TRAVAUX PRATIQUES (SOLUTIONS)

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

```
SET search_path = magasin;
```

Index « simples »

Considérons le cas d'usage d'une recherche de commandes par date. Le besoin fonctionnel est le suivant : renvoyer l'intégralité des commandes passées au mois de janvier 2014.

> Créer la requête affichant l'intégralité des commandes passées au mois de janvier 2014.

Pour renvoyer l'ensemble de ces produits, la requête est très simple :

```
SELECT * FROM commandes date_commande
WHERE date_commande >= '2014-01-01'
AND date_commande < '2014-02-01';
```

> Afficher le plan de la requête , en utilisant `EXPLAIN (ANALYZE, BUFFERS)`. Que constate-t-on ?

Le plan de celle-ci est le suivant :

```
EXPLAIN (ANALYZE, BUFFERS) SELECT * FROM commandes
WHERE date_commande >= '2014-01-01' AND date_commande < '2014-02-01';
```

```
                                QUERY PLAN
-------------------------------------------------------------------------
 Seq Scan on commandes  (cost=0.00..25158.00 rows=19674 width=50)
                        (actual time=2.436..102.300 rows=19204 loops=1)
   Filter: ((date_commande >= '2014-01-01'::date)
            AND (date_commande < '2014-02-01'::date))
   Rows Removed by Filter: 980796
   Buffers: shared hit=10158
 Planning time: 0.057 ms
 Execution time: 102.929 ms
```

> Réécrire la requête par ordre de date croissante. Afficher de nouveau son plan. Que constate-t-on ?

Ajoutons la clause ORDER BY :

```
EXPLAIN (ANALYZE, BUFFERS) SELECT * FROM commandes
WHERE date_commande >= '2014-01-01' AND date_commande < '2014-02-01'
ORDER BY date_commande;
```

```
                              QUERY PLAN
-----------------------------------------------------------------------
 Sort  (cost=26561.15..26610.33 rows=19674 width=50)
       (actual time=103.895..104.726 rows=19204 loops=1)
```

```
   Sort Key: date_commande
   Sort Method: quicksort  Memory: 2961kB
   Buffers: shared hit=10158
   ->  Seq Scan on commandes  (cost=0.00..25158.00 rows=19674 width=50)
                              (actual time=2.801..102.181
                               rows=19204 loops=1)
         Filter: ((date_commande >= '2014-01-01'::date)
                  AND (date_commande < '2014-02-01'::date))
         Rows Removed by Filter: 980796
         Buffers: shared hit=10158
 Planning time: 0.096 ms
 Execution time: 105.410 ms
```

On constate ici que lors du parcours séquentiel, 980 796 lignes ont été lues, puis écartées car ne correspondant pas au prédicat, nous laissant ainsi avec un total de 19 204 lignes. Les valeurs précises peuvent changer, les données étant générées aléatoirement. De plus, le tri a été réalisé en mémoire. On constate de plus que 10 158 blocs ont été parcourus, ici depuis le cache, mais ils auraient pu l'être depuis le disque.

> Créer un index permettant de répondre à ces requêtes.

Création de l'index :

```
CREATE INDEX idx_commandes_date_commande ON commandes(date_commande);
```

> Afficher de nouveau le plan des deux requêtes. Que constate-t-on ?

```
EXPLAIN (ANALYZE, BUFFERS) SELECT * FROM commandes
WHERE date_commande >= '2014-01-01' AND date_commande < '2014-02-01';
```

```
                        QUERY PLAN
-----------------------------------------------------------
 Index Scan using idx_commandes_date_commande on commandes
        (cost=0.42..822.60 rows=19674 width=50)
        (actual time=0.015..3.311 rows=19204
   Index Cond: ((date_commande >= '2014-01-01'::date)
                AND (date_commande < '2014-02-01'::date))
   Buffers: shared hit=254
 Planning time: 0.074 ms
 Execution time: 4.133 ms
```

Le temps d'exécution a été réduit considérablement : la requête est 25 fois plus rapide. On constate notamment que seuls 254 blocs ont été parcourus.

