Module J3

Optimisation SQL

22.09

Dalibo SCOP

https://dalibo.com/formations

Optimisation SQL

Module J3

TITRE : Optimisation SQL
SOUS-TITRE : Module J3

REVISION: 22.09
DATE: 02 septembre 2022

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

LICENCE CREATIVE COMMONS BY-NC-SA 2.0 FR

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

Vous êtes autorisé à :

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

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

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

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

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

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

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

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

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

[1] mailto:contact@dalibo.com

Chers lectrices & lecteurs,

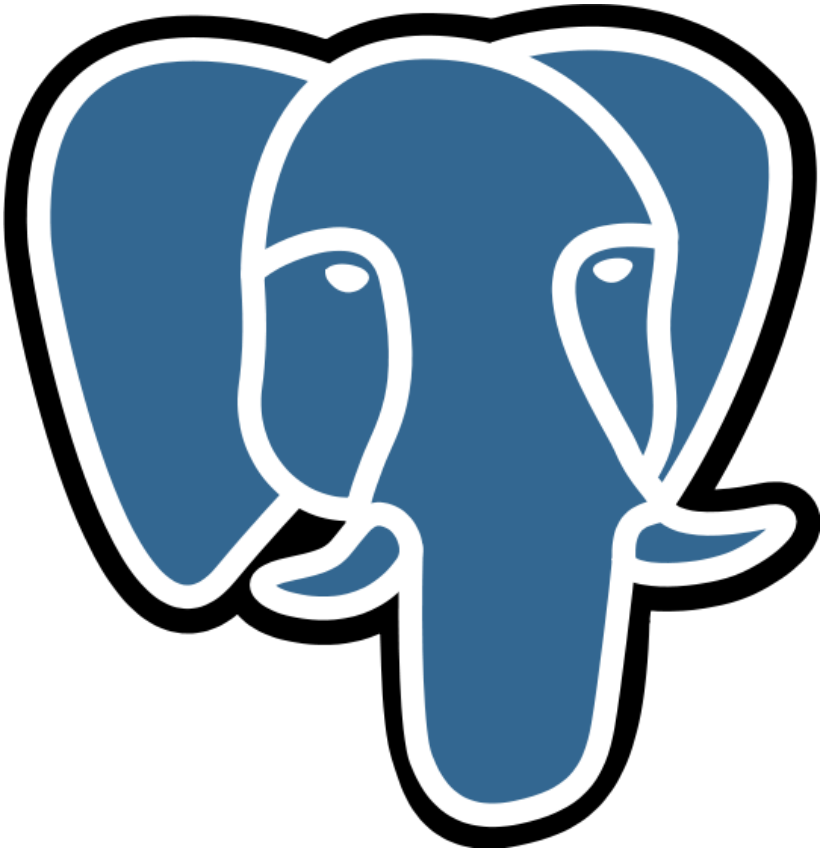
Nos formations PostgreSQL sont issues de nombreuses années d'études, d'expérience de terrain et de passion pour les logiciels libres. Pour Dalibo, l'utilisation de PostgreSQL n'est pas une marque d'opportunisme commercial, mais l'expression d'un engagement de longue date. Le choix de l'Open Source est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

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

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

Table des Matières

1 POSTGRESQL : OPTIMISATIONS SQL



1.0.1 INTRODUCTION

- L'optimisation doit porter sur :
  - le serveur qui héberge le SGBDR : matériel, distribution, kernel.
  - le moteur de la base : `postgresql.conf`
  - la base de données : l'organisation des fichiers de PostgreSQL
  - l'application : schéma, requêtes, vues...
- Ce module se focalise sur ce dernier point

Les bases de données sont des systèmes très complexes. Afin d'en tirer toutes les performances possibles, l'optimisation doit porter sur un très grand nombre de composants différents : le serveur qui héberge la base de données, les processus faisant fonctionner la base de données, les fichiers et disques servant à son stockage, mais aussi, et surtout, l'application elle-même. C'est sur ce dernier point que les gains sont habituellement les plus importants.

Ce module se focalise sur ce dernier point. Il n'aborde pas les plans d'exécution à proprement parler, ceux-ci étant traités ailleurs.

