Module N3

Requêtes SQL

DALIBO
L'expertise PostgreSQL

22.09

Dalibo SCOP

https://dalibo.com/formations

Requêtes SQL

Module N3

TITRE : Requêtes SQL
SOUS-TITRE : Module N3

REVISION: 22.09
DATE: 02 septembre 2022

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

LICENCE CREATIVE COMMONS BY-NC-SA 2.0 FR

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

Vous êtes autorisé à :

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

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

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

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

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

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

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

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

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

[1] mailto:contact@dalibo.com

Chers lectrices & lecteurs,

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

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

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

Table des Matières

1 REQUÊTES SQL

1.1 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.

1.2 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.

1.2.1 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 à 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.

1.2.2 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 simulter 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[2]

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

1.2.3 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
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)
```

1.2.4 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. Quoiqu'il en soit, la plupart des index utilisés sont des index de type `BTREE`.

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

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

Tablespaces

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

Types utilisateur

L'ensemble des types pouvant être défini par un utilisateur sont supportés avec plus ou moins d'adaptation. Il peut notamment être nécessaire de redéfinir des fonctions d'entrée/sortie définissant le comportement lors d'une insertion/lecture sur les données

du type. Dans la plupart des cas, il s'agit de types composites ou de tableaux parfaitement supportés par PostgreSQL.

Exemple de type composite version Oracle :

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

et la version PostgreSQL :

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

Exemple d'un tableau de type :

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

qui sera traduit en :

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

dblink

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

1.2.5 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.

1.3 TYPES DE DONNÉES

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

1.3.1 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.

1.3.2 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`).

1.3.3 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)
```

1.3.4 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.

1.3.5 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
```

1.3.6 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`).

1.3.7 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[3] , 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.

---

[3] https://postgis.net/

1.4 DIFFÉRENCES DE SYNTAXES

- Expressions conditionnelles DECODE et NVL
- 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.

1.4.1 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.

1.4.2 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;
```

1.4.3 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.

1.4.4 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.

1.4.5 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;
```

1.4.6 ROWNUM ET ORDER BY

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

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

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

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

```
test=# EXPLAIN SELECT * FROM t1 ORDER BY col DESC LIMIT 10;
                           QUERY PLAN
-----------------------------------------------------------------
 Limit  (cost=4.16..4.19 rows=10 width=4)
   ->  Sort  (cost=4.16..4.41 rows=100 width=4)
         Sort Key: col
         ->  Seq Scan on t1  (cost=0.00..2.00 rows=100 width=4)
(4 rows)
```

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

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

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

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

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

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

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

1.4.7 JOINTURES

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

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

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

```
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;
```

1.4.8 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;
```

1.4.9 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;
```

1.4.10 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;
```

1.4.11 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.

1.4.12 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.

1.4.13 WITH RECURSIVE

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

1.4.14 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;
```

1.4.15 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
```

1.4.16 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;
```

1.4.17 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[4] .

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

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

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

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

1.5 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 un nouvelle automatiquement.

De même, le pilote JDBC de la communauté PostgreSQL[5] 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);
```

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

```
                  ^
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)
```

1.5.1 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[6] , est appelée *Serializable Snapshot Isolation* et corrige les défauts des implémentations[7] précédentes du niveau `SERIALIZABLE`.

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

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

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

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

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

Pour plus de détails sur les niveaux d'isolation, consulter la documentation de PostgreSQL sur l'isolation des transactions[8] .

1.5.2 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 :

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

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

1.5.3 VERROUS EXPLICITES

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

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

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

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

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

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

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

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

1.6 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 commnu entre Oracle et PostgreSQL. La norme SQL:2016 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-ensemble 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 fonctionnalité 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[9] 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();
```

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

1.6.1 QUESTIONS

N'hésitez pas, c'est le moment !

1.7 QUIZ

https://dali.bo/n3_quiz

1.8 TRAVAUX PRATIQUES

Question 1

Les traitements sur les dates

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

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

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

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

Faites de même avec la requête :

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

Valeur retournée sur Oracle :

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

Question 2

Jointures (+)

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

Réécrire le code suivant dans le respect de la norme :

```
SELECT last_name, first_name, department_name
  FROM employees e, departments d
 WHERE e.employee_id = d.manager_id (+);
```

Puis ce code :

```
SELECT city, country_name
  FROM locations l, countries c
 WHERE l.country_id (+) = c.country_id;
```

Question 3

ROWNUM

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

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

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

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

Question 4

Portage de DECODE

La construction suivante utilise la fonction `DECODE` :

```
SELECT last_name, job_id, salary,
       DECODE(job_id,
              'PU_CLERK', salary * 1.05,
              'SH_CLERK', salary * 1.10,
              'ST_CLERK', salary * 1.15,
              salary) "Proposed Salary"
  FROM employees
 WHERE job_id LIKE '%_CLERK'
   AND last_name < 'E'
 ORDER BY last_name;
```

Réécrire cette requête pour son exécution sous PostgreSQL.

Autre exemple à convertir :

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

Question 5

FUNCTION

Porter sur PostgreSQL la fonction Oracle suivante :

```
CREATE OR REPLACE FUNCTION text_length(a CLOB)
  RETURN NUMBER DETERMINISTIC IS
BEGIN
  RETURN DBMS_LOB.GETLENGTH(a);
END;
```

Les fonctions sur les chaînes de caractères sont listées dans la documentation : http://docs.postgresql.fr/current/functions-string.html

---

Question 6

CONNECT BY

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