Pour la requête avec la clause ORDER BY, nous obtenons le plan d'exécution suivant :

```
                        QUERY PLAN
-----------------------------------------------------------
 Index Scan using idx_commandes_date_commande on commandes
        (cost=0.42..822.60 rows=19674 width=50)
        (actual time=0.032..3.378 rows=19204
   Index Cond: ((date_commande >= '2014-01-01'::date)
                AND (date_commande < '2014-02-01'::date))
   Buffers: shared hit=254
```

```
 Planning time: 0.516 ms
 Execution time: 4.049 ms
```

Celui-ci est identique ! En effet, l'index permettant un parcours trié, l'opération de tri est ici « gratuite ».

> Écrire la requête affichant `commandes.nummero_commande` et `clients.type_client` pour `client_id = 3`. Afficher son plan. Que constate-t-on ?

```
EXPLAIN (ANALYZE, BUFFERS) SELECT numero_commande, type_client FROM commandes
      INNER JOIN clients ON commandes.client_id = clients.client_id
    WHERE clients.client_id = 3;
```

```
                                  QUERY PLAN
------------------------------------------------------------------------------
 Nested Loop  (cost=0.29..22666.42 rows=11 width=101)
              (actual time=8.799..80.771 rows=14 loops=1)
   Buffers: shared hit=10161
   ->  Index Scan using clients_pkey on clients
            (cost=0.29..8.31 rows=1 width=51)
            (actual time=0.017..0.018 rows=1 loops=1)
         Index Cond: (client_id = 3)
         Buffers: shared hit=3
   ->  Seq Scan on commandes  (cost=0.00..22658.00 rows=11 width=50)
                              (actual time=8.777..80.734 rows=14 loops=1)
         Filter: (client_id = 3)
         Rows Removed by Filter: 999986
         Buffers: shared hit=10158
 Planning time: 0.281 ms
 Execution time: 80.853 ms
```

> Créer un index pour accélérer cette requête.

```
CREATE INDEX ON commandes (client_id) ;
```

> Afficher de nouveau son plan. Que constate-t-on ?

```
EXPLAIN (ANALYZE, BUFFERS) SELECT * FROM commandes
      INNER JOIN clients on commandes.client_id = clients.client_id
      WHERE clients.client_id = 3;
```

```
                                  QUERY PLAN
------------------------------------------------------------------------------
 Nested Loop  (cost=4.80..55.98 rows=11 width=101)
              (actual time=0.064..0.189 rows=14 loops=1)
   Buffers: shared hit=23
   ->  Index Scan using clients_pkey on clients
            (cost=0.29..8.31 rows=1 width=51)
            (actual time=0.032..0.032 rows=1 loops=1)
         Index Cond: (client_id = 3)
         Buffers: shared hit=6
   ->  Bitmap Heap Scan on commandes  (cost=4.51..47.56 rows=11 width=50)
                                      (actual time=0.029..0.147
                                       rows=14 loops=1)
         Recheck Cond: (client_id = 3)
```

```
         Heap Blocks: exact=14
         Buffers: shared hit=17
         ->  Bitmap Index Scan on commandes_client_id_idx
                    (cost=0.00..4.51 rows=11 width=0)
                    (actual time=0.013..0.013 rows=14 loops=1)
               Index Cond: (client_id = 3)
               Buffers: shared hit=3
 Planning time: 0.486 ms
 Execution time: 0.264 ms
```

On constate ici un temps d'exécution divisé par 160 : en effet, on ne lit plus que 17 blocs pour la commande (3 pour l'index, 14 pour les données) au lieu de 10 158.

Sélectivité

> Écrire une requête renvoyant l'intégralité des clients qui sont du type entreprise ('E'), une autre pour l'intégralité des clients qui sont du type particulier ('P').

Les requêtes :

```sql
SELECT * FROM clients WHERE type_client = 'P';
SELECT * FROM clients WHERE type_client = 'E';
```

> Ajouter un index sur la colonne `type_client`, et rejouer les requêtes précédentes.

Pour créer l'index :

```sql
CREATE INDEX ON clients (type_client);
```

> Afficher leurs plans d'exécution. Que se passe-t-il ? Pourquoi ?