1.1 AXES D'OPTIMISATION

- Il est illusoire d'essayer d'optimiser une application sans déterminer au préalable les sources de ralentissement
- Principe de Pareto (dérivé) : « 80% des effets sont produits par 20% des causes. »
- L'optimisation doit donc être ciblée :
  - il s'agit de trouver ces « 20% » de causes.
  - ne pas micro-optimiser ce qui n'a pas d'influence

Le principe de Pareto et la loi de Pareto sont des constats empiriques. On peut définir mathématiquement une distribution vérifiant la loi de Pareto[2] .

> Le pourcentage de la population dont la richesse est supérieure à une valeur x est proportionnel à A/x^α »
> Vilfredo Pareto, économiste du XIXe siècle

De nombreux phénomènes suivent cette distribution, comme par exemple les files d'attente.

Dans le cadre de PostgreSQL, cela se vérifie souvent. Une poignée de requêtes peut être responsable de l'essentiel du ralentissement. Une table mal conçue peut être responsable de la majorité de vos problèmes. Il ne sert à rien de passer beaucoup de temps à optimiser chaque paramètre sans avoir identifié les principaux consommateurs de ressource.

---

[2] https://fr.wikipedia.org/wiki/Principe_de_Pareto

1.1.1 QUELLES REQUÊTES OPTIMISER ?

- Seules un certain nombre de requêtes sont critiques
  - utilisation d'outil de profiling pour les identifier
  - le travail d'optimisation se porte sur celles-ci uniquement
- Détermination des requêtes critiques :
  - longues en temps cumulé, coûteuses en ressources serveur
  - longues et interactives, mauvais ressenti des utilisateurs

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

Il y a deux façon 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, `pg_stat_statements`). 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). Les requêtes les plus fréquemment exécutées méritent également qu'on leur porte attention, leur optimisation peut permettre d'économiser quelques ressources du serveur.

En utilisant l'extension `pg_stat_statements`, la requête suivante permet de déterminer les requêtes dont les temps d'exécution cumulés sont les plus importants :

```
SELECT r.rolname, d.datname, s.calls, s.total_time,
       s.calls / s.total_time AS avg_time, s.query
  FROM pg_stat_statements s
  JOIN pg_roles r
    ON (s.userid=r.oid)
  JOIN pg_database d
    ON (s.dbid = d.oid)
 ORDER BY s.total_time DESC
 LIMIT 10;
```

Toujours avec `pg_stat_statements`, la requête suivante permet de déterminer les requêtes les plus fréquemment appelées :

```
SELECT r.rolname, d.datname, s.calls, s.total_time,
       s.calls / s.total_time AS avg_time, s.query
  FROM pg_stat_statements s
```

```
  JOIN pg_roles r
    ON (s.userid=r.oid)
  JOIN pg_database d
    ON (s.dbid = d.oid)
ORDER BY s.calls DESC
LIMIT 10;
```

1.1.2 RECHERCHE DES AXES D'OPTIMISATION

- Utilisation de profiler
  - PostgreSQL : pgBadger, PoWA, pg_stat_statements
  - Oracle : tkprof, statspack, AWR…
  - SQL Server : SQL Server Profiler

Ces outils permettent rapidement d'identifier les requêtes les plus consommatrices sur un serveur. Les outils pour PostgreSQL ont le fonctionnement suivant :

- `pgBadger` est un analyseur de log. On trace donc dans les journaux applicatifs de PostgreSQL toutes les requêtes et leur durée. L'outil les analyse et retourne les requêtes les plus fréquemment exécutées, les plus gourmandes unitairement, les plus gourmandes en temps cumulé (somme des temps unitaires) ;
- `pg_stat_statements` est une vue de PostgreSQL (installée grâce une extension standard) qui trace pour chaque ordre exécuté sur l'instance son nombre d'exécution, sa durée cumulée, et un certain nombre d'autres statistiques très utiles ;
- `PoWA` est similaire à `Oracle AWR`. Cette extension s'appuie sur `pg_stat_statements` pour permettre d'historiser l'activité du serveur. Une interface web permet ensuite de visualiser l'activité ainsi historisée et repérer les requêtes problématiques avec les fonctionnalités de *drill-down* de l'interface.

