Livre Blanc DALIBO #05

Bonnes pratiques de développement avec PostgreSQL

18.10

DALIBO
L'expertise PostgreSQL

Dalibo SCOP

https ://wwww.dalibo.com/

Bonnes pratiques de développement avec PostgreSQL

Livre Blanc DALIBO #05

TITRE : Bonnes pratiques de développement avec PostgreSQL
SOUS-TITRE : Livre Blanc DALIBO #05

REVISION : 18.10

Remerciements : Ce livre blanc est le fruit d’un travail collectif. Nous remercions chaleureusement ici toutes les personnes qui ont contribué directement ou indirectement à cet ouvrage, notamment : Carole Arnaud, Damien Clochard, Léo Cossic, Adrien Nayrat, Thomas Reiss, Maël Rimbault. Nous remercions également la société DSIA qui a contribué au financement de ce livre blanc.

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 livres blancs sur https ://dalibo.com/

Chers lectrices & lecteurs,

Nos livres blancs sont issus de plus de 12 ans d'études, d'expérience de terrain et de passion pour les logiciels libres. Pour Dalibo, l'utilisation de PostgreSQL n'est pas une marque d'opportunisme commercial, mais l'expression d'un engagement de longue date. Le choix de l'Open Source est aussi le choix de l'implication dans la communauté du logiciel.

Au-delà du contenu technique en lui-même, notre intention est de transmettre les valeurs qui animent et unissent les développeurs de PostgreSQL depuis toujours : partage, ouverture, transparence, créativité, dynamisme… Le but premier de nos livres blancs est de vous aider à mieux exploiter toute la puissance de PostgreSQL mais nous espérons également qu'ils 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 productions 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 contact@dalibo.com !

Table des Matières

EXPLOITER TOUTE LA PUISSANCE DE POSTGRESQL

PostgreSQL est un moteur de base de données à la fois robuste et puissant. Depuis plus de 20 ans, il est porté par une communauté internationale, décentralisée et très dynamique. Chaque année, une nouvelle version majeure est publiée avec de nouvelles fonctionnalités, ce qui en fait le SGBD le plus innovant du secteur.

Cette richesse et ce dynamisme font qu'il est parfois difficile de suivre les dernières avancées ou simplement de connaitre toutes les possibilités offertes par le moteur.

Le but premier de ce guide est de faire un tour d'horizon des bonnes pratiques et des conventions pour développer des applications qui tirent le meilleur de PostgreSQL. Il se concentre sur l'écriture de requêtes : détecter les requêtes lentes, optimiser certaines opérations, analyser les statistiques et la consommation mémoire, etc.

Ce manuel est construit comme un boite à outils dans laquelle le lecteur pourra piocher des conseils et des idées en fonction du contexte et des constraintes de chaque projet.

UTILISATION DE LA BASE DE DONNÉES

BONNES PRATIQUES D'ÉCRITURE DES REQUÊTES

FORMATAGE DU CODE SQL

SQL est un langage. On utilisera donc des règles de formatage et d'écriture de la même façon qu'on le ferait pour n'importe quel langage de programmation, en distinguant les mots clés du langage des identifiants de colonnes et de tables, et en utilisant l'indentation et les commentaires.

Un exemple vaut probablement mieux qu'un long discours. La requête suivante n'est typiquement pas lisible :

```
select  groupeid,datecreationitem from itemagenda where typeitemagenda = 5
and groupeid in(12225,12376) and datecreationitem > now() order by groupeid,
datecreationitem ;
```

Cette requête est strictement identique mais est bien plus lisible :

```
SELECT groupeid, datecreationitem
  FROM itemagenda
 WHERE typeitemagenda = 5
   AND groupeid IN (12225,12376)
   AND datecreationitem > now()
 ORDER BY groupeid, datecreationitem ;
```

La principale recommandation est de choisir des règles de formatage claires, et de s'y tenir.

Un exemple de telles règles peut se trouver sur le site SQL Style Guide[1] .

La documentation PostgreSQL utilise des règles de formatage proches de celles présentées sur ce site.

COMMENTAIRES

Une requête SQL peut être commentée au même titre qu'un programme standard.

Le marqueur -- permet de signifier à l'analyseur syntaxique que le reste de la ligne est commenté, il n'en tiendra donc pas compte dans son analyse de la requête.

Un commentaire peut aussi se présenter sous la forme d'un bloc de commentaire, le bloc pouvant occuper plusieurs lignes :

1. http://www.sqlstyle.guide/

```
/* Ceci est un commentaire
   sur plusieurs
   lignes

*/
```

Aucun des éléments compris entre le marqueur de début de bloc `/*` et le marqueur de fin de bloc `*/` ne sera pris en compte. Certains SGBDR propriétaires utilisent ces commentaires pour y placer des informations (appelées hints sur Oracle) qui permettent d'influencer le comportement de l'optimiseur, mais PostgreSQL ne possède pas ce genre de mécanisme.

OPÉRATEURS RELATIONNELS VS OPÉRATEURS NON-RELATIONNELS

PostgreSQL est avant tout un Système de Gestion de Bases de Données Relationnel (SGBDR). Cela signifie qu'il appartient à la famille des moteurs de bases de données qui reposent sur la théorie mathématique de l'algèbre relationnelle. PostgreSQL permet l'accès et la manipulation des données dans les relations (tables) en utilisant le langage SQL, qui définit des opérateurs relationnels et non-relationnels.

Les opérateurs purement relationnels sont les suivants :

- Projection : clause `SELECT` (choix des attributs : colonnes)
- Sélection : clause `WHERE` (choix des tuples : lignes)
- Jointure : clause `FROM` / `JOIN` (choix des relations : tables)

Pour résumer, les opérateurs relationnels sont les opérateurs qui déterminent sur quelles données on travaille. Tous ces opérateurs sont optimisables : il y a 40 ans de théorie mathématique développée afin de permettre l'optimisation de ces traitements. L'optimiseur fera un excellent travail sur ces opérations, et les organisera de façon efficace pour répondre le plus rapidement possible à la requête.

Par exemple : `a JOIN b JOIN c WHERE c.col=constante` sera très probablement réordonné en `c JOIN b JOIN a WHERE c.col=constante` ou `c JOIN a JOIN b WHERE c.col=constante`. Le moteur se débrouillera aussi pour choisir le meilleur algorithme de jointure pour chacune, suivant les volumétries ramenées.

À contrario, les autres opérateurs sont non-relationnels :

- `ORDER BY`
- `GROUP BY`/`DISTINCT`
- `HAVING`
- Sous-requête, vue
- Fonction (classique, d'agrégat, analytique)

- Jointure externe

Ceux-ci sont plus difficilement optimisables : ils introduisent par exemple des contraintes d'ordre dans l'exécution.

Si l'on prend la requête suivante, l'appel à la fonction d'agrégation impose d'exécuter la sous-requête avant la requête :

```
SELECT *
FROM table1
WHERE montant > (
  SELECT avg(montant)
  FROM table1
  WHERE departement='44'
);
```

À contrario, si la sous-requête ne présente pas d'opérateur non-relationnel, elle est transformée en jointure et aucun ordre d'exécution n'est imposé :

```
EXPLAIN
SELECT *
FROM employes emp
JOIN (SELECT * FROM services WHERE num_service = 1) ser
  ON (emp.num_service = ser.num_service);

                            QUERY PLAN
------------------------------------------------------------------
 Nested Loop  (cost=0.00..2.25 rows=2 width=64)
   ->  Seq Scan on services  (cost=0.00..1.05 rows=1 width=21)
         Filter : (num_service = 1)
   ->  Seq Scan on employes emp  (cost=0.00..1.18 rows=2 width=43)
         Filter : (num_service = 1)
(5 rows)
```

Du point de vue de l'optimiseur, une vue est considérée comme une sous-requête : la définition de la vue est intégrée comme une sous-requête là où elle est appelée dans la requête initiale. Les mêmes contraintes d'optimisations s'appliquent donc.

Les jointures externes peuvent parfois être optimisées par le moteur, mais posent tout de même des problèmes et doivent être traitées prudemment. Par exemple dans le cas de cette requête : `SELECT * FROM t1 LEFT JOIN (t2 JOIN t3 ON (t2.ref = t3.id)) ON (t1.id = t2.id);`

Le moteur n'a pas le choix, il faut faire les jointures dans l'ordre indiqué : joindre t1 à t3 puis le résultat à t2 pourrait ne pas amener le même résultat (un `LEFT JOIN` peut produire des `NULL`). Il est donc préférable de toujours mettre les jointures externes en fin de requête, sauf besoin précis : on laisse bien plus de liberté à l'optimiseur.

Enfin, le mot clé `DISTINCT` ne doit être utilisé qu'en dernière extrémité. On le rencontre très fréquemment dans des requêtes mal écrites qui produisent des doublons, afin de maquiller le résultat. C'est bien sûr extrêmement simple de produire des doublons pour ensuite dédoublonner (au moyen d'un tri de l'ensemble du résultat, bien sûr).

Pour aller un peu plus loin, on pourra lire la page Wikipedia sur l'algèbre relationnelle[2]

LIMITER LE VOLUME DE DONNÉES MANIPULÉ

Le volume de données récupéré a un impact sur les performances. Lors de l'écriture d'une requête SQL, il est donc recommandé de n'accéder qu'aux tables et aux colonnes nécessaires. Il faut également avoir à l'esprit que plus le volume de données à traiter est élevé, plus les opérations seront lentes : les tris et jointures et, éventuellement, le stockage temporaire sur disque pour certains algorithmes.

De manière générale, c'est une mauvaise pratique d'utiliser la syntaxe `SELECT *`. Indiquer explicitement quelles sont les colonnes de la table que l'on désire manipuler permet d'éviter de rencontrer des problèmes ultérieurs, par exemple lors de modifications de schémas comme l'ajout ou le changement de position d'une colonne.

Certains moteurs suppriment d'eux-mêmes les colonnes qui ne sont pas retournées à l'appelant, par exemple dans le cas de :

```
SELECT col1, col2 FROM (SELECT * FROM t1 JOIN t2 USING (t2id) ) ;
```

Mais PostgreSQL n'implémente pas cette fonctionnalité il est donc important de ne récupérer que les colonnes utilisées.

Éviter les jointures sur des tables inutiles : il n'y a que peu de cas où l'optimiseur peut supprimer de lui-même l'accès à une table inutile.

PostgreSQL le fait dans le cas d'une jointure externe sur une table inutilisée dans le `SELECT`, à condition qu'aucune colonne de la table externe ne soit accédée et que la jointure est réalisée sur une contrainte d'unicité de la table externe. Avec cela, l'optimiseur sait qu'aucune ligne en doublon ne sera produite et peut alors considérer la jointure externe comme inutile et va donc l'éliminer du plan d'exécution :

```
EXPLAIN
SELECT e.matricule, e.nom, e.prenom
FROM employes e
LEFT JOIN services s
  ON (e.num_service = s.num_service)
WHERE e.num_service = 4;
```

2. https://fr.wikipedia.org/wiki/Alg%C3%A8bre_relationnelle

```
                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on employes e  (cost=0.00..1.18 rows=5 width=19)
   Filter : (num_service = 4)
(2 rows)
```

TRAITEMENTS ENSEMBLISTES VS ITÉRATIFS/PROCÉDURAL

Le langage SQL (et même les autres langages relationnels qui ont existé comme QUEL, SEQUEL) est un langage ensembliste et déclaratif. Il est conçu pour permettre la manipulation d'un gros volume de données, et la mise en correspondance (jointure) d'informations. Une base de données relationnelle n'est pas une simple couche de persistance, il est important de prendre en considération les contraintes posées par l'organisation des données.

Il est recommandé de ne pas faire de traitement unitaire par enregistrement. Les bases de données relationnelles sont conçues pour manipuler des relations, pas des enregistrements unitaires.

Le fait de récupérer en une seule opération l'ensemble des informations pertinentes est aussi, indépendamment du langage, un gain de performance énorme, car il permet de s'affranchir en grande partie des latences de communication entre la base et l'application.

L'exemple ci-dessous permet de montrer la différence de performance que l'on peut observer entre un traitement itératif et un traitement ensembliste.

Préparons un jeu de test :

```
CREATE TABLE test (a int, b varchar);
INSERT INTO test SELECT i,i FROM generate_series (1,1000000) g(i);
ALTER TABLE test ADD PRIMARY KEY (a);
```

Récupérons 10 000 enregistrements un par un. Le script Perl suivant récupère 10 000 lignes une par une, avec une requête préparée pour être dans le cas le plus efficace :

```
#!/usr/bin/perl -w
print "PREPARE ps (int) AS SELECT * FROM test WHERE a=\$1;\n";
for (my $i=0;$i<=10000;$i++)
{
        print "EXECUTE ps($i);\n";
}
```

Exécutons ce code :

```
time perl demo.pl  | psql > /dev/null

real    0m1.025s
```

```
user    0m0.213s
sys     0m0.057s
```

Voici maintenant la même chose, en un seul ordre SQL :

```
time psql -c "SELECT * FROM test WHERE a >=0 AND a `<= 10000" >` /dev/null

real    0m0.052s
user    0m0.030s
sys     0m0.003s
```

Pour accéder aux données, il faut utiliser des jointures et non pas accéder à chaque table une-par-une. C'est un comportement fréquent avec les ORM. La plupart des ORM fournissent un moyen de traverser des liens entre objets. Par exemple, si une commande est liée à plusieurs articles, un ORM permettra d'écrire un code similaire à celui-ci (exemple en Java avec Hibernate) :

```
List commandes = sess.createCriteria(Commande.class);

for(Commande cmd : commandes)
{
  // Un traitement utilisant les produits
  // Génère une requête par commande !!
  System.out.println(cmd.getProduits());
}
```

Tel quel, ce code générera N+1 requête, N étant le nombre de commandes (on l'appelle le "problème N+1"[3] . En effet, pour chaque accès à l'attribut "produits", l'ORM générera une requête pour récupérer les produits correspondants à la commande.

