :

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

Enfin, le choix de porter les variables globales dans une table de paramètres est tout aussi judicieux. Certains langages, comme PL/Perl par exemple, disposent quant à eux, de variables globales.

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.

4.4 CONVERSION AUTOMATIQUE DU CODE

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

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

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

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

4.4.1 Conversions globales

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

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

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

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

```
CREATE TRIGGER trigger_name
    BEFORE
```

```
    DELETE OR INSERT OR UPDATE
    ON table_name
       -- pl/sql block
```

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

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

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

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

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

deviendra :

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

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

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

on obtient :

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

```
$body$
LANGUAGE PLPGSQL;
```

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

Par exemple :

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

sera converti de la sorte dans PostgreSQL :

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

4.4.2 Correspondance des fonctions - 1

Les noms diffèrent :

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

Renommage des fonctions par Ora2Pg

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

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, ' ')`

Remplacement des paramètres par Ora2Pg

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

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

- `TO_NUMBER(num)` et `to_number(num, '999999999999999999D999999999999999999')` 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)
```

Réécriture complète des fonctions et paramètres par Ora2Pg

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

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

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 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'::in`
- `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 fonctions de trigger

4.4.7 Réécriture de parties de code - 2

- Supprime le caractère `:` devant les noms de variable Oracle
- Convertit 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`
- Remplace les appels à `MINUS` par `EXCEPT`

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

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

Avec PostgreSQL, la notation équivalente 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 réécrits en `LIMIT N`
- `ROWNUM` = `N` est réécrit en `LIMIT 1 OFFSET N`
- `ROWNUM` > or >= `N` sont réécrits 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
```

4.4.9 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;`

4.4.10 Réécriture de parties de code - 5

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

Empty string vs NULL

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

```
'' = NULL
```

Dans PostgreSQL et dans le SQL standard :

```
'' <> NULL
```

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

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

Si la directive `NULL_EQUAL_EMPTY` est activée, Ora2Pg remplace toutes les conditions avec un test sur `NULL` par un 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 sûr d'avoir 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.

4.4.11 Réécriture de parties de code - 6

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

4.4.12 Réécriture de parties de code - 7

- Remplacement des `REGEX_LIKE( string, pattern )` en syntaxe avec l'opérateur PostgreSQL de recherche regex `string ~ pattern`.
- Remplacement des appels aux variables d'environnement `SYS_CONTECT('USERENV', ...)` en équivalent PostgreSQL.
- Remplacement des fonctions spatiales `SDO_GEOM.*` en appels aux fonctions 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 d'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 transactions autonomes
- Import des fonctions et paquets de fonctions
- Absence de fonctions ou paquets

C'est la partie la plus importante en termes 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 suffixe `_atx`
- La fonction de substitution prend le nom originel de la fonction
- La fonction de substitution appelle la fonction `_atx` au travers d'un dblink
- Utilisation possible de l'extension `pg_background` à partir de PostgreSQL 9.5 et Ora2Pg 17.5

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

Code Oracle :

```
CREATE PROCEDURE LOG_ACTION (username VARCHAR2, msg VARCHAR2)
IS
  PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
  INSERT INTO table_tracking VALUES (username, msg);
  COMMIT;
END log_action;
```

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

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

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

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

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

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

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

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

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

4.5.2 Import des procédures et paquets avec Ora2Pg

Chargement des fonctions et procédures :

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

Chargement des paquets de procédures stockées :

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

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

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

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

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

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

```
\set ON_ERROR_STOP ON
```

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

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

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 PL/pgSQL ait été importé sans erreurs.

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

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

4.6.1 Ora2Pg : tests intégrés

Deux actions permettent de tester à minima :

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

Dénombrement des objets migrés

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

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

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

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

- les index par table ;
- les contraintes d'unicité par table ;
- les contraintes check par table ;
- les contraintes NOT NULL par table ;
- les clés primaires par table ;
- les colonnes avec valeurs par défaut par table ;
- les clés étrangères par table ;
- les triggers par table ;
- les partitions par table partitionnée ;
- les tables dans la base ;
- les triggers dans la base ;
- les vues dans la base ;
- les vues matérialisées dans la base ;
- les séquences dans la base ;
- les types utilisateurs dans la base ;
- les tables distantes (FDW) dans la base.