1.2 SQL - REQUÊTES

- Le `SQL` est un langage déclaratif :
  - on décrit le résultat et pas la façon de l'obtenir
  - comme `Prolog`
  - c'est le travail de la base de déterminer le traitement à effectuer
- Traitement ensembliste :
  - par opposition au traitement procédural
  - « *on effectue des opérations sur des relations pour obtenir des relations* »

Le langage SQL a été normalisé par l'ANSI en 1986 et est devenu une norme ISO internationale en 1987. Il s'agit de la norme `ISO 9075`. Elle a subi plusieurs évolutions dans le but d'ajouter des fonctionnalités correspondant aux attentes de l'industrie logicielle. Parmi ces améliorations, notons l'intégration de quelques fonctionnalités objet pour le modèle relationnel-objet. La dernière version stable de la norme est `SQL:2016`.

1.2.1 OPÉRATEURS RELATIONNELS

Les opérateurs purement relationnels sont les suivants :

- Projection
  - clause `SELECT` (choix des colonnes)
- Sélection
  - clause `WHERE` (choix des enregistrements)
- Jointure
  - clause `FROM/JOIN` (choix des tables)
- Bref, tout ce qui détermine 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.

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.

1.2.2 OPÉRATEURS NON-RELATIONNELS

- 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 :

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

On doit exécuter la sous-requête avant la requête.

Les jointures externes sont relationnelles, mais posent tout de même des problèmes et doivent être traitées prudemment.

```
SELECT * FROM t1
LEFT JOIN t2 on (t1.t2id=t2.id)
JOIN t3 on (t1.t3id=t3.id) ;
```

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.

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).

1.2.3 DONNÉES UTILES

Le volume de données récupéré a un impact sur les performances.

- N'accéder qu'aux tables nécessaires
- N'accéder qu'aux colonnes nécessaires
  - Index Only Scan
  - stockage TOAST
- Plus le volume de données à traiter est élevé, plus les opérations seront lentes :
  - tris et Jointures
  - éventuellement stockage temporaire sur disque pour certains algorithmes

Éviter donc les `SELECT *` : c'est une bonne pratique de toute façon, car la requête peut changer de résultat suite à un changement de schéma, ce qui risque d'entraîner des conséquences sur le reste du code.

Ne récupérer que les colonnes utilisées. Certains moteurs suppriment d'eux-même les colonnes qui ne sont pas retournées à l'appelant, comme par exemple dans le cas de :

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

PostgreSQL ne le fait pas pour le moment.

En précisant uniquement les colonnes nécessaires, le moteur peut utiliser un parcours appelé Index Only Scan qui lui permet d'éviter d'avoir aussi à lire des morceaux de la table. Il peut aussi éviter de lire les colonnes à gros contenu qui sont généralement déportés dans le fichier TOAST d'une table.

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'un `LEFT JOIN` sur une table inutilisée dans le `SELECT`, au travers d'une clé étrangère, car on peut garantir que cette table est effectivement inutile.

1.2.4 LIMITER LE NOMBRE DE REQUÊTES

SQL : langage ensembliste et déclaratif

- Ne pas faire de traitement unitaire par enregistrement
- Utiliser les jointures, ne pas accéder à chaque table une par une
- Une seule requête, parcours de curseur
- Fréquent avec les ORM

Les bases de données relationnelles sont conçues pour manipuler des relations, pas des enregistrements unitaires.

Le langage SQL (et même les autres langages relationnels qui ont existé comme QUEL, SEQUEL) 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.

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.

Préparons un jeu de test :

```
CREATE TABLE test (id int, valeur varchar);
INSERT INTO test SELECT i,chr(i%94+32) FROM generate_series (1,1000000) g(i);
ALTER TABLE test ADD PRIMARY KEY (id);
```

Le script perl suivant récupère 1000 enregistrements un par un, 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 id=\$1;\n";
```

```
for (my $i=0;$i<=1000;$i++)
{
        print "EXECUTE ps($i);\n";
}
```

Exécutons ce code :

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

real    0m44,476s
user    0m0,137s
sys     0m0,040s
```

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

```
time psql -c "select * from test where id >=0 and id <= 1000" > /dev/null

real    0m0,129s
user    0m0,063s
sys     0m0,018s
```