Les plans d'éxécution :

```sql
EXPLAIN ANALYZE SELECT * FROM clients WHERE type_client = 'P';
```

```
                             QUERY PLAN
--------------------------------------------------------------------
 Seq Scan on clients  (cost=0.00..2276.00 rows=89803 width=51)
                      (actual time=0.006..12.877 rows=89800 loops=1)
   Filter: (type_client = 'P'::bpchar)
   Rows Removed by Filter: 10200
 Planning time: 0.374 ms
 Execution time: 16.063 ms
```

```sql
EXPLAIN ANALYZE SELECT * FROM clients WHERE type_client = 'E';
```

```
                                QUERY PLAN
---------------------------------------------------------------------------
 Bitmap Heap Scan on clients  (cost=154.50..1280.84 rows=8027 width=51)
                              (actual time=2.094..4.287 rows=8111 loops=1)
   Recheck Cond: (type_client = 'E'::bpchar)
   Heap Blocks: exact=1026
```

```
   ->  Bitmap Index Scan on clients_type_client_idx
         (cost=0.00..152.49 rows=8027 width=0)
         (actual time=1.986..1.986 rows=8111 loops=1)
       Index Cond: (type_client = 'E'::bpchar)
 Planning time: 0.152 ms
 Execution time: 4.654 ms
```

L'optimiseur sait estimer, à partir des statistiques (consultables via la vue `pg_stats`), qu'il y a approximativement 89 000 clients particuliers, contre 8 000 clients entreprise.

Dans le premier cas, la majorité de la table sera parcourue, et renvoyée : il n'y a aucun intérêt à utiliser l'index.

Dans l'autre, le nombre de lignes étant plus faible, l'index est bel et bien utilisé (via un *Bitmap Scan*, ici).

Index partiels

> Sur la base fournie pour les TPs, les lots non livrés sont constamment requêtés. Notamment, un système d'alerte est mis en place afin d'assurer un suivi qualité sur les lots expédié depuis plus de 3 jours (selon la date d'expédition), mais non réceptionné (date de réception à NULL).
>
> Écrire la requête correspondant à ce besoin fonctionnel (il est normal qu'elle ne retourne rien).

La requête est la suivante :

```sql
SELECT * FROM lots
    WHERE date_reception IS NULL
    AND   date_expedition < now() - '3d'::interval;
```

> Afficher le plan d'exécution.

Le plans (ci-dessous avec ANALYZE) opère un *Seq Scan* parallélisé, lit et rejette toutes les lignes, ce qui est évidemment lourd :

```
                          QUERY PLAN
---------------------------------------------------------------
 Gather  (cost=1000.00..17764.65 rows=1 width=43) (actual time=28.522..30.993 rows=0
↪  loops=1)
   Workers Planned: 2
   Workers Launched: 2
   ->  Parallel Seq Scan on lots  (cost=0.00..16764.55 rows=1 width=43) (actual
↪  time=24.887..24.888 rows=0 loops=3)
         Filter: ((date_reception IS NULL) AND (date_expedition < (now() - '3
↪  days'::interval)))
         Rows Removed by Filter: 335568
 Planning Time: 0.421 ms
 Execution Time: 31.012 ms
```

> Quel index partiel peut-on créer pour optimiser ?

On peut optimiser ces requêtes sur les critères de recherche à l'aide des index partiels suivants :

```
CREATE INDEX ON lots (date_expedition) WHERE date_reception  IS NULL;
```

> Afficher le nouveau plan d'exécution et vérifier l'utilisation du nouvel index.

```
EXPLAIN (ANALYZE)
SELECT * FROM lots
    WHERE date_reception IS NULL
    AND   date_expedition < now() - '3d'::interval;

                          QUERY PLAN
---------------------------------------------------------------
 Index Scan using lots_date_expedition_idx on lots  (cost=0.13..4.15 rows=1
 ↪  width=43) (actual time=0.008..0.009 rows=0 loops=1)
   Index Cond: (date_expedition < (now() - '3 days'::interval))
 Planning Time: 0.243 ms
 Execution Time: 0.030 ms
```

Il est intéressant de noter que seul le test sur la condition indexée (`date_expedition`) est présent dans le plan : la condition `date_reception IS NULL` est implicitement validée par l'index partiel.