Le SQL généré sera alors similaire à celui-ci :

```
SELECT * FROM commande;
SELECT * from produits where commande_id = 1;
SELECT * from produits where commande_id = 2;
SELECT * from produits where commande_id = 3;
SELECT * from produits where commande_id = 4;
SELECT * from produits where commande_id = 5;
SELECT * from produits where commande_id = 6;
...
```

La plupart des ORM proposent des options pour personnaliser la stratégie d'accès aux collections. Il est extrêmement important de connaître celles-ci afin de permettre à l'ORM de générer des requêtes optimales.

Par exemple, dans le cas précédent, nous savons que tous les produits de toutes les commandes seront utilisés. Nous pouvons donc informer l'ORM de ce fait :

3. https://use-the-index-luke.com/fr/sql/l-operation-de-jointure/boucles-imbriquees

```
List commandes = sess.createCriteria(Commande.class)
  .setFetchMode('produits', FetchMode.EAGER);

for(Commande cmd : commandes)
{
  // Un traitement utilisant les produits
  System.out.println(cmd.getProduits());
}
```

Ceci générera une seule et unique requête du type, qui sera bien plus efficace :

```
SELECT * FROM commandes
LEFT JOIN produits ON commandes.id = produits.commande_id;
```

ÉCRITURE DES JOINTURES

Les jointures permettent d'écrire des requêtes qui impliquent plusieurs tables. Elles permettent de combiner les colonnes de plusieurs tables selon des critères particuliers, appelés conditions de jointures.

Les jointures permettent de tirer parti du modèle de données dans lequel les tables sont associées à l'aide de clés étrangères.

Bien qu'il soit possible de décrire une jointure interne sous la forme d'une requête `SELECT` portant sur deux tables dont la condition de jointure est décrite dans la clause `WHERE`, comme ceci :

```
SELECT apl.libelle AS appellation, reg.libelle AS region
FROM appellation apl, region reg
WHERE apl.region_id = reg.id
  AND reg.libelle LIKE 'A%';
```

Cette forme d'écriture n'est pas recommandée. En effet, les conditions de jointures se trouveront mélangées avec les clauses de filtrage, rendant ainsi la compréhension et la maintenance difficiles. Il arrive aussi que, noyé dans les autres conditions de filtrage, l'utilisateur oublie la configuration de jointure, ce qui aboutit à un produit cartésien, n'ayant rien à voir avec le résultat attendu, sans même parler de la lenteur de la requête.

Il est recommandé d'utiliser la syntaxe SQL :92 et d'exprimer les jointures à l'aide de la clause `JOIN`. Cette syntaxe facilite la compréhension de la requête mais facilite également le travail de l'optimiseur SQL qui peut déduire beaucoup plus rapidement les jointures qu'en analysant la clause `WHERE` pour déterminer les conditions de jointure et les tables auxquelles elles s'appliquent le cas échéant.

La requête de l'exemple plus haut s'écrira donc :

```
SELECT apl.libelle AS appellation, reg.libelle AS region
FROM appellation apl
JOIN region reg
  ON (apl.region_id = reg.id)
WHERE reg.libelle LIKE 'A%';
```

D'ailleurs, cette syntaxe est la seule qui soit utilisable pour exprimer simplement et efficacement une jointure externe, à travers les syntaxes `LEFT JOIN` et `RIGHT JOIN`. De plus c'est la seule syntaxe normalisée et donc commune à tous les SGBD. L'exemple suivant présente une jointure externe à gauche (`LEFT JOIN`) :

```
SELECT article.art_titre, auteur.aut_nom
FROM article
LEFT JOIN auteur
  ON (article.aut_id=auteur.aut_id) ;
```

Dans cet exemple, la table de gauche correspond à la table `article` et la table de droite à la table `auteur`. On les distingue ainsi car elles se trouvent soit à gauche soit à droite de la clause `JOIN`. Cette requête va ramener le résultat de la jointure interne et aussi ramener l'ensemble de la table de gauche qui ne peut être joint avec la table de droite. Les attributs de la table de droite sont alors NULL.

Pour exprimer les conditions de jointure, il existe aussi une clause `NATURAL` pour les trois types de jointures interne et externes. Cette clause permet de réaliser la jointure entre deux tables en utilisant les colonnes qui portent le même nom sur les deux tables comme condition de jointure. La forme `NATURAL JOIN` est de ce fait déconseillée car elle entraîne des comportements inattendus et amène donc à des résultats faux, en particulier si l'on ajoute une table à la requête.

La clause `CROSS JOIN` permet d'exprimer un produit cartésien, c'est à dire qu'elle réalise toutes les combinaisons entre les lignes d'une table et les lignes d'une autre. Cette clause est à éviter dans la mesure du possible, sauf lorsque l'on souhaite réellement réaliser un produit cartésien. Dans ce dernier cas, l'utilisation de cette clause permettra de souligner l'intention et lèvera toute ambiguïté quant à la production du produit cartésien.

L'exemple suivant montre l'utilisation d'un produit cartésien parfaitement légitime pour repérer les relevés manquants entre 12h et 14h pour une sonde donnée :

```
SELECT id_sonde,
       heures_releves
FROM sondes
CROSS JOIN generate_series('2013-01-01 12 :00 :00','2013-01-01 14 :00 :00',
                           interval '1 hour') series(heures_releves)
WHERE NOT EXISTS
    (SELECT 1
     FROM releves_horaires
```

```
WHERE releves_horaires.id_sonde=sondes.id_sonde
  AND releves_horaires.heure_releve=series.heures_releves) ;
```

IMPACT DU NOMBRE DE JOINTURES SUR UNE REQUÊTE

Déterminer le bon ordre de jointure est un point clé dans la recherche de performances. Mais le nombre de possibilités en augmentation factorielle avec le nombre de tables. Si le nombre de jointures à réaliser est minime, on peut se permettre de réaliser une recherche exhaustive. Mais s'il est trop important, il est contre-productif de réaliser une recherche exhaustive, en particulier en environnement OLTP.

Les paramètres `join_collapse_limit` et `from_collapse_limit` permettent de contrôler le nombre de jointures que PostgreSQL essaye d'optimiser. Au-delà, l'optimiseur limite ses combinaisons aux premières tables de la requête, et prendra les tables suivantes dans l'ordre de la requête. Cela permet de limiter le temps de planification et la consommation mémoire durant la planification. Enfin, si la limite `geqo_threshold` est atteinte, PostgreSQL utilisera un algorithme différent nommé GEQO.

La règle importante est de limiter le nombre de tables jointes au sein d'une requête au strict minimum. Cela signifie qu'il ne faut pas joindre de tables qui ne servent pas au résultat final, même si c'est fait de façon invisible au moment de l'écriture de la requête, par exemple en utilisant une vue. Cela signifie également que le modèle de données doit être pensé pour éviter d'avoir besoin d'un nombre excessif de jointures pour reconstituer une information utile.

De plus amples informations sont disponibles dans la partie *Ordre de jointure* du module de formation Dalibo à propos d'EXPLAIN.

Plus de détails sur : https://dali.bo/sql2_html#exécution-globale-dune-requête

CLAUSE IN / NOT IN / EXISTS

Les requêtes qui expriment un Semi-Join (test d'existence) ou Anti-Join (test de non-existence) sont presque des jointures : la seule différence est qu'elles ne récupèrent pas l'enregistrement de la table cible. Elles s'expriment par le biais d'une clause `IN`, `EXISTS` ou `NOT IN`, `NOT EXISTS`.

La clause `IN` peut être utilisée dans différents cas. On la retrouve habituellement lorsque l'utilisateur souhaite récupérer des données en fonction d'une liste de valeurs. Cette liste de valeurs peut-être fixée ou générée à partir du résultat d'une sous-requête.

La requête suivante permet par exemple de sélectionner les bouteilles du stock de la cave dont la contenance est comprise entre 0,3 litre et 1 litre. Pour répondre à la question, la

sous-requête retourne les identifiants de contenant qui correspondent à la condition. La requête principale ne retient alors que les lignes dont la colonne contenant_id correspond à une valeur d'identifiant retournée par la sous-requête.

```
SELECT *
FROM stock
WHERE contenant_id IN (SELECT id
                       FROM contenant
                       WHERE contenance BETWEEN 0.3 AND 1.0) ;
```

La clause `EXISTS` permet de réécrire une clause `IN` assez facilement, en utilisant une corrélation entre la requête principale et la sous-requête. Ainsi, la requête précédente pourrait également s'écrire :

```
SELECT *
FROM stock s
WHERE EXISTS (SELECT 1
              FROM contenant c
              WHERE c.id = s.contenant_id
                AND contenance BETWEEN 0.3 AND 1.0) ;
```

Les deux requêtes génèrent strictement le même plan :

```
Hash Join  (cost=28.07..28754.04 rows=5125 width=16)
  Hash Cond : (s.contenant_id = c.id)
  ->  Seq Scan on stock s  (cost=0.00..15791.85 rows=1025085 width=16)
  ->  Hash  (cost=28.00..28.00 rows=6 width=4)
        ->  Seq Scan on contenant c  (cost=0.00..28.00 rows=6 width=4)
              Filter : (
                (contenance >= '0.3'::double precision)
                AND (contenance <= '1'::double precision)
              )
```

Dans le cas où la liste de valeurs est obtenue à partir d'une sous-requête, si le nombre d'identifiants retournés est important, il est recommandé de les dédoublonner à l'aide de `DISTINCT` avant l'application de la clause `IN` :

```
SELECT * FROM t1
WHERE val IN (SELECT DISTINCT …)
```

La clause `NOT IN` permet de retourner les lignes dont un identifiant n'est pas inclue dans la liste du `NOT IN`. Elle n'est efficace que pour une liste de valeurs peu nombreuses. L'implémentation sous-jacente de `NOT IN` empêche tout simplement d'obtenir de bonnes performances lorsque la liste de valeurs devient importante : il faut rechercher la valeur à tester dans la liste complète de valeurs du `NOT IN`. On préférera utiliser la clause `NOT EXISTS` qui est beaucoup plus efficace.

Ainsi, sur un volume de données conséquent, la requête suivante utilisant `NOT IN` ne rendra la main qu'au bout de plusieurs heures :

```
SELECT *
FROM commandes
WHERE numero_commande NOT IN (SELECT numero_commande
                              FROM lignes_commandes) ;
```

Le plan d'exécution de cette requête est le suivant, on peut voir le coût prohibitif donné à ce plan par l'optimiseur. Le problème de ce plan est que la table `lignes_commandes` est lue, puis les clés sont matérialisées en mémoire et/ou sur disque. Ensuite, pour chaque ligne de la table `commandes`, PostgreSQL valide que le `numero_commande` ne se trouve pas dans la liste des clés issues de `lignes_commandes`. PostgreSQL relit dont ces clés autant de fois qu'il y a de lignes dans la table principale. C'est totalement inefficace, surtout sur une volumétrie importante :

```
Seq Scan on commandes
      (cost=0.00..54729813909.00 rows=500000 width=51)
Filter : (NOT (SubPlan 1))
 SubPlan 1
 ->  Materialize  (cost=0.00..101604.79 rows=3141919 width=8)
  ->  Seq Scan on lignes_commandes
        (cost=0.00..73621.19 rows=3141919 width=8)
```

En comparaison, une requête équivalente utilisant `NOT EXISTS` répond relativement rapidement pour un nombre important de lignes :

```
EXPLAIN (ANALYZE, BUFFERS, COSTS OFF, TIMING OFF)
SELECT *
FROM commandes
WHERE NOT EXISTS (SELECT 1
                  FROM lignes_commandes l
                  WHERE l.numero_commande = commandes.numero_commande) ;

                    QUERY PLAN
-------------------------------------------------------------------------------
Merge Anti Join (actual rows=0 loops=1)
  Merge Cond : (commandes.numero_commande = l.numero_commande)
  Buffers : shared hit=243 read=30390 written=11
  ->  Index Scan using commandes_pkey on commandes
            (actual rows=1000000 loops=1)
        Buffers : shared hit=172 read=18352 written=7
  ->  Index Only Scan using lignes_commandes_numero_commande_quantite_idx
            on lignes_commandes l
            (actual rows=3141963 loops=1)
        Heap Fetches : 0
        Buffers : shared hit=71 read=12038 written=4
```

```
Planning time : 0.634 ms
Execution time : 602.734 ms
```

DÉDOUBLONNEMENT DE LIGNES

Cela a déjà été rappelé plus haut. Le mot clé `DISTINCT` ne doit être utilisé qu'en dernière extrémité. On le rencontre très fréquemment dans des requêtes mal écrites qui produisent des doublons, afin de maquiller le résultat.

On voit ici l'effet d'un `DISTINCT` sur la lecture d'une table. On dédoublonne la clé primaire donc on utilise l'index pour récupérer les données triées, puis on applique l'opération `Unique` :

```
explain SELECT DISTINCT numero_commande FROM commandes ;
                            QUERY PLAN
-------------------------------------------------------------------
 Unique  (cost=0.42..50432.43 rows=1000000 width=8)
   ->  Index Only Scan using commandes_pkey on commandes
                 (cost=0.42..47932.43 rows=1000000 width=8)
(2 rows)
```

Comme on récupère la clé primaire, on sait que les valeurs sont uniques, donc on sait qu'il est inutile de dédoublonner. Le plan sans le `DISTINCT` inutile est bien moins coûteux :

```
explain SELECT numero_commande FROM commandes ;
                            QUERY PLAN
-------------------------------------------------------------------
 Seq Scan on commandes  (cost=0.00..20159.00 rows=1000000 width=8)
(1 row)
```

Dans le même ordre d'idée, il est important de prendre en compte que le mot-clé `UNION` indique qu'il faut dédoublonner les résultats, il implique donc également l'application d'un noeud `Unique`, et donc impose généralement la réalisation de tris coûteux. Si un dédoublonnage des résultats n'est pas nécessaire, il faut privilégier `UNION ALL`.