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. 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 :

```
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 :

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

1.2.5 ÉVITER LES VUES NON-RELATIONNELLES

Une vue est simplement une requête pré-déclarée en base.

- C'est l'équivalent relationnel d'une fonction
- Quand elle est utilisée dans une autre requête, elle est initialement traitée comme une sous-requête
- Attention aux vues avec `DISTINCT`, `GROUP BY` etc.
  - impossible de l'*inliner*
  - barrière d'optimisation
  - donc mauvaises performances
- Les vues sont dangereuses en termes de performance
  - masquent la complexité

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 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. La vue a été remplacée par une sous-requête équivalente :

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

                                QUERY PLAN
--------------------------------------------------------------------------------
 Subquery Scan on tmp  (cost=114082.84..131582.84 rows=5000 width=6)
   Filter: ((tmp.valeur)::text = 'b'::text)
   ->  Unique  (cost=114082.84..119082.84 rows=1000000 width=6)
         ->  Sort  (cost=114082.84..116582.84 rows=1000000 width=6)
               Sort Key: test.id
               ->  Seq Scan on test  (cost=0.00..14425.00 rows=1000000 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.

1.2.6 SOUS-REQUÊTES 1/2

- Si `IN`, limiter le nombre d'enregistrements grâce à `DISTINCT`

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

- Éviter une requête liée :

```
SELECT a,b
  FROM t1
  WHERE val IN (SELECT f(b))
```

1.2.7 SOUS-REQUÊTES 2/2

- Certaines sous-requêtes sont l'expression de `Semi-join` ou `Anti-join`

```
SELECT * FROM t1 WHERE fk
   [NOT] IN (SELECT pk FROM t2 WHERE xxx)
SELECT * FROM t1 WHERE [NOT] EXISTS
   (SELECT 1 FROM t2 WHERE t2.pk=t1.fk AND xxx)
SELECT t1.* FROM t1 LEFT JOIN t2 ON (t1.fk=t2.pk)
   WHERE t2.id IS [NOT] NULL`
```

- Elles sont strictement équivalentes !

- L'optimiseur les exécute à l'identique (sauf `NOT IN`)

Les seules sous-requêtes sans danger sont celles qui retournent un ensemble constant et ne sont exécutés qu'une fois, ou celles qui expriment un `Semi-Join` (test d'existence) ou `Anti-Join` (test de non-existence), qui sont presque des jointures : la seule différence est qu'elles ne récupèrent pas l'enregistrement de la table cible.

Attention toutefois à l'utilisation du prédicat `NOT IN`, ils peuvent générer des plans d'exécution catastrophiques :

```
tpc=# EXPLAIN SELECT *
  FROM commandes
 WHERE numero_commande NOT IN (SELECT numero_commande
                               FROM lignes_commandes);
                                        QUERY PLAN
-------------------------------------------------------------------------------------------
 Gather  (cost=1000.00..1196748507.51 rows=84375 width=77)
   Workers Planned: 1
   ->  Parallel Seq Scan on commandes  (cost=0.00..1196739070.01 rows=49632 width=77)
         Filter: (NOT (SubPlan 1))
         SubPlan 1
           ->  Materialize  (cost=0.00..22423.15 rows=675543 width=8)
                 ->  Seq Scan on lignes_commandes
                       (cost=0.00..16406.43 rows=675543 width=8)
(7 rows)

Time: 0.460 ms
```

La requête suivante est strictement équivalente et produit un plan d'exécution largement plus intéressant :

```
tpc=# EXPLAIN SELECT *
  FROM commandes
 WHERE NOT EXISTS (SELECT 1
                   FROM lignes_commandes l
                   WHERE l.numero_commande = commandes.numero_commande);

                                QUERY PLAN
---------------------------------------------------------------------------
 Gather  (cost=18084.21..28274.34 rows=26537 width=77)
   Workers Planned: 1
   ->  Parallel Hash Anti Join  (cost=17084.21..24620.64 rows=15610 width=77)
         Hash Cond: (commandes.numero_commande = l.numero_commande)
         ->  Parallel Seq Scan on commandes
             (cost=0.00..3403.65 rows=99265 width=77)
         ->  Parallel Hash  (cost=12465.76..12465.76 rows=281476 width=8)
               ->  Parallel Seq Scan on lignes_commandes l
```

```
                    (cost=0.00..12465.76 rows=281476 width=8)
(7 rows)
```

La raison, c'est que si un `NULL` est présent dans la liste du `NOT IN`, `NOT IN` vaut systématiquement `FALSE`. Nous, nous savons qu'il n'y aura pas de `numero_commandes` à `NULL`.

1.3 ACCÈS AUX DONNÉES

L'accès aux données est coûteux.

- Quelle que soit la base
- Dialogue entre client et serveur
  - plusieurs aller/retours potentiellement
- Analyse d'un langage complexe
  - SQL PostgreSQL : `gram.y` de 14000 lignes
- Calcul de plan :
  - langage déclaratif => converti en impératif à chaque exécution

Dans les captures réseau ci-dessous, le serveur est sur le port 5932.

`SELECT * FROM test`, 0 enregistrement :