Attention, il peut être tentant d'utiliser une formulation de la sorte pour ces requêtes :

```
SELECT * FROM lots
WHERE date_reception IS NULL
AND   now() - date_expedition > '3d'::interval;
```

D'un point de vue logique, c'est la même chose, mais l'optimiseur n'est pas capable de réécrire cette requête correctement. Ici, le nouvel index sera tout de même utilisé, le volume de lignes satisfaisant au critère étant très faible, mais il ne sera pas utilisé pour filtrer sur la date :

```
EXPLAIN (ANALYZE) SELECT * FROM lots
      WHERE date_reception IS NULL
      AND   now() - date_expedition > '3d'::interval;

                              QUERY PLAN
-----------------------------------------------------------------------
 Index Scan using lots_date_expedition_idx on lots
        (cost=0.12..4.15 rows=1 width=43)
        (actual time=0.007..0.007 rows=0 loops=1)
   Filter: ((now() - (date_expedition)::timestamp with time zone) >
           '3 days'::interval)
 Planning time: 0.204 ms
 Execution time: 0.132 ms
```

La ligne importante et différente ici concerne le `Filter` en lieu et place du `Index Cond` du plan précédent. Ici tout l'index partiel (certes tout petit) est lu intégralement et les lignes testées une à une.

C'est une autre illustration des points vus précédemment sur les index non utilisés.

Index fonctionnel

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

```
curl -L https://dali.bo/tp_magasin -o magasin.dump
pg_restore -d magasin  magasin.dump
```

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

> Écrire une requête permettant de renvoyer l'ensemble des produits (table `magasin.produits`) dont le volume ne dépasse pas 1 litre (les unités de longueur sont en mm, 1 litre = 1 000 000 mm^3).

Concernant le volume des produits, la requête est assez simple :

```
SELECT * FROM produits WHERE longueur * hauteur * largeur < 1000000 ;
```

> Quel index permet d'optimiser cette requête ? (Utiliser une fonction est possible, mais pas obligatoire.)

L'option la plus simple est de créer l'index de cette façon, sans avoir besoin d'une fonction :

```
CREATE INDEX ON produits((longueur * hauteur * largeur));
```

En général, il est plus propre de créer une fonction. On peut passer la ligne entière en paramètre pour éviter de fournir 3 paramètres. Il faut que cette fonction soit `IMMUTABLE` pour être indexable :

```
CREATE OR REPLACE function volume (p produits)
RETURNS numeric
AS $$
 SELECT p.longueur * p.hauteur * p.largeur;
$$ language SQL
PARALLEL SAFE
IMMUTABLE ;
```

(Elle est même `PARALLEL SAFE` pour la même raison qu'elle est `IMMUTABLE` : elle dépend uniquement des données de la table.)

On peut ensuite indexer le résultat de cette fonction :

```
CREATE INDEX ON produits (volume(produits)) ;
```

Il est ensuite possible d'écrire la requête de plusieurs manières, la fonction étant ici écrite en SQL et non en PL/pgSQL ou autre langage procédural :

```
SELECT * FROM produits WHERE longueur * hauteur * largeur < 1000000 ;
SELECT * FROM produits WHERE volume(produits) < 1000000 ;
```

En effet, l'optimiseur est capable de « regarder » à l'intérieur de la fonction SQL pour déterminer que les clauses sont les mêmes, ce qui n'est pas vrai pour les autres langages.

En revanche, la requête suivante, où la multiplication est faite dans un ordre différent, n'utilise pas l'index :

```
SELECT * FROM produits WHERE largeur * longueur * hauteur < 1000000 ;
```

et c'est notamment pour cette raison qu'il est plus propre d'utiliser la fonction.

De part l'origine « relationnel-objet » de PostgreSQL, on peut même écrire la requête de la manière suivante :

```
SELECT * FROM produits WHERE produits.volume < 1000000;
```

Cas d'index non utilisés

> Afficher le plan de la requête.

```
SELECT * FROM lignes_commandes WHERE numero_lot_expedition = '190774'::numeric;
```

```
EXPLAIN (ANALYZE,BUFFERS) SELECT * FROM lignes_commandes
                            WHERE numero_lot_expedition = '190774'::numeric;
```

```
                     QUERY PLAN
-----------------------------------------------------------------------------
 Seq Scan on lignes_commandes
                        (cost=0.00..89331.51 rows=15710 width=74)
                        (actual time=0.024..1395.705 rows=6 loops=1)
   Filter: ((numero_lot_expedition)::numeric = '190774'::numeric)
   Rows Removed by Filter: 3141961
   Buffers: shared hit=97 read=42105
 Planning time: 0.109 ms
 Execution time: 1395.741 ms
```

Le moteur fait un parcours séquentiel et retire la plupart des enregistrements pour n'en conserver que 6.