COMMON TABLE EXPRESSIONS

PostgreSQL propose le support des Common Table Expressions[4] (CTE) depuis de nombreuses versions, par le biais de la clause `WITH`.

Il faut garder à l'esprit que cette fonctionnalité est une barrière d'optimisation en l'état actuel.

4. https://dali.bo/sql2_html#common-table-expressions

Lorsque cela est possible, PostgreSQL essaye d'appliquer les prédicats au plus tôt, comme c'est le cas dans cette requête simple, qui n'utilise pas de CTE :

```
EXPLAIN
SELECT MAX(date_embauche)
FROM (SELECT * FROM employes 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 employes  (cost=0.00..1.21 rows=2 width=4)
         Filter : ((date_embauche < '2006-01-01'::date)
               AND (num_service = 4))
(3 rows)
```

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.

Si l'on retranscrit la requête pour exécuter la sous-requête dans une CTE (clause WITH), on voit que l'on peut bloquer cette optimisation :

```
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=1.29..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)
(6 rows)
```

Le moteur n'a pas appliqué les deux prédicats en même temps. Il a d'abord appliqué le prédicat `num_service = 4` car il s'agit du prédicat de la CTE `e`. Il applique ensuite le second prédicat sur la CTE, après coup. Ce n'était probablement pas le chemin d'accès le plus optimal, bien que la faible volumétrie mise en jeu ne permettra pas de le souligner aisément.

Cette fonctionnalité, bien qu'alléchante pour faciliter l'écriture et la compréhension d'une requête SQL complexe, est assez dangereuse et peut amener le moteur à proposer des plans d'exécution qui ne seront absolument pas pertinents et à l'inverse de ce qu'un utili-

sateur attendrait de la fonctionnalité.

Ensuite, parmi les fonctionnalités inédites des CTE dans PostgreSQL, il y a la possibilité d'utiliser des ordres d'écritures. Lorsqu'ils sont combinés à la clause RETURNING[5] , elles permettent de réaliser simplement des traitements importants de mises à jour des données. Par exemple, la requête suivante 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 :

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

Dans le même ordre d'idée, il existe une clause INSERT … ON CONFLICT[6] qui permet de gérer des conflits d'insertion de clés dupliquées, soit pour mettre à jour en cas de conflit sur un `INSERT`, soit pour ne rien faire en cas de conflit sur un `INSERT`. 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.

Par exemple, pour une table `personnel` possédant une contrainte d'unicité sur la colonne `matricule`, on aurait :

```
INSERT INTO personnel (matricule, nom, prenom, version, fonction)
  SELECT matricule, nom, prenom, 1, fonction FROM temp_integ_employes
 ON CONFLICT (matricule)
    DO UPDATE SET matricule = excluded.matricule,
                  nom       = excluded.nom,
                  prenom    = excluded.prenom ;
```

L'ordre `MERGE` n'existe pas encore dans PostgreSQL mais il est en cours de développement, et sera probablement intégré dans une version future.

VUES

Les vues sont très pratiques en SQL et en théorie permettent de séparer le modèle physique (les tables) de ce que voient les développeurs, et donc de faire évoluer le modèle

5. https://dali.bo/sql2_html#returning
6. https://dali.bo/sql2_html#upsert

physique sans impact pour le développement. En pratique, elles vont souvent être source de ralentissement : elles masquent la complexité, et peuvent rapidement conduire à l'écriture implicite de requêtes très complexes, mettant en jeu des dizaines de tables (voire des dizaines de fois les MÊMES tables).

Il faut donc se méfier des vues. En particulier, des vues contenant des opérations non-relationnelles, qui peuvent empêcher de nombreuses optimisations. En voici un exemple simple.

```
CREATE TABLE test
AS SELECT
    i AS id,
    (i % 4) ::text AS valeur
FROM generate_series(1, 10000) i ;

CREATE VIEW tmp
AS SELECT DISTINCT ON (id) id,valeur
FROM test ;
```

La requête suivante :

```
SELECT id, valeur
FROM tmp
WHERE valeur='b';
```

sera réécrite par l'optimiseur de PostgreSQL par une sous-requête équivalente :

```
SELECT id,valeur
FROM
  (SELECT DISTINCT ON (id) id,valeur FROM test ) AS tmp
WHERE valeur='b' ;
```

On obtient ainsi le plan d'exécution suivant :

```
Subquery Scan on tmp  (cost=809.39..984.39 rows=50 width=6)
  Filter : (tmp.valeur = 'b'::text)
  ->  Unique  (cost=809.39..859.39 rows=10000 width=6)
        ->  Sort  (cost=809.39..834.39 rows=10000 width=6)
              Sort Key : test.id
              ->  Seq Scan on test  (cost=0.00..145.00 rows=10000 width=6)
```

On constate que la condition de filtrage sur b n'est appliquée qu'à la fin. C'est normal, à cause du DISTINCT ON, l'optimiseur ne peut pas savoir si l'enregistrement qui sera retenu dans la sous-requête vérifiera valeur='b' ou pas, et doit donc attendre l'étape suivante pour filtrer. Le coût en performances, même avec un volume de données raisonnable, peut être astronomique.

FONCTIONS ET PROCÉDURES STOCKÉES

Les langages de procédures (« PL ») fournissent :

- des fonctionnalités procédurales dans un univers relationnel ;
- des fonctionnalités avancées du langage *PL* choisi ;
- la possibilité d'exécuter un traitement applicatif sans avoir à subir le transport des données entre le serveur de bases de données et le serveur applicatif.

Il peut y avoir de nombreuses raisons différentes à l'utilisation d'un langage *PL*. Simplifier et centraliser des traitements clients directement dans la base est l'argument le plus fréquent. Les langages PL permettent donc de rajouter une couche d'abstraction et d'effectuer des traitements avancés directement en base.

Il convient de noter que dans les versions de PostgreSQL jusqu'à la 10 inclue, il n'y a pas de possibilité de déclarer de procédure, seulement des fonctions. Les caractéristiques de ces fonctions sont :

- elles peuvent effectuer des modifications de données ;
- elles s'exécutent toujours dans une seule et même transaction, celle de l'ordre SQL qui les a appelées ;
- elles ne peuvent pas démarrer de transaction autonome ;
- elles ne peuvent réaliser de validation partielle des données modifiées ;
- elles s'exécutent par défaut avec les privilèges de l'appelant, mais peuvent être définies pour utiliser les privilèges du propriétaire de la fonction ;
- elles ne sont pas obligées d'avoir d'arguments, ni de retourner un résultat ;
- il n'y a pas de possibilité de déclarer de *packages*, ou de variables globales partagées entre plusieurs fonctions.

Les fonctions sont fréquemment utilisées pour mettre en place des clauses de vérification et des contraintes de cohérence avancées, en combinaison avec des triggers[7]

Néanmoins, l'utilisation de fonctions pour réaliser des traitements applicatifs directement en base de données peut poser un certain nombre de problèmes.

L'utilisation excessive des langages de procédures peut amener à manipuler les données de façon procédurale plutôt qu'ensembliste, et il a déjà été indiqué précédemment que cela abouti généralement à de mauvaises performances. L'utilisation de curseurs est dans ce contexte un piège fréquent.

Par ailleurs, l'utilisation d'étapes dans le traitement, comme des variables ou des tables temporaires, peut là encore provoquer des problèmes de performances (barrière d'optimisation) et des effets de bord inattendus en fonction du niveau d'isolation choisi.

7. https://www.postgresql.org/docs/current/static/plpgsql-trigger.html

Enfin, intégrer de nombreux traitements applicatifs directement dans PostgreSQL peut être source de problèmes pour la facilité du développement et l'évolution de l'architecture. Notamment, la gestion du code *PL* par des outils de versionnement et de contrôle de source n'est pas toujours bien intégrée, et il convient également de penser aux difficultés éventuelles de migration vers un autre moteur de gestion de bases de données.

Un des intérêts de l'utilisation des langages de procédures est le mécanisme de gestion des exceptions proposé par PostgreSQL car il permet une gestion simplifié des erreurs. La gestion des exceptions utilise le mécanisme de `SAVEPOINT` (voir la section suivante sur le fonctionnement des transactions pour plus de détails). À chaque nouveau bloc de traitement d'exception exécuté, un `SAVEPOINT` est placé. Si un bloc de gestion d'exception est exécuté à chaque itération d'une boucle, le traitement se verra fortement pénalisé. Il faudra trouver d'autres façons de faire, l'idéal étant de réaliser le traitement de manière ensembliste et non plus itérative.

L'exemple ci-dessus montre un mauvais cas d'utilisation, destiné à traiter l'intégration de clés dupliquées de manière unitaire :

```
CREATE OR REPLACE FUNCTION integration_employes ()
  RETURNS boolean
AS $FUNC$
DECLARE
    integ_employes CURSOR IS
        SELECT * FROM temp_integ_employes
         ORDER BY matricule ;
BEGIN
  FOR emp IN integ_employes LOOP
    BEGIN  -- début du bloc de traitement d'exception

      INSERT INTO employes (matricule, nom, prenom, version, fonction)
    VALUES (emp.matricule, emp.nom, emp.prenom, 1, emp.fonction) ;

      -- traitement de l'exception si violation de contrainte d'unicité
      -- càd modification d'un employé déjà existant
      EXCEPTION WHEN unique_violation THEN
        UPDATE employes
           SET matricule = emp.matricule,
              nom = emp.nom,
              prenom = emp.prenom
        WHERE matricule = emp.matricule ;
      END;
    END LOOP;
    RETURN true;
EXCEPTION WHEN OTHERS THEN
  RETURN false;
```

```
END;
$FUNC$
LANGUAGE plpgsql ;
```

On pourra aisément contourner le problème avec l'ordre INSERT ON CONFLICT[8] pour résoudre le problème de manière ensembliste :

```
CREATE OR REPLACE FUNCTION integration_employes ()
  RETURNS boolean
AS $FUNC$
BEGIN
    INSERT INTO employes (matricule, nom, prenom, version, fonction)
        SELECT matricule, nom, prenom, 1, fonction FROM temp_integ_employes
     ON CONFLICT (matricule)
        DO UPDATE SET matricule = excluded.matricule,
                      nom       = excluded.nom,
                      prenom    = excluded.prenom ;
    RETURN true;
EXCEPTION WHEN OTHERS THEN
  RETURN false;
END;
$FUNC$
LANGUAGE plpgsql ;
```

Le traitement sera ainsi beaucoup plus efficace. Bien entendu, cet exemple est simple et simpliste : dans ce cas, nous pourrions même nous passer complètement de la fonction, qui est devenue inutile, il suffit d'exécuter la requête `INSERT` avec la clause `ON CONFLICT`, cela serait plus simple à maintenir, et plus encore efficace. La réalité d'un processus d'intégration de données nécessitera très probablement des traitements supplémentaires.

Requêtes préparées

Les requêtes préparées permettent dans de nombreux cas d'obtenir des gains importants en terme de performance.

Lorsque l'on soumet une requête SQL au serveur de base de données, elle passe par différentes étapes[9] :

- le parser, l'étape d'analyse syntaxique de la requête ;
- le rewritter va réécrire la requête, notamment en injectant la définition des vues dans l'arbre syntaxique ;
- le planner va calculer le plan d'exécution le moins coûteux à partir d'un ensemble de plans d'exécution qu'il aura généré ;

8. https://www.postgresql.org/docs/current/static/sql-insert.html#SQL-ON-CONFLICT
9. https://dali.bo/sql2_html#exécution-globale-dune-requête

- l'exécuteur va exécuter la requête et retourner le résultat à l'utilisateur.

L'ordre EXPLAIN ANALYZE permet d'obtenir le temps nécessaire à la planification de la requête, donc le temps nécessaire pour les 3 premières étapes, et le temps d'exécution, donc le temps nécessaire pour la dernière étape, hors temps de récupération du résultat par le client :

```
EXPLAIN ANALYZE SELECT * FROM employes ;
                        QUERY PLAN
-----------------------------------------------------------------------
 Seq Scan on employes  (cost=0.00..1.02 rows=2 width=166)
                       (actual time=0.020..0.022 rows=4 loops=1)
 Planning time : 0.087 ms
 Execution time : 0.070 ms
(3 rows)
```

On voit que pour une requête simple, le temps de planification est égal, voire supérieur au temps d'exécution. Pour des requêtes plus lourdes, le temps de planification sera négligeable par rapport au temps d'exécution.

De plus, une application n'exécute souvent qu'un nombre assez limité de requêtes SQL. Il serait donc intéressant de pouvoir réaliser la planification qu'une seule fois et de pouvoir exécuter la requête à volonté, et ainsi économiser le temps de planification. Les requêtes préparées permettent cela. L'inconvénient est que nous n'aurons qu'un plan générique car le plan d'exécution peut varier en fonction des valeurs utilisées dans les prédicats.

L'ordre PREPARE permet d'effectuer les trois premières actions :

```
PREPARE employe_detail
AS SELECT
    matricule,
    nom,
    prenom,
    fonction,
    date_embauche
FROM employes
WHERE matricule = $1;
```

On pourra exécuter la requête autant de fois que souhaité avec l'ordre EXECUTE, sans passer par la préparation du plan :

```
EXECUTE employe_detail(109) ;
EXECUTE employe_detail(127) ;
```

Si l'on regarde en détail, le temps de planification apparaît encore pour les premières exécutions :

```
EXPLAIN ANALYZE EXECUTE employe_detail(109) ;
```

```
                        QUERY PLAN
-------------------------------------------------------------
Seq Scan on employes  (cost=0.00..1.02 rows=1 width=162)
                      (actual time=0.019..0.021 rows=1 loops=1)
  Filter : (matricule = 109)
  Rows Removed by Filter : 3
Planning time : 0.207 ms
Execution time : 0.074 ms
5 rows)
```

Ensuite le temps de planification reste faible mais constant, aux erreurs de chronométrage prêt :

```
EXPLAIN ANALYZE EXECUTE employe_detail(109);
                        QUERY PLAN
-------------------------------------------------------------
 Seq Scan on employes  (cost=0.00..1.02 rows=1 width=162)
                       (actual time=0.008..0.008 rows=1 loops=1)
   Filter : (matricule = $1)
   Rows Removed by Filter : 3
 Planning time : 0.009 ms
 Execution time : 0.028 ms
(5 rows)
```

Ceci s'explique par le fait que PostgreSQL recalcule systématiquement les premiers plans d'exécution en fonction des valeurs spécifiques des prédicats. Il compare ensuite le plan obtenu avec le plan générique qu'il aura calculé au moment du `PREPARE`. Si le nouveau plan est plus intéressant, il recalculera systématiquement le plan à chaque exécution. En général, c'est bénéfique, par exemple pour des requêtes sur des tables partitionnées. Si le nouveau plan n'est pas plus intéressant, PostgreSQL restera sur le plan générique. C'est ce qui arrive ici au bout de quelques exécutions.