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

```
[TEST INDEXES COUNT]
ORACLEDB:COUNTRIES:1
POSTGRES:countries:1
ORACLEDB:C50_LEX_CARTES:1
POSTGRES:c50_lex_cartes:1
ORACLEDB:SUPPLIER:1
POSTGRES:supplier:1
ORACLEDB:DEPARTMENTS:2
POSTGRES:departments:2
ORACLEDB:JOB_HISTORY:4
POSTGRES:job_history:4
ORACLEDB:REGIONS:1
POSTGRES:regions:1
ORACLEDB:MYTABLE:1
POSTGRES:mytable:1
ORACLEDB:LOCATIONS:4
POSTGRES:locations:4
ORACLEDB:EMPLOYEES:6
POSTGRES:employees:6
ORACLEDB:JOBS:1
POSTGRES:jobs:1
```

```
[ERRORS INDEXES COUNT]
OK, Oracle and PostgreSQL have the same number of indexes.

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

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

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

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

[TEST NOT NULL CONSTRAINTS COUNT]
ORACLEDB:TIME_TZ2:0
POSTGRES:time_tz2:0
ORACLEDB:COUNTRIES:1
POSTGRES:countries:1
ORACLEDB:C50_LEX_CARTES:1
POSTGRES:c50_lex_cartes:1
ORACLEDB:SUPPLIER:2
POSTGRES:supplier:2
ORACLEDB:DEPARTMENTS:2
POSTGRES:departments:2
ORACLEDB:MYTABLE:1
POSTGRES:mytable:1
ORACLEDB:EMPLOYEES:5
POSTGRES:employees:5
ORACLEDB:JOBS:2
POSTGRES:jobs:2
ORACLEDB:VAL_RESULTS:0
POSTGRES:val_results:0
ORACLEDB:JOB_HISTORY:4
POSTGRES:job_history:4
ORACLEDB:TESTA:0
POSTGRES:testa:0
ORACLEDB:REGIONS:1
POSTGRES:regions:1
ORACLEDB:TEST_TZ:0
POSTGRES:test_tz:0
ORACLEDB:MESURE:1
POSTGRES:mesure:1
ORACLEDB:FICHIER_DONNEE:1
POSTGRES:fichier_donnee:1
ORACLEDB:TEST_NUM:0
POSTGRES:test_num:0
ORACLEDB:LOCATIONS:2
POSTGRES:locations:2
[ERRORS NOT NULL CONSTRAINTS COUNT]
OK, Oracle and PostgreSQL have the same number of null constraints.

[TEST COLUMN DEFAULT VALUE COUNT]
```

```
ORACLEDB:TIME_TZ2:0
POSTGRES:time_tz2:0
ORACLEDB:COUNTRIES:0
POSTGRES:countries:0
ORACLEDB:C50_LEX_CARTES:0
POSTGRES:c50_lex_cartes:0
ORACLEDB:SUPPLIER:0
POSTGRES:supplier:0
ORACLEDB:DEPARTMENTS:0
POSTGRES:departments:0
ORACLEDB:MYTABLE:0
POSTGRES:mytable:0
ORACLEDB:EMPLOYEES:0
POSTGRES:employees:0
ORACLEDB:JOBS:0
POSTGRES:jobs:0
ORACLEDB:VAL_RESULTS:0
POSTGRES:val_results:0
ORACLEDB:JOB_HISTORY:0
POSTGRES:job_history:0
ORACLEDB:TESTA:0
POSTGRES:testa:0
ORACLEDB:REGIONS:0
POSTGRES:regions:0
ORACLEDB:TEST_TZ:0
POSTGRES:test_tz:0
ORACLEDB:MESURE:0
POSTGRES:mesure:0
ORACLEDB:FICHIER_DONNEE:0
POSTGRES:fichier_donnee:0
ORACLEDB:TEST_NUM:0
POSTGRES:test_num:0
ORACLEDB:LOCATIONS:0
