

Module S5

SQL avancé pour le transactionnel



22.09

Dalibo SCOP

<https://dalibo.com/formations>

SQL avancé pour le transactionnel

Module S5

TITRE : SQL avancé pour le transactionnel

SOUS-TITRE : Module S5

REVISION: 22.09

DATE: 02 septembre 2022

COPYRIGHT: © 2005-2022 DALIBO SARL SCOP

LICENCE: Creative Commons BY-NC-SA

Postgres®, PostgreSQL® and the Slonik Logo are trademarks or registered trademarks of the PostgreSQL Community Association of Canada, and used with their permission.

(Les noms PostgreSQL® et Postgres®, et le logo Slonik sont des marques déposées par PostgreSQL Community Association of Canada.

Voir <https://www.postgresql.org/about/policies/trademarks/>)

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, Benoît Lobréau, Jean-Louis Louër, Thibaut Madelaine, Adrien Nayrat, Alexandre Pereira, Flavie Perette, Robin Portigliatti, Thomas Reiss, Maël Rimbault, Julien Rouhaud, Stéphane Schildknecht, Julien Tachaires, Nicolas Thauvin, Be Hai Tran, Christophe Truffier, Cédric Villemain, Thibaud Walkowiak, Frédéric Yhuel.

À propos de DALIBO : DALIBO est le spécialiste français de PostgreSQL. Nous proposons du support, de la formation et du conseil depuis 2005. Retrouvez toutes nos formations sur <https://dalibo.com/formations>

LICENCE CREATIVE COMMONS BY-NC-SA 2.0 FR

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

Vous êtes autorisé 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.com¹ !

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

Licence Creative Commons BY-NC-SA 2.0 FR	5
1 SQL avancé pour le transactionnel	10
1.1 LIMIT	11
1.2 RETURNING	16
1.3 UPSERT	18
1.4 LATERAL	24
1.5 Common Table Expressions	28
1.6 Concurrence d'accès	41
1.7 Serializable Snapshot Isolation	47
1.8 Conclusion	49
1.9 Travaux pratiques	50
1.10 Travaux pratiques (solutions)	53

1 SQL AVANCÉ POUR LE TRANSACTIONNEL

1.0.1 PRÉAMBULE

- SQL et PostgreSQL proposent de nombreuses possibilités 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

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 normalisée vient de DB2 d'IBM et va être amenée à apparaître sur la plupart des bases de données. La syntaxe **LIMIT** reste néanmoins disponible sur de nombreux SGBD 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 le jeu d'essai suivant :

```
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)

SQL avancé pour le transactionnel

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 employees ;
```

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 employees
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 employees
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
- [Article sur le sujet^a](#)

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'exécution²](#) suivant :

```
QUERY PLAN
-----
Limit  (cost=0.43..18.26 rows=10 width=115)
  (actual time=0.043..0.053 rows=10 loops=1)
```

^a<https://use-the-index-luke.com/fr/no-offset>

²<https://explain.dalibo.com/plan/xEs>

```

-> 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**³ 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](#)⁴.

Dans notre cas, le principe est d'abord de 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

³<https://explain.dalibo.com/plan/V05>

⁴<https://use-the-index-luke.com/fr/no-offset>

SQL avancé pour le transactionnel

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. Ainsi, la clause **RETURNING** associée à l'ordre **INSERT** permet d'obtenir une ou plusieurs colonnes des lignes insérées.

1.2.1 RETURNING : EXEMPLE

```
CREATE TABLE test_returning (id serial primary key, val integer);
```

```
INSERT INTO test_returning (val)
```

```
VALUES (10)
```

```
RETURNING id, val;
```

```
id | val
```

```
---+---
```

```
1 | 10
```

```
(1 ligne)
```

Cela permet par exemple de récupérer la valeur de colonnes portant une valeur par défaut, comme la valeur affectée par une séquence, comme sur l'exemple ci-dessus.

La clause **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
```

```
(1 ligne)
```

Associée à l'ordre **DELETE**, il est possible d'obtenir les lignes supprimées :

```
DELETE FROM test_returning
```

```
WHERE val < 30
```

```
RETURNING id, val;
```

```
id | val
```

```
---+---
```

```
1 | 20
```

```
(1 ligne)
```

