Module S60

SQL avancé pour le transactionnel

25.09

Table des matières

Sur ce document

Formation	Module S60
Titre	SQL avancé pour le transactionnel
Révision	25.09
PDF	https://dali.bo/s60_pdf
EPUB	https://dali.bo/s60_epub
HTML	https://dali.bo/s60_html
Slides	https://dali.bo/s60_slides
TP	https://dali.bo/s60_tp
TP (solutions)	https://dali.bo/s60_solutions

Vous trouverez en ligne les différentes versions complètes de ce document.

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 :

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

Cette licence interdit la réutilisation pour l'apprentissage d'une IA. Elle couvre 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 13 à 17.

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/ SQL avancé pour le transactionnel

1.0.1 Préambule

— SQL et PostgreSQL proposent de nombreuses fonctionnalités avancées
  — normes SQL :99, 2003, 2008 et 2011
  — parfois, extensions propres à PostgreSQL

La norme SQL a continué d'évoluer et a bénéficié d'un grand nombre d'améliorations. Beaucoup de requêtes qu'il était difficile d'exprimer avec les premières incarnations de la norme sont maintenant faciles à réaliser avec les dernières évolutions.

Ce module a pour objectif de voir les fonctionnalités pouvant être utiles pour développer une application transactionnelle.

1.0.2 Menu

— `LIMIT` / `OFFSET`
— Jointures `LATERAL`
— `UPSERT` : `INSERT` ou `UPDATE`
— *Common Table Expressions*
— Serializable Snapshot Isolation

1.0.3 Objectifs

— Aller au-delà de SQL :92
— Concevoir des requêtes simples pour résoudre des problèmes complexes

Beaucoup de personnes écrivant du SQL ne connaissent que les bases du SQL :92. Le langage a cependant de nombreuses fonctionnalités très puissantes à connaître.

1.1 LIMIT

- Clause `LIMIT`
- ou syntaxe en norme SQL : `FETCH FIRST xx ROWS`
- Utilisation :
  - limite le nombre de lignes du résultat

La clause `LIMIT`, ou sa déclinaison normalisée par le comité ISO `FETCH FIRST xx ROWS`, permet de limiter le nombre de lignes résultant d'une requête SQL. La syntaxe `LIMIT` est cependant plus connue et est plus concise.

1.1.1 LIMIT : exemple

```
SELECT *
  FROM employes
LIMIT 2;

 matricule |   nom    | service  | salaire
-----------+----------+----------+----------
 00000001  | Dupuis   |          | 10000.00
 00000004  | Fantasio | Courrier |  4500.00
(2 lignes)
```

L'exemple ci-dessous s'appuie sur un jeu d'essai créé ainsi :

```
-- Si la table existe déjà, la détruire
DROP TABLE IF EXISTS employes CASCADE ;

-- Création de la table
CREATE TABLE employes (
matricule char(8) PRIMARY KEY,
nom       text NOT NULL,
service   text,
salaire   numeric(7,2)
);

-- Données
INSERT INTO employes (matricule, nom, service, salaire)
VALUES ('00000001', 'Dupuis', 'Direction', 10000.00),
('00000004', 'Fantasio', 'Courrier', 4500.00),
('00000006', 'Prunelle', 'Publication', 4000.00),
('00000020', 'Lagaffe', 'Courrier', 3000.00),
('00000040', 'Lebrac', 'Publication', 3000.00);

SELECT * FROM employes ;
```

```
 matricule |   nom    |   service   | salaire
-----------+----------+-------------+----------
 00000001  | Dupuis   |             | 10000.00
 00000004  | Fantasio | Courrier    |  4500.00
 00000006  | Prunelle | Publication |  4000.00
 00000020  | Lagaffe  | Courrier    |  3000.00
 00000040  | Lebrac   | Publication |  3000.00
(5 lignes)
```

Il faut faire attention au fait que ces fonctions ne permettent pas d'obtenir des résultats stables si les données ne sont pas triées explicitement. En effet, le standard SQL ne garantit en aucune façon l'ordre des résultats à moins d'employer la clause `ORDER BY`, et que l'ensemble des champs sur lequel on trie soit unique et non `NULL`.

Si une ligne était modifiée, changeant sa position physique dans la table, le résultat de la requête ne serait pas le même. Par exemple, en réalisant une mise à jour fictive de la ligne correspondant au matricule `00000001` :

```
UPDATE employes
   SET nom = nom
 WHERE matricule = '00000001';
```

L'ordre du résultat n'est pas garanti :

```
SELECT *
  FROM employes
 LIMIT 2;
```

```
 matricule |   nom    |   service   | salaire
-----------+----------+-------------+---------
 00000004  | Fantasio | Courrier    | 4500.00
 00000006  | Prunelle | Publication | 4000.00
(2 lignes)
```

L'application d'un critère de tri explicite permet d'obtenir la sortie souhaitée :

```
SELECT *
  FROM employes
 ORDER BY matricule
 LIMIT 2;
```

```
 matricule |   nom    | service  | salaire
-----------+----------+----------+----------
 00000001  | Dupuis   |          | 10000.00
 00000004  | Fantasio | Courrier |  4500.00
```

1.1.2 OFFSET

— Clause OFFSET
  — à utiliser avec `LIMIT`
— Utilité :
  — pagination de résultat
  — sauter les *n* premières lignes avant d'afficher le résultat

Ainsi, en reprenant le jeu d'essai utilisé précédemment :

```
SELECT * FROM employes ;
```

```
 matricule |   nom    |   service   | salaire
-----------+----------+-------------+----------
 00000001  | Dupuis   |             | 10000.00
 00000004  | Fantasio | Courrier    |  4500.00
 00000006  | Prunelle | Publication |  4000.00
 00000020  | Lagaffe  | Courrier    |  3000.00
 00000040  | Lebrac   | Publication |  3000.00
(5 lignes)
```

1.1.3 OFFSET : exemple (1/2)

— Sans offset :

```
SELECT *
  FROM employes
 LIMIT 2
 ORDER BY matricule;

 matricule |   nom    | service  | salaire
-----------+----------+----------+----------
 00000001  | Dupuis   |          | 10000.00
 00000004  | Fantasio | Courrier |  4500.00
```

1.1.4 OFFSET : exemple (2/2)

— En sautant les deux premières lignes :

```
SELECT *
  FROM employes
 ORDER BY matricule
 LIMIT 2
 OFFSET 2;

 matricule |   nom    |   service   | salaire
-----------+----------+-------------+---------
 00000006  | Prunelle | Publication | 4000.00
 00000020  | Lagaffe  | Courrier    | 3000.00
```

1.1.5 OFFSET : problèmes

— `OFFSET` est problématique
  — beaucoup de données lues
  — temps de réponse dégradés
— Alternative possible
  — utilisation d'un index sur le critère de tri
  — critère de filtrage sur la page précédente
— https://use-the-index-luke.com/fr/no-offset

Cependant, sur un jeu de données conséquent et une pagination importante, ce principe de fonctionnement peut devenir contre-performant. En effet, la base de données devra lire malgré tout les enregistrements qui n'apparaîtront pas dans le résultat de la requête, simplement dans le but de les compter.

Soit la table `posts` suivante (téléchargeable sur https://dali.bo/tp_posts, à laquelle on ajoute un index sur `(id_article_id, id_post)`) :

```
\d posts
                                Table « public.posts »
  Colonne   |           Type           | Collationnement | NULL-able | Par défaut
------------+--------------------------+-----------------+-----------+------------
 id_article | integer                  |                 |           |
 id_post    | integer                  |                 |           |
 ts         | timestamp with time zone |                 |           |
 message    | text                     |                 |           |
Index :
    "posts_id_article_id_post" btree (id_article, id_post)
    "posts_ts_idx" btree (ts)
```

Si l'on souhaite récupérer les 10 premiers enregistrements :

```
SELECT  *
FROM    posts
WHERE   id_article =12
ORDER BY id_post
LIMIT   10 ;
```

On obtient le plan d'exécution[1] suivant :

```
                                  QUERY PLAN
------------------------------------------------------------------------------
 Limit  (cost=0.43..18.26 rows=10 width=115)
        (actual time=0.043..0.053 rows=10 loops=1)
   ->  Index Scan using posts_id_article_id_post on posts
                 (cost=0.43..1745.88 rows=979 width=115)
               (actual time=0.042..0.051 rows=10 loops=1)
         Index Cond: (id_article = 12)
 Planning Time: 0.204 ms
 Execution Time: 0.066 ms
```

La requête est rapide car elle profite d'un index bien trié et elle ne lit que peu de données, ce qui est bien.

En revanche, si l'on saute un nombre conséquent d'enregistrements grâce à la clause `OFFSET`, la situation devient problématique :

```
SELECT  *
FROM    posts
WHERE   id_article = 12
ORDER BY id_post
LIMIT   10
OFFSET  900 ;
```

Le plan[2] n'est plus le même :

```
Limit  (cost=1605.04..1622.86 rows=10 width=115)
        (actual time=0.216..0.221 rows=10 loops=1)
   ->  Index Scan using posts_id_article_id_post on posts
                  (cost=0.43..1745.88 rows=979 width=115)
                  (actual time=0.018..0.194 rows=910 loops=1)
         Index Cond: (id_article = 12)
 Planning Time: 0.062 ms
 Execution Time: 0.243 ms
```

Pour répondre à la requête, PostgreSQL choisit la lecture de l'ensemble des résultats, puis leur tri, pour enfin appliquer la limite. En effet, `LIMIT` et `OFFSET` ne peuvent s'opérer que sur le résultat trié : il faut lire les 910 posts avant de pouvoir choisir les 10 derniers.

Le problème de ce plan est que, plus le jeu de données sera important, plus les temps de réponse seront importants. Ils seront encore plus importants si le tri n'est pas utilisable dans un index, ou si l'on déclenche un tri sur disque. Il faut donc trouver une solution pour les minimiser.

Les problèmes de l'utilisation de la clause `OFFSET` sont parfaitement expliqués dans cet article[3].

[1] https://explain.dalibo.com/plan/xEs
[2] https://explain.dalibo.com/plan/V05
[3] https://use-the-index-luke.com/fr/no-offset

Dans notre cas, la solution revient à créer un index qui contient le critère ainsi que le champ qui fixe la pagination (l'index existant convient). Puis on mémorise à quel `post_id` la page précédente s'est arrêtée, pour le donner comme critère de filtrage (ici `12900`). Il suffit donc de récupérer les 10 articles pour lesquels `id_article = 12` et `id_post > 12900` :

```
EXPLAIN ANALYZE
SELECT  *
FROM    posts
WHERE   id_article = 12
AND     id_post> 12900
ORDER BY id_post
LIMIT   10 ;
```

```
                            QUERY PLAN
------------------------------------------------------------------
 Limit  (cost=0.43..18.29 rows=10 width=115)
        (actual time=0.018..0.024 rows=10 loops=1)
   ->  Index Scan using posts_id_article_id_post on posts
                        (cost=0.43..1743.02 rows=976 width=115)
                        (actual time=0.016..0.020 rows=10 loops=1)
         Index Cond: ((id_article = 12) AND (id_post > 12900))
 Planning Time: 0.111 ms
 Execution Time: 0.039 ms
```

1.2 RETURNING

- Clause `RETURNING`
- Utilité :
  - récupérer les enregistrements modifiés
  - avec `INSERT`
  - avec `UPDATE`
  - avec `DELETE`

La clause `RETURNING` permet de récupérer les valeurs modifiées par un ordre DML.