```
10:57:15.087777 IP 127.0.0.1.39508 > 127.0.0.1.5932:
    Flags [P.], seq 109:134, ack 226, win 350,
    options [nop,nop,TS val 2270307 ecr 2269578], length 25
10:57:15.088130 IP 127.0.0.1.5932 > 127.0.0.1.39508:
    Flags [P.], seq 226:273, ack 134, win 342,
    options [nop,nop,TS val 2270307 ecr 2270307], length 47
10:57:15.088144 IP 127.0.0.1.39508 > 127.0.0.1.5932:
    Flags [.], ack 273, win 350,
    options [nop,nop,TS val 2270307 ecr 2270307], length 0
```

`SELECT * FROM test`, 1000 enregistrements :

```
10:58:08.542660 IP 127.0.0.1.39508 > 127.0.0.1.5932:
    Flags [P.], seq 188:213, ack 298, win 350,
    options [nop,nop,TS val 2286344 ecr 2285513], length 25
10:58:08.543281 IP 127.0.0.1.5932 > 127.0.0.1.39508:
    Flags [P.], seq 298:8490, ack 213, win 342,
    options [nop,nop,TS val 2286344 ecr 2286344], length 8192
10:58:08.543299 IP 127.0.0.1.39508 > 127.0.0.1.5932:
    Flags [.], ack 8490, win 1002,
    options [nop,nop,TS val 2286344 ecr 2286344], length 0
10:58:08.543673 IP 127.0.0.1.5932 > 127.0.0.1.39508:
    Flags [P.], seq 8490:14241, ack 213, win 342,
```

```
    options [nop,nop,TS val 2286344 ecr 2286344], length 5751
10:58:08.543682 IP 127.0.0.1.39508 > 127.0.0.1.5932:
    Flags [.], ack 14241, win 1012,
    options [nop,nop,TS val 2286344 ecr 2286344], length 0
```

Un client JDBC va habituellement utiliser un aller/retour de plus, en raison des requêtes préparées : un dialogue pour envoyer la requête et la préparer, et un autre pour envoyer les paramètres. Le problème est la latence du réseau, habituellement : de 50 à 300 µs. Cela limite à 3 000 à 20 000 le nombre d'opérations maximum par seconde par socket. On peut bien sûr paralléliser sur plusieurs sessions, mais cela complique le traitement.

En ce qui concerne le parser : comme indiqué dans ce message[3] : `gram.o`, le parser fait 1 Mo une fois compilé !

1.3.1 MAINTIEN DES CONNEXIONS

Se connecter coûte cher :

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

=> Maintenir les connexions côté applicatif ou utiliser un pooler.

On peut tester l'impact d'une connexion/déconnexion avec pgbench :

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 :

[3] https://www.postgresql.org/message-id/CA+TgmoaaYvJ7yDKJHrWN1BVk_7fcV16rvc93udSo59gfxG_t7A@mail.gmail.com

```
$ 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 38 000 transactions par seconde.

1.3.2 PENSER RELATIONNEL

- Les spécifications sont souvent procédurales, voire objet !
- Il faut prendre du recul, et réfléchir de façon ensembliste
  - on travaille sur des ensembles de données
  - on peut faire encore mieux avec les CTE (clauses `WITH`)

Si les spécifications disent (version simplifiée bien sûr) :

- Vérifier la présence du client
  - s'il est présent, mettre à jour son adresse
  - sinon, créer le client avec la bonne adresse

Peut être écrit (pseudo-code client) :

```
SELECT count(*) from clients where client_name = 'xxx' INTO compte
IF compte > 0
  UPDATE clients set adresse='yyy' WHERE client_name='xxx'
ELSE
  INSERT client SET client_name='xxx', adresse='yyy'
END IF
```

D'où 3 requêtes, systématiquement 2 appels à la base.

On peut très bien l'écrire comme suit :

```
UPDATE clients set adresse='yyy' WHERE client_name='xxx'
IF NOT FOUND
  INSERT client SET client_name='xxx', adresse='yyy'
END IF
```

Dans ce cas, on n'aura 2 requêtes que dans le plus mauvais cas. Bien sûr, cet exemple est simpliste.

