Module J4

Techniques d'indexation

22.09

Dalibo SCOP

https://dalibo.com/formations

Techniques d'indexation

Module J4

TITRE : Techniques d'indexation
SOUS-TITRE : Module J4

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 TECHNIQUES D'INDEXATION

Photo de Maksym Kaharlytskyi[2] , Unsplash licence

1.1 INTRODUCTION

- Qu'est-ce qu'un index ?
- Comment indexer une base ?
- Les différents types d'index

[2] https://unsplash.com/@qwitka

1.1.1 OBJECTIFS

- Comprendre ce qu'est un index
- Maîtriser le processus de création d'index
- Connaître les différents types d'index et leurs cas d'usages

---

1.1.2 INTRODUCTION AUX INDEX

- Uniquement destinés à l'optimisation
- À gérer d'abord par le développeur
  - **Markus Winand** : *SQL Performance Explained*

Les index ne sont pas des objets qui font partie de la théorie relationnelle. Ils sont des objets physiques qui permettent d'accélérer l'accès aux données. Et comme ils ne sont que des moyens d'optimisation des accès, les index ne font pas non plus partie de la norme SQL. C'est d'ailleurs pour cette raison que la syntaxe de création d'index est si différente d'une base de données à une autre.

La création des index est à la charge du développeur ou du DBA, leur création n'est pas automatique, sauf exception.

Pour Markus Winand, c'est d'abord au développeur de poser les index, car c'est lui qui sait comment ses données sont utilisées. Un DBA d'exploitation n'a pas cette connaissance, mais il connaît généralement mieux les différents types d'index et leurs subtilités, et voit comment les requêtes réagissent en production. Développeur et DBA sont complémentaires dans l'analyse d'un problème de performance.

Le site de Markus Winand, Use the index, Luke[3] , propose une version en ligne de son livre *SQL Performance Explained*, centré sur les index B-tree (les plus courants). Une version française est par ailleurs disponible sous le titre *SQL : au cœur des performances*.

---

[3] https://use-the-index-luke.com

1.1.3 UTILITÉ D'UN INDEX

- Un index permet de :
  - trouver un enregistrement dans une table directement
  - récupérer une série d'enregistrements dans une table
  - voire tout récupérer dans l'index (*Index Only Scan*)
- Un index facilite :
  - certains tris
  - certains agrégats
- Obligatoires et automatique pour clés primaires & unicité
  - conseillé pour clés étrangères (FK)

Les index ne changent pas le résultat d'une requête, mais l'accélèrent. L'index permet de pointer l'endroit de la table où se trouve une donnée, pour y accéder directement. Parfois c'est toute une plage de l'index, voire sa totalité, qui sera lue, ce qui est généralement plus rapide que lire toute la table.

Le cas le plus favorable est l'*Index Only Scan* : toutes les données nécessaires sont contenues dans l'index, lui seul sera lu et PostgreSQL ne lira pas la table elle-même.

PostgreSQL propose différentes formes d'index :

- index classique sur une seule colonne d'une table ;
- index composite sur plusieurs colonnes d'une table ;
- index partiel, en restreignant les données indexées avec une clause `WHERE` ;
- index fonctionnel, en indexant le résultat d'une fonction appliquée à une ou plusieurs colonnes d'une table ;
- index couvrants, contenant plus de champs que nécessaire au filtrage, pour ne pas avoir besoin de lire la table, et obtenir un *Index Only Scan*.

La création des index est à la charge du développeur. Seules exceptions : ceux créés automatiquement quand on déclare des contraintes de clé primaire ou d'unicité. La création est alors automatique.

Les contraintes de clé étrangère imposent qu'il existe déjà une clé primaire sur la table pointée, mais ne crée pas d'index sur la table portant la clé.