On a ainsi une modification et la récupération des valeurs générées, modifiées ou supprimées en un seul ordre.

1.2.1 RETURNING : exemple

```
CREATE TABLE test_returning (id serial primary key, val integer);
-- Insertion de la valeur val seulement
INSERT INTO test_returning (val)
  VALUES (10)
RETURNING id, val;

 id | val
----+-----
  1 |  10
(1 ligne)
```

La clause `RETURNING` permet, par exemple, de récupérer la valeur de colonnes portant une valeur par défaut, comme la valeur créée par une séquence, comme sur l'exemple ci-dessus.

`RETURNING` permet également de récupérer les valeurs des colonnes mises à jour :

```
UPDATE test_returning
   SET val = val + 10
 WHERE id = 1
 RETURNING id, val;

 id | val
----+-----
  1 |  20
```

Associée à l'ordre `DELETE`, `RETURNING` renvoie les lignes supprimées :

```
DELETE FROM test_returning
 WHERE val < 30
 RETURNING id, val;

 id | val
----+-----
  1 |  20
```

1.3 UPSERT (INSERT … ON CONFLICT)

- `INSERT` ou `UPDATE` ?
  - `INSERT … ON CONFLICT DO { NOTHING | UPDATE }`
- Utilité :
  - mettre à jour en cas de conflit sur un `INSERT`
  - ne rien faire en cas de conflit sur un `INSERT`
- Plutôt préférer `MERGE`

Par *upsert*, on entend la syntaxe `INSERT … ON CONFLICT DO { NOTHING | UPDATE }` qui permet d'insérer une ligne, ou de la mettre à jour si elle existe déjà (c'est-à-dire si la clé qui sert à l'identifier est mise à jour).

L'implémentation de PostgreSQL de `ON CONFLICT DO UPDATE` est une opération atomique, c'est-à-dire que PostgreSQL garantit que l'une ou l'autre seulement des conditions sera effectuée, sans souci en cas de traitements en parallèle et de suppression (sauf erreur pour une autre raison). En comparaison, plusieurs approches naïves présentent des problèmes de concurrences d'accès. (Voir les différentes approches décrites dans cet article de Depesz[4]).

Il faut évidemment qu'il y ait une contrainte d'unicité pour l'identification des lignes.

Une bonne conception évitera autant que possible de modifier les mêmes lignes par des traitements simultanés, ne serait-ce que pour éviter des ralentissements à cause de verrous. Mais ce n'est pas toujours possible.

La clause `INSERT … ON CONFLICT` est très proche de la commande `MERGE` (voir plus bas), plus proche du standard, mais qui n'est apparue dans PostgreSQL que plus tard.

1.3.1 UPSERT : problème à résoudre

- Insérer une ligne déjà existante provoque une erreur :

```
INSERT INTO employes (matricule, nom, service, salaire)
 VALUES ('00000001', 'Marsupilami', 'Direction', 50000.00);

ERROR:  duplicate key value violates unique constraint
        "employes_pkey"
DETAIL:  Key (matricule)=(00000001) already exists.
```

Si l'on souhaite insérer une ligne contenant un matricule déjà existant, une erreur de clé dupliquée est levée et toute la transaction est annulée.

[4] https://www.depesz.com/2012/06/10/why-is-upsert-so-complicated/

1.3.2 ON CONFLICT DO NOTHING

— La clause `ON CONFLICT DO NOTHING` évite d'insérer une ligne existante :

```
INSERT INTO employes (matricule, nom, service, salaire)
   VALUES ('00000001', 'Marsupilami', 'Direction', 50000.00)
   ON CONFLICT DO NOTHING;

INSERT 0 0
```

Les données n'ont pas été modifiées :

```
SELECT * FROM employes ;
```

```
 matricule |   nom     |   service    | salaire
-----------+-----------+--------------+----------
 00000004  | Fantasio  | Courrier     |  4500.00
 00000006  | Prunelle  | Publication  |  4000.00
 00000020  | Lagaffe   | Courrier     |  3000.00
 00000040  | Lebrac    | Publication  |  3000.00
 00000001  | Dupuis    | Direction    | 10000.00
(5 rows)
```

La transaction est toujours valide.

1.3.3 ON CONFLICT DO NOTHING : syntaxe

```
INSERT …
ON CONFLICT
  DO NOTHING;
```

Il suffit d'indiquer à PostgreSQL de ne rien faire en cas de conflit sur une valeur dupliquée avec la clause `ON CONFLICT DO NOTHING` placée à la fin de l'ordre `INSERT` qui peut poser problème.

Dans ce cas, si une rupture d'unicité est détectée, alors PostgreSQL ignorera l'erreur, silencieusement. En revanche, si une erreur apparaît sur une autre contrainte, l'erreur sera levée.

En prenant l'exemple suivant :

```
CREATE TABLE test_upsert (
  i serial PRIMARY KEY,
  v text UNIQUE,
  x integer CHECK (x > 0)
);

INSERT INTO test_upsert (v, x) VALUES ('x', 1);
```

L'insertion d'une valeur dupliquée provoque bien une erreur d'unicité :

```
INSERT INTO test_upsert (v, x) VALUES ('x', 1);

ERROR:  duplicate key value violates unique constraint "test_upsert_v_key"
```

L'erreur d'unicité est bien ignorée si la ligne existe déjà, le résultat est `INSERT 0 0` qui indique qu'aucune ligne n'a été insérée :

```
INSERT INTO test_upsert (v, x)
VALUES ('x', 1)
 ON CONFLICT DO NOTHING;

INSERT 0 0
```

L'insertion est aussi ignorée si l'on tente d'insérer des lignes rompant la contrainte d'unicité mais ne comportant pas les mêmes valeurs pour d'autres colonnes :

```
INSERT INTO test_upsert (v, x)
VALUES ('x', 4)
 ON CONFLICT DO NOTHING;

INSERT 0 0
```

Si l'on insère une valeur interdite par la contrainte `CHECK`, une erreur est bien levée :

```
INSERT INTO test_upsert (v, x)
VALUES ('x', 0)
 ON CONFLICT DO NOTHING;

ERROR:  new row for relation "test_upsert" violates check constraint
        "test_upsert_x_check"
DETAIL:  Failing row contains (4, x, 0).
```

1.3.4 ON CONFLICT DO UPDATE

```
INSERT INTO employes (matricule, nom, service, salaire)
VALUES ('00000001', 'M. Pirate', 'Direction', 0.00)
ON CONFLICT (matricule)
   DO UPDATE SET salaire = employes.salaire,
                 nom = excluded.nom
RETURNING *;

 matricule |    nom    |   service   | salaire
-----------+-----------+-------------+----------
 00000001  | M. Pirate | Direction   | 50000.00
```

La clause `ON CONFLICT` permet de déterminer une colonne sur laquelle le conflit peut arriver. Cette colonne ou ces colonnes doivent porter une contrainte d'unicité ou une contrainte d'exclusion, c'est à dire une contrainte portée par un index. La clause `DO UPDATE` associée fait référence aux valeurs

rejetées par le conflit à l'aide de la pseudo-table `excluded`. Les valeurs courantes sont accessibles en préfixant les colonnes avec le nom de la table. L'exemple montre cela.

Avec la requête de l'exemple, on voit que le salaire du directeur n'a pas été modifié, mais son nom l'a été :

```
SELECT * FROM employes ;
```

```
 matricule |    nom    |   service   | salaire
-----------+-----------+-------------+----------
 00000004  | Fantasio  | Courrier    |  4500.00
 00000006  | Prunelle  | Publication |  4000.00
 00000020  | Lagaffe   | Courrier    |  3000.00
 00000040  | Lebrac    | Publication |  3000.00
 00000001  | M. Pirate | Direction   | 10000.00
(5 rows)
```

La clause `ON CONFLICT` permet également de définir une contrainte d'intégrité sur laquelle on réagit en cas de conflit :

```
INSERT INTO employes (matricule, nom, service, salaire)
VALUES ('00000001', 'Marsupilami', 'Direction', 50000.00)
ON CONFLICT ON CONSTRAINT employes_pkey
   DO UPDATE SET salaire = excluded.salaire;
```

On remarque que seul le salaire du directeur a changé :

```
SELECT * FROM employes ;
```

```
 matricule |   nom     |   service   | salaire
-----------+-----------+-------------+----------
 00000004  | Fantasio  | Courrier    |  4500.00
 00000006  | Prunelle  | Publication |  4000.00
 00000020  | Lagaffe   | Courrier    |  3000.00
 00000040  | Lebrac    | Publication |  3000.00
 00000001  | M. Pirate | Direction   | 50000.00
(5 rows)
```

1.3.5 ON CONFLICT DO UPDATE

— Avec plusieurs lignes insérées :

```
INSERT INTO employes (matricule, nom, service, salaire)
VALUES ('00000002', 'Moizelle Jeanne', 'Publication', 3000.00),
       ('00000040', 'Lebrac', 'Publication', 3100.00)
ON CONFLICT (matricule)
   DO UPDATE SET salaire = employes.salaire,
                 nom = excluded.nom
RETURNING *;
```

```
 matricule |       nom       |   service   | salaire
-----------+-----------------+-------------+----------
 00000002  | Moizelle Jeanne | Publication |  3000.00
 00000040  | Lebrac          | Publication |  3000.00
```

Bien sûr, on peut insérer plusieurs lignes, `INSERT ON CONFLICT` réagira uniquement sur les doublons :

La nouvelle employée, *Moizelle Jeanne* a été intégrée dans la table `employes`, et *Lebrac* a été traité comme un doublon, en appliquant la règle de mise à jour vue plus haut : seul le nom est mis à jour et le salaire est inchangé.

```
SELECT * FROM employes ;
```

```
 matricule |       nom       |   service   | salaire
-----------+-----------------+-------------+----------
 00000004  | Fantasio        | Courrier    |  4500.00
 00000006  | Prunelle        | Publication |  4000.00
 00000020  | Lagaffe         | Courrier    |  3000.00
 00000001  | M. Pirate       | Direction   | 50000.00
 00000002  | Moizelle Jeanne | Publication |  3000.00
 00000040  | Lebrac          | Publication |  3000.00
(6 rows)
```

À noter que la clause `SET salaire = employes.salaire` est inutile, c'est ce que fait PostgreSQL implicitement.

1.3.6 ON CONFLICT DO UPDATE : syntaxe

— Colonne(s) portant(s) une contrainte d'unicité
— Pseudo-table *excluded*

```
INSERT ….
ON CONFLICT (<colonne clé>)
  DO UPDATE
        SET colonne_a_modifier = excluded.colonne,
            autre_colonne_a_modifier = excluded.autre_colonne,
            …;
```

Si l'on choisit de réaliser une mise à jour plutôt que de générer une erreur, on utilisera la clause `ON CONFLICT DO UPDATE`. Il faudra dans ce cas préciser là ou les colonnes qui portent une contrainte d'unicité. Cette contrainte d'unicité permettra de détecter la duplication de valeur, PostgreSQL pourra alors appliquer la règle de mise à jour édictée.