On peut aussi, grâce aux CTE, réaliser tout cela en un seul ordre SQL, qui ressemblerait à :

```
WITH
  enregistrements_a_traiter AS (
    SELECT * FROM (VALUES ('toto' , 'adresse1' ),('tata','adresse2'))
    AS val(nom_client,adresse)
  ),
  mise_a_jour AS (
    UPDATE client SET adresse=enregistrements_a_traiter.adresse
    FROM enregistrements_a_traiter
    WHERE enregistrements_a_traiter.nom_client=client.nom_client
    RETURNING client.nom_client
  )
INSERT INTO client (nom_client,adresse)
  SELECT nom_client,adresse from enregistrements_a_traiter
  WHERE NOT EXISTS (
    SELECT 1 FROM mise_a_jour
    WHERE mise_a_jour.nom_client=enregistrements_a_traiter.nom_client
  );
```

Plus d'information sur la fusion des enregistrements dans PostgreSQL avec des CTE[4] .

À partir de la version 9.5, cette requête devient même beaucoup plus simple grâce à la clause `ON CONFLICT ... DO UPDATE` :

```
INSERT INTO client (nom_client,adresse) VALUES ('toto' , 'adresse1' ), ('tata','adresse2')
    ON CONFLICT (nom_client) DO UPDATE
    SET adresse = EXCLUDED.adresse
    WHERE client.nom_client=EXCLUDED.nom_client;
```

1.3.3 PAS DE DDL APPLICATIF

- Le schéma représente la modélisation des données
  - une application n'a pas à y toucher lors de son fonctionnement normal
  - parfois tables temporaires locales à des sessions
  - toujours voir si une autre solution est possible
- SQL manipule les données en flux continu :
  - chaque étape d'un plan d'exécution n'attend pas la fin de la précédente
  - passer par une table temporaire est probablement une perte de temps

[4] https://vibhorkumar.wordpress.com/2011/10/26/upsertmerge-using-writable-cte-in-postgresql-9-1/

Si on reprend l'exemple précédent, il aurait pu être écrit :

```
=> CREATE TEMP TABLE temp_a_inserer  (nom_client text, adresse text);
=> INSERT INTO temp_a_inserer SELECT * FROM (VALUES ('toto' , 'adresse1' ),
              ('tata','adresse2')) AS tmp;
=> UPDATE client SET adresse=temp_a_inserer.adresse
   FROM temp_a_inserer
   WHERE temp_a_inserer.nom_client=client.nom_client;
=> INSERT INTO client (nom_client,adresse)
      SELECT nom_client,adresse from temp_a_inserer
      WHERE NOT EXISTS (
          SELECT 1 FROM client
          WHERE client.nom_client=temp_a_inserer.nom_client);
=> DROP TABLE temp_a_inserer;
```

1000 exécutions de cette méthode prennent 5 s, alors que la solution précédente ne dure que 500 ms.

1.3.4 OPTIMISER CHAQUE ACCÈS

Un ordre SQL peut effectuer de nombreuses choses :

- Les moteurs SQL sont très efficaces, et évoluent en permanence
- Ils ont de nombreuses méthodes de tri, de jointure, qu'ils choisissent en fonction du contexte
- Si vous utilisez le langage SQL, votre requête profitera des futures évolutions
- Si vous codez tout dans votre programme, vous devrez le maintenir et l'améliorer
- Faites un maximum du côté SQL : agrégats, fonctions analytiques, tris, numérotations, `CASE`, etc.
- Commentez votre code avec `--` et `/* */`

L'avantage du code SQL est, encore une fois, qu'il est déclaratif. Il aura donc de nombreux avantages sur un code procédural.

L'exécution évoluera pour prendre en compte les variations de volumétrie des différentes tables. Les optimiseurs sont la partie la plus importante d'un moteur SQL. Ils progressent en permanence. Chaque nouvelle version va donc potentiellement améliorer vos performances.

Si vous écrivez du procédural avec des appels unitaires à la base dans des boucles, le moteur ne pourra rien optimiser

Si vous faites vos tris ou regroupements côté client, vous êtes limités aux algorithmes fournis par vos langages, voire à ceux que vous aurez écrit manuellement. Une base de

données bascule automatiquement entre une dizaine d'algorithmes différents suivant le volume, le type de données à trier, ce pour quoi le tri est ensuite utilisé, etc., voire évite de trier en utilisant des tables de hachage ou des index disponibles.