Un client JDBC utilise habituellement des requêtes préparées par défaut. Ce n'est pas forcément le cas pour les autres langages de programmation. Tous les pilotes PostgreSQL proposent une API pour appeler les ordres `PREPARE` et `EXECUTE` sans avoir à écrire les ordres SQL correspondant explicitement.

Il convient néanmoins de prendre en considération les limitations suivantes lorsque l'on utilise les requêtes préparées :

- le texte de la requête et le plan associé sont stockés en mémoire, ainsi que le nombre et les types de paramètres, cela peut contribuer à une consommation excessive de la mémoire sur le serveur ;
- il n'y a pas de cache partagé de plan entre les sessions, chaque session doit construire son propre cache.

OFFSET ET PAGINATION DES RÉSULTATS

Les requêtes de pagination peuvent facilement poser des problèmes de performance lorsqu'elles n'ont pas été conçues en vue d'obtenir les meilleures performances possibles. Le mot-clé `OFFSET` (ne pas afficher les *n* premières lignes remontées par la requête) est parfois utilisé en combinaison avec le mot-clé `LIMIT` (n'afficher que les *n* première lignes remontées par la requête, après application du `OFFSET`) pour écrire ce type de fonctionnalité.

Malheureusement, cette méthode d'écriture implique que le temps d'exécution de la requête va se dégrader lors de l'affichage des pages après la première. En effet, il est nécessaire de ramener tous les résultats avant de pouvoir retourner les enregistrements souhaités. Sur une requête répondant rapidement, le problème de performance ne se ressent pas de façon marquée. En revanche, il peut devenir important si le volume de données traité est important, ou si la requête pour obtenir le jeu de données à paginer est lente.

Par exemple, un jeu d'essai simple montrant les impacts de la technique :

```
CREATE TABLE pagination (
    id serial primary key,
    id_classement int,
    contenu text
);

INSERT INTO pagination (id_classement, contenu)
SELECT random()*100::int, md5(i ::text) FROM generate_series(1, 1000000) i;

CREATE INDEX ON pagination (id_classement, id);
```

Voici le plan d'exécution de la requête qui serait utilisé pour paginer les résultats d'une sélection simple. On voit que 17 blocs sont accédés via un parcours d'index sur la table `pagination` :

```
EXPLAIN (ANALYZE, BUFFERS)
SELECT id, id_classement, contenu
FROM pagination
WHERE id_classement = 85
ORDER BY id
LIMIT 10
OFFSET 10;
                            QUERY PLAN
-------------------------------------------------------------------
Limit  (cost=27.46..54.49 rows=10 width=41)
               (actual time=0.051..0.065 rows=10 loops=1)
  Buffers : shared hit=17
  ->  Index Scan using pagination_id_classement_id_idx on pagination
```

```
            (cost=0.42..25050.46 rows=9267 width=41)
            (actual time=0.032..0.059 rows=20 loops=1)
        Index Cond : (id_classement = 85)
        Buffers : shared hit=17
Planning time : 0.198 ms
Execution time : 0.127 ms
```

Cette requête ne pose aucune difficulté en soi. Mais les problèmes arrivent lorsque l'on souhaite obtenir des données sur une plage plus lointaine, en sautant par exemple les 1000 premiers enregistrements :

```
EXPLAIN (ANALYZE, BUFFERS)
SELECT id, id_classement, contenu
FROM pagination
WHERE id_classement = 85
ORDER BY id
LIMIT 10
OFFSET 1000;
                          QUERY PLAN
-----------------------------------------------------------------------
 Limit  (cost=2703.57..2730.60 rows=10 width=41)
        (actual time=1.553..1.569 rows=10 loops=1)
   Buffers : shared hit=637
   ->  Index Scan using pagination_id_classement_id_idx on pagination
               (cost=0.42..25050.46 rows=9267 width=41)
               (actual time=0.038..1.399 rows=1010 loops=1)
         Index Cond : (id_classement = 85)
         Buffers : shared hit=637
 Planning time : 0.259 ms
 Execution time : 1.645 ms
(7 rows)
```

Le temps d'exécution a été multiplié par 10. On lit également beaucoup plus : 637 blocs contre 17 précédemment, soit environ 5 Mo contre 130 Ko.

Le problème est encore plus visible si l'on saute les 10 000 premières lignes, les temps explosent et le volume de données est multiplié par 10 :

```
EXPLAIN (ANALYZE, BUFFERS)
SELECT id, id_classement, contenu
FROM pagination
WHERE id_classement = 85
ORDER BY id
LIMIT 10
OFFSET 10000;
                         QUERY PLAN
-----------------------------------------------------------------------
 Limit  (cost=10575.09..10575.09 rows=1 width=41)
```

```
        (actual time=28.331..28.331 rows=0 loops=1)
   Buffers : shared hit=6109
   ->  Sort     (cost=10551.92..10575.09 rows=9267 width=41)
                (actual time=25.250..26.956 rows=9925 loops=1)
         Sort Key : id
         Sort Method : quicksort  Memory : 1160kB
         Buffers : shared hit=6109
         ->  Bitmap Heap Scan on pagination
                     (cost=176.24..9941.32 rows=9267 width=41)
                     (actual time=5.542..22.197 rows=9925 loops=1)
               Recheck Cond : (id_classement = 85)
               Heap Blocks : exact=6079
               Buffers : shared hit=6109
               ->  Bitmap Index Scan on pagination_id_classement_id_idx
                           (cost=0.00..173.93 rows=9267 width=0)
                           (actual time=2.820..2.820 rows=9925 loops=1)
                     Index Cond : (id_classement = 85)
                     Buffers : shared hit=30
 Planning time : 0.245 ms
 Execution time : 28.428 ms
(15 rows)
```

Cependant, si l'on se réfère à un scénario où l'utilisateur passe de page en page, on peut se contenter d'une requête plus simple qui filtre sur la colonne `id_classement`, en triant sur la colonne `id` et en se limitant aux 10 premiers résultats :

```
SELECT id, id_classement, contenu
FROM pagination
WHERE id_classement = 85
ORDER BY id
LIMIT 10;
  id  | id_classement |             contenu
------+---------------+----------------------------------
   43 |            85 | 17e62166fc8586dfa4d1bc0e1742c08b
  191 |            85 | 0aa1883c6411f7873cb83dacb17b0afc
  497 |            85 | 7380ad8a673226ae47fce7bff88e9c33
  582 |            85 | 46922a0880a8f11f8f69cbb52b1396be
 1299 |            85 | a0833c8a1817526ac555f8d67727caf6
 1314 |            85 | 50905d7b2216bfeccb5b41016357176b
 1400 |            85 | f0dd4a99fba6075a9494772b58f95280
 1450 |            85 | c5cc17e395d3049b03e0f1ccebb02b4d
 1460 |            85 | 07042ac7d03d3b9911a00da43ce0079a
 1482 |            85 | ab541d874c7bc19ab77642849e02b89f
(10 rows)
```

On connaît ainsi l'id le plus élevé qui a été récupéré. La requête qui nous permet de récupérer les résultats à afficher sur la seconde page, on peut donc récupérer les 10 enregistre-

ments en ajoutant un prédicat `id > 1482` pour passer les 10 premiers enregistrements :

```
SELECT id, id_classement, contenu
FROM pagination
WHERE id_classement = 85
  AND id > 1482
ORDER BY id
LIMIT 10;
  id  | id_classement |             contenu
------+---------------+----------------------------------
 1498 |            85 | ced556cd9f9c0c8315cfbe0744a3baf0
 1579 |            85 | ed4227734ed75d343320b6a5fd16ce57
 1672 |            85 | 2451041557a22145b3701b0184109cab
 1844 |            85 | 06a15eb1c3836723b53e4abca8d9b879
 1997 |            85 | 06964dce9addb1c5cb5d6e3d9838f733
 2029 |            85 | 093b60fd0557804c8ba0cbf1453da22f
 2320 |            85 | a70dc40477bc2adceef4d2c90f47eb82
 2338 |            85 | b6cda17abb967ed28ec9610137aa45f7
 2349 |            85 | 70fcb77e6349f4467edd7227baa73222
 2574 |            85 | 8f125da0b3432ed853c0b6f7ee5aaa6b
(10 rows)
```

Le plan d'exécution montre que l'on n'accède qu'à 10 blocs pour récupérer les données. La requête est extrêmement rapide.

```
EXPLAIN (ANALYZE, BUFFERS)
SELECT id, id_classement, contenu
FROM pagination
WHERE id_classement = 85
  AND id > 1482
ORDER BY id
LIMIT 10;
                          QUERY PLAN
---------------------------------------------------------------------
 Limit  (actual time=0.039..0.055 rows=10 loops=1)
   Buffers : shared hit=10
   ->  Index Scan using pagination_id_classement_id_idx on pagination
               (actual time=0.037..0.050 rows=10 loops=1)
         Index Cond : ((id_classement = 85) AND (id > 1482))
         Buffers : shared hit=10
 Planning time : 0.259 ms
 Execution time : 0.109 ms
(7 rows)
```

Cette technique de pagination permet d'avoir des temps d'accès extrêmement court, mais elle a les inconvénients suivants :

- pas de comptage possible des résultats,

- pas de possibilité d'accès en sautant plusieurs pages,
- nécessite que les colonnes sur lesquelles on filtre et sur lesquelles on réalise le tri soient indexées.

Cette technique est présentée en détail sur le site de Markus Winand[10] .

DÉTERMINER LES REQUÊTES À OPTIMISER

Toutes les requêtes ne sont pas critiques, seul un certain nombre d'entre elles méritent une attention particulière.

Il y a deux façons de déterminer les requêtes qui nécessitent d'être travaillées. La première dépend du ressenti utilisateur, il faudra en priorité traiter les requêtes interactives. Certaines auront déjà d'excellents temps de réponse, d'autres pourront être améliorées encore. Il faudra déterminer non seulement le temps de réponse maximal attendu pour une requête, mais vérifier aussi le temps total de réponse de l'application.

L'autre méthode pour déterminer les requêtes à optimiser consiste à utiliser des outils de *profiling* habituels :

- pgBadger[11] ,
- pg_stat_statements[12] ,
- PoWA[13] .

Ces outils permettront de déterminer les requêtes les plus fréquemment exécutées et permettront d'établir un classement des requêtes qui ont nécessité le plus de temps cumulé à leur exécution (voir onglet *Time consuming queries (N)* d'un rapport pgBadger par exemple). Les requêtes les plus fréquemment exécutées méritent également qu'on leur porte attention, leur optimisation peut permettre d'économiser quelques ressources du serveur. Un outil comme PoWA permet d'aller encore plus loin, avec la possibilité de visualiser les ratios d'utilisation du cache des requêtes, ou de faire de la suggestion de création d'index.

GESTION DE LA CONCURRENCE D'ACCÈS

PostgreSQL est un SGBD relationnel qui respecte l'ensemble des propriétés *ACID*. Il s'agit de quatre règles fondamentales :

- A pour atomicité : Une transaction est entière : "tout ou rien".

10. http://use-the-index-luke.com/fr/no-offset
11. https://github.com/dalibo/pgbadger
12. https://www.postgresql.org/docs/current/static/pgstatstatements.html
13. https://github.com/powa-team/powa

- C pour cohérence : Une transaction amène le système d'un état stable à un autre.
- I pour isolation : Les transactions n'agissent pas les unes sur les autres.
- D pour durabilité : Une transaction validée provoque des changements permanents.

Pour respecter la plupart de ces propriétés, PostgreSQL s'appuie sur le modèle MVCC pour MultiVersion Concurrency[14] .

Le principe est de faciliter l'accès concurrent de plusieurs utilisateurs (sessions) à la base en disposant en permanence de plusieurs versions différentes d'un même enregistrement. Chaque session peut travailler simultanément sur la version qui s'applique à son contexte (on parle d'instantané ou de *snapshot*).

IMPLÉMENTATION MVCC

Pour gérer la concurrence d'accès et les contraintes d'isolation tout en garantissant de bonnes performances, PostgreSQL implémente le MVCC, une table peut donc stocker plusieurs versions d'une même ligne. La modification d'une ligne ne modifie pas la ligne existante mais entraîne la création d'une nouvelle version de cette ligne, via un mécanisme de *Copy On Write*, à un nouvel emplacement physique dans les fichiers de la table.

L'ancienne version de la ligne continuera d'exister tant qu'un processus de nettoyage n'aura pas recyclé l'espace qu'elle occupait. Une ancienne version de ligne ne sera recyclée qu'à la condition que toutes les transactions en cours n'aient plus besoin d'accéder à cette vieille version. C'est-à-dire qu'aucune transaction n'a un instantané de la base plus ancien que l'opération qui a créé cette version de ligne. Cette version est alors invisible pour tout le monde et l'espace qu'elle occupe peut être recyclé, on parle alors de ligne morte (`dead tuple`). Chaque version d'enregistrement contient bien sûr les informations permettant de déterminer s'il est visible ou non dans un contexte donné. L'opération de maintenance qui réalise ce nettoyage s'appelle `VACUUM`. Elle ne nécessite pas de prendre de verrou lourd et n'entraîne donc pas le blocage des sessions concurrentes. Seule sa déclinaison `VACUUM FULL` est bloquante car elle doit réécrire intégralement la table, c'est donc une opération lourde et risquée, qui ne devrait être réalisée que manuellement sur une plage de maintenance. Le processus `autovacuum` permet de déclencher régulièrement et automatiquement `VACUUM` suite à des modifications de données, il est activé par défaut sur les versions courantes de PostgreSQL.