1.1.4 INDEX ET LECTURES

Un index améliore les `SELECT`

- Sans index :

```
=# SELECT * FROM test WHERE id = 10000;
Temps : 1760,017 ms
```

- Avec index :

```
=# CREATE INDEX idx_test_id ON test (id);
=# SELECT * FROM test WHERE id = 10000;
Temps : 27,711 ms
```

L'index est une structure de données qui permet d'accéder rapidement à l'information recherchée. À l'image de l'index d'un livre, pour retrouver un thème rapidement, on préférera utiliser l'index du livre plutôt que lire l'intégralité du livre jusqu'à trouver le passage qui nous intéresse. Dans une base de données, l'index a un rôle équivalent. Plutôt que de lire une table dans son intégralité, la base de données utilisera l'index pour ne lire qu'une faible portion de la table pour retrouver les données recherchées.

Pour la requête d'exemple (avec une table de 20 millions de lignes), on remarque que l'optimiseur n'utilise pas le même chemin selon que l'index soit présent ou non. Sans index, PostgreSQL réalise un parcours séquentiel de la table :

```
postgres=# EXPLAIN SELECT * FROM test WHERE id = 10000;

                            QUERY PLAN
-------------------------------------------------------------------
 Gather  (cost=1000.00..193661.66 rows=1 width=4)
   Workers Planned: 2
   ->  Parallel Seq Scan on test  (cost=0.00..192661.56 rows=1 width=4)
         Filter: (id = 10000)
```

Lorsqu'il est présent, PostgreSQL l'utilise car l'optimiseur estime que son parcours ne récupérera qu'une seule ligne sur les 10 millions que compte la table :

```
postgres=# EXPLAIN SELECT * FROM test WHERE id = 10000;

                             QUERY PLAN
---------------------------------------------------------------------------
 Index Only Scan using idx_test_id on test  (cost=0.44..8.46 rows=1 width=4)
   Index Cond: (id = 10000)
```

Mais l'index n'accélère pas seulement la simple lecture de données, il permet également d'accélérer les tris et les agrégations, comme le montre l'exemple suivant sur un tri :

```
postgres=# EXPLAIN SELECT id FROM test
           WHERE id BETWEEN 1000 AND 1200 ORDER BY id DESC;
```

```
                                  QUERY PLAN
------------------------------------------------------------------------------
 Index Only Scan Backward using idx_test_id on test
                                              (cost=0.44..12.26 rows=191 width=4)
   Index Cond: ((id >= 1000) AND (id <= 1200))
```

1.1.5 INDEX : INCONVÉNIENTS

- L'index n'est pas gratuit !
- Ralentit les écritures
  - maintenance
- Place disque
- Compromis à trouver

La présence d'un index ralentit les écritures sur une table. En effet, il faut non seulement ajouter ou modifier les données dans la table, mais il faut également maintenir le ou les index de cette table.

Les index dégradent surtout les temps de réponse des insertions. Les mises à jour et les suppressions (`UPDATE` et `DELETE`) tirent en général parti des index pour retrouver les lignes concernées par les modifications. Le coût de maintenance de l'index est secondaire par rapport au coût de l'accès aux données.

Soit une table `test2` telle que :

```
CREATE TABLE test2 (
    id INTEGER GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
    valeur INTEGER,
    commentaire TEXT
);
```

La table est chargée avec pour seul index présent celui sur la clé primaire :

```
=# INSERT INTO test2 (valeur, commentaire)
   SELECT i, 'commentaire ' || i FROM generate_series(1, 10000000) i;
INSERT 0 10000000
Durée : 35253,228 ms (00:35,253)
```

Un index supplémentaire est créé sur une colonne de type entier :

```
=# CREATE INDEX idx_test2_valeur ON test2 (valeur);
=# INSERT INTO test2 (valeur, commentaire)
   SELECT i, 'commentaire ' || i FROM generate_series(1, 10000000) i;
INSERT 0 10000000
Durée : 44410,775 ms (00:44,411)
```

Un index supplémentaire est encore créé, mais cette fois sur une colonne de type texte :

```
=# CREATE INDEX idx_test2_commentaire ON test2 (commentaire);
=# INSERT INTO test2 (valeur, commentaire)
   SELECT i, 'commentaire ' || i FROM generate_series(1, 10000000) i;
INSERT 0 10000000
Durée : 207075,335 ms (03:27,075)
```

On peut comparer ces temps à l'insertion dans une table similaire dépourvue d'index :

```
=# CREATE TABLE test3 AS SELECT * FROM test2;
=# INSERT INTO test3 (valeur, commentaire)
   SELECT i, 'commentaire ' || i FROM generate_series(1, 10000000) i;
INSERT 0 10000000
Durée : 14758,503 ms (00:14,759)
```

La table `test2` a été vidée préalablement pour chaque test.