La règle de mise à jour permet de définir très finement les colonnes à mettre à jour et les colonnes à ne pas mettre à jour. Dans ce contexte, la pseudo-table `excluded` représente l'ensemble rejeté par l'`INSERT`. Il faudra explicitement indiquer les colonnes dont la valeur sera mise à jour à partir des valeurs que l'on tente d'insérer, reprise de la pseudo-table `excluded` :

```
ON CONFLICT (…)
  DO UPDATE
  SET colonne = excluded.colonne,
      autre_colonne = excluded.autre_colonne,
      …
```

En alternative, il est possible d'indiquer un nom de contrainte plutôt que le nom d'une colonne portant une contrainte d'unicité :

```
INSERT ….
ON CONFLICT ON CONSTRAINT nom_contrainte
  DO UPDATE
        SET colonne_a_modifier = excluded.colonne,
            autre_colonne_a_modifier = excluded.autre_colonne,
            …;
```

De plus amples informations quant à la syntaxe sont disponibles dans la documentation[5].

1.3.7 MERGE

```
MERGE INTO mesures_capteurs c
USING import_mesures_capteurs i
ON c.id = i .id
WHEN NOT MATCHED THEN
  INSERT (id, top_mesure, derniere_mesure, derniere_maj)
  VALUES (i.id, i.mesure, i.mesure, current_timestamp)
WHEN MATCHED AND ( c.derniere_maj + INTERVAL '10 days' <=
 ↪  current_timestamp ) THEN
  DELETE
WHEN MATCHED AND ( c.top_mesure > i.mesure ) THEN
  UPDATE
  SET derniere_mesure = i.mesure, derniere_maj = current_timestamp
WHEN MATCHED THEN
  UPDATE SET top_mesure = i.mesure, derniere_mesure = i.mesure,
   ↪  derniere_maj = current_timestamp
;
```

`MERGE` est très voisin de `INSERT … ON CONFLICT` mais n'est apparu qu'avec PostgreSQL 15. Leurs mécanismes diffèrent complètement et les deux syntaxes ont leur utilité.

`MERGE` est conforme au standard SQL (avec quelques extensions de syntaxe propres à PostgreSQL), a une syntaxe plus complexe et permet aussi des `DELETE` dans la table cible.

`MERGE` ne s'appuie pas sur les contraintes d'unicité et est plus susceptible de mener à des erreurs en cas de concurrence d'accès forte, au contraire de `INSERT … ON CONFLICT`.

Lors de son exécution, la commande commence par réaliser une jointure entre la source de donnée et la table cible. La source de donnée peut être une table ou une requête quelconque. La condition de

[5] https://docs.postgresql.fr/current/sql-insert.html

jointure ne doit contenir que des colonnes des tables source et cible qui participent à la jointure, et la jointure ne doit produire qu'une ligne pour chaque ligne candidate.

Chaque ligne candidate se voit assigner le statut `[NOT] MATCHED`, suivant que la jointure a été un succès ou non. Ensuite, les clauses `WHEN` sont évaluées dans l'ordre où elles sont spécifiées. Seule l'action associée à la première clause `WHEN` qui renvoie « vrai » est exécutée.

Lorsqu'elles sont exécutées, les actions ont les mêmes effets que des ordres `INSERT`, `UPDATE` ou `DELETE` classiques. La syntaxe est similaire, à la différence prêt qu'il n'y a ni clause `FROM` ni clause `WHERE`. Les actions agissent sur la cible, utilisent les lignes courantes de la jointure et agissent sur la cible.

Il est possible de spécifier `DO NOTHING` si on souhaite ignorer la ligne en cours. Ce résultat peut également être obtenu si aucune clause n'est évaluée à vrai.

`INSERT … ON CONFLICT UPDATE` garantit l'exécution atomique d'un `INSERT` ou d'un `UPDATE` même en cas de forte concurrence d'accès. La commande `MERGE` n'a pas ce genre de garantie. Si un `INSERT` est exécutée en même temps que le `MERGE`, il est possible que le `MERGE` ne la voit pas et choisisse d'utiliser son action `INSERT`, ce qui aboutira à une erreur de violation de contrainte d'unicité. C'est la raison pour laquelle la commande `MERGE` avait été initialement refusée et remplacée par `INSERT … ON CONFLICT`.

Pour les détails et les différences avec `INSERT … ON CONFLICT`, voir :

— Notre workshop de la version 15[6], avec l'exemple ci-dessus complet ;
— la documentation de PostgreSQL[7].

[6] https://dali.bo/workshop15_html#développement-changement-syntaxe-sql
[7] https://www.postgresql.fr/docs/current/sql-merge.html

1.4 LATERAL

— Jointures `LATERAL`
  — SQL :99
  — équivalent d'une boucle `foreach`

`LATERAL` apparaît dans la révision de la norme SQL de 1999. Elle permet d'appliquer une requête ou une fonction sur le résultat d'une table. L'implémentation dans la plupart des SGBD reste cependant relativement récente.

C'est une manière d'introduire une forme de fonctionnement procédural, alors que le SQL est de nature ensembliste. Ce peut être très utile dans certains cas.

1.4.1 LATERAL : utilité

— Utilité :
  — Top-N à partir de plusieurs tables
  — jointure avec une fonction retournant un ensemble
  — exemple : *afficher les 5 derniers messages des 5 derniers sujets actifs d'un forum*

`LATERAL` permet d'utiliser les données de la requête principale dans une sous-requête. La sous-requête sera appliquée à chaque enregistrement retourné par la requête principale.

1.4.2 LATERAL : exemple

```
SELECT titre,
       top_5_messages.date_publication,
       top_5_messages.extrait
  FROM sujets,
  LATERAL(SELECT date_publication,
                 substr(message, 0, 100) AS extrait
            FROM messages
           WHERE sujets.sujet_id = messages.sujet_id
           ORDER BY date_publication DESC
           LIMIT 5) top_5_messages
 ORDER BY sujets.date_modification DESC,
           top_5_messages.date_publication DESC
 LIMIT 25;
```

L'exemple ci-dessus montre comment afficher les 5 derniers messages postés sur les 5 derniers sujets actifs d'un forum avec la clause `LATERAL`.

Une autre forme d'écriture emploie le mot clé `JOIN`, inutile dans cet exemple. Il peut avoir son intérêt si l'on utilise une jointure externe (`LEFT JOIN` par exemple si un sujet n'impliquait pas forcément la présence d'un message) :

```
SELECT titre, top_5_messages.date_publication, top_5_messages.extrait
  FROM sujets
  JOIN LATERAL(SELECT date_publication, substr(message, 0, 100) AS extrait
          FROM messages
         WHERE sujets.sujet_id = messages.sujet_id
         ORDER BY date_publication DESC
         LIMIT 5) top_5_messages
    ON (true) -- condition de jointure toujours vraie
 ORDER BY sujets.date_modification DESC, top_5_messages.date_publication DESC
 LIMIT 25;
```

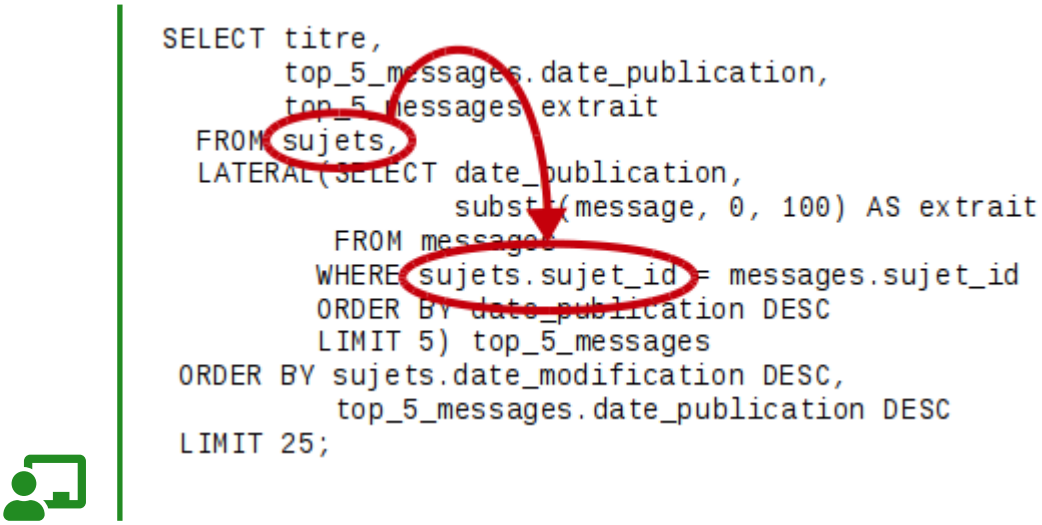
Il aurait été possible de réaliser cette requête par d'autres moyens, mais `LATERAL` permet d'obtenir la requête la plus performante. Une autre approche quasiment aussi performante aurait été de faire appel à une fonction retournant les 5 enregistrements souhaités.

À noter qu'une colonne `date_modification` a été ajouté à la table `sujets` afin de déterminer rapidement les derniers sujets modifiés. Sans cela, il faudrait parcourir l'ensemble des sujets, récupérer la date de publication des derniers messages avec une jointure `LATERAL` et récupérer les 5 derniers sujets actifs. Cela nécessite de lire beaucoup de données. Un trigger positionné sur la table `messages` permettra d'entretenir la colonne `date_modification` sur la table `sujets` sans difficulté. Il s'agit donc ici d'une entorse aux règles de modélisation en vue d'optimiser les traitements.

Un index sur les colonnes `sujet_id` et `date_publication` permettra de minimiser les accès pour cette requête :

```
CREATE INDEX ON messages (sujet_id, date_publication DESC);
```

1.4.3 LATERAL : principe



Si nous n'avions pas la clause `LATERAL`, nous pourrions être tentés d'écrire la requête suivante :

```
SELECT titre, top_5_messages.date_publication, top_5_messages.extrait
  FROM sujets
  JOIN (SELECT date_publication, substr(message, 0, 100) AS extrait
          FROM messages
         WHERE sujets.sujet_id = messages.sujet_id
         ORDER BY date_message DESC
         LIMIT 5) top_5_messages
 ORDER BY sujets.date_modification DESC
 LIMIT 25;
```

Cependant, la norme SQL interdit une telle construction, il n'est pas possible de référencer la table principale dans une sous-requête. Mais avec la clause `LATERAL`, la sous-requête peut faire appel à la table principale.

1.4.4 LATERAL : avec une fonction

- Utilisation avec une fonction retournant un ensemble
  - clause `LATERAL` optionnelle
- Utilité :
  - extraire les données d'un tableau ou d'une structure JSON sous forme tabulaire
  - utiliser une fonction métier qui retourne un ensemble X selon un ensemble Y fourni

L'exemple ci-dessous montre qu'il est possible d'utiliser une fonction retournant un ensemble (SRF pour *Set Returning Functions*).