1.3 UPSERT

- **INSERT** ou **UPDATE** ?
 - `INSERT ... ON CONFLICT DO { NOTHING | UPDATE }`
 - à partir de la version 9.5
- Utilité :
 - mettre à jour en cas de conflit sur un **INSERT**
 - ne rien faire en cas de conflit sur un **INSERT**

L'implémentation de l'UPSERT peut poser des questions sur la concurrence d'accès. L'implémentation de PostgreSQL de `ON CONFLICT DO UPDATE` est une opération atomique, c'est-à-dire que PostgreSQL garantit qu'il n'y aura pas de conditions d'exécution qui pourront amener à des erreurs. L'utilisation d'une contrainte d'unicité n'est pas étrangère à cela, elle permet en effet de pouvoir vérifier que la ligne n'existe pas, et si elle existe déjà, de verrouiller la ligne à mettre à jour de façon atomique.

En comparaison, plusieurs approches naïves présentent des problèmes de concurrences d'accès. Les différentes approches sont décrites dans [cet article de depezsz](#)⁵. Elle présente toutes des problèmes de *race conditions* qui peuvent entraîner des erreurs. Une autre possibilité aurait été d'utiliser une CTE en écriture, mais elle présente également les problèmes de concurrence d'accès décrits dans l'article.

Sur des traitements d'intégration de données, il s'agit d'un comportement qui n'est pas toujours souhaitable. La norme SQL propose l'ordre **MERGE** pour palier à des problèmes de ce type, mais il est peu probable de le voir rapidement implémenté dans PostgreSQL⁶. L'ordre **INSERT** s'est toutefois vu étendu avec PostgreSQL 9.5 pour gérer les conflits à l'insertion.

Les exemples suivants s'appuient sur le jeu de données suivant :

```
\d employees
      Table "public.employees"
  Column |      Type      | Modifiers
-----+-----+-----
 matricule | character(8) | not null
    nom    | text         | not null
  service | text         |
  salaire | numeric(7,2) |
```

Indexes:

⁵<https://www.depsz.com/2012/06/10/why-is-upsert-so-complicated/>

⁶La solution actuelle semble techniquement meilleure et la solution actuelle a donc été choisie. Le wiki du projet PostgreSQL montre que l'ordre **MERGE** a été étudié et qu'un certain nombre d'aspects cruciaux n'ont pas été spécifiés, amenant le projet PostgreSQL à utiliser sa propre version. Voir la documentation : https://wiki.postgresql.org/wiki/UPSERT#MERGE_disadvantages.

```
"employees_pkey" PRIMARY KEY, btree (matricule)
```

```
SELECT * FROM employees ;
```

matricule	nom	service	salaire
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

1.3.1 UPSERT : PROBLÈME À RÉSOUDRE

- Insérer une ligne déjà existante provoque une erreur :

```
INSERT INTO employees (matricule, nom, service, salaire)
VALUES ('00000001', 'Marsupilami', 'Direction', 50000.00);
ERROR: duplicate key value violates unique constraint
        "employees_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.

1.3.2 ON CONFLICT DO NOTHING

- la clause **ON CONFLICT DO NOTHING** évite d'insérer une ligne existante :

```
== INSERT INTO employees (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 employees ;
```

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)

```

SQL avancé pour le transactionnel

```
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 tables des employés, 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 = employees.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 la 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](#)⁷

1.4 LATERAL

- Jointures **LATERAL**
 - SQL:99
 - PostgreSQL 9.3
 - équivalent d'une boucle **foreach**
- Utilisations
 - top-N à partir de plusieurs tables
 - jointure avec une fonction retournant un ensemble

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.

1.4.1 LATERAL : AVEC UNE SOUS-REQUÊTE

- Jointure **LATERAL**
 - équivalent de *foreach*
- Utilité :
 - Top-N à partir de plusieurs tables
 - exemple : *afficher les 5 derniers messages des 5 derniers sujets actifs d'un forum*

La clause **LATERAL** existe dans la norme SQL depuis plusieurs années. L'implémentation de cette clause dans la plupart des SGBD reste cependant relativement récente.

Elle 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.

⁷<https://www.postgresql.org/docs/current/static/sql-insert.html>

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 ajoutée 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