Par exemple, un `UPDATE` de toute une table peut entraîner la duplication de toutes les lignes affectées. L'espace ne pourra être récupéré que grâce à des opérations de maintenance, et il est probable qu'à la fin d'une transaction de ce type le processus `autovacuum` se déclenche automatiquement. Par ailleurs, lors de la mise à jour d'une ligne, et donc la

14. https://fr.wikipedia.org/wiki/Multiversion_Concurrency_Control

création d'une nouvelle version de cette ligne, il faut également mettre à jour les index : la dernière version de l'enregistrement a changé d'emplacement physique.

Toutefois il existe une exception à ce principe : sous certaines conditions, le moteur peut enregistrer plusieurs versions d'une même ligne dans le même bloc

Voir https://github.com/postgres/postgres/blob/master/src/backend/access/heap/README.HOT

et

http://www.interdb.jp/pg/pgsql07.html.

Ceci permettant au fur et à mesure des mises à jour de supprimer automatiquement les anciennes versions, sans besoin de VACUUM. Cela permet aussi de ne pas toucher aux index, qui pointent donc grâce à cela sur plusieurs versions du même enregistrement. Les conditions sont que :

- le bloc contienne assez de place pour la nouvelle version (on ne chaîne pas les enregistrements entre plusieurs blocs) ;
- aucune colonne indexée n'a été modifiée par l'opération.

Cette implémentation nécessite donc de faire attention à ne pas utiliser des transactions trop longues. Des transactions ouvertes depuis trop longtemps (actives ou non, donc cela inclut les sessions à l'état *idle in transaction*) auront un impact fort sur le recyclage des lignes mortes. L'opération de nettoyage `VACUUM` ne pourra purement et simplement pas supprimer les anciennes versions de lignes qui auront été supprimées depuis le début de la transaction ouverte.

Il est important de prendre en compte ces spécificités lors de l'utilisation de PostgreSQL :

- éviter les index inutiles, ou d'indexer trop de colonnes ;
- éviter de réaliser des modifications successives des mêmes données dans une transaction ;
- éviter de réaliser des `UPDATE` si les données n'ont pas changé ;
- éviter les transactions longues, ne pas laisser de transaction ouverte et inactive.

GESTION DES TRANSACTIONS

Une transaction est un ensemble atomique d'opérations. C'est à dire que la transaction est exécutée en entier ou pas du tout. On ouvre une transaction avec l'ordre `BEGIN`. La transaction est validée avec l'ordre `COMMIT` mais elle peut aussi être annulée avec l'ordre `ROLLBACK`. Cependant, si aucune transaction n'est ouverte explicitement, PostgreSQL travaille en auto-commit. Dans ce mode, chaque ordre SQL est exécuté dans sa propre tran-

saction. Ce principe de fonctionnement est valable pour tous les langages, à l'exception de Java avec le pilote JDBC.

Voici un exemple de transaction :

```
BEGIN;
CREATE TABLE capitaines (id serial, nom text, ageinteger) ;
INSERT INTO capitaines VALUES (1, 'Haddock',35) ;
SELECT age FROM capitaines ;
 age
-----
  35
(1 ligne)
ROLLBACK;

SELECT age FROM capitaines ;
ERROR :  relation "capitaines" does not exist
LINE 1: SELECT age FROM capitaines ;
```

On voit que la table capitaine a existé à l'intérieur de la transaction. Mais puisque cette transaction a été annulée (`ROLLBACK`), la table n'a pas été créée au final. Cela montre aussi le support du DDL transactionnel au sein de PostgreSQL.

Un point de sauvegarde est une marque spéciale à l'intérieur d'une transaction qui autorise l'annulation de toutes les commandes exécutées après son établissement, restaurant la transaction dans l'état où elle était au moment de l'établissement du point de sauvegarde :

```
BEGIN;
CREATE TABLE capitaines (id serial, nom text, ageinteger) ;
INSERT INTO capitaines VALUES(1,'Haddock',35) ;
SAVEPOINT insert_sp ;
UPDATE capitaines SET age=45 WHERE nom='Haddock';
ROLLBACK TO SAVEPOINT insert_sp ;
COMMIT;

SELECT age FROM capitaines WHERE nom='Haddock';
 age
-----
  35
(1 row)
```

Malgré le `COMMIT` après l'`UPDATE`, la mise à jour n'est pas prise en compte. En effet, le `ROLLBACK TO SAVEPOINT` a permis d'annuler cet `UPDATE` mais pas les opérations précédant le `SAVEPOINT`. Ce mécanisme est intéressant dans certains cas bien précis, mais il n'est pas recommandé d'en abuser, car de trop nombreux `SAVEPOINT` peuvent entraîner des pertes de performances importants. Pour simplifier, ce n'est pas une bonne idée de

placer un `SAVEPOINT` avant chaque ordre SQL.

Comme indiqué précédemment, les transactions longues sont fortement pénalisantes pour le moteur, car elles réservent des ressources et peuvent bloquer d'autres opérations comme les tâches de maintenance.

NIVEAUX D'ISOLATION

Chaque transaction (et donc session) est isolée à un certain point :

- elle ne voit pas les opérations des autres ;
- elle s'exécute indépendamment des autres.

Le standard SQL spécifie quatre niveaux d'isolation, mais PostgreSQL n'en supporte que trois :

- `READ COMMITTED` (par défaut) : les requêtes ne peuvent lire que des données qui ont été validées, pas les données modifiées par les transactions en cours. Toute donnée modifiée par une transaction validée devient visible pour les autres transactions utilisant ce niveau ;
- `REPEATABLE READ` : une session ne voit que les transactions commitées depuis le début de la session et ne peut pas modifier les données lues par ailleurs. C'est le niveau utilisé pour les exports `pg_dump`, pour avoir un équivalent de snapshot au début de la sauvegarde, et donc une sauvegarde cohérente ;
- `SERIALIZABLE` : ce niveau est un renforcement du mode `REPEATABLE READ` qui permet de développer comme si chaque transaction se déroulait seule sur la base. Plus précisément, une session ouverte à ce niveau ne pourra pas créer de nouvel enregistrement dont la clé est dans une plage de valeurs lue par ailleurs. En cas d'incompatibilité entre les opérations réalisées par plusieurs transactions, PostgreSQL annule celle qui déclenchera le moins de perte de données.

Le niveau manquant `READ UNCOMMITTED` permet à une transaction de voir les données des transactions non commitées, ce qui implique que chaque requête obtient une vision incohérente des données. Ce niveau d'isolation est avant tout prévu pour améliorer la concurrence d'accès en limitant le verrouillage, ce qui serait redondant et moins efficace que l'utilisation de MVCC, il n'est donc pas implémenté dans PostgreSQL. Si une transaction demande ce niveau, PostgreSQL utilisera le niveau `READ COMMITTED` à la place.

On peut spécifier le niveau d'isolation au démarrage d'une transaction, par exemple ici en `SERIALIZABLE` :

```
BEGIN ISOLATION LEVEL READ SERIALIZABLE;
```

Le niveau de séparation demandé sera un compromis entre le besoin applicatif (pouvoir ignorer sans risque ce que font les autres transactions) et les contraintes imposées au niveau de PostgreSQL (performances, risque d'échec d'une transaction).

GESTIONNAIRE DE VERROUS

Afin de garantir une isolation correcte entre les différentes sessions, le SGBD a besoin de protéger certaines opérations. On ne peut, par exemple, pas autoriser une session à modifier un enregistrement déjà modifié par une autre transaction, tant qu'on ne sait pas si cette dernière a validé ou annulé sa modification. On utilise donc une méthode de verrouillage pour éviter cette situation.

Le gestionnaire de verrous de PostgreSQL est capable de gérer des verrous sur des tables, sur des enregistrements, sur des ressources virtuelles.

De nombreux types de verrous sont disponibles, chacun entrant en conflit avec d'autres[15]

Chaque opération doit tout d'abord prendre un verrou sur les objets à manipuler.

Les noms des verrous peuvent prêter à confusion : `ROW SHARE` par exemple est un verrou de table, pas un verrou d'enregistrement. Il signifie qu'on a pris un verrou sur une table pour y faire des `SELECT FOR UPDATE` par exemple. Ce verrou est en conflit avec les verrous pris pour un `DROP TABLE`, ou pour un `LOCK TABLE`.

Lorsque PostgreSQL effectue la lecture d'une table, il pose un verrou de consultation (`ACCESS SHARE`) sur la table et sur un éventuel index lu. Cela évite qu'une autre transaction ne viennent supprimer la table au cours de la lecture.

Un verrou exclusif est également posé implicitement sur chaque ligne modifiée. Cela impacte surtout les mises à jour et les suppressions, les verrous exclusifs bloqueront la mise à jour ou la suppression de ces lignes par d'autres transactions. Ce n'est pas un comportement problématique en soit car il permet d'assurer l'atomicité d'une transaction. Mais les verrous posés n'empêchent pas les mises à jour fantômes que l'on peut facilement obtenir avec le niveau d'isolation `READ COMMITTED` (Cette notion est expliquée dans la section Problèmes d'accès concurrents).

Certains SGBD verrouillent totalement l'enregistrement modifié. Celui-ci n'est plus accessible même en lecture tant que la modification n'a pas été validée ou annulée. Cela a l'avantage d'éviter aux sessions en attente de voir une ancienne version de l'enregistrement, mais le défaut de les bloquer, et donc de fortement dégrader les performances. La gestion d'un grand nombre de verrous pose également des problèmes.

15. https://www.postgresql.org/docs/current/static/explicit-locking.html

Le modèle MVCC utilisé par PostgreSQL permet à chaque enregistrement de cohabiter en plusieurs versions simultanées en base. Cela permet d'éviter que les écrivains ne bloquent les lecteurs ou les lecteurs ne bloquent les écrivains, tout en limitant fortement le nombre de verrous. Cela permet aussi de garantir un instantané de la base à une requête, sur toute sa durée, voire sur toute la durée de sa transaction si la session le demande (`BEGIN ISOLATION LEVEL REPEATABLE READ`).

Certaines opérations (DDL et certaines opérations de maintenance comme `REINDEX` ou `VACUUM FULL`) utilisent des verrous dits exclusifs, qui bloquent tout type d'accès aux relations concernées tant que le verrou n'est pas relâché.

Il est important de prendre en considération les éléments suivants lorsque l'on développe une application avec PostgreSQL :

- les verrous ont une existence en mémoire partagée, ils sont donc limités en nombre ;
- le gestionnaire de verrous ne crée pas un verrou individuel pour chaque ligne accédée, ce qui limite la consommation de verrous ;
- il n'y a pas de mécanisme d'escalade de verrous ;
- tous les accès aux relations consomment des verrous, y compris les manipulations d'objets créés temporairement au sein de la session ;
- les verrous ne sont libérés qu'une fois la transaction validée (`COMMIT`) ou annulée (`ROLLBACK`), les transactions très longues peuvent donc poser des problèmes.

Par ailleurs, pour éviter des contentions, voire des *deadlocks*, on fera toujours les mises à jour ou les acquisitions des verrous dans le même ordre dans toutes les transactions : par exemple toutes les transactions prennent `table1` puis `table2`. En effet, une transaction qui prend d'abord `table1` puis `table2` et une autre transaction qui prend d'abord `table2` puis `table1`, c'est la garantie d'avoir de nombreux *deadlocks*.

DEADLOCKS

Les *deadlocks* se produisent quand plusieurs sessions acquièrent simultanément des verrous et s'interbloquent. Par exemple :

```
Session 1          Session 2
BEGIN              BEGIN
UPDATE demo
   SET a=10
 WHERE a=1;
                   UPDATE demo
                   SET a=11
                   WHERE a=2;
```

```
UPDATE demo
   SET a=11
 WHERE a=2;                          -- Session 1 bloquée. Attend session 2.
                 UPDATE demo
                 SET a=10
                 WHERE a=1;          --  Session 2 bloquée. Attend session 1.
```

Bien sûr, la situation ne reste pas en l'état. Une session qui attend un verrou appelle au bout d'un temps court (une seconde par défaut sous PostgreSQL) le gestionnaire de *deadlock*, qui finira par tuer une des deux sessions. Dans cet exemple, il sera appelé par la session 2, ce qui débloquera la situation.

Une application qui a beaucoup de *deadlocks* a plusieurs problèmes :

- Les transactions attendent beaucoup (utilisation de toutes les ressources machine difficile),
- Certaines finissent annulées et doivent donc être rejouées (travail supplémentaire).

Dans notre exemple, on aurait pu éviter le problème, en définissant une règle simple : toujours verrouiller par valeurs de `a` croissante. Dans la pratique, sur des cas complexes, c'est bien sûr bien plus difficile à faire. Par ailleurs, un *deadlock* peut impliquer plus de deux transactions. Mais simplement réduire le volume de *deadlocks* aura toujours un impact très positif sur les performances.

On peut aussi déclencher plus rapidement le gestionnaire de *deadlock*. 1 seconde, c'est quelquefois une éternité dans la vie d'une application. Sous PostgreSQL, il suffit de modifier le paramètre `deadlock_timeout`. Plus cette variable sera basse, plus le traitement de détection de *deadlock* sera déclenché souvent. Et celui-ci peut être assez gourmand si de nombreux verrous sont présents, puisqu'il s'agit de détecter des cycles dans les dépendances de verrous.