1.4.5 LATERAL : exemple avec une fonction

```
SELECT titre,
       top_5_messages.date_publication,
       top_5_messages.extrait
  FROM sujets
       -- cette fonction peut retourner plusieurs lignes
       get_top_5_messages(sujet_id) AS top_5_messages
 ORDER BY sujets.date_modification DESC
 LIMIT 25;
```

La fonction `get_top_5_messages()` est la suivante :

```
CREATE OR REPLACE FUNCTION get_top_5_messages (p_sujet_id integer)
RETURNS TABLE (date_publication timestamp, extrait text)
AS $PROC$
BEGIN
  RETURN QUERY SELECT date_publication, substr(message, 0, 100) AS extrait
    FROM messages
   WHERE messages.sujet_id = p_sujet_id
   ORDER BY date_publication DESC
   LIMIT 5;
END;
$PROC$ LANGUAGE plpgsql;
```

La clause `LATERAL` n'est pas obligatoire, mais elle s'utiliserait ainsi :

```
SELECT titre, top_5_messages.date_publication, top_5_messages.extrait
  FROM sujets, LATERAL get_top_5_messages(sujet_id) AS top_5_messages
 ORDER BY sujets.date_modification DESC LIMIT 25;
```

1.5 COMMON TABLE EXPRESSIONS (CTE)

- *Common Table Expressions*
  - clauses `WITH` et `WITH RECURSIVE`
- Utilité :
  - factoriser des sous-requêtes

1.5.1 CTE et SELECT

- Utilité
  - factoriser des sous-requêtes
  - améliorer la lisibilité d'une requête

Les *Common Table Expressions*, ou CTE, permettent de factoriser la définition d'une sous-requête qui pourrait être appelée plusieurs fois.

Une CTE est exprimée avec la clause `WITH`. Cette clause permet de définir des « vues éphémères » qui seront utilisées les unes après les autres, éventuellement en cascade, et au final utilisées dans la requête principale.

1.5.2 CTE et SELECT : exemple

```
WITH resultat AS (
   /* requête complexe */
)
SELECT *
  FROM resultat
  WHERE nb < 5;
```

La première utilité d'une CTE est de factoriser la définition d'une sous-requête commune, comme `resultat` dans l'exemple ci-dessus.

Cela est logiquement équivalent à écrire une vue `resultat` de même définition et à l'utiliser dans la requête. La CTE évite la création, la maintenance et la suppression de ces vues. La souplesse et, nous le verrons, même les performances peuvent y gagner. Évidemment, si la définition de `resultat` est utilisée dans plusieurs requêtes, une vue à part entière sera plus pertinente.

L'exemple suivant illustre une réutilisation multiple d'une même CTE. Il utilise la base d'exemple **magasin**. La base **magasin** (dump de 96 Mo, pour 667 Mo sur le disque au final) peut être téléchargée et restaurée comme suit dans une nouvelle base **magasin** :

```
createdb magasin
curl -kL https://dali.bo/tp_magasin -o /tmp/magasin.dump
pg_restore -d magasin  /tmp/magasin.dump
# le message sur public préexistant est normal
rm -- /tmp/magasin.dump
```

Les données sont dans deux schémas, **magasin** et **facturation**. Penser au `search_path`.

Pour ce TP, figer les paramètres suivants :

```
SET max_parallel_workers_per_gather to 0;
SET seq_page_cost TO 1 ;
SET random_page_cost TO 4 ;
```

La CTE `commandes_2014` définit un ensemble de commandes, et ce jeu de données est utilisé deux fois, dans chacun des membres du `UNION ALL`.

```
WITH commandes_2014 AS (
-- vue des données sur 2014
SELECT c.numero_commande, c.client_id, quantite*prix_unitaire AS montant
  FROM magasin.commandes c
  JOIN magasin.lignes_commandes l
    ON (c.numero_commande = l.numero_commande)
 WHERE date_commande BETWEEN '2014-01-01' AND '2014-12-31'
)
-- Regroupement par type_client
SELECT type_client, NULL AS pays, SUM(montant) AS montant_total_commande
  FROM commandes_2014
  JOIN magasin.clients
    ON (commandes_2014.client_id = clients.client_id)
 GROUP BY type_client
--
UNION ALL
--
-- Regroupement par type_pays
SELECT NULL, code_pays AS pays, SUM(montant)
  FROM commandes_2014 r
  JOIN magasin.clients cl
    ON (r.client_id = cl.client_id)
  JOIN magasin.contacts co
    ON (cl.contact_id = co.contact_id)
 GROUP BY code_pays;
```

Le plan d'exécution de la requête ci-dessous peut être obtenu en précédant la requête de `EXPLAIN (ANALYZE, COSTS OFF, SUMMARY OFF)`. Une vision graphique plus complète est visible sur https://explain.dalibo.com/plan/92hce934hg875a80.

Ce plan montre que la vue `commandes_2014` est exécutée une seule fois, au début. Son résultat est ensuite relu par les deux opérations de regroupements définies dans la requête principale (voir les mentions `CTE Scan on commandes_2014`) :

```
                                QUERY PLAN
-------------------------------------------------------------------------
 Append (actual time=1295.413..1835.434 rows=12 loops=1)
   CTE commandes_2014
     ->  Hash Join (actual time=97.041..781.457 rows=1226456 loops=1)
           Hash Cond: (l.numero_commande = c.numero_commande)
           ->  Seq Scan on lignes_commandes l (actual time=0.041..175.880
↪  rows=3141967 loops=1)
           ->  Hash (actual time=96.483..96.484 rows=390331 loops=1)
                 Buckets: 524288  Batches: 1  Memory Usage: 22393kB
                 ->  Seq Scan on commandes c (actual time=1.403..53.373 rows=390331
↪  loops=1)
                       Filter: ((date_commande >= '2014-01-01'::date) AND
↪  (date_commande <= '2014-12-31'::date))
                       Rows Removed by Filter: 609669
   ->  HashAggregate (actual time=1295.412..1295.415 rows=3 loops=1)
         Group Key: clients.type_client
         Batches: 1  Memory Usage: 24kB
         ->  Hash Join (actual time=109.251..1130.601 rows=1226456 loops=1)
               Hash Cond: (commandes_2014.client_id = clients.client_id)
               ->  CTE Scan on commandes_2014 (actual time=97.045..936.931
↪  rows=1226456 loops=1)
               ->  Hash (actual time=12.139..12.140 rows=100000 loops=1)
                     Buckets: 131072  Batches: 1  Memory Usage: 5712kB
                     ->  Seq Scan on clients (actual time=0.009..4.834 rows=100000
↪  loops=1)
   ->  HashAggregate (actual time=540.009..540.013 rows=9 loops=1)
         Group Key: co.code_pays
         Batches: 1  Memory Usage: 24kB
         ->  Hash Join (actual time=30.661..386.341 rows=1226456 loops=1)
               Hash Cond: (cl.contact_id = co.contact_id)
               ->  Hash Join (actual time=12.125..224.655 rows=1226456 loops=1)
                     Hash Cond: (r.client_id = cl.client_id)
                     ->  CTE Scan on commandes_2014 r (actual time=0.001..46.190
↪  rows=1226456 loops=1)
                     ->  Hash (actual time=12.059..12.060 rows=100000 loops=1)
                           Buckets: 131072  Batches: 1  Memory Usage: 5712kB
                           ->  Seq Scan on clients cl (actual time=0.010..4.815
↪  rows=100000 loops=1)
               ->  Hash (actual time=18.476..18.477 rows=110005 loops=1)
                     Buckets: 131072  Batches: 1  Memory Usage: 6181kB
                    ->  Seq Scan on contacts co (actual time=0.007..9.229 rows=110005
↪  loops=1)
```

Sans CTE, il aurait fallu décrire deux fois la même sous-requête, ou créer une vue `commandes_2014` qui aurait été appelée deux fois. Le coût d'exécution aurait été multiplié par deux, car il aurait fallu exécuter la sous-requête deux fois au lieu d'une.

Le temps de la requête, le résultat de la CTE peut être stocké en mémoire ou sur disque : PostgreSQL s'en occupe. Dans l'exemple ci-dessus, le résultat de la CTE est matérialisé (en mémoire ou sur disque) pour être relu. Cette méthode évite souvent le recours à des alternatives beaucoup plus lourdes comme les vues matérialisées ou les tables de travail temporaires. De nombreuses CTE ne nécessitent pas de matérialisation.

1.5.3 CTE et SELECT : syntaxe

```
WITH nom_vue1 AS (
 <requête pour générer la vue 1>
), nom_vue2 AS (
 <requête pour générer la vue 2, pouvant utiliser la vue 1>
)
<requête principale utilisant vue 1 et/ou vue2> ;
```

On utilise aussi les CTE pour améliorer la lisibilité des requêtes complexes.

On peut enchaîner plusieurs vues les unes à la suite des autres comme dans l'exemple ci-dessous.

Il est donc possible de définir un jeu de données en plusieurs étapes dans la même requête, dans plusieurs clauses séparées, avec par exemple une première CTE pour une sélection des données, puis de l'utiliser dans une deuxième CTE pour une jointure, puis faire l'agrégat dans une troisième CTE, avec un `SELECT` final ne procédant qu'à un formatage. Une requête plus complexe peut devenir plus lisible. PostgreSQL, en interne, sait rassembler toutes ces vues en une requête unique.

1.5.4 CTE, MATERIALIZED, et barrière d'optimisation

— Clause `MATERIALIZED` pour forcer une barrière d'optimisation
— Normalement automatique, et pour le meilleur
— Parfois un souci

Normalement, PostgreSQL « fusionne » le contenu d'un CTE avec les autres de la même requête pour choisir la manière dont il va récupérer les données

```
EXPLAIN
 WITH e AS ( SELECT * FROM employes WHERE num_service = 4 )
 SELECT MAX(date_embauche)
 FROM e
 WHERE e.date_embauche < '2006-01-01';
```

```
                                QUERY PLAN
---------------------------------------------------------------------------
 Aggregate  (cost=1.21..1.22 rows=1 width=4)
   ->  Seq Scan on employes  (cost=0.00..1.21 rows=2 width=4)
         Filter: ((date_embauche < '2006-01-01'::date) AND (num_service = 4))
```

Il pourrait y avoir des cas où le planificateur de PostgreSQL échoue à comprendre le bon plan. Un CTE peut permettre de le forcer à exécuter les CTE dans l'ordre qui a été précisé. Cela s'appelle une « barrière d'optimisation », et cela peut se faire grâce au mot-clé `MATERIALIZED` :

```
-- CTE avec MATERIALIZED
EXPLAIN
  WITH e AS MATERIALIZED ( SELECT * FROM employes WHERE num_service = 4 )
  SELECT MAX(date_embauche)
  FROM e
  WHERE e.date_embauche < '2006-01-01';

                            QUERY PLAN
------------------------------------------------------------------
 Aggregate  (cost=1.29..1.30 rows=1 width=4)
   CTE e
     ->  Seq Scan on employes  (cost=0.00..1.18 rows=5 width=43)
           Filter: (num_service = 4)
   ->  CTE Scan on e  (cost=0.00..0.11 rows=2 width=4)
         Filter: (date_embauche < '2006-01-01'::date)
```

La CTE est alors intégralement exécutée avec son filtre propre, matérialisée, puis son contenu est relu dans un autre nœud et le deuxième filtre appliqué .

En général, ce genre de manipulation n'est pas nécessaire. (Jusqu'en version 11 incluse, le `MATERIALIZED` était celui par défaut, et les CTE étaient une source fréquente de problèmes de performances.)

À l'inverse, PostgreSQL peut choisir de « matérialiser » une CTE, notamment quand elle est utilisée plusieurs fois dans une requête (voir l'exemple plus haut).