Enfin, la place disque utilisée par ces index n'est pas négligeable :

```
# \di+ *test2*
                                 Liste des relations
 Schéma |          Nom          | Type  | Propriétaire | Table | Taille |  …
--------+-----------------------+-------+--------------+-------+--------+-
 public | idx_test2_commentaire | index | postgres     | test2 | 387 MB |
 public | idx_test2_valeur      | index | postgres     | test2 | 214 MB |
 public | test2_pkey            | index | postgres     | test2 | 214 MB |

# SELECT pg_size_pretty(pg_relation_size('test2')),
         pg_size_pretty(pg_indexes_size('test2')) ;

 pg_size_pretty | pg_size_pretty
----------------+----------------
 574 MB         | 816 MB
```

Pour ces raisons, on ne posera pas des index systématiquement avant de se demander s'ils seront utilisés. L'idéal est d'étudier les plans de ses requêtes et de chercher à optimiser.

1.1.6 INDEX : CONTRAINTES PRATIQUES À LA CRÉATION

- Lourd…

```
CREATE INDEX ON matable ( macolonne ) ;              -- bloque les écritures !
CREATE INDEX CONCURRENTLY ON matable ( macolonne ) ;  -- ne bloque pas, peut échouer
```

- Si fragmentation :

```
REINDEX INDEX nomindex ;
REINDEX TABLE CONCURRENTLY nomtable ;
```

- Paramètres :
  - `maintenance_work_mem` (sinon : fichier temporaire !)
  - `max_parallel_maintenance_workers`

Création d'un index :

Bien sûr, la durée de création de l'index dépend fortement de la taille de la table. PostgreSQL va lire toutes les lignes et trier les valeurs rencontrées. Ce peut être lourd et impliquer la création de fichiers temporaires.

Si l'on utilise la syntaxe classique, toutes les écritures sur la table sont bloquées (mises en attente) pendant la durée de la création de l'index (verrou *ShareLock*). Les lectures restent possibles, mais cette contrainte est parfois rédhibitoire pour les grosses tables.

Clause CONCURRENTLY :

Ajouter le mot clé `CONCURRENTLY` permet de rendre la table accessible en écriture. Malheureusement, cela nécessite au minimum deux parcours de la table, et donc alourdit et ralentit la construction de l'index. Dans quelques cas défavorables, la création échoue et l'index existe mais devient invalide :

```
postgres=# \d tab
    Table "public.tab"
 Column |  Type   | Collation | Nullable | Default
--------+---------+-----------+----------+---------
 col    | integer |           |          |
    Indexes:
    "idx" btree (col) INVALID
```

L'index est inutilisable et doit être supprimé et recréé, ou bien réindexé.

Pour les détails, voir la documentation officielle[4] .

De tels index invalides sont détectés avec cette requête, qui ne doit jamais rien ramener :

```
SELECT indexrelid::regclass AS index, indrelid::regclass AS table
FROM pg_index
WHERE indisvalid = false ;
```

[4] https://docs.postgresql.fr/15/sql-createindex.html#SQL-CREATEINDEX-CONCURRENTLY

Réindexation :

Comme les tables, les index sont soumis à la fragmentation. Celle-ci peut cependant monter assez haut sans grande conséquence pour les performances. De plus, le nettoyage des index est une des étapes des opérations de VACUUM[5] .

Une reconstruction de l'index est automatique lors d'un `VACUUM FULL` de la table.