POSTGRES:locations:0
[ERRORS COLUMN DEFAULT VALUE COUNT]
OK, Oracle and PostgreSQL have the same number of column default value.

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

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

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

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

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

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

[TEST MVIEW COUNT]
ORACLEDB:MVIEW:1

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

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

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

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

Il est aussi possible de demander à Ora2Pg de dénombrer et de comparer le nombre de lignes de chaque table avec le type `TEST_COUNT` :

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

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

Dénombrement des résultats des vues

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

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

4.6.2 Outils de tests unitaires pour PostgreSQL

- pgTap
  - http://www.pgtap.org/
- pgUnit
  - https://github.com/adrianandrei-ca/pgunit

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[7], vous trouverez un exemple d'implémentation Java avec le projet tap4j[8].

[7] https://testanything.org/
[8] https://sourceforge.net/projects/tap4j/

4.6.3 Plans de tests complets

- Tests sur la base de 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 !

4.8 QUIZ

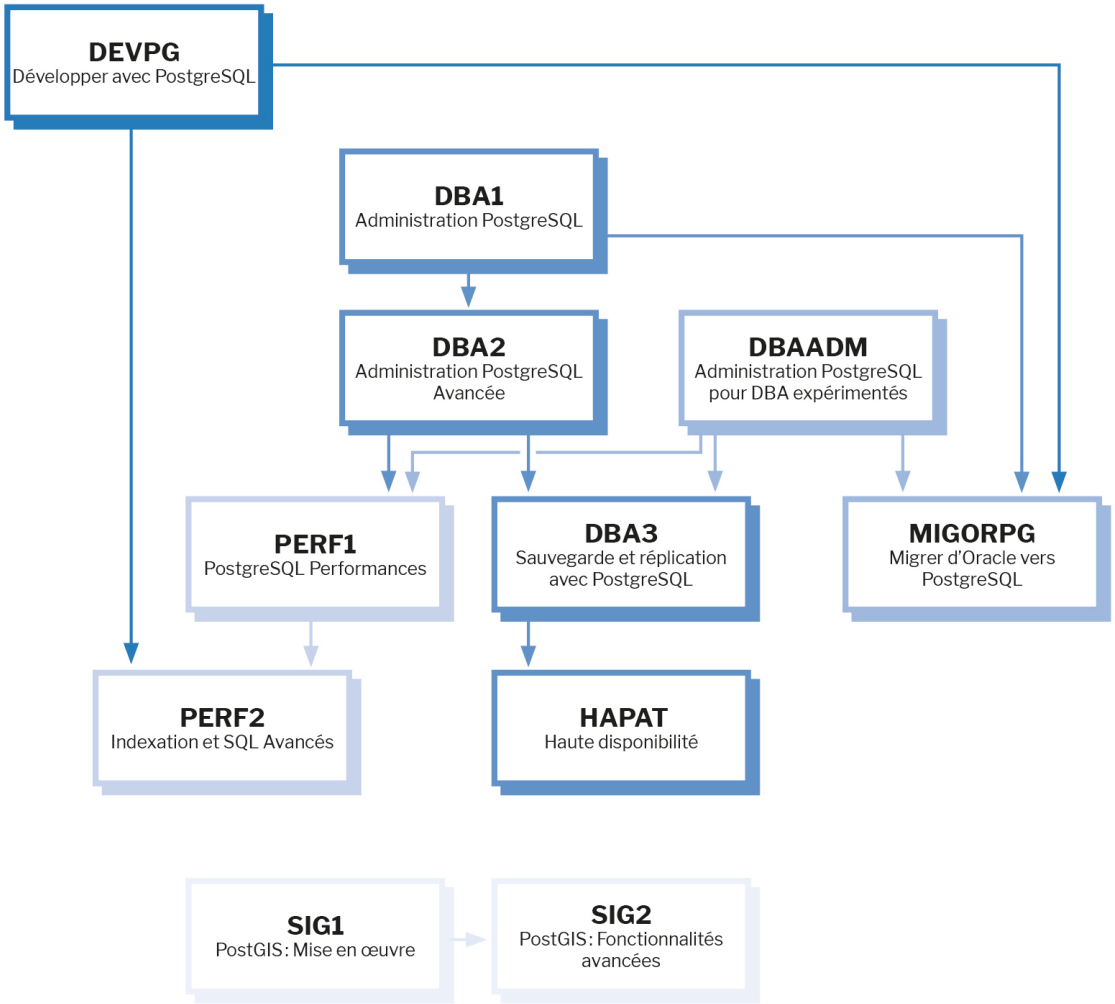
https://dali.bo/n4_quiz

Les formations Dalibo

Retrouvez nos formations et le calendrier sur https://dali.bo/formation

Pour toute information ou question, n'hésitez pas à nous écrire sur contact@dalibo.com.

Cursus des formations



Retrouvez nos formations dans leur dernière version :

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

Les livres blancs

- Migrer d'Oracle à PostgreSQL
  https://dali.bo/dlb01
- Industrialiser PostgreSQL
  https://dali.bo/dlb02
- Bonnes pratiques de modélisation avec PostgreSQL
  https://dali.bo/dlb04
- Bonnes pratiques de développement avec PostgreSQL
  https://dali.bo/dlb05

Téléchargement gratuit

Les versions électroniques de nos publications sont disponibles gratuitement sous licence open source ou sous licence Creative Commons.