1.5.5 CTE en écriture

- CTE avec des requêtes en modification
  - avec `INSERT` / `UPDATE` / `DELETE`
  - et éventuellement `RETURNING`
- Exemples d'utilisation :
  - requête complexe
  - écriture dans plusieurs tables
  - déplacement de données (archivage…)

Les CTE peuvent contenir des ordres d'écriture. Grâce à `RETURNING`, les données modifiées/insérées/effacées peuvent être réutilisées dans une autre partie de la requête.

Cette technique permet des requêtes très complexes capables d'écrire dans plusieurs lignes, ou déplaçant des données d'une table à une autre, dans un seul ordre. Sans les CTE, il faudrait des transactions plus complexes, avec des tables de travail, du code procédural, voire des curseurs…

1.5.6 CTE en écriture : exemple

```
WITH donnees_a_archiver AS (
DELETE FROM donnes_courantes
 WHERE date < '2015-01-01'
 RETURNING *
)
INSERT INTO donnes_archivees
SELECT * FROM donnees_a_archiver ;
```

La requête d'exemple permet d'archiver des données dans une table dédiée à l'archivage en utilisant une CTE en écriture. L'emploi de la clause `RETURNING` permet de récupérer les lignes purgées.

En plus de ce cas d'usage simple, il est possible d'utiliser cette fonctionnalité pour déboguer une requête complexe.

```
WITH sous-requete1 AS (

),
debug_sous-requete1 AS (
INSERT INTO debug_sousrequete1
SELECT * FROM sous-requete1
), sous-requete2 AS (
SELECT …
  FROM sous-requete1
  JOIN …
 WHERE …
 GROUP BY …
),
debug_sous-requete2 AS (
INSERT INTO debug_sousrequete2
SELECT * FROM sous-requete2
)
SELECT *
  FROM sous-requete2;
```

On peut également envisager une requête CTE en écriture pour émuler une requête `MERGE` pour réaliser une intégration de données complexe, là où l'`UPSERT` ne serait pas suffisant. Il faut toutefois avoir à l'esprit qu'une telle requête présente des problèmes de concurrences d'accès, pouvant entraîner des résultats inattendus si elle est employée alors que d'autres sessions modifient les données. On se contentera d'utiliser une telle requête dans des traitements batchs.

1.5.7 CTE récursive

- SQL permet d'exprimer des récursions
  - `WITH RECURSIVE`
- Utilité :
  - récupérer une arborescence de menu hiérarchique
  - parcourir des graphes (réseaux sociaux, etc.)

Le langage SQL permet de réaliser des récursions avec des CTE récursives. Son principal intérêt est de pouvoir parcourir des arborescences, comme par exemple des arbres généalogiques, des arborescences de service ou des entrées de menus hiérarchiques.

Il permet également de réaliser des parcours de graphes, mais les possibilités en SQL sont plus limitées de ce côté-là. En effet, SQL utilise un algorithme de type *Breadth First* (parcours en largeur) où PostgreSQL produit tout le niveau courant, et approfondit ensuite la récursion. Ce fonctionnement est à l'opposé d'un algorithme *Depth First* (parcours en profondeur) où chaque branche est explorée à fond individuellement avant de passer à la branche suivante. Ce principe de fonctionnement de l'implémentation dans SQL peut poser des problèmes sur des recherches de types réseaux sociaux où des bases de données orientées graphes, tel que Neo4J.

1.5.8 CTE récursive : exemple (1/2)

```sql
WITH RECURSIVE suite AS (
SELECT 1 AS valeur
UNION ALL
SELECT valeur + 1
  FROM suite
 WHERE valeur < 10
)
SELECT * FROM suite;
```

Voici le résultat de cette requête :

```
valeur
--------
       1
       2
       3
       4
       5
       6
       7
       8
       9
```

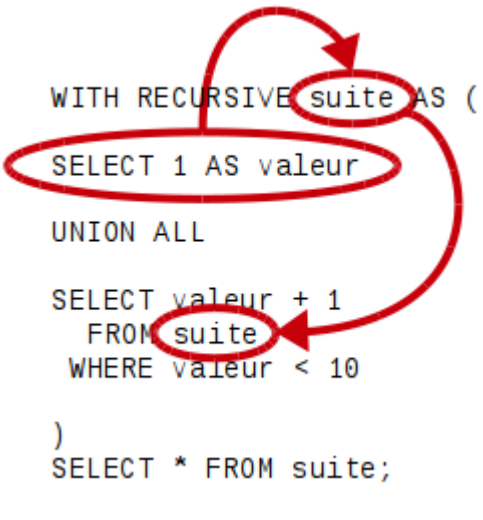
```
 10
```

L'exécution de cette requête commence avec le `SELECT 1 AS valeur` (la requête avant le `UNION ALL`), d'où la première ligne avec la valeur 1. Puis PostgreSQL exécute le `SELECT valeur + 1 FROM suite WHERE valeur < 10` tant que cette requête renvoie des lignes. À la première exécution, il additionne 1 avec la valeur précédente (1), ce qui fait qu'il renvoie 2. À la deuxième exécution, il additionne 1 avec la valeur précédente (2), ce qui fait qu'il renvoie 3. Etc. La récursivité s'arrête quand la requête ne renvoie plus de ligne, autrement dit quand la colonne vaut 10.

Cet exemple n'a aucun autre intérêt que de présenter la syntaxe permettant de réaliser une récursion en langage SQL.

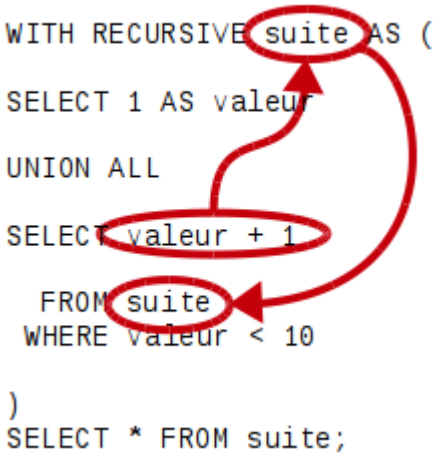
1.5.9 CTE récursive : principe

— 1ère étape : initialisation de la récursion



1.5.10 CTE récursive : principe

— récursion : la requête s'appelle elle-même



```
WITH RECURSIVE suite AS (

SELECT 1 AS valeur

UNION ALL

SELECT valeur + 1

  FROM suite
 WHERE valeur < 10

)
SELECT * FROM suite;
```

1.5.11 CTE récursive : exemple (2/2)

```
WITH RECURSIVE parcours_menu AS (
SELECT menu_id, libelle, parent_id,
       libelle AS arborescence
  FROM entrees_menu
 WHERE libelle = 'Terminal'
   AND parent_id IS NULL
UNION ALL
SELECT menu.menu_id, menu.libelle, menu.parent_id,
       arborescence || '/' || menu.libelle
  FROM entrees_menu menu
  JOIN parcours_menu parent
    ON (menu.parent_id = parent.menu_id)
)
SELECT * FROM parcours_menu;
```

Cet exemple suivant porte sur le parcours d'une arborescence de menu hiérarchique.

Une table `entrees_menu` est créée :

```
CREATE TABLE entrees_menu (menu_id serial primary key, libelle text not null,
                           parent_id integer);
```

Elle dispose du contenu suivant :

```
SELECT * FROM entrees_menu;
```

```
 menu_id |          libelle           | parent_id
---------+----------------------------+-----------
       1 | Fichier                    |
       2 | Edition                    |
       3 | Affichage                  |
       4 | Terminal                   |
       5 | Onglets                    |
       6 | Ouvrir un onglet           |         1
       7 | Ouvrir un terminal         |         1
       8 | Fermer l'onglet            |         1
       9 | Fermer la fenêtre          |         1
      10 | Copier                     |         2
      11 | Coller                     |         2
      12 | Préférences                |         2
      13 | Général                    |        12
      14 | Apparence                  |        12
      15 | Titre                      |        13
      16 | Commande                   |        13
      17 | Police                     |        14
      18 | Couleur                    |        14
      19 | Afficher la barre d'outils |         3
      20 | Plein écran                |         3
      21 | Modifier le titre          |         4
      22 | Définir l'encodage         |         4
      23 | Réinitialiser              |         4
      24 | UTF-8                      |        22
      25 | Europe occidentale         |        22
      26 | Europe centrale            |        22
      27 | ISO-8859-1                 |        25
      28 | ISO-8859-15                |        25
      29 | WINDOWS-1252               |        25
      30 | ISO-8859-2                 |        26
      31 | ISO-8859-3                 |        26
      32 | WINDOWS-1250               |        26
      33 | Onglet précédent           |         5
      34 | Onglet suivant             |         5
(34 rows)
```

Nous allons définir une CTE récursive qui va afficher l'arborescence du menu *Terminal*. La récursion va donc commencer par chercher la ligne correspondant à cette entrée de menu dans la table `entrees_menu`. Une colonne calculée `arborescence` est créée, elle servira plus tard dans la récursion :

```
SELECT menu_id, libelle, parent_id, libelle AS arborescence
  FROM entrees_menu
 WHERE libelle = 'Terminal'
   AND parent_id IS NULL
```

La requête qui réalisera la récursion est une jointure entre le résultat de l'itération précédente, obtenu par la vue `parcours_menu` de la CTE, qui réalisera une jointure avec la table `entrees_menu` sur la colonne `entrees_menu.parent_id` qui sera jointe à la colonne `menu_id` de l'itération précédente.

La condition d'arrêt de la récursion n'a pas besoin d'être exprimée. En effet, les entrées terminales des menus ne peuvent pas être jointes avec de nouvelles entrées de menu, car il n'y a pas d'autre

correspondance avec `parent_id`).

On obtient ainsi la requête CTE récursive présentée ci-dessus.

À titre d'exemple, voici l'implémentation du jeu des six degrés de Kevin Bacon en utilisant pgRouting :

```
WITH dijkstra AS (
SELECT seq, id1 AS node, id2 AS edge, cost
  FROM pgr_dijkstra('
  SELECT f.film_id AS id,
         f.actor_id::integer AS source,
         f2.actor_id::integer AS target,
         1.0::float8 AS cost
    FROM film_actor f
    JOIN film_actor f2
      ON (f.film_id = f2.film_id and f.actor_id <> f2.actor_id)'
  , 29539, 29726, false, false)
)
SELECT *
  FROM actors
  JOIN dijkstra
  on (dijkstra.node = actors.actor_id) ;

 actor_id |   actor_name    | seq | node  | edge | cost
----------+-----------------+-----+-------+------+------
    29539 | Kevin Bacon     |   0 | 29539 | 1330 |    1
    29625 | Robert De Niro  |   1 | 29625 |   53 |    1
    29726 | Al Pacino       |   2 | 29726 |   -1 |    0
(3 lignes)
```

1.6 CONCURRENCE D'ACCÈS

- Problèmes pouvant se poser :
  - `UPDATE` perdu
  - lecture non répétable
- Plusieurs solutions possibles
  - versionnement des lignes
  - `SELECT FOR UPDATE`
  - `SERIALIZABLE`

Plusieurs problèmes de concurrences d'accès peuvent se poser quand plusieurs transactions modifient les mêmes données en même temps.