```
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;
```

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 la 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,
       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

- 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 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 et au final utilisées dans la requête principale.

Avant la version 12, une CTE était forcément matérialisée. À partir de la version 12, ce n'est plus le cas. Le seul moyen de s'en assurer revient à ajouter la clause **MATERIALIZED**.

1.5.2 CTE ET SELECT : EXEMPLE

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

On utilise principalement une CTE pour factoriser la définition d'une sous-requête commune, comme dans l'exemple ci-dessus.

Un autre exemple un peu plus complexe :

```
WITH resume_commandes AS (  
SELECT c.numero_commande, c.client_id, quantite*prix_unitaire AS montant  
FROM commandes c  
JOIN lignes_commandes l  
ON (c.numero_commande = l.numero_commande)  
WHERE date_commande BETWEEN '2014-01-01' AND '2014-12-31'
```

```

)
SELECT type_client, NULL AS pays, SUM(montant) AS montant_total_commande
FROM resume_commandes
JOIN clients
    ON (resume_commandes.client_id = clients.client_id)
GROUP BY type_client
UNION ALL
SELECT NULL, code_pays AS pays, SUM(montant)
FROM resume_commandes r
JOIN clients cl
    ON (r.client_id = cl.client_id)
JOIN contacts co
    ON (cl.contact_id = co.contact_id)
GROUP BY code_pays;

```

Le plan d'exécution de la requête montre que la vue `resume_commandes` est exécutée une seule fois et son résultat est utilisé par les deux opérations de regroupements définies dans la requête principale :

QUERY PLAN

```

Append (cost=244618.50..323855.66 rows=12 width=67)
  CTE resume_commandes
    -> Hash Join (cost=31886.90..174241.18 rows=1216034 width=26)
        Hash Cond: (l.numero_commande = c.numero_commande)
        -> Seq Scan on lignes_commandes l
            (cost=0.00..73621.47 rows=3141947 width=18)
        -> Hash (cost=25159.00..25159.00 rows=387032 width=16)
            -> Seq Scan on commandes c
                (cost=0.00..25159.00 rows=387032 width=16)
                Filter: ((date_commande >= '2014-01-01'::date)
                    AND (date_commande <= '2014-12-31'::date))
    -> HashAggregate (cost=70377.32..70377.36 rows=3 width=34)
        Group Key: clients.type_client
        -> Hash Join (cost=3765.00..64297.15 rows=1216034 width=34)
            Hash Cond: (resume_commandes.client_id = clients.client_id)
            -> CTE Scan on resume_commandes
                (cost=0.00..24320.68 rows=1216034 width=40)
            -> Hash (cost=2026.00..2026.00 rows=100000 width=10)
                -> Seq Scan on clients
                    (cost=0.00..2026.00 rows=100000 width=10)
        -> HashAggregate (cost=79236.89..79237.00 rows=9 width=35)
            Group Key: co.code_pays
            -> Hash Join (cost=12624.57..73156.72 rows=1216034 width=35)
                Hash Cond: (r.client_id = cl.client_id)
                -> CTE Scan on resume_commandes r

```

SQL avancé pour le transactionnel

```
(cost=0.00..24320.68 rows=1216034 width=40)
-> Hash (cost=10885.57..10885.57 rows=100000 width=11)
    -> Hash Join
        (cost=3765.00..10885.57 rows=100000 width=11)
        Hash Cond: (co.contact_id = cl.contact_id)
        -> Seq Scan on contacts co
            (cost=0.00..4143.05 rows=110005 width=11)
        -> Hash (cost=2026.00..2026.00 rows=100000 width=16)
            -> Seq Scan on clients cl
                (cost=0.00..2026.00 rows=100000 width=16)
```

Si la requête avait été écrite sans CTE, donc en exprimant deux fois la même sous-requête, le coût d'exécution aurait été multiplié par deux car il aurait fallu exécuter la sous-requête deux fois au lieu d'une.

On utilise également les CTE pour améliorer la lisibilité des requêtes complexes, mais cela peut poser des problèmes d'optimisations, comme cela sera discuté plus bas.

1.5.3 CTE ET SELECT : SYNTAXE