```
 SELECT employee_id, last_name, manager_id
   FROM employees
  START WITH manager_id IS NULL
CONNECT BY PRIOR employee_id = manager_id
  ORDER BY employee_id;
```

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

Voici le retour de cette requête sous Oracle :

```
EMPLOYEE_ID LAST_NAME      MANAGER_ID
----------- -------------- ----------
        100 King
        101 Kochhar               100
```

```
        102 De Haan             100
        103 Hunold              102
        104 Ernst               103
(...)
```

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

1.9 TRAVAUX PRATIQUES (SOLUTIONS)

Question 1

Les traitements sur les dates

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

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

```
-- SELECT SYSDATE + 1 FROM DUAL;
-- devient
SELECT CURRENT_DATE + 1;
```

Ce qui est équivalent à :

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

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

```
-- SELECT add_months(to_date('14-MAR-2014'), 2) FROM DUAL;
-- devient
SELECT '2014-03-14'::date + '2 months'::interval;
```

renvoie :

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

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

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

ou, ce qui revient au même :

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

Question 2

Jointures (+)

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

```
-- SELECT last_name, first_name, department_name
--   FROM employees e, departments d
```

```
--  WHERE e.employee_id = d.manager_id (+);
-- devient
SELECT last_name, first_name, department_name
  FROM employees e
  LEFT OUTER JOIN departments d ON e.employee_id = d.manager_id;
```

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

```
-- SELECT city, country_name
--   FROM locations l, countries c
--  WHERE l.country_id (+) = c.country_id;
-- devient
SELECT city, country_name
  FROM locations l
 RIGHT JOIN countries c ON l.country_id = c.country_id;
```

Question 3

ROWNUM

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

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

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

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

```
-- SELECT country_name, region_name
--   FROM countries c
--   JOIN regions r ON (c.region_id = r.region_id)
--  WHERE ROWNUM < 21;
-- devient
SELECT country_name, region_name
  FROM countries c
  JOIN regions r ON (c.region_id = r.region_id)
 LIMIT 20 OFFSET 0;
```

Pour une requête pouvant interroger un grand nombre de données, la réécriture avec `OFFSET` limitera les performances de PostgreSQL.

Il faudra alors réécrire la requête en combinant `FETCH FIRST N ROWS ONLY` et l'utilisation de la dernière région de la pagination dans la clause `WHERE` ainsi que dans les colonnes retournées par le `SELECT`.

```
SELECT c.region_id, country_name, region_name
  FROM countries c
  JOIN regions r ON (c.region_id = r.region_id)
 WHERE c.region_id < 100
 ORDER BY region_id DESC
 FETCH FIRST 20 ROWS ONLY;
```

Question 4

Portage de DECODE

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

```
SELECT last_name, job_id, salary,
       DECODE(job_id,
              'PU_CLERK', salary * 1.05,
              'SH_CLERK', salary * 1.10,
              'ST_CLERK', salary * 1.15,
              salary) "Proposed Salary"
  FROM employees
 WHERE job_id LIKE '%_CLERK'
   AND last_name < 'E'
 ORDER BY last_name;
```

Cette construction doit être réécrite de cette façon :

```
SELECT last_name, job_id, salary,
       CASE job_id
            WHEN 'PU_CLERK' THEN salary * 1.05
            WHEN 'SH_CLERK' THEN salary * 1.10
            WHEN 'ST_CLERK' THEN salary * 1.15
            ELSE salary
       END AS "Proposed salary"
  FROM employees
 WHERE job_id LIKE '%_CLERK'
   AND last_name < 'E'
 ORDER BY last_name;
```

Réécriture du second exemple :

```
-- DECODE("user_status",'active',"username",NULL)
-- devient
CASE WHEN user_status='active' THEN username ELSE NULL END
```

Question 5

FUNCTION

La fonction Oracle

```
CREATE OR REPLACE FUNCTION text_length(a CLOB)
  RETURN NUMBER DETERMINISTIC IS
BEGIN
  RETURN DBMS_LOB.GETLENGTH(a);
END;
```

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

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

ou simplement en SQL :

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

Question 6

CONNECT BY

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

```
 SELECT employee_id, last_name, manager_id
   FROM employees
```

```
  START WITH manager_id IS NULL
CONNECT BY PRIOR employee_id = manager_id
  ORDER BY employee_id;
```

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

```
WITH RECURSIVE employees_hierarchie (employee_id, last_name, manager_id) AS
(
  SELECT employee_id, last_name, manager_id
    FROM employees
   WHERE manager_id IS NULL
   UNION ALL
  SELECT e.employee_id, e.last_name, e.manager_id
    FROM employees e
    JOIN employees_hierarchie prior ON (e.manager_id = prior.employee_id)
)
SELECT * FROM employees_hierarchie
 ORDER BY employee_id;
```

Résultat :

```
 employee_id |  last_name  | manager_id
-------------+-------------+------------
         100 | King        |
         101 | Kochhar     |        100
         102 | De Haan     |        100
         103 | Hunold      |        102
         104 | Ernst       |        103
```

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

La récursion continue ensuite avec la requête suivante qui réalise une jointure entre la table `employees` et la vue virtuelle `employees_hierarchie` qui est définie par la clause `WITH RECURSIVE`. La condition de jointure correspond à la clause `CONNECT BY`. La vue virtuelle `employees_hierarchie` a pour alias `prior` pour mieux représenter la transposition de la clause `CONNECT BY`.

NOTES

NOTES

NOTES

NOTES

NOTES

NOS AUTRES PUBLICATIONS

FORMATIONS

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

LIVRES BLANCS

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

TÉLÉCHARGEMENT GRATUIT

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

DALIBO, L'EXPERTISE POSTGRESQL

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

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

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