PROBLÈMES D'ACCÈS CONCURRENTS

Pris à la légère, les problématiques d'accès concurrents peuvent entraîner différentes anomalies, qui peuvent avoir des conséquences graves sur la cohérence des données manipulées par les sessions.

Pour approfondir le sujet : https://en.wikipedia.org/wiki/Isolation_(database_systems)#Read_phenomena

Le premier problème théorique lié à la concurrence est le phénomène de *dirty read* : la possibilité de lire des données invalides, temporaires, ou d'une manière générale liées à une autre transaction toujours en cours. Ce phénomène est lié au niveau d'isolation `READ UNCOMMITED`, et ne peut donc pas apparaître sur PostgreSQL.

Par défaut et sauf configuration particulière, une transaction s'exécute sur PostgreSQL dans le niveau d'isolation `READ COMMITTED`. Dans ce niveau d'isolation, chaque **requête SQL** voit les données dans un nouvel instantané. Ainsi, **dans une même transaction**, le même `SELECT` exécuté à deux moments différents ne verra pas les mêmes données si d'autres transactions ont validé des modifications entre temps. C'est le phénomène des *non repeatable reads*.

Par exemple :

```
Session 1                Session 2
BEGIN;
SELECT nom, prenom
  FROM employes ;
   nom    | prenom
----------+--------
 Prunelle |
(1 row)

                         BEGIN;
                         UPDATE employes
                         SET prenom = 'Léon'
                         WHERE nom = 'Prunelle';
                         COMMIT;

SELECT nom, prenom       -- la session 1 verra l'enregistrement à jour
  FROM employes ;        -- alors que c'est toujours la même transaction
   nom    | prenom
----------+--------
 Prunelle | Léon
(2 rows)

COMMIT;
```

Les *phantom reads* sont un cas particulier des *non repeatable read*, également observable avec le niveau `READ COMMITED`, dans lequel des enregistrements qui ne satisfaisaient pas les prédicats de la clause `WHERE` à la première exécution peuvent apparaître lors d'exécutions suivantes de la même transaction.

Par exemple :

```
Session 1                Session 2
BEGIN;
SELECT nom, prenom
  FROM employes
 WHERE age < 50;
   nom    | prenom
----------+--------
```

```
 Prunelle | Léon
(1 row)
```

```
                        BEGIN;
                        INSERT INTO employes
                        VALUES ('Lagaffe', 'Gaston', 18);
                        COMMIT;
```

```
SELECT nom, prenom          -- la session 1 verra le nouvel employé Gaston Lagaffe
  FROM employes;            -- alors que c'est toujours la même transaction
 WHERE age < 50;
   nom     | prenom
-----------+--------
 Prunelle | Léon
 Lagaffe  | Gaston
(2 rows)

COMMIT;
```

Ces deux phénomènes disparaissent sous PostgreSQL lorsque l'on ouvre une transaction utilisant le niveau d'isolation `REPEATABLE READ` (PostgreSQL est plus strict que la norme puisque les *phantom reads* sont corrigés sans avoir besoin de passer en `SERIALIZABLE`).

Ce niveau permet en effet d'obtenir une transaction travaillant sur un *snapshot* des données de la base telles qu'elles étaient **au début de la transaction**.

Deux ordres SQL consécutifs dans la même transaction retourneront donc les mêmes enregistrements, dans la même version, sans considération pour les modifications apportées par les autres transactions validées. En lecture seule, ce type de transaction ne peut pas échouer (ce niveau d'isolation est, entre autres, utilisé pour réaliser des exports cohérents des données par pg_dump).

En écriture par contre (ou `SELECT FOR UPDATE`, `FOR SHARE`), si les données que la transaction tente de modifier ont déjà été modifiées par une autre transaction validée (la nouvelle version étant invisible à la transaction en cours), la transaction en `REPEATABLE READ` va échouer immédiatement avec l'erreur suivante :

```
ERROR : could not serialize access due to concurrent update
```

Il faut donc que l'application utilisant ce niveau d'isolation soit conçue pour être capable de rejouer la transaction au besoin.

Par ailleurs, même dans ce niveau d'isolation, des anomalies indiquant que les transactions ne sont pas complètement isolées peuvent se produire. Un exemple est le phénomène des *write skews*, lorsque deux transactions accèdent chacune aux données d'une table, mais ne modifient qu'un sous-ensemble disjoint de données.

Un exemple détaillé en REPEATABLE READ, montrant qu'à la fin des deux transactions le résultat est différent de ce qu'il aurait été si les deux transactions s'étaient exécutées de façon sérialisée, en isolation complète :

```
SELECT * FROM dots ;
 id | color
----+-------
  1 | black
  2 | white
  3 | black
  4 | white
  5 | black
  6 | white
  7 | black
  8 | white
  9 | black
 10 | white
(10 lignes)
```

```
Session 1
BEGIN ISOLATION LEVEL
      REPEATABLE READ;
UPDATE dots
   SET color = 'black'
 WHERE color = 'white';
```

```
Session 2

BEGIN ISOLATION LEVEL
      REPEATABLE READ;
UPDATE dots
   SET color = 'white'
 WHERE color = 'black';

COMMIT ;

SELECT * FROM dots ;
 id | color
----+-------
  2 | white
  4 | white
  6 | white
  8 | white
 10 | white
  1 | white
  3 | white
  5 | white
  7 | white
```

```
                          9 | white
                         (10 lignes)

COMMIT;

SELECT * FROM dots ;
 id | color
----+-------
  2 | black
  4 | black
  6 | black
  8 | black
 10 | black
  1 | white
  3 | white
  5 | white
  7 | white
  9 | white
(10 lignes)
```

Pour éviter ce comportement, on peut utiliser le niveau d'isolation SERIALIZABLE[16]

Dans ce niveau d'isolation, la transaction s'exécute comme si elle était seule sur la base. PostgreSQL pourra le vérifier au moment de la validation de la transaction (`COMMIT`). Si un conflit de mise à jour est détecté, PostgreSQL choisira d'annuler une des transactions en conflit. L'application doit alors être capable de rejouer la transaction échouée, ou au moins de traiter l'erreur correctement. Si les deux sessions de l'exemple *write skews* avaient été dans le niveau `SERIALIZABLE`, le `COMMIT` aurait échoué avec l'erreur suivante :

```
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.
```

Ce mode empêche notamment les anomalies dues à une transaction effectuant un `SELECT` d'enregistrements, puis d'autres traitements, pendant qu'une autre transaction modifie les enregistrements vus par le `SELECT` : il est probable que le SELECT initial de notre transaction utilise les enregistrements récupérés, et que le reste du traitement réalisé par notre transaction dépende de ces enregistrements. Si ces enregistrements sont modifiés par une transaction concurrente, notre transaction ne s'est plus déroulée comme si elle était seule sur la base, et on a donc une violation de sérialisation.

Utiliser le mode `SERIALIZABLE` permettra également de s'affranchir des ordres `SELECT`

16. https://dali.bo/sql2_html#serializable-snapshot-isolation

`FOR UPDATE` que l'on positionne en général au niveau `READ COMMITTED` pour verrouiller explicitement les enregistrements lus à un moment de la transaction.

Attention, 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.

À noter que certaines commandes particulières (`TRUNCATE`, `COPY FREEZE`, etc.) peuvent provoquer des anomalies spécifiques à PostgreSQL[17] .

VERROUILLAGE EXPLICITE

Pour pallier aux différents problèmes liés à la concurrence d'accès, il est également possible d'utiliser des techniques de verrouillage explicites. On peut utiliser différentes techniques de verrouillage en fonction du contexte, soit *pessimiste* avec les possibilités de la base de données, soit *optimiste* pour les contextes Web.

Attention, par nature les techniques de verrouillage réduisent les capacités de PostgreSQL à optimiser les accès concurrents : l'objectif de poser un verrou explicite est de sérialiser les accès, donc **d'empêcher les accès concurrents**. Lorsque cela est possible, il est préférable d'éviter d'y recourir. Lorsque leur usage est nécessaire, il est extrêmement important de bien concevoir la gestion des verrous par l'application, de toujours prendre les verrous dans le même ordre, de les libérer le plus tôt possible (voir le Dining_philosophers_problem[18]).

La gestion *pessimiste* des verrous va consister à utiliser les outils de verrouillage explicite à disposition dans PostgreSQL.

Par exemple, on utilisera `SELECT FOR UPDATE` pour lire des données et mettre les tentatives de verrouillages ou mises à jour concurrentes en attente, sans pour autant empêcher les lectures parallèles :

```
BEGIN;

SELECT *
FROM stocks
WHERE nom_livre = 'Voyage au centre de la Terre '
FOR UPDATE;     -- la ligne correspondante est maintenant verrouillée

-- autres opérations de la transactions
UPDATE stocks
SET quantite = 42
```

17. https://wiki.postgresql.org/wiki/MVCC_violations
18. https://en.wikipedia.org/wiki/Dining_philosophers_problem

```
WHERE nom_livre = 'Voyage au centre de la Terre ';

COMMIT;
```

Dans le contexte d'applications en mode Web, les transactions ne durent souvent que le temps de calcul d'une nouvelle page. Cela a pour avantage d'éviter des transactions longues, qui posent de nombreux problèmes. En contrepartie, il n'est pas possible de maintenir des verrous actifs sur des actions applicatives qui peuvent se réaliser sur plusieurs écrans et donc plusieurs transactions côté base de données. Il devient alors impossible de maintenir l'atomicité et l'isolation pendant le temps de cette "transaction métier". La solution à ce problème est alors d'utiliser une méthode de verrouillage *optimiste*.

Cette méthode s'appuie sur la présence d'une colonne permettant de versionner les enregistrements. Ainsi, lors de la première lecture des données, nous connaîtrons la version des enregistrements présentés à l'utilisateur. Après un temps donné, nous mettons les données à jour. Le fait de connaître la version de l'enregistrement permettra de valider que l'enregistrement n'a pas été mis à jour depuis sa lecture. Si elle a changé depuis la première lecture, alors un autre utilisateur a mis cette ligne à jour, alors nous ne faisons pas la mise à jour et retournons un message à l'utilisateur pour le lui indiquer. Si le numéro de version n'a pas changé, nous pouvons faire la mise à jour.

Concrètement, il s'agit donc d'ajouter un numéro de version à chaque enregistrement, par le biais de la colonne `version` de table `employes` :

```
CREATE TABLE employes (
    matricule     serial primary key,
    version       integer     not null,
    nom           varchar(15) not null,
    prenom        varchar(15) not null,
    fonction      varchar(20) not null,
    date_embauche date
);

INSERT INTO employes VALUES
    (109, 1, 'Lagaffe', 'Gaston', 'Inventeur', '2000-06-01'),
    (127, 1, 'Moizelle', 'Jeanne', 'Soutient d`inventeur', '2000-06-01');
```

Marrions maintenant nos deux protagonistes. Sur un premier écran, l'utilisateur recherche *Jeanne Moizelle* dans son fichier et récupère son identité ainsi que le numéro de version de la ligne correspondante :

```
BEGIN;
SELECT matricule, nom, prenom, version
FROM employes WHERE nom = 'Moizelle' AND prenom = 'Jeanne';
COMMIT;
```

La modification du nom est ensuite réalisée et l'application réalise la mise à jour, cette fois à partir du matricule qui est la clé primaire de la table :

```
BEGIN;
-- verrouillage de la ligne
SELECT
  matricule, nom, prenom, version
  FROM employes
WHERE matricule = 127
FOR UPDATE;
```

La transaction n'est pas terminée. On peut valider que le numéro de version n'a pas été modifié de différente manière ici. Dans l'application en comparant le numéro de version obtenu sur le premier écran avec le numéro de version obtenu sur le `SELECT FOR UPDATE`. Une autre façon de procéder est de valider cela dans la requête `UPDATE` directement, mais il faudra vérifier le nombre de lignes mises à jour par l'ordre SQL. Une troisième méthode consiste à réaliser l'ordre `UPDATE` directement plutôt que de passer par un premier `SELECT FOR UPDATE` puis un `UPDATE`. Cette dernière façon de faire évite un aller-retour entre le serveur d'application et la base de données.

Dans tous les cas, nous réalisons la mise à jour :

```
UPDATE employes
SET nom = 'Lagaffe',
    version = version + 1 -- incrément de version calculé par la base
WHERE matricule = 127
  AND version = 1; -- prédicat ajouté si l'on utilise la seconde ou la
                   -- troisième méthode de validation.
                   -- à ce stade, l'application est capable de savoir
                   -- combien de lignes ont été mises à jour par UPDATE
                   -- si aucune ligne n'est mise à jour, l'UPDATE a échoué
                   -- du fait d'une mise à jour concurrente
COMMIT;
```

STATISTIQUES

L'optimiseur se base principalement sur les statistiques issues de l'échantillonnage des données pour ses décisions. Le choix du parcours, le choix des jointures, le choix de l'ordre des jointures, tout cela dépend des statistiques (et un peu de la configuration). Sans statistiques à jour, le choix du planificateur a un fort risque d'être mauvais. Il est donc important que les statistiques soient mises à jour fréquemment. La mise à jour se fait avec l'instruction `ANALYZE` qui peut être exécuté manuellement ou automatiquement (via un cron ou `autovacuum` par exemple).

MISE À JOUR DES STATISTIQUES

Une cause habituelle de problèmes de performances est le manque de fraicheur des statistiques. Cela arrive souvent après le chargement massif d'une table ou une mise à jour massive sans avoir fait une nouvelle collecte des statistiques à l'issue de ces changements.

On utilisera l'ordre `ANALYZE table` pour déclencher explicitement la collecte des statistiques après un tel traitement. En effet, bien qu'autovacuum soit présent, il peut se passer un certain temps entre le moment où le traitement est fait et le moment où autovacuum déclenche une collecte de statistiques. Ou autovacuum peut simplement ne pas se déclencher car le traitement complet est imbriqué dans une seule transaction et il ne voit donc pas les modifications réalisées par la transaction, ou parce qu'il s'agit d'une table temporaire donc totalement invisible à autovacuum.