Tout d'abord, des `UPDATE` peuvent être perdus, dans le cas où plusieurs transactions lisent la même ligne, puis la mettent à jour sans concertation. Par exemple, si la transaction 1 ouvre une transaction et effectue une lecture d'une ligne donnée :

```
BEGIN TRANSACTION;
SELECT * FROM employes WHERE matricule = '00000004';
```

La transaction 2 effectue les mêmes traitements :

```
BEGIN TRANSACTION;
SELECT * FROM employes WHERE matricule = '00000004';
```

Après un traitement applicatif, la transaction 1 met les données à jour pour noter l'augmentation de 5 % du salarié. La transaction est validée dans la foulée avec `COMMIT` :

```
UPDATE employes
   SET salaire = <valeur récupérée préalablement * 1.05>
 WHERE matricule = '00000004';
COMMIT;
```

Après un traitement applicatif, la transaction 2 met également les données à jour pour noter une augmentation exceptionnelle de 100 € :

```
UPDATE employes
   SET salaire = <valeur récupérée préalablement + 100>
 WHERE matricule = '00000004';
COMMIT;
```

Le salarié a normalement droit à son augmentation de 100 € ET l'augmentation de 5 %, or l'augmentation de 5 % a été perdue car écrasée par la transaction n°2. Ce problème aurait pu être évité de trois façons différentes :

- en effectuant un `UPDATE` utilisant la valeur lue par l'ordre `UPDATE` (`… SET salaire = salaire*1.05 WHERE …`) au lieu ;
- en verrouillant les données lues avec `SELECT FOR UPDATE` : ces verrous peuvent cependant ralentir l'application ;

— en utilisant le niveau d'isolation `SERIALIZABLE`.

La première solution n'est pas toujours envisageable, il faut donc se tourner vers les deux autres solutions.

Le problème des lectures sales (*dirty reads*) ne peut pas se poser car PostgreSQL n'implémente pas le niveau d'isolation `READ UNCOMMITTED`. Si ce niveau d'isolation est sélectionné, PostgreSQL utilise alors le niveau `READ COMMITTED`.

1.6.1 SELECT FOR UPDATE

— `SELECT FOR UPDATE`
— Utilité :
  — « réserver » des lignes en vue de leur mise à jour
  — éviter les problèmes de concurrence d'accès

L'ordre `SELECT FOR UPDATE` permet de lire des lignes tout en les réservant en posant un verrou dessus en vue d'une future mise à jour. Le verrou n'interdira pas la lecture par une autre session, mais mettra toute mise à jour en attente.

Reprenons l'exemple précédent et utilisons `SELECT FOR UPDATE` pour voir si le problème de concurrence d'accès peut être résolu.

Session 1 :

```
BEGIN TRANSACTION;
SELECT * FROM employes WHERE matricule = '00000004'
FOR UPDATE;
```

```
 matricule |   nom    | service  | salaire
-----------+----------+----------+---------
 00000004  | Fantasio | Courrier | 4500.00
(1 row)
```

La requête `SELECT` a retourné les données souhaitées.

Session 2 :

```
BEGIN TRANSACTION;
SELECT * FROM employes WHERE matricule = '00000004'
FOR UPDATE;
```

La requête SELECT ne rend pas la main, elle est mise en attente.

Session 3 :

Une troisième session effectue une lecture, sans poser de verrou explicite :

```
SELECT * FROM employes WHERE matricule = '00000004';
```

```
 matricule |   nom    | service  | salaire
-----------+----------+----------+---------
 00000004  | Fantasio | Courrier | 4500.00
(1 row)
```

Le `SELECT` n'a pas été bloqué par la session 1. Seule la session 2 est bloquée car elle tente d'obtenir le même verrou.

Session 1 :

L'application a effectué ses calculs et met à jour les données en appliquant l'augmentation de 5 % :

```
UPDATE employes
   SET salaire = 4725
 WHERE matricule = '00000004';
```

Les données sont vérifiées :

```
SELECT * FROM employes WHERE matricule = '00000004';
```

```
 matricule |   nom    | service  | salaire
-----------+----------+----------+---------
 00000004  | Fantasio | Courrier | 4725.00
(1 row)
```

Enfin, la transaction est validée :

```
COMMIT;
```

Session 2 :

La session 2 a rendu la main, le temps d'attente a été important pour réaliser ces calculs complexes :

```
 matricule |   nom    | service  | salaire
-----------+----------+----------+---------
 00000004  | Fantasio | Courrier | 4725.00
(1 row)

Time: 128127,105 ms
```

Le salaire obtenu est bien le salaire mis à jour par la session 1. Sur cette base, l'application applique l'augmentation de 100 € :

```
UPDATE employes
   SET salaire = 4825.00
 WHERE matricule = '00000004';

SELECT * FROM employes WHERE matricule = '00000004';
```

```
 matricule |   nom    | service  | salaire
-----------+----------+----------+---------
 00000004  | Fantasio | Courrier | 4825.00
```

La transaction est validée :

```
COMMIT;
```

Les deux transactions ont donc été effectuée de manière sérialisée, l'augmentation de 100 € ET l'augmentation de 5 % ont été accordées à Fantasio. En contrepartie, l'une des deux transactions concurrentes a été mise en attente afin de pouvoir sérialiser les transactions. Cela implique de penser les traitements en verrouillant les ressources auxquelles on souhaite accéder.

L'ordre `SELECT FOR UPDATE` dispose également d'une option `NOWAIT` qui permet d'annuler la transaction courante si un verrou ne pouvait être acquis. Si l'on reprend les premières étapes de l'exemple précédent :

Session 1 :

```
BEGIN TRANSACTION;
SELECT * FROM employes WHERE matricule = '00000004'
FOR UPDATE NOWAIT;

 matricule |   nom    | service  | salaire
-----------+----------+----------+---------
 00000004  | Fantasio | Courrier | 4500.00
(1 row)
```

Aucun verrou préalable n'avait été posé, la requête `SELECT` a retourné les données souhaitées.

Session 2 :

On effectue la même chose sur la session n°2 :

```
BEGIN TRANSACTION;
SELECT * FROM employes WHERE matricule = '00000004'
FOR UPDATE NOWAIT;

ERROR:  could not obtain lock on row in relation "employes"
```

Comme la session n°1 possède déjà un verrou sur la ligne qui nous intéresse, l'option `NOWAIT` sur le `SELECT` a annulé la transaction.

Une application bien conçue interceptera l'erreur et décidera d'abandonner ou recommencer plus tard la mise à jour, au lieu d'attendre, peut-être très longtemps, que le verrou soit levé.

Il faut maintenant effectuer un `ROLLBACK` explicite pour pouvoir recommencer les traitements au risque d'obtenir le message suivant :

```
ERROR:  current transaction is aborted, commands ignored until
        end of transaction block
```

1.6.2 SKIP LOCKED

- `SELECT FOR UPDATE SKIP LOCKED`
- Utilité :
  - files d'attentes parallélisables

Une dernière fonctionnalité intéressante de `SELECT FOR UPDATE` est idéal dans le cas de différents *workers* qui consomment des tâches d'une table contenant une file d'attente. Il s'agit de la clause `SKIP LOCKED`.

En prenant une table représentant la file d'attente suivante, peuplée avec des données générées :

```
CREATE TABLE liste_taches_a_faire (id serial primary key, val text);
-- Données
INSERT INTO liste_taches_a_faire (val) SELECT md5(i::text)
FROM generate_series(1, 1000) i;
```

Une première transaction est ouverte et tente d'obtenir un verrou sur les 10 premières lignes :

```
BEGIN TRANSACTION;

SELECT *
FROM liste_taches_a_faire
LIMIT 10
FOR UPDATE SKIP LOCKED;
```

```
 id |               val
----+----------------------------------
  1 | c4ca4238a0b923820dcc509a6f75849b
  2 | c81e728d9d4c2f636f067f89cc14862c
  3 | eccbc87e4b5ce2fe28308fd9f2a7baf3
  4 | a87ff679a2f3e71d9181a67b7542122c
  5 | e4da3b7fbbce2345d7772b0674a318d5
  6 | 1679091c5a880faf6fb5e6087eb1b2dc
  7 | 8f14e45fceea167a5a36dedd4bea2543
  8 | c9f0f895fb98ab9159f51fd0297e236d
  9 | 45c48cce2e2d7fbdea1afc51c7c6ad26
 10 | d3d9446802a44259755d38e6d163e820
(10 rows)
```

La session va s'occuper ensuite de ces lignes avant de rendre la main, ce qui peut prendre plus ou moins de temps.

Si on démarre une seconde transaction en parallèle, avec la première transaction toujours ouverte, le fait d'exécuter la requête `SELECT FOR UPDATE` sans la clause `SKIP LOCKED` aurait pour effet de la mettre en attente. Le deuxième `SELECT` ne rendra la main lorsque la transaction n°1 se terminera, et s'apercevra peut-être ensuite que les lignes ont déjà été traitées par la première session.

Avec la clause `SKIP LOCKED`, les 10 premières verrouillées par la transaction n°1 seront ignorées puisque déjà verrouillées, et ce sont les 10 lignes suivantes qui seront selectionnées, verrouillées et retournées par l'ordre `SELECT` :

```
BEGIN TRANSACTION;

SELECT *
FROM liste_taches_a_faire
LIMIT 10
FOR UPDATE SKIP LOCKED;
```

```
 id |               val
----+----------------------------------
```

```
 11 | 6512bd43d9caa6e02c990b0a82652dca
 12 | c20ad4d76fe97759aa27a0c99bff6710
 13 | c51ce410c124a10e0db5e4b97fc2af39
 14 | aab3238922bcc25a6f606eb525ffdc56
 15 | 9bf31c7ff062936a96d3c8bd1f8f2ff3
 16 | c74d97b01eae257e44aa9d5bade97baf
 17 | 70efdf2ec9b086079795c442636b55fb
 18 | 6f4922f45568161a8cdf4ad2299f6d23
 19 | 1f0e3dad99908345f7439f8ffabdffc4
 20 | 98f13708210194c475687be6106a3b84
(10 rows)
```

Ensuite, la première transaction supprime les lignes verrouillées et valide la transaction :

```
DELETE FROM liste_taches_a_faire
 WHERE id IN (…);
COMMIT;
```

De même pour la seconde transaction, qui aura traité d'autres lignes en parallèle de la transaction n°1.

1.7 SERIALIZABLE SNAPSHOT ISOLATION

SSI : Serializable Snapshot Isolation
- Chaque transaction est seule sur la base
- Si on ne peut maintenir l'illusion
  - une des transactions en cours est annulée
- Sans blocage
- On doit être capable de rejouer la transaction
- Toutes les transactions impliquées doivent être `serializable`
- `default_transaction_isolation=serializable` dans la configuration

PostgreSQL fournit plusieurs modes d'isolation, pour un bon compromis entre fiabilité et performances. Par défaut, en mode `READ COMMITTED`, une transaction en cours d'exécution peut voir les modifications committés par d'autres transactions parallèles. En mode `REPEATABLE READ`, une transaction ne voit que les données telles qu'au début de la transaction, et ses propres modifications.