1.3.5 NE FAIRE QUE LE NÉCESSAIRE

Encore une fois, prendre de la distance vis-à-vis des spécifications fonctionnelles :

- Si le client existe, le mettre à jour :
  - le mettre à jour, et regarder combien d'enregistrements ont été mis à jour
- Si le client existe :
  - surtout pas de `COUNT(*)`, éventuellement un test de l'existence d'un seul enregistrement
- Gérer les exceptions plutôt que de vérifier préalablement que les conditions sont remplies (si l'exception est rare)

Toujours coder les accès aux données pour que la base fasse le maximum de traitement, mais uniquement les traitements nécessaires : l'accès aux données est coûteux, il faut l'optimiser. Et le gros des pièges peut être évité avec les quelques règles d'« hygiène » simples qui viennent d'être énoncées.

1.4 INDEX

- Objets destinés à l'optimisation des accès
- À poser par les développeurs

```
CREATE INDEX ON ma_table (nom colonne) ;
```

Les index sont des objets uniquement destinés à accélérer les requêtes (filtrage mais aussi jointures et tris), non à modifier le résultat. Il est capital pour un développeur d'en maîtriser les bases car il est celui qui sait quels sont les champs interrogés dans son application.

Les index sont un sujet en soi qui sera traité par ailleurs.

1.5 IMPACT DES TRANSACTIONS

- Prise de verrous : ils ne sont relâchés qu'à la fin
  - `COMMIT`
  - `ROLLBACK`
- Validation des données sur le disque au `COMMIT`
  - écriture synchrone : coûteux
- Faire des transactions qui correspondent au fonctionnel
- Si traitement lourd, préférer des transactions de grande taille

Réaliser des transactions permet de garantir l'atomicité des opérations : toutes les modifications sont validées (`COMMIT`), ou tout est annulé (`ROLLBACK`). Il n'y a pas d'état intermédiaire. Le `COMMIT` garantit aussi la durabilité des opérations : une fois que le `COMMIT` a réussi, la base de données garantit que les opérations ont bien été stockées, et ne seront pas perdues… sauf perte du matériel (disque) sur lequel ont été écrites ces opérations bien sûr.

L'opération `COMMIT` a donc bien sûr un coût : il faut garantir que les données sont bien écrites sur le disque, il faut les écrire sur le disque (évidemment), mais aussi attendre la confirmation du disque. Que les disques soient mécaniques ou SSD ne change pas grand chose, cette opération est coûteuse :

- Un disque dur doit se positionner au bon endroit (journal de transaction), écrire la donnée, et confirmer au système que c'est fait. Il faudra donc compter le temps de déplacement de la tête et de rotation du disque pour se positionner, et le temps d'écriture (celui-ci sera plus court). Compter 1 à 5 millisecondes.
- Un disque SSD doit écrire réellement le bloc demandé. Il faudra donc faire un `ERASE` du bloc puis une nouvelle écriture de celui-ci. C'est l'`ERASE` qui est lent (de l'ordre de 2 millisecondes, donc à peine plus rapide qu'un disque dur mécanique).

On peut limiter l'impact des écritures synchrone en utilisant un cache en écriture comme en proposent les serveurs haut de gamme.

Les transactions devant garantir l'unicité des opérations, il est nécessaire qu'elles prennent des verrous : sur les enregistrements modifiés, sur les tables accédées (pour éviter les changements de structure pendant leur manipulation), sur des prédicats (dans certains cas compliqués comme le niveau d'isolation `serializable`)… tout ceci a un impact :

- Le temps d'acquisition des verrous, bien sûr
- Mais aussi les sources de contention entre chaque session

Il est donc très difficile de déterminer la bonne durée d'une transaction. Trop courte : on génère beaucoup d'opérations synchrones. Trop longue : on risque de bloquer d'autres

sessions. Le mieux est de coller au besoin fonctionnel.

1.5.1 VERROUILLAGE ET CONTENTION

- Chaque transaction prend des verrous :
  - sur les objets (tables, index, etc.) pour empêcher au moins leur suppression ou modification de structure pendant leur travail
  - sur les enregistrements
  - libérés à la fin de la transaction : les transactions très longues peuvent donc être problématiques
- Sous PostgreSQL, on peut quand même lire un enregistrement en cours de modification : on voit l'ancienne version (MVCC)

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 le même enregistrement qu'une autre, tant qu'on ne sait pas si cette dernière a validé ou annulé sa modification. On a donc un verrouillage des enregistrements modifiés.