Un traitement batch devra comporter des ordres `ANALYZE` juste après les ordres SQL qui modifient fortement les données :

```
COPY table_travail FROM '/tmp/fichier.csv';
ANALYZE table_travail ;
SELECT ... FROM table_travail ;
```

MAUVAISE ÉCRITURE DES PRÉDICATS

Dans un prédicat, lorsque les valeurs des colonnes sont transformées par un calcul ou par une fonction, l'optimiseur n'a aucun moyen pour connaître la sélectivité du prédicat. Il utilise donc une estimation codée en dur dans le code de l'optimiseur : 0,5 % du nombre de lignes de la table.

Dans la requête suivante, l'optimiseur estime que la requête va ramener 2495 lignes :

```
EXPLAIN
SELECT * FROM employes_big
WHERE extract('year' from date_embauche) = 2006;
                        QUERY PLAN
-------------------------------------------------------------
 Gather  (cost=1000.00..9552.15 rows=2495 width=40)
   Workers Planned : 2
   ->  Parallel Seq Scan on employes_big
       (cost=0.00..8302.65 rows=1040 width=40)
         Filter : (date_part('year'::text,
                  (date_embauche) ::timestamp without time zone)
                  = '2006'::double precision)
(4 rows)
```

Ces 2495 lignes correspondent à 0,5 % de la table commandes :

```
SELECT relname, reltuples, round(reltuples*0.005) AS estimé
FROM pg_class
WHERE relname = 'employes_big';

   relname    | reltuples | estimé
--------------+-----------+--------
 employes_big |    499015 |   2495
(1 row)
```

TAILLE D'ÉCHANTILLON DES STATISTIQUES

Un autre problème qui peut se poser avec les statistiques concerne les tables de très forte volumétrie, particulièrement si la répartition des valeurs n'est pas uniforme. Dans certains cas, l'échantillon de données ramené par `ANALYZE` n'est pas assez précis pour donner à l'optimiseur de PostgreSQL une vision suffisamment précise des données. Il choisira alors de mauvais plans d'exécution.

Il est possible d'augmenter la précision de l'échantillon de données collecté pour une colonne à l'aide de l'ordre :

```
ALTER TABLE ... ALTER COLUMN ... SET STATISTICS ... ;
```

CORRÉLATIONS DE DONNÉES

PostgreSQL conserve des statistiques par colonne simple. Imaginons la requête suivante :

```
SELECT *
FROM t1
WHERE c1=1
  AND c2=1;
```

- `c1=1` est vrai pour 20 % des lignes
- `c2=1` est vrai pour 10 % des lignes

Le planificateur sait grâce à l'échantillonnage que l'estimation pour `c1=1` est de 20 % et que l'estimation pour `c2=1` est de 10 %.

Par contre, il n'a aucune idée de l'estimation pour `c1=1 AND c2=1`.

En réalité, l'estimation pour cette formule va de 0 % (valeurs inversement corrélées) à 10 % (valeurs complètements corrélées) mais le planificateur doit statuer sur une seule valeur.

Ce sera le résultat de la multiplication des deux estimations, soit 2 % (20 % * 10 % : valeurs complètement indépendantes, aucune corrélation entre les colonnes). Dans le cas où les

valeurs sont fortement corrélées, l'estimation de cardinalité de l'optimiseur va donc être largement erronée, ce qui peut mener à de très mauvais plans d'exécution.

La version 10 de PostgreSQL corrige cela en ajoutant la possibilité d'ajouter des statistiques sur plusieurs colonnes spécifiques. Ce n'est pas automatique, il faut créer un objet statistique avec l'ordre `CREATE STATISTICS`. Dans les versions précédentes de PostgreSQL, contourner ce problème implique de créer un index fonctionnel sur la combinaison des deux colonnes pour forcer la collecte des statistiques. Malheureusement, cela implique de devoir modifier la requête pour utiliser la fonction de combinaison.

UTILISER LES INDEX

Le premier usage d'un index est de réduire le volume de données à parcourir.

Dans de nombreux cas, les problèmes de performances d'une requête sont liés à la non utilisation d'un index. Il est donc important lors de l'écriture ou de l'optimisation d'une requête de s'assurer qu'un index pertinent existe bien.

PostgreSQL offre de nombreuses possibilités d'indexation des données :

- Type d'index : B-tree, GiST, GIN, SP-GiST, BRIN et hash.
- Index multi-colonnes : `CREATE INDEX ... ON (col1, col2...);`
- Index partiel : `CREATE INDEX ... WHERE colonne = valeur`
- Index fonctionnel : `CREATE INDEX ... ON (fonction(colonne))`
- Extensions offrant des fonctionnalités supplémentaires : pg_trgm, bloom

Malgré toutes ces possibilités, une question revient souvent lorsqu'un index vient d'être ajouté : pourquoi cet index n'est-il pas utilisé ?

L'optimiseur de PostgreSQL est très avancé et il y a peu de cas où il est mis en défaut. Malgré cela, certains index ne sont pas utilisés comme on le souhaiterait. Il peut y avoir plusieurs raisons à cela.

L'INDEX N'EST PAS PERTINENT

Un cas qui n'est pas à exclure est que l'index soit effectivement moins efficace qu'un autre type de parcours.

Les situations suivantes peuvent toutes expliquer qu'un index soit délaissé par l'optimiseur :

- la table considérée est petite, passer par un index ne réduit pas de façon conséquente le nombre de blocs accédés ;

- le prédicat n'est pas suffisamment sélectif, le nombre de lignes retournées représente une portion importante de la table ;
- l'index est très volumineux, ne tient pas en mémoire et provoque de nombreux accès disque de type `random` (accès aléatoires, ou directs) ;
- une lecture complète de la table en utilisant le parallélisme de requête est plus efficace.

Il est également possible que l'index soit pertinent, mais que l'optimiseur estime à tort qu'il ne l'est pas.

Cela peut être dû au paramétrage de PostgreSQL, à la fraicheur des statistiques, ou à des limitations de l'optimiseur.

Voir la section sur les statistiques à ce sujet.

PRÉDICAT INCLUANT UNE TRANSFORMATION

Si une colonne est indexée, mais que le prédicat filtre non pas sur la colonne elle-même, mais sur le résultat d'une transformation de son contenu (application d'une fonction ou d'un opérateur sur la donnée avant comparaison), alors l'index ne peut pas être utilisé.

Un exemple trivial de prédicat de ce type, si `col1` est indexée :

```
WHERE col1 + 2 > 5
```

Dans ce cas, comparer `col1 > 3` permet d'utiliser l'index sans modifier le modèle de données.

Autre exemple, avec `col2` qui contient des données de type `text` :

```
WHERE upper(col2) = 'IN PROGRESS'
```

Dans ce cas précis, si la requête ne peut être modifiée, il faut créer un index fonctionnel pour qu'il puisse répondre à la requête :

```
CREATE INDEX ON t1 (upper(col2)) ;
```

Évidemment, dans ce cas le nouvel index ne peut pas répondre à des requêtes qui n'utilisent pas la fonction `upper()`.

Encore un exemple, si `col1` est toujours de type `int` :

```
WHERE col1 < 3.5
```

Ce dernier cas est plus difficile à détecter, il s'agit d'un cas de conversion implicite. On peut comprendre ce qui se passe en examinant le plan d'exécution :

```
EXPLAIN
SELECT *
```

```
FROM t1
WHERE col1 < 3.5;
                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on t1  (cost=0.00..19425.00 rows=333333 width=4)
   Filter : ((col1) ::numeric < 3.5)
(2 lignes)
```

La colonne `col1` se voit appliquer un `CAST` vers le type `numeric` qui empêche l'utilisation de l'index.

Dans ces différents cas, plusieurs solutions sont envisageables :

- réécrire la requête pour éviter la transformation ;
- adapter le modèle de données pour stocker directement la donnée transformée et l'indexer (ce choix peut violer la normalisation du modèle de données) ;
- créer un index fonctionnel.

OPÉRATEURS NON-SUPPORTÉS

Les index B-tree supportent la plupart des opérateurs généraux sur les variables scalaires (entiers, chaînes, dates, mais pas types composés comme géométries, hstore…), mais pas la différence (`<>` ou `!=`). Par nature, il n'est pas possible d'utiliser un index pour déterminer toutes les valeurs sauf une. Mais ce type de construction est parfois utilisé pour exclure les valeurs les plus fréquentes d'une colonne. Dans ce cas, il est possible d'utiliser un index partiel qui, en plus, sera très petit car il n'indexera qu'une faible quantité de données par rapport à la totalité de la table :

```
EXPLAIN SELECT * FROM employes_big WHERE num_service`<>`4;
                    QUERY PLAN
-------------------------------------------------
 Gather  (cost=1000.00..8264.74 rows=17 width=41)
   Workers Planned : 2
   ->  Parallel Seq Scan on employes_big
            (cost=0.00..7263.04 rows=7 width=41)
         Filter : (num_service `<>` 4)
(4 rows)
```

La création d'un index partiel permet d'en tirer parti :

```
CREATE INDEX ON employes_big(num_service) WHERE num_service`<>`4;
EXPLAIN SELECT * FROM employes_big WHERE num_service`<>`4;
                           QUERY PLAN
-----------------------------------------------------------------
 Index Scan using employes_big_num_service_idx1 on employes_big
```

```
   (cost=0.14..12.35 rows=17 width=40)
(1 row)
```

Il est intéressant de noter que dans le cas d'un index partiel, les colonnes utilisées en prédicat n'ont pas forcément besoin d'être parmi les colonnes à indexer. Ainsi, cet index est parfaitement valide :

```
CREATE INDEX ON employes_big(last_name, first_name) WHERE num_service`<>`4;
```

Et pourrait être utilisé de façon efficace pour répondre à une requête de ce type :

```
SELECT *
FROM employes_big
WHERE num_service `<>` 4
ORDER BY last_name, first_name ;
```

PROBLÈME AVEC LIKE

Dans le cas d'une recherche avec préfixe, PostgreSQL peut utiliser un index B-tree qui se trouverait sur la colonne. Il existe cependant une spécificité à PostgreSQL. Si le jeu de caractères est autre chose que `C` (par exemple, `UTF-8`), il faut utiliser une classe d'opérateur lors de la création de l'index[19] .

Voici une requête de ce type :

```
SELECT *
FROM t1
WHERE c2 LIKE 'x%';
```

En considérant que la colonne `c2` est de type `varchar`, l'index suivant sera approprié pour répondre à la requête :

```
CREATE INDEX i1 ON t1 (c2 varchar_pattern_ops) ;
```

À noter que cette méthode d'indexation avec des index B-tree ne permet des recherches efficaces que dans un cas particulier : si le seul joker de la chaîne est à la fin de celle-ci (`LIKE 'hello%'` par exemple).

De plus, à partir de la version 9.1, il est important de faire attention au collationnement. Si le collationnement de la requête diffère du collationnement de la colonne de l'index, 'index ne pourra pas être utilisé[20] .

Le module pg_trgm[21] , qui permet de décomposer en trigramme les chaînes qui lui sont proposées, permet d'aller encore plus loin.

19. https://www.postgresql.org/docs/current/static/indexes-opclass.html
20. https://www.postgresql.org/docs/current/static/collation.html
21. https://www.postgresql.org/docs/current/static/pgtrgm.html

Plutôt que d'indexer les données, il s'agit d'indexer les trigrammes qui la composent, avec un index GIN ou GiST. Une fois les trigrammes indexés, on peut réaliser de la recherche floue, ou utiliser des clauses `LIKE` malgré la présence de jokers (`%`) n'importe où dans la chaîne.

Une troisième méthode pour améliorer ce type de requêtes et favoriser l'utilisation d'index est de passer par les fonctionnalités Full Text Search[22] (FTS) de PostgreSQL.

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

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

PostgreSQL doit donc permettre de rechercher de façon efficace dans un champ texte. L'avantage de cette solution est d'être intégrée au SGBD. Le moteur de recherche est donc toujours parfaitement à jour avec le contenu de la base, puisqu'il est intégré avec le reste des transactions. Contrairement aux méthodes présentées précédemment, la méthode FTS nécessite une réécriture des requêtes pour utiliser les fonctions et opérateurs spécifiques.

MÉTHODES D'INDEXATION AVANCÉES

PostgreSQL fournit de nombreux types d'index, afin de répondre à des problématiques de recherches complexes.

L'index B-tree est l'index le plus fréquemment utilisé, car il a de nombreux avantages :

- performances se dégradant peu avec la taille de l'arbre : les temps de recherche sont en O(log(n)), c'est-à-dire qu'ils varient en fonction du logarithme du nombre d'enregistrements contenus dans l'index. Plus le nombre d'enregistrements est élevé, plus la variation est faible ;
- excellente concurrence d'accès : on peut facilement avoir plusieurs processus en train d'insérer simultanément dans un index B-tree, avec très peu de contention entre ces processus.

Toutefois ils ne permettent de répondre qu'à des questions très simples : il faut qu'elles ne portent que sur la colonne indexée, et uniquement sur des opérateurs courants (égalité, comparaison). Cela couvre le gros des cas, mais connaître les autres possibilités du moteur vous permettra d'accélérer des requêtes plus complexes, ou d'indexer des types de données inhabituels.

22. https://www.postgresql.org/docs/current/static/textsearch.html

Les méthodes d'indexation avancées suivantes sont disponibles nativement :

- Index multi-colonnes
- Index fonctionnels
- Index partiels
- Index couvrants
- Classes d'opérateurs
- GiN
- GiST
- BRIN
- Hash

Plus de détails sur : https://cloud.dalibo.com/p/exports/formation/manuels/modules/j4/j4.handout.html

COÛT D'ACCÈS ET UTILISATION DES INDEX

Plusieurs paramètres de PostgreSQL influencent l'optimiseur sur l'utilisation ou non d'un index, parmi lesquels :

- `random_page_cost` : indique à PostgreSQL la vitesse d'un accès aléatoire par rapport à un accès séquentiel (`seq_page_cost`) ;
- `effective_cache_size` : indique à PostgreSQL une estimation de la taille du cache disque du système.