```
WITH nom_vue1 AS [ [ NOT ] MATERIALIZED ] (
    <requête pour générer la vue 1>
)
SELECT *
FROM nom_vue1;
```

La syntaxe de définition d'une vue est donnée ci-dessus.

On peut néanmoins enchaîner plusieurs vues les unes à la suite des autres :

```
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>;
```

1.5.4 CTE ET BARRIÈRE D'OPTIMISATION

- Attention, une CTE est une barrière d'optimisation !
 - pas de transformations
 - pas de propagation des prédicats
- Sauf à partir de la version 12
 - clause **MATERIALIZED** pour obtenir cette barrière

Il faut néanmoins être vigilant car l'optimiseur n'inclut pas la définition des CTE dans la requête principale quand il réalise les différentes passes d'optimisations.

Par exemple, sans CTE, si un prédicat appliqué dans la requête principale peut être remonté au niveau d'une sous-requête, l'optimiseur de PostgreSQL le réalisera :

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

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

Les deux prédicats `num_service = 4` et `date_embauche < '2006-01-01'` ont été appliqués en même temps, réduisant ainsi le jeu de données à considérer dès le départ. En anglais, on parle de *predicate push-down*.

Une requête équivalente basée sur une CTE ne permet pas d'appliquer le filtre au plus tôt : ici le filtre inclus dans la CTE est appliqué, pas le second.

```
EXPLAIN
WITH e AS
  (SELECT * FROM employees 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 employees  (cost=0.00..1.18 rows=5 width=43)
       Filter: (num_service = 4)
```

SQL avancé pour le transactionnel

```
-> CTE Scan on e (cost=0.00..0.11 rows=2 width=4)
    Filter: (date_embauche < '2006-01-01'::date)
```

On peut se faire piéger également en voulant calculer trop de choses dans les CTE. Dans cet autre exemple, on cherche à afficher les 7 commandes d'un client donné, le cumul des valeurs des lignes par commande étant réalisé dans un CTE :

```
EXPLAIN ANALYZE
WITH nos_commandes AS
(
    SELECT c.numero_commande, c.client_id, SUM(quantite*prix_unitaire) AS montant
    FROM   commandes c
    JOIN   lignes_commandes l
    ON     (c.numero_commande = l.numero_commande)
    GROUP BY 1,2
)
SELECT clients.client_id, type_client, nos_commandes.*
FROM   nos_commandes
INNER JOIN clients
ON     (nos_commandes.client_id = clients.client_id)
WHERE  clients.client_id = 6845
;
```

QUERY PLAN

```
-----
Nested Loop (cost=154567.68..177117.90 rows=5000 width=58)
    (actual time=7.757..5526.148 rows=7 loops=1)
    CTE nos_commandes
    -> GroupAggregate (cost=3.51..154567.39 rows=1000000 width=48)
        (actual time=0.043..5076.121 rows=1000000 loops=1)
        Group Key: c.numero_commande
        -> Merge Join (cost=3.51..110641.89 rows=3142550 width=26)
            (actual time=0.017..2511.385 rows=3142632 loops=1)
            Merge Cond: (c.numero_commande = l.numero_commande)
            -> Index Scan using commandes_pkey on commandes c
                (cost=0.42..16290.72 rows=1000000 width=16)
                (actual time=0.008..317.547 rows=1000000 loops=1)
            -> Index Scan using lignes_commandes_pkey on lignes_commandes l
                (cost=0.43..52570.08 rows=3142550 width=18)
                (actual time=0.006..1030.420 rows=3142632 loops=1)
        -> Index Scan using clients_pkey on clients
            (cost=0.29..0.51 rows=1 width=10)
            (actual time=0.009..0.009 rows=1 loops=1)
            Index Cond: (client_id = 6845)
    -> CTE Scan on nos_commandes (cost=0.00..22500.00 rows=5000 width=48)
        (actual time=7.746..5526.128 rows=7 loops=1)
        Filter: (client_id = 6845)
```


Rows Removed by Filter: 999993

Notez que la construction de la CTE fait un calcul sur l'intégralité des 5000 commandes et brasse un million de lignes. Puis, une fois connu le `client_id`, PostgreSQL parcourt cette CTE pour en récupérer une seule ligne. C'est évidemment extrêmement coûteux et dure plusieurs secondes.

Alors que sans la CTE, l'optimiseur se permet de faire la jointure avec les tables, donc à filtrer sur le client demandé, et ne fait la somme des lignes qu'après, en quelques millisecondes.