> Créer un index pour améliorer son exécution.

```
CREATE INDEX ON lignes_commandes (numero_lot_expedition);
```

> L'index est-il utilisé ? Quel est le problème ?

L'index n'est pas utilisé à cause de la conversion `bigint` vers `numeric`. Il est important d'utiliser les bons types :

```
EXPLAIN (ANALYZE,BUFFERS)
SELECT * FROM lignes_commandes
WHERE numero_lot_expedition = '190774' ;
```

```
                          QUERY PLAN
-----------------------------------------------------------------------------
 Index Scan using lignes_commandes_numero_lot_expedition_idx
    on lignes_commandes
        (cost=0.43..8.52 rows=5 width=74)
        (actual time=0.054..0.071 rows=6 loops=1)
   Index Cond: (numero_lot_expedition = '190774'::bigint)
   Buffers: shared hit=1 read=4
 Planning time: 0.325 ms
 Execution time: 0.100 ms
```

Sans conversion la requête est bien plus rapide. Faites également le test sans index, le *Seq Scan* sera également plus rapide, le moteur n'ayant pas à convertir toutes les lignes parcourues.

> Écrire une requête pour obtenir les commandes dont la quantité est comprise entre 1 et 8 produits.

```
EXPLAIN (ANALYZE,BUFFERS) SELECT * FROM lignes_commandes
                                    WHERE quantite BETWEEN 1 AND 8;

                            QUERY PLAN
-------------------------------------------------------------------------------
 Seq Scan on lignes_commandes
        (cost=0.00..89331.51 rows=2504357 width=74)
        (actual time=0.108..873.666 rows=2512740 loops=1)
   Filter: ((quantite >= 1) AND (quantite <= 8))
   Rows Removed by Filter: 629227
   Buffers: shared hit=16315 read=25887
 Planning time: 0.369 ms
 Execution time: 1009.537 ms
```

> Créer un index pour améliorer l'exécution de cette requête.

```
CREATE INDEX ON lignes_commandes(quantite);
```

> Pourquoi celui-ci n'est-il pas utilisé ? (Conseil : regarder la vue `pg_stats`)

La table `pg_stats` nous donne des informations de statistiques. Par exemple, pour la répartition des valeurs pour la colonne quantite:

```
SELECT * FROM pg_stats
WHERE tablename='lignes_commandes' AND attname='quantite'
\gx

…
n_distinct             | 10
most_common_vals       | {0,6,1,8,2,4,7,9,5,3}
most_common_freqs      | {0.1037,0.1018,0.101067,0.0999333,0.0999,0.0997,
                                  0.0995,0.0992333,0.0978333,0.0973333}
…
```

Ces quelques lignes nous indiquent qu'il y a 10 valeurs distinctes et qu'il y a environ 10 % d'enregistrements correspondant à chaque valeur.

Avec le prédicat `quantite BETWEEN 1 and 8`, le moteur estime récupérer environ 80 % de la table. Il est donc bien plus coûteux de lire l'index et la table pour récupérer 80 % de la table. C'est pourquoi le moteur fait un *Seq Scan* qui moins coûteux.

> Faire le test avec les commandes dont la quantité est comprise entre 1 et 4 produits.

```
EXPLAIN (ANALYZE,BUFFERS) SELECT * FROM lignes_commandes
                                    WHERE quantite BETWEEN 1 AND 4;
```

```
                            QUERY PLAN
------------------------------------------------------------------------------
 Bitmap Heap Scan on lignes_commandes
            (cost=26538.09..87497.63 rows=1250503 width=74)
            (actual time=206.705..580.854 rows=1254886 loops=1)
   Recheck Cond: ((quantite >= 1) AND (quantite <= 4))
   Heap Blocks: exact=42202
   Buffers: shared read=45633
   ->  Bitmap Index Scan on lignes_commandes_quantite_idx
            (cost=0.00..26225.46 rows=1250503 width=0)
            (actual time=194.250..194.250 rows=1254886 loops=1)
         Index Cond: ((quantite >= 1) AND (quantite <= 4))
         Buffers: shared read=3431
 Planning time: 0.271 ms
 Execution time: 648.414 ms
(9 rows)
```

Cette fois, la sélectivité est différente et le nombre d'enregistrements moins élevé. Le moteur passe donc par un parcours d'index.

Cet exemple montre qu'on indexe selon une requête et non selon une table.