Le paramètre `random_page_cost` a une grande influence sur l'utilisation des index en général. Il indique à PostgreSQL le coût d'un accès disque aléatoire. Il est à comparer au paramètre `seq_page_cost` qui indique à PostgreSQL le coût d'un accès disque séquentiel. Ces coûts d'accès sont purement arbitraires et n'ont aucune réalité physique, mais devraient être configurés en prenant en considération les capacités réelles du stockage utilisé.

Dans sa configuration par défaut, PostgreSQL estime qu'un accès aléatoire est 4 fois plus coûteux qu'un accès séquentiel. Les accès aux index étant par nature aléatoires alors que les parcours de table sont par nature séquentiels, modifier ce paramètre revient à favoriser l'un par rapport à l'autre. Cette valeur est bonne dans la plupart des cas. Mais si le serveur de bases de données dispose d'un système disque rapide, c'est-à-dire une bonne carte RAID et plusieurs disques SAS rapides en RAID10, ou du SSD, il est possible de baisser la valeur de ce paramètre.

Enfin, le paramètre `effective_cache_size` indique à PostgreSQL une estimation de la taille du cache disque du système (total du shared buffers et du cache du système). Une bonne pratique est de positionner ce paramètre à 2/3 de la quantité totale de RAM du

serveur. Sur un système Linux, il est possible de donner une estimation plus précise en s'appuyant sur la valeur de colonne `cached` de la commande `free`. Mais si le cache n'est que peu utilisé (par exemple après un redémarrage du serveur), la valeur trouvée peut être trop basse pour pleinement favoriser l'utilisation des index.

Plus de détails sur : https://www.postgresql.org/docs/current/static/runtime-config-query.html

OPTIMISER LA CONSOMMATION DES RESSOURCES

GESTION DES CONNEXIONS

Ouvrir une connexion coûte cher :

- Vérification authentification, permissions
- Création de processus, de contexte d'exécution
- Éventuellement négociation SSL
- Acquisition de verrous

Il est donc recommandé d'utiliser un mécanisme de maintien des connexions côté applicatif ou d'utiliser un pooler comme pgBouncer[23] . Les serveurs d'applications J2EE fournissent en général un pool de connexions JDBC particulièrement efficace.

Pour montrer l'impact du maintien des connexions, PostgreSQL fournit un outil de benchmark synthétique[24] .

Voici les résultats :

- Option -C : se connecter à chaque requête :

```
$ pgbench pgbench -T 20 -c 10 -j5 -S -C
starting vacuum...end.
transaction type : `<builtin : select only>`
scaling factor : 2
query mode : simple
number of clients : 10
number of threads : 5
duration : 20 s
number of transactions actually processed : 16972
latency average = 11.787 ms
tps = 848.383850 (including connections establishing)
tps = 1531.057609 (excluding connections establishing)
```

- Sans se reconnecter à chaque requête :

23. https://pgbouncer.github.io/
24. https://www.postgresql.org/docs/current/static/pgbench.html

```
$ pgbench pgbench -T 20 -c 10 -j5 -S
starting vacuum...end.
transaction type : `<builtin : select only>`
scaling factor : 2
query mode : simple
number of clients : 10
number of threads : 5
duration : 20 s
number of transactions actually processed : 773963
latency average = 0.258 ms
tps = 38687.524110 (including connections establishing)
tps = 38703.239556 (excluding connections establishing)
```

On passe de 900 à 40 000 transactions par seconde.

Par ailleurs, chaque connexion va consommer des ressources, notamment en terme de mémoire privée et partagée, qu'il est important de libérer lorsqu'elles ne sont plus nécessaires.

Même dans le cas d'utilisation d'un pooler de connexions, il est impératif que l'application ferme correctement et explicitement les connexions une fois les traitements terminés. Dans le cas contraire, le pooler ne sera par défaut pas capable de réutiliser les sessions inactives. Certains poolers comme pgBouncer permettent de fonctionner en mode *transaction* plutôt que *session* pour contourner ce problème, mais cela limite beaucoup les fonctionnalités de PostgreSQL utilisables (requêtes préparées, variables de sessions, tables temporaires, etc.).

Plus de détails sur : https://wiki.postgresql.org/wiki/PgBouncer#Feature_matrix_for_pooling_modes

PARTITIONNEMENT

Le partitionnement consiste à fragmenter une table de forte volumétrie en plusieurs tables, ou partitions, plus petites. Cela permet de réduire la volumétrie de chaque partition et de limiter le volume de données accédé, à condition que la requête spécifie bien la ou les partitions concernées par la lecture.

Avant la version 10 de PostgreSQL, le partitionnement[25] n'existait pas nativement. On détournait des fonctionnalités de PostgreSQL pour réaliser du partitionnement en utilisant des notions d'héritage de table. Depuis PostgreSQL 10, le partitionnement est natif mais il est encore jeune et s'appuie sur des mécanismes qui ne sont pas toujours performants et imposent plusieurs limitations.

25. https://www.postgresql.org/docs/current/static/ddl-partitioning.html

Parmi ces limitations :

- pas de support d'index global, donc pas de possibilité de contrainte unique sur la table partitionnée, ni de gestion de clé étrangère ;
- pas de possibilité native de déplacer automatiquement les lignes modifiées d'une partition à une autre ;
- problèmes de performances si le nombre de partitions est très élevé ;
- pas de **dynamic pruning** (exclusion des partitions à la phase d'exécution plutôt qu'à la phase de planification) : attention à ne pas utiliser dans les requêtes de fonctions *stable* comme `to_date()`, utiliser des fonctions *immutable* (l'affectation d'une constante par exemple, comme `DATE '2018-04-19'`) pour que l'exclusion de partition ait lieu. (voir https://www.postgresql.org/docs/current/static/xfunc-volatility.html)

De nouvelles améliorations sont prévues dans les versions à venir de PostgreSQL.

De plus amples informations sur le partitionnement dans PostgreSQL sont disponibles dans le Workshop PostgreSQL 10 de Dalibo[26] .

CONSOMMATION MÉMOIRE

Chaque processus, en plus de la mémoire partagée à laquelle il accède en permanence, peut allouer de la mémoire pour ses besoins propres. L'allocation de cette mémoire est temporaire (elle est libérée dès qu'elle n'est plus utile). Cette mémoire n'est utilisable que par le processus l'ayant allouée.

Plus de détails sur : https://www.postgresql.org/docs/current/static/runtime-config-resource.html#RUNTIME-CONFIG-RESOURCE-MEMORY

Cette mémoire est utilisée dans plusieurs contextes :

- pour des tris et hachages lors de l'exécution de requêtes : un `ORDER BY`, par exemple, peut nécessiter un tri. Ce tri sera effectué à hauteur de `work_mem` en mémoire, puis sera poursuivi sur le disque au besoin ;
- pour des opérations de maintenance : un `CREATE INDEX` ou un `VACUUM` par exemple. Ces besoins étant plus rares, mais plus gourmands en mémoire, on dispose d'un second paramètre `maintenance_work_mem`, habituellement plus grand que `work_mem` ;
- pour la gestion de l'accès à des tables temporaires : afin de minimiser les appels systèmes dans le cas d'accès à des tables temporaires (locales à chaque session), chaque session peut allouer `temp_buffers` de cache dédié à ces tables.

26. https://dali.bo/wk10_html#partitionnement

Comme elle n'est pas partagée, cette mémoire est totalement dynamique.

Il n'y a pas de limite globale de la mémoire pouvant être utilisée par ces paramètres. Par exemple, il est possible, potentiellement, que tous les processus allouent simultanément plusieurs fois `work_mem` (si la requête en cours d'exécution nécessite plusieurs tris par exemple ou si le processus qui a reçu la requête a demandé l'aide à d'autres processus). Il faut donc rester prudent sur les valeurs de ces paramètres, work_mem tout particulièrement, et superviser les conséquences d'une modification de celui-ci.

Le paramètre `work_mem` est un paramètre de session, il est donc possible de modifier sa valeur juste avant d'exécuter une requête nécessitant beaucoup de mémoire, puis de réinitialiser sa valeur.

Attention, si une requête ne dispose pas de suffisamment de mémoire pour travailler à cause de la configuration du paramètre `work_mem`, elle va basculer vers un travail utilisant des fichiers temporaires sur disque, ce qui risque de dégrader beaucoup les performances de la requête.

Il est donc important de trouver le bon équilibre au niveau de l'affectation de mémoire pour conserver des performances acceptables, tout en évitant de saturer la mémoire du serveur de bases de données.

PARALLÉLISME DE REQUÊTES

PostgreSQL est un système multi-processus. Chaque connexion d'un client est gérée par un processus, responsable de l'exécution des requêtes et du renvoi des données au client. Ce processus n'est pas multi-threadé. Par conséquent, chaque requête exécutée est traitée par un cœur de processeur. Plus vous voulez pouvoir exécuter de requêtes en parallèle, plus vous devez avoir de processeurs (ou plus exactement de cœurs).

À partir de la version 9.6, un processus exécutant une requête peut demander l'aide d'autre processus (appelés *workers*) pour l'aider à traiter cette requête. Les différents processus utiliseront des CPU différents, permettant ainsi une exécution parallélisée d'une requête. Ceci n'est possible qu'à partir de la version 9.6 et uniquement pour des requêtes en lecture seule. De plus, seules certaines actions sont parallélisables : parcours séquentiel, jointure, calcul d'agrégats … la liste s'enrichit dans les versions suivantes de PostgreSQL.

Pour plus de détail voir : https://www.postgresql.org/docs/current/static/parallel-query.html

Cette fonctionnalité a un impact important pour les requêtes consommatrices en temps CPU. C'est donc un facteur important à considérer pour optimiser au mieux l'utilisation

des ressources CPU du serveur de bases de données, particulièrement pour une application utilisant des requêtes complexes et gourmandes en ressources (entrepôt de données, calculs complexes, etc.).

BIBLIOGRAPHIE

Documentation officielle de PostgreSQL :
https://www.postgresql.org/docs/
Version française
https://docs.postgresql.fr/
Manuels de formation de Dalibo
https://www.dalibo.com/formations
***The World and the Machine*, Michael Jackson**
version en ligne http://users.mct.open.ac.uk/mj665/icse17kn.pdf
***The Art of SQL*, Stéphane Faroult,**
ISBN-13 : 978-0596008949
***Refactoring SQL Applications*, Stéphane Faroult,**
ISBN-13 : 978-0596514976
***SQL Performance Explained*, Markus Winand**
https://use-the-index-luke.com/fr
FR : ISBN-13 : 978-3950307832
EN : ISBN-13 : 978-3950307825
***Introduction aux bases de données*, Chris Date**
FR : ISBN-13 : 978-2711748389 (8e édition)
EN : ISBN-13 : 978-0321197849
Vidéos de Stéphane Faroult,
Chaine https://www.youtube.com/channel/UCW6zsYGFckfczPKUUVdvYjg
Vidéo 1 https://www.youtube.com/watch?v=40Lnoyv-sXg&list=PL767434BC92D459A7
Vidéo 2 https://www.youtube.com/watch?v=GbZgnAINjUw&list=PL767434BC92D459A7
Vidéo 3 https://www.youtube.com/watch?v=y70FmugnhPU&list=PL767434BC92D459A7
Conférence sur les contraintes au PGDay Paris 2018, *Constraints : a Developer's Secret Weapon*
https://www.postgresql.eu/events/pgdayparis2018/sessions/session/1835/slides/70/2018-03-15%20constraints%20a%20developers%20secret%20weapon%20pgday%20paris.pdf
Article de blog associé, *Database constraints in Postgres : The last line of defense* :
https://www.citusdata.com/blog/2018/03/19/postgres-database-constraints/
***Mastering PostgreSQL in Application Development*, Dimitri Fontaine**
https://masteringpostgresql.com/
***SQL avancé*, Joe Celko**
ISBN-13 : 978-2711786503

NOTES

NOTES

NOS AUTRES PUBLICATIONS

FORMATIONS

- **DBA1 - PostgreSQL Administration** Révision 18.09 / ISBN : 979-10-97371-30-2
- **DBA2 - PostgreSQL Avancé** Révision 18.09 / ISBN : 979-10-97371-31-9
- **DBA3 - PostgreSQL Réplication** Révision 18.09 / ISBN : 979-10-97371-32-6
- **DBA4 - PostgreSQL Performances** Révision 18.09 / ISBN : 979-10-97371-33-3
- **DBA5 - PostgreSQL Sauvegardes avancées** Révision 18.09 / ISBN : 979-10-97371-34-0
- **SQL1 - SQL Conception et Mise en Œuvre** Révision 18.09 / ISBN : 979-10-97371-35-7
- **SQL2 - SQL Avancé** Révision 18.09 / ISBN : 979-10-97371-36-4
- **SQL3 - Développer en PL/pgSQL** Révision 18.09 / ISBN : 979-10-97371-37-1
- **SQL4 - PostgreSQL : L'état de l'art** Révision 18.09 / ISBN : 979-10-97371-38-8
- **SQL5 - Migrer d'Oracle vers PostgreSQL** Révision 18.09 / ISBN : 979-10-97371-39-5

LIVRES BLANCS

- **Migrer d'Oracle à PostgreSQL**
- **Industrialiser 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 activement aux développements de la communauté PostgreSQL avec notamment une implication forte sur les projets gAdmin, phpPgAdmin, pgpool et le code de PostgreSQL lui-même. Le société est également à l'origine de nombreux outils open-source de monitoring, 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 perennité. Dalibo a intégré ces principes dans son ADN en optant pour le statut de SCOP : la société est controlée à 100% par ses salariés, les décisions sont prises collectivement et les bénéfices sont partagés à parts égales.