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.

PostgreSQL, comme Oracle, utilise un modèle dit MVCC (Multi-Version Concurrency Control), qui 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. 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`).

Dans le cas où il est réellement nécessaire de verrouiller un enregistrement sans le mettre à jour immédiatement (pour éviter une mise à jour concurrente), il faut utiliser l'ordre SQL `SELECT FOR UPDATE`.

1.5.2 DEADLOCKS

- Du fait de ces verrous :
  - on peut avoir des `deadlocks` (verrous mortels)
  - en théorie, on peut les éviter (en prenant toujours les verrous dans le même ordre)
  - en pratique, ça n'est pas toujours possible ou commode
  - les SGBD tuent une des transactions responsables du `deadlock`
  - une application générant de nombreux `deadlocks` est ralentie

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.

1.6 BASE DISTRIBUÉE

Pour les performances, on envisage souvent de distribuer la base sur plusieurs nœuds.

- La complexité augmente (code applicatif et/ou exploitation)
  - et le risque d'erreur (programmation, fausse manipulation)
- Le retour à un état stable après un incident est complexe

Il est toujours tentant d'augmenter la quantité de ressources matérielles pour résoudre un problème de performance. Il ne faut surtout pas négliger tous les coûts de cette solution : non seulement l'achat de matériel, mais aussi les coûts humains : procédures d'exploitation, de maintenance, complexité accrue de développement, etc.

Performance et robustesse peuvent être des objectifs contradictoires.

1.7 BIBLIOGRAPHIE

- Quelques références :
  - *The Art of SQL*, **Stéphane Faroult**
  - *Refactoring SQL Applications*, **Stéphane Faroult**
  - *SQL Performance Explained*, **Markus Winand**
  - *Introduction aux bases de données*, **Chris Date**
  - *The Art of PostgreSQL*, **Dimitri Fontaine**
  - Vidéos de **Stéphane Faroult** sous Youtube

Il existe bien des livres sur le développement en SQL. Voici quelques sources intéressantes parmi bien d'autres :

Livres pratiques non propres à PostgreSQL :

- *The Art of SQL*, **Stéphane Faroult**, 2006 (ISBN-13: 978-0596008949)
- *Refactoring SQL Applications*, **Stéphane Faroult**, 2008 (ISBN-13: 978-0596514976)

- *SQL Performance Explained*, **Markus Winand**, 2012 : même si ce livre ne tient pas compte des dernières nouveautés des index de PostgreSQL, il contient l'essentiel de ce qu'un développeur doit savoir sur les index B-tree sur diverses bases de données courantes
  - site internet (fr)[5]
  - ISBN-13 en français : 978-3950307832, en anglais : 978-3950307825

En vidéos :

- **Stéphane Faroult** (roughsealtd sur Youtube[6]) : *SQL Best Practices in less than 20 minutes* partie 1[7] , partie 2[8] , partie 3[9] .

Livres spécifiques à PostgreSQL :

- *The Art of PostgreSQL*[10] de **Dimitri Fontaine** (2020) ; l'ancienne édition de 2017 se nommait *Mastering PostgreSQL in Application Development*.

Sur la théorie des bases de données :

- *An Introduction to Database Systems*, **Chris Date** (8è édition de 2003, ISBN-13 en français : 978-2711748389 ; en anglais : 978-0321197849) ;
- *The World and the Machine*, **Michael Jackson** (version en ligne[11]).

1.8 QUIZ

https://dali.bo/j3_quiz

[5] https://use-the-index-luke.com/fr
[6] https://www.youtube.com/channel/UCW6zsYGFckfczPKUUVdvYjg
[7] https://www.youtube.com/watch?v=40Lnoyv-sXg&list=PL767434BC92D459A7
[8] https://www.youtube.com/watch?v=GbZgnAINjUw&list=PL767434BC92D459A7
[9] https://www.youtube.com/watch?v=y70FmugnhPU&list=PL767434BC92D459A7
[10] https://theartofpostgresql.com/
[11] http://users.mct.open.ac.uk/mj665/icse17kn.pdf

NOTES

NOTES

NOS AUTRES PUBLICATIONS

FORMATIONS

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

LIVRES BLANCS

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

TÉLÉCHARGEMENT GRATUIT

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

DALIBO, L'EXPERTISE POSTGRESQL

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

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

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