```
EXPLAIN ANALYZE
SELECT clients.client_id, type_client, nos_commandes.*
FROM
(
  SELECT c.numero_commande, c.client_id, SUM(quantite*prix_unitaire) AS montant
  FROM   commandes c
  JOIN   lignes_commandes l
  ON     (c.numero_commande = l.numero_commande)
  GROUP BY 1,2
) AS nos_commandes
INNER JOIN clients
ON     (nos_commandes.client_id = clients.client_id)
WHERE  clients.client_id = 6845
;
```

QUERY PLAN

```
Nested Loop (cost=12.83..13.40 rows=11 width=58)
  (actual time=0.113..0.117 rows=7 loops=1)
    -> Index Scan using clients_pkey on clients (cost=0.29..0.51 rows=1 width=10)
        (actual time=0.007..0.007 rows=1 loops=1)
      Index Cond: (client_id = 6845)
    -> HashAggregate (cost=12.54..12.67 rows=11 width=48)
        (actual time=0.106..0.108 rows=7 loops=1)
      Group Key: c.numero_commande
      -> Nested Loop (cost=0.85..12.19 rows=35 width=26)
          (actual time=0.028..0.087 rows=23 loops=1)
            -> Index Scan using commandes_clients_fkey on commandes c
                (cost=0.42..1.82 rows=11 width=16)
                (actual time=0.022..0.028 rows=7 loops=1)
              Index Cond: (client_id = 6845)
            -> Index Scan using lignes_commandes_pkey on lignes_commandes l
                (cost=0.43..0.89 rows=5 width=18)
                (actual time=0.006..0.007 rows=3 loops=7)
              Index Cond: (numero_commande = c.numero_commande)
```

En plus d'améliorer la lisibilité et d'éviter la duplication de code, le mécanisme des CTE est aussi un moyen contourner certaines limitations de l'optimiseur de PostgreSQL en vue de contrôler précisément le plan d'exécution d'une requête.

Ce principe de fonctionnement a changé avec la version 12 de PostgreSQL. Par défaut, il n'y a pas de matérialisation mais celle-ci peut être forcée avec l'option **MATERIALIZED**.

1.5.5 CTE EN ÉCRITURE

- CTE avec des requêtes en modification
 - avec **INSERT/UPDATE/DELETE**
 - et éventuellement **RETURNING**
 - obligatoirement exécuté sur PostgreSQL
- Exemple d'utilisation :
 - archiver des données
 - partitionner les données d'une table
 - déboguer une requête complexe

1.5.6 CTE EN ÉCRITURE : EXEMPLE

```
WITH donnees_a_archiver AS (  
  DELETE FROM donnees_courantes  
  WHERE date < '2015-01-01'  
  RETURNING *  
)  
INSERT INTO donnees_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.

Le même principe s'applique pour une table que l'on vient de partitionner. Les enregistrements se trouvent initialement dans la table mère, il faut les répartir sur les différentes partitions. On utilisera une requête reposant sur le même principe que la précédente. L'ordre INSERT visera la table principale si l'on souhaite utiliser le trigger de partition pour répartir les données. Il pourra également viser une partition donnée afin d'éviter le surcoût du trigger de partition.

En plus de ce cas d'usage simple, il est possible d'utiliser cette fonctionnalité pour debug-

ger 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 concurrence 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.

Il est important de noter que sur PostgreSQL, chaque sous-partie d'une CTE qui exécute une opération de mise à jour sera exécutée, même si elle n'est pas explicitement appelée. Par exemple :

```
WITH del AS (DELETE FROM nom_table),
fonction_en_ecriture AS (SELECT * FROM fonction_en_ecriture())
SELECT 1;
```

supprimera l'intégralité des données de la table **nom_table**, mais n'appellera pas la fonction **fonction_en_ecriture()**, même si celle-ci effectue des écritures.

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, seront bien plus efficaces. À noter que l'extension pgRouting implémente des algorithmes de parcours de graphes plus efficace. Cela permet de rester dans PostgreSQL mais nécessite un certain formalisme et il faut avoir conscience que pgRouting n'est pas l'outil le plus efficace car il génère un graphe en mémoire à chaque requête à résoudre, qui est perdu après l'appel.