PostgreSQL fournit un autre mode d'isolation appelé `SERIALIZABLE`. Dans ce mode, toutes les transactions déclarées comme telles s'exécutent comme si elles étaient seules sur la base. Dès que cette garantie ne peut plus être apportée, une des transactions est annulée.

L'intérêt est par exemple de pouvoir garantir que certaines validations touchant d'autres tables pourront s'effectuer sans *race conditions*, et ce sans poser des verrous trop gênants. L'application doit bien sûr être prête à ce que ses transactions échouent souvent, et à retenter ensuite.

Toute transaction non déclarée comme `SERIALIZABLE` peut en théorie s'exécuter n'importe quand, ce qui rend inutile le mode `SERIALIZABLE` sur les autres. C'est donc un mode qui doit être mis en place sur un domaine assez large.

Dans l'exemple suivant, des enregistrements avec une colonne couleur contiennent 'blanc' ou 'rouge'. Deux utilisateurs essayent simultanément de convertir tous les enregistrements vers une couleur unique, mais chacun dans une direction opposée. Un utilisateur veut passer tous les blancs en rouge, et l'autre tous les rouges en blanc.

L'exemple peut être mis en place avec ces ordres :

```
create table points
  (
    id int not null primary key,
    couleur text not null
  );
insert into points
  with x(id) as (select generate_series(1,10))
  select id, case when id % 2 = 1 then 'rouge'
    else 'blanc' end from x;
```

Session 1 :

```
set default_transaction_isolation = 'serializable';
```

```
begin;
update points set couleur = 'rouge'
where couleur = 'blanc';
```

Session 2 :

```
set default_transaction_isolation = 'serializable';
begin;
update points set couleur = 'blanc'
where couleur = 'rouge';
```

À ce moment, une des deux transaction est condamnée à mourir.

Session 2 :

```
COMMIT;
```

Le premier à valider gagne.

```
  select * from points order by id;

  id | couleur
 ----+-------
   1 | blanc
   2 | blanc
   3 | blanc
   4 | blanc
   5 | blanc
   6 | blanc
   7 | blanc
   8 | blanc
   9 | blanc
  10 | blanc
 (10 rows)
```

Session 1 :

Celle-ci s'est exécutée comme si elle était seule.

```
COMMIT ;

 ERROR:  could not serialize access
         due to read/write dependencies
         among transactions
 DETAIL:  Cancelled on identification
          as a pivot, during commit attempt.
 HINT:  The transaction might succeed if retried.
```

Une erreur de sérialisation. On annule et on réessaye.

```
rollback;
 begin;
 update points set couleur = 'rouge'
   where couleur = 'blanc';
 commit;
```

Il n'y a pas de transaction concurrente pour gêner.

```
select * from points order by id;
```

```
 id | couleur
----+-------
  1 | rouge
  2 | rouge
  3 | rouge
  4 | rouge
  5 | rouge
  6 | rouge
  7 | rouge
  8 | rouge
  9 | rouge
 10 | rouge
(10 rows)
```

La transaction s'est exécutée seule, après l'autre.

Le mode `SERIALIZABLE` permet de s'affranchir des `SELECT FOR UPDATE` qu'on écrit habituellement, dans les applications en mode `READ COMMITTED`. Toutefois, il fait bien plus que ça, puisqu'il réalise du verrouillage de prédicats. Un enregistrement qui « apparaît » ultérieurement suite à une mise à jour réalisée par une transaction concurrente déclenchera aussi une erreur de sérialisation. Il permet aussi de gérer les problèmes ci-dessus avec plus de deux sessions.

Pour des exemples plus complets, le mieux est de consulter la documentation officielle[8].

[8] https://wiki.postgresql.org/wiki/SSI/fr

1.8 CONCLUSION

- SQL est un langage très riche
- Connaître les nouveautés des versions de la norme depuis 20 ans permet de
  - gagner énormément de temps de développemment
  - mais aussi de performance

1.9 TRAVAUX PRATIQUES

La version en ligne des solutions de ces TP est disponible sur https://dali.bo/s60_solutions.

Jointure latérale

Cette série de question utilise la base de TP **magasin**. La base **magasin** (dump de 96 Mo, pour 667 Mo sur le disque au final) peut être téléchargée et restaurée comme suit dans une nouvelle base **magasin** :

```
createdb magasin
curl -kL https://dali.bo/tp_magasin -o /tmp/magasin.dump
pg_restore -d magasin  /tmp/magasin.dump
# le message sur public préexistant est normal
rm -- /tmp/magasin.dump
```

Les données sont dans deux schémas, **magasin** et **facturation**. Penser au `search_path`.

Pour ce TP, figer les paramètres suivants :

```
SET max_parallel_workers_per_gather to 0;
SET seq_page_cost TO 1 ;
SET random_page_cost TO 4 ;
```

> Afficher les 10 derniers articles commandés.

> Pour chacune des 10 dernières commandes passées, afficher le premier article commandé.

CTE récursive

La table `genealogie` peut être téléchargée depuis https://dali.bo/tp_genealogie et restaurée à l'aide de `pg_restore` :

```
curl -kL https://dali.bo/tp_genealogie  -o genealogie.dump
createdb genealogie
pg_restore -O -d genealogie genealogie.dump
# le message d'erreur sur le schéma public est normal
```

Voici la description de la table `genealogie` qui sera utilisée :

```
\d genealogie
                          Table "public.genealogie"
     Column     |  Type   |                Modifiers
----------------+---------+-----------------------------------------
 id             | integer | not null default                       +
                |         | nextval('genealogie_id_seq'::regclass)
 nom            | text    |
 prenom         | text    |
 date_naissance | date    |
 pere           | integer |
 mere           | integer |
Indexes:
    "genealogie_pkey" PRIMARY KEY, btree (id)
```

> À partir de la table `genealogie`, déterminer qui sont les descendants de Fernand DEVAUX.

> À l'inverse, déterminer qui sont les ancètres de Adèle TAILLANDIER

Réseau social

La table `socialnet` peut être téléchargée et restaurée ainsi :

```
curl -kL https://dali.bo/tp_socialnet -o /tmp/socialnet.dump
createdb socialnet
pg_restore -O -d socialnet /tmp/socialnet.dump
# le message d'erreur sur le schéma public est normal
```

Cet exercice est assez similaire au précédent et propose de manipuler des arborescences.

> Les tableaux et la fonction `unnest()` peuvent être utiles pour résoudre plus facilement ce problème.

La table `personnes` contient la liste de toutes les personnes d'un réseau social.

```
                        Table "public.personnes"
 Column |  Type   |                       Modifiers
--------+---------+-------------------------------------------------------
 id     | integer | not null default nextval('personnes_id_seq'::regclass)
 nom    | text    | not null
 prenom | text    | not null
Indexes:
    "personnes_pkey" PRIMARY KEY, btree (id)
```

La table `relation` contient les connexions entre ces personnes.

```
   Table "public.relation"
 Column |  Type   | Modifiers
--------+---------+-----------
 gauche | integer | not null
 droite | integer | not null
Indexes:
    "relation_droite_idx" btree (droite)
    "relation_gauche_idx" btree (gauche)
```

> Déterminer le niveau de connexions entre Sadry Luettgen et Yelsi Kerluke et afficher le chemin de relation le plus court qui permet de les connecter ensemble.

Dépendance de vues

Les dépendances entre objets est un problème classique dans les bases de données :

— dans quel ordre charger des tables selon les clés étrangères ?
— dans quel ordre recréer des vues ?

— etc.

Le catalogue de PostgreSQL décrit l'ensemble des objets de la base de données. Deux tables vont nous intéresser pour mener à bien cet exercice :

— `pg_depend` liste les dépendances entre objets
— `pg_rewrite` stocke les définitions des règles de réécritures des vues (RULES)
— `pg_class` liste les objets que l'on peut interroger comme une table, hormis les fonctions retournant des ensembles

La définition d'une vue peut être obtenue à l'aide de la fonction `pg_get_viewdef`.

Pour plus d'informations sur ces tables, se référer à la documentation :

— Catalogue pg_depend[9]
— Catalogue pg_rewrite[10]
— Catalogue pg_class[11]
— Fonction d'information du catalogue système[12]

L'objectif de se TP consiste à récupérer l'ordre de suppression et de recréation des vues de la base `brno2015` en fonction du niveau de dépendances entre chacune des vues. Brno est une ville de Tchéquie, dans la région de Moravie-du-Sud. Le circuit Brno-Masaryk est situé au nord-ouest de la ville. Le Grand Prix moto de Tchéquie s'y déroule chaque année.

La table `brno2015` peut être téléchargée et restaurée ainsi :

```
curl -kL https://dali.bo/tp_brno2015 -o /tmp/brno2015.dump
createdb  brno2015
pg_restore -O -d brno2015 /tmp/brno2015.dump
# une erreur sur l'existence du schéma public est normale
```

> Retrouver les dépendances de la vue `pilotes_brno`. Déduisez également l'ordre de suppression et de recréation des vues.

[9] https://www.postgresql.org/docs/current/static/catalog-pg-depend.html
[10] https://www.postgresql.org/docs/current/static/catalog-pg-rewrite.html
[11] https://www.postgresql.org/docs/current/static/catalog-pg-class.html
[12] https://www.postgresql.org/docs/current/static/functions-info.html#FUNCTIONS-INFO-CATALOG-TABLE

1.10 TRAVAUX PRATIQUES (SOLUTIONS)

Jointure latérale

> Afficher les 10 derniers articles commandés.

Tout d'abord, nous positionnons le `search_path` pour chercher les objets du schéma **magasin** :

```
SET search_path = magasin;
```

On commence par afficher les 10 dernières commandes :

```
SELECT *
  FROM commandes
 ORDER BY numero_commande DESC
 LIMIT 10;
```

Une simple jointure nous permet de retrouver les 10 derniers articles commandés :

```
SELECT lc.produit_id, p.nom
  FROM commandes c
  JOIN lignes_commandes lc
    ON (c.numero_commande = lc.numero_commande)
  JOIN produits p
    ON (lc.produit_id = p.produit_id)
 ORDER BY c.numero_commande DESC, numero_ligne_commande DESC
 LIMIT 10;
```

> Pour chacune des 10 dernières commandes passées, afficher le premier article commandé.

La requête précédente peut être dérivée pour répondre à la question demandée. Ici, pour chacune des dix dernières commandes, nous voulons récupérer le nom du dernier article commandé, ce qui sera transcrit sous la forme d'une jointure latérale :

```
SELECT numero_commande, produit_id, nom
  FROM commandes c,
  LATERAL (SELECT p.produit_id, p.nom
             FROM lignes_commandes lc
             JOIN produits p
               ON (lc.produit_id = p.produit_id)
            WHERE (c.numero_commande = lc.numero_commande)
            ORDER BY numero_ligne_commande ASC
            LIMIT 1
 ) premier_article_par_commande
 ORDER BY c.numero_commande DESC
 LIMIT 10;
```

CTE récursive

> À partir de la table `genealogie`, déterminer qui sont les descendants de Fernand DEVAUX.

```
WITH RECURSIVE arbre_genealogique AS (
SELECT id, nom, prenom, date_naissance, pere, mere
```

```
  FROM genealogie
 WHERE nom = 'DEVAUX'
   AND prenom = 'Fernand'
UNION ALL
SELECT g.*
  FROM arbre_genealogique ancetre
  JOIN genealogie g
    ON (g.pere = ancetre.id OR g.mere = ancetre.id)
)
SELECT id, nom, prenom, date_naissance
  FROM arbre_genealogique;
```

> À l'inverse, déterminer qui sont les ancètres de Adèle TAILLANDIER

```
WITH RECURSIVE arbre_genealogique AS (
SELECT id, nom, prenom, date_naissance, pere, mere
  FROM genealogie
 WHERE nom = 'TAILLANDIER'
   AND prenom = 'Adèle'
UNION ALL
SELECT ancetre.id, ancetre.nom, ancetre.prenom, ancetre.date_naissance,
       ancetre.pere, ancetre.mere
  FROM arbre_genealogique descendant
  JOIN genealogie ancetre
    ON (descendant.pere = ancetre.id OR descendant.mere = ancetre.id)
)
SELECT id, nom, prenom, date_naissance
  FROM arbre_genealogique;
```

Réseau social

> Déterminer le niveau de connexions entre Sadry Luettgen et Yelsi Kerluke et afficher le chemin de relation le plus court qui permet de les connecter ensemble.