1.5.8 CTE RÉCURSIVE : EXEMPLE (1/2)

```
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
```

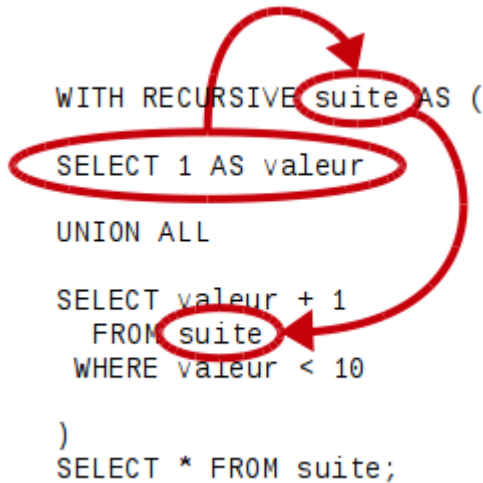
4
5
6
7
8
9
10

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. A 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

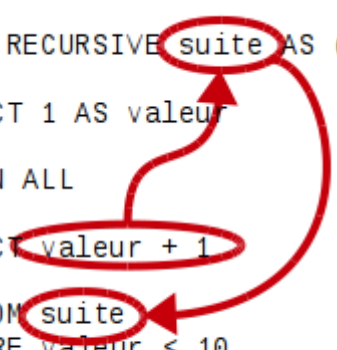


```
WITH RECURSIVE suite AS (
  SELECT 1 AS valeur
  UNION ALL
  SELECT valeur + 1
     FROM suite
    WHERE valeur < 10
)
SELECT * FROM suite;
```

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 :

SQL avancé pour le transactionnel

```
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`,
- en verrouillant les données lues avec `SELECT FOR UPDATE`,

SQL avancé pour le transactionnel

- 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 permettra une lecture parallèle, 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
```

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';
```

SQL avancé pour le transactionnel

```
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 contre-partie, 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.

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**
 - PostgreSQL 9.5
- Utilité :
 - implémente des files d'attente parallélisables

Une dernière fonctionnalité intéressante de **SELECT FOR UPDATE**, apparue avec PostgreSQL 9.5, permet de mettre en oeuvre différents workers qui consomment des données issues d'une table représentant une file d'attente. Il s'agit de la clause **SKIP LOCKED**, dont le principe de fonctionnement est identique à son équivalent sous Oracle.

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

```
CREATE TABLE test_skiplocked (id serial primary key, val text);
INSERT INTO test_skiplocked (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 test_skiplocked
LIMIT 10
FOR UPDATE SKIP LOCKED;
```

id	val
1	c4ca4238a0b923820dcc509a6f75849b
2	c81e728d9d4c2f636f067f89cc14862c
3	eccbc87e4b5ce2fe28308fd9f2a7baf3
4	a87ff679a2f3e71d9181a67b7542122c
5	e4da3b7fbfce2345d7772b0674a318d5
6	1679091c5a880faf6fb5e6087eb1b2dc
7	8f14e45fcee167a5a36dedd4bea2543
8	c9f0f895fb98ab9159f51fd0297e236d
9	45c48cce2e2d7fbdea1afc51c7c6ad26
10	d3d9446802a44259755d38e6d163e820

(10 rows)

Si on démarre une seconde transaction en parallèle, avec la première transaction toujours

SQL avancé pour le transactionnel

ouverte, le fait d'exécuter la requête `SELECT FOR UPDATE` sans la clause `SKIP LOCKED` aurait pour effet de la mettre en attente. L'ordre `SELECT` rendra la main lorsque la transaction #1 se terminera.

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

```
BEGIN TRANSACTION;
```

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

id	val
11	6512bd43d9caa6e02c990b0a82652dca
12	c20ad4d76fe97759aa27a0c99bfff6710
13	c51ce410c124a10e0db5e4b97fc2af39
14	aab3238922bcc25a6f606eb525ffdc56
15	9bf31c7ff062936a96d3c8bd1f8f2ff3
16	c74d97b01eae257e44aa9d5bade97baf
17	70efdf2ec9b086079795c442636b55fb
18	6f4922f45568161a8cdf4ad2299f6d23
19	1f0e3dad99908345f7439f8ffabdfc4
20	98f13708210194c475687be6106a3b84

(10 rows)

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

```
DELETE FROM test_skiplocked  
  WHERE id IN (...);  
COMMIT;
```

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

1.7 SERIALIZABLE SNAPSHOT ISOLATION

SSI : Serializable Snapshot Isolation (9.1+)

- 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 depuis la version 9.1 un 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.

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 globalement.

Voici un exemple.

Dans cet exemple, il y a des enregistrements avec une colonne couleur contenant 'blanc' ou 'noir'. Deux utilisateurs essaient simultanément de convertir tous les enregistrements vers une couleur unique, mais chacun dans une direction opposée. Un veut passer tous les blancs en noir, et l'autre tous les noirs 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 'noir'
  else 'blanc' end from x;
```

Session 1 :

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

SQL avancé pour le transactionnel

Session 2 :

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

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 = 'noir'
where couleur = 'blanc';
commit;
```


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

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

```
id | couleur
---+-----
 1 | noir
 2 | noir
 3 | noir
 4 | noir
 5 | noir
 6 | noir
 7 | noir
 8 | noir
 9 | noir
10 | noir
(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](https://www.postgresql.org/docs/12/serializable.html)⁸.

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 développement
 - mais aussi de performance

⁸<https://wiki.postgresql.org/wiki/SSI/fr>

1.9 TRAVAUX PRATIQUES

Jointure latérale

Cette série de question utilise la base de TP **magasin**. La base **magasin** peut être téléchargée depuis https://dali.bo/tp_magasin (dump de 96 Mo, pour 667 Mo sur le disque au final). Elle s'importe de manière très classique (une erreur sur le schéma **public** déjà présent est normale), ici dans une base nommée aussi **magasin** :

```
pg_restore -d magasin magasin.dump
```

Toutes les données sont dans deux schémas nommés **magasin** et **facturation**.

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** :

```
createdb genealogie
pg_restore -O -d genealogie genealogie.dump
```

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

```
\d 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 ancêtres de Adèle TAIL-
LANDIER

Réseau social

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

```
createdb socialnet
pg_restore -O -d socialnet socialnet.dump
```

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 Luetngen 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 :

<https://dalibo.com/formations>

SQL avancé pour le transactionnel

- 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 système¹²

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 depuis https://dali.bo/tp_brno2015 et restaurée à l'aide de `pg_restore` :

```
createdb brno2015
pg_restore -O -d brno2015 brno2015.dump
```

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
```

SQL avancé pour le transactionnel

```
ORDER BY c.numero_commande DESC
LIMIT 10;
```

CTE récursive

Cet exercice propose de manipuler des données généalogiques.

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

```
SET search_path = genealogie;
```

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.

```
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 ancêtres 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

Cet exercice est assez similaire au précédent.

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

```
SET search_path = socialnet;
```

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
```

SQL avancé pour le transactionnel

```
droite | integer | not null
Indexes:
    "relation_droite_idx" btree (droite)
    "relation_gauche_idx" btree (gauche)
```

Déterminer le niveau de connexions entre Sadry Luetngen 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^a](https://docs.pgRouting.org/2.0/fr/src/kdijkstra/doc/index.html#pgr-kdijkstra), 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.

^a<https://docs.pgRouting.org/2.0/fr/src/kdijkstra/doc/index.html#pgr-kdijkstra>

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_depend¹³](#)
- [Catalogue pg_rewrite¹⁴](#)
- [Catalogue pg_class¹⁵](#)
- [Fonction d'information du catalogue système¹⁶](#)

Retrouver les dépendances de la vue `pilotes_brno`. Déduisez également l'ordre de suppression et de création 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;
```

¹³<http://www.postgresql.org/docs/current/static/catalog-pg-depend.html>

¹⁴<http://www.postgresql.org/docs/current/static/catalog-pg-rewrite.html>

¹⁵<http://www.postgresql.org/docs/current/static/catalog-pg-class.html>

¹⁶<http://www.postgresql.org/docs/current/static/functions-info.html#FUNCTIONS-INFO-CATALOG-TABLE>

SQL avancé pour le transactionnel

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  
)  
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));
```

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.