La requête suivante permet de répondre à cette question :

```
WITH RECURSIVE connexions AS (
SELECT gauche, droite, ARRAY[gauche] AS personnes_connectees,0::integer AS level
  FROM relation
 WHERE gauche = 1
UNION ALL
SELECT p.gauche, p.droite, personnes_connectees || p.gauche, level + 1 AS level
  FROM connexions c
  JOIN relation p ON (c.droite = p.gauche)
 WHERE level < 4
   AND p.gauche <> ANY (personnes_connectees)
), plus_courte_connexion AS (
SELECT *
  FROM connexions
 WHERE gauche = (
    SELECT id FROM personnes WHERE nom = 'Kerluke' AND prenom = 'Yelsi'
 )
 ORDER BY level ASC
 LIMIT 1
)
```

```
SELECT list.id, p.nom, p.prenom, list.level - 1 AS level
  FROM plus_courte_connexion,
       unnest(personnes_connectees) WITH ORDINALITY AS list(id, level)
  JOIN personnes p on (list.id = p.id)
 ORDER BY list.level;
```

Cet exemple fonctionne sur une faible volumétrie, mais les limites des bases relationnelles sont rapidement atteintes sur de telles requêtes.

Une solution consisterait à implémenter un algorithme de parcours de graphe avec pgRouting[13], mais cela nécessitera de présenter les données sous une forme particulière. Pour les problématiques de traitement de graphe, notamment sur de grosses volumétries, une base de données orientée graphe comme Neo4J sera probablement plus adaptée.

Dépendance de vues

> Retrouver les dépendances de la vue `pilotes_brno`. Déduisez également l'ordre de suppression et de recréation des vues.

Tout d'abord, nous positionnons le `search_path` pour chercher les objets du schéma **brno2015** :

```
SET search_path = brno2015;
```

Si la jointure entre `pg_depend` et `pg_rewrite` est possible pour l'objet de départ, alors il s'agit probablement d'une vue. En discriminant sur les objets qui référencent la vue `pilotes_brno`, nous arrivons à la requête de départ suivante :

```
SELECT DISTINCT pg_rewrite.ev_class as objid, refobjid as refobjid, 0 as depth
  FROM pg_depend
  JOIN pg_rewrite ON pg_depend.objid = pg_rewrite.oid
  WHERE refobjid = 'pilotes_brno'::regclass;
```

La présence de doublons nous oblige à utiliser la clause DISTINCT.

Nous pouvons donc créer un graphe de dépendances à partir de cette requête de départ, transformée en requête récursive :

```
WITH RECURSIVE graph AS (
  SELECT distinct pg_rewrite.ev_class as objid, refobjid as refobjid, 0 as depth
  FROM pg_depend
  JOIN pg_rewrite ON pg_depend.objid = pg_rewrite.oid
  WHERE refobjid = 'pilotes_brno'::regclass
  UNION ALL
  SELECT distinct pg_rewrite.ev_class as objid, pg_depend.refobjid as refobjid,
                  depth + 1 as depth
  FROM pg_depend
  JOIN pg_rewrite ON pg_depend.objid = pg_rewrite.oid
  JOIN graph on pg_depend.refobjid = graph.objid
  WHERE pg_rewrite.ev_class != graph.objid
```

[13] https://docs.pgrouting.org/2.0/fr/src/kdijkstra/doc/index.html#pgr-kdijkstra

```
)
SELECT * FROM graph;
```

Il faut maintenant résoudre les OID pour déterminer les noms des vues et leur schéma. Pour cela, nous ajoutons une vue `resolved` telle que :

```
WITH RECURSIVE graph AS (
  SELECT distinct pg_rewrite.ev_class as objid, refobjid as refobjid, 0 as depth
  FROM pg_depend
  JOIN pg_rewrite ON pg_depend.objid = pg_rewrite.oid
  WHERE refobjid = 'pilotes_brno'::regclass
  UNION ALL
  SELECT distinct pg_rewrite.ev_class as objid, pg_depend.refobjid as refobjid,
                  depth + 1 as depth
  FROM pg_depend
  JOIN pg_rewrite ON pg_depend.objid = pg_rewrite.oid
  JOIN graph on pg_depend.refobjid = graph.objid
  WHERE pg_rewrite.ev_class != graph.objid
),
resolved AS (
  SELECT n.nspname AS dependent_schema, d.relname as dependent,
         n2.nspname AS dependee_schema, d2.relname as dependee,
         depth
  FROM graph
  JOIN pg_class d ON d.oid = objid
  JOIN pg_namespace n ON d.relnamespace = n.oid
  JOIN pg_class d2 ON d2.oid = refobjid
  JOIN pg_namespace n2 ON d2.relnamespace = n2.oid
)
SELECT * FROM resolved;
```

Nous pouvons maintenant présenter les ordres de suppression et de recréation des vues, dans le bon ordre. Les vues doivent être supprimées selon le numéro d'ordre décroissant et recrées selon le numéro d'ordre croissant :

```
WITH RECURSIVE graph AS (
  SELECT distinct pg_rewrite.ev_class as objid, refobjid as refobjid, 0 as depth
  FROM pg_depend
  JOIN pg_rewrite ON pg_depend.objid = pg_rewrite.oid
  WHERE refobjid = 'pilotes_brno'::regclass
  UNION ALL
  SELECT distinct pg_rewrite.ev_class as objid, pg_depend.refobjid as refobjid,
                  depth + 1 as depth
  FROM pg_depend
  JOIN pg_rewrite ON pg_depend.objid = pg_rewrite.oid
  JOIN graph on pg_depend.refobjid = graph.objid
  WHERE pg_rewrite.ev_class != graph.objid
),
resolved AS (
  SELECT n.nspname AS dependent_schema, d.relname as dependent,
         n2.nspname AS dependee_schema, d2.relname as dependee,
         d.oid as dependent_oid,
         depth
  FROM graph
  JOIN pg_class d ON d.oid = objid
  JOIN pg_namespace n ON d.relnamespace = n.oid
```

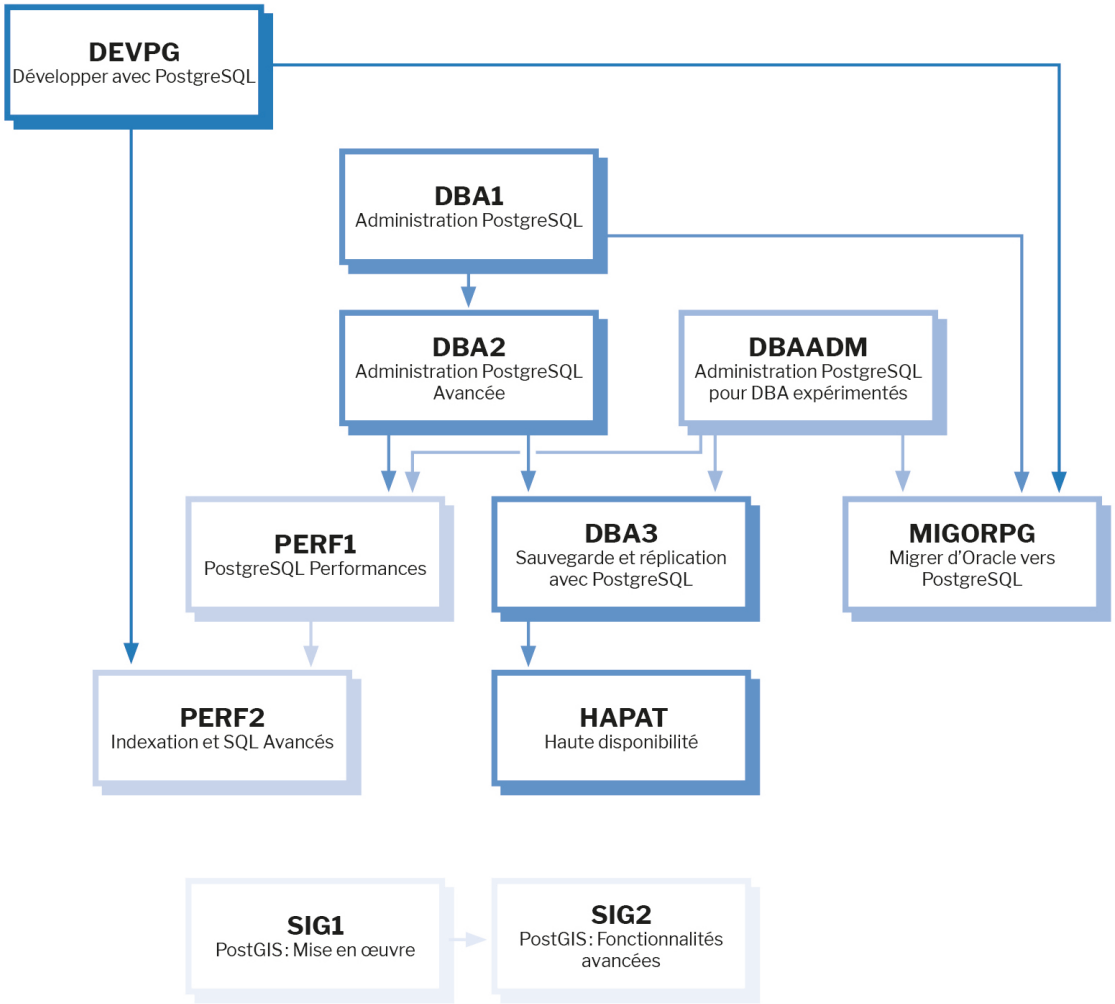
```
  JOIN pg_class d2 ON d2.oid = refobjid
  JOIN pg_namespace n2 ON d2.relnamespace = n2.oid
)
(SELECT 'DROP VIEW ' || dependent_schema || '.' || dependent || ';'
   FROM resolved
  GROUP BY dependent_schema, dependent
  ORDER BY max(depth) DESC)
UNION ALL
(SELECT 'CREATE OR REPLACE VIEW ' || dependent_schema || '.' || dependent ||
        ' AS ' || pg_get_viewdef(dependent_oid)
   FROM resolved
  GROUP BY dependent_schema, dependent, dependent_oid
  ORDER BY max(depth));
```

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
— DEV1 : Introduction à SQL
https://dali.bo/dev1
— 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.