Certaines charges provoquent une fragmentation assez élevée, typiquement les tables gérant des files d'attente. Une réindexation reconstruit totalement l'index. Voici quelques variantes de l'ordre :

```
REINDEX INDEX pgbench_accounts_bid_idx ;  -- un seul index
REINDEX TABLE pgbench_accounts ;          -- tous les index de la table
REINDEX (VERBOSE) DATABASE pgbench ;      -- tous deux de la base, avec détails
```

À partir de la version 12, cet ordre accepte une clause `CONCURRENTLY` pour les mêmes raisons que le `CREATE INDEX` :

```
REINDEX (VERBOSE) INDEX CONCURRENTLY pgbench_accounts_bid_idx ;
```

Paramètres :

La rapidité de création d'un index dépend essentiellement de la mémoire accordée, définie dans `maintenance_work_mem`. Si elle ne suffit pas, le tri se fera dans des fichiers temporaires plus lents. Sur les serveurs modernes, le défaut de 64 Mo est ridicule, et on peut monter aisément à :

```
SET maintenance_work_mem = '2GB' ;
```

Attention de ne pas saturer la mémoire en cas de création simultanée de nombreux gros index (lors d'une restauration avec `pg_restore` notamment).

Si le serveur est bien doté en CPU, la parallélisation de la création d'index peut apporter un gain en temps appréciable. La valeur par défaut est :

```
SET max_parallel_maintenance_workers = 2 ;
```

et devrait même être baissée sur les plus petites configurations.

---

[5] https://dali.bo/m5_html#fonctionnement-de-vacuum

1.1.7 TYPES D'INDEX DANS POSTGRESQL

- Défaut : B-tree classique (balancé)
- `UNIQUE` (préférer la contrainte)
- Mais aussi multicolonne, fonctionnel, partiel, couvrant
- Index spécialisés : hash, GiST, GIN, BRIN...

Par défaut un `CREATE INDEX` créera un index de type B-tree, de loin le plus courant. Il est stocké sous forme d'arbre balancé, avec de nombreux avantages :

- performances se dégradant peu avec la taille de l'arbre (les temps de recherche sont en O(log(n)), donc fonction du logarithme du nombre d'enregistrements dans l'index) ;
- excellente concurrence d'accès, avec très peu de contention entre processus qui insèrent simultanément.

Toutefois les B-tree ne permettent de répondre qu'à des questions très simples, portant sur la colonne indexée, et uniquement sur des opérateurs courants (égalité, comparaison). Cela couvre tout de même la majorité des cas.

Contrainte d'unicité et index :

Un index peut être déclaré `UNIQUE` pour provoquer une erreur en cas d'insertion de doublons. Mais on préférera généralement déclarer une *contrainte* d'unicité (notion fonctionnelle), qui techniquement, entraînera la création d'un index.

Par exemple, sur cette table `personne` :

```
$ CREATE TABLE personne (id int, nom text);

$ \d personne
                Table « public.personne »
 Colonne |  Type   | Collationnement | NULL-able | Par défaut
---------+---------+-----------------+-----------+------------
 id      | integer |                 |           |
 nom     | text    |                 |           |
```

on peut créer un index unique :

```
$ CREATE UNIQUE INDEX ON personne (id);

$ \d personne
                 Table « public.personne »
 Colonne |  Type   | Collationnement | NULL-able | Par défaut
---------+---------+-----------------+-----------+------------
 id      | integer |                 |           |
 nom     | text    |                 |           |
```

```
Index :
    "personne_id_idx" UNIQUE, btree (id)
```

La contrainte d'unicité est alors implicite. La suppression de l'index se fait sans bruit :

```
DROP INDEX personne_id_idx;
```

Définissons une contrainte d'unicité sur la colonne plutôt qu'un index :

```
ALTER TABLE personne ADD CONSTRAINT unique_id UNIQUE (id);
```

```
$ \d personne
                 Table « public.personne »
 Colonne |  Type   | Collationnement | NULL-able | Par défaut
---------+---------+-----------------+-----------+------------
 id      | integer |                 |           |
 nom     | text    |                 |           |
Index :
    "unique_id" UNIQUE CONSTRAINT, btree (id)
```

Un index est également créé. La contrainte empêche sa suppression :

```
DROP INDEX unique_id ;
```

```
ERREUR:  n'a pas pu supprimer index unique_id car il est requis par contrainte
 unique_id sur table personne
ASTUCE : Vous pouvez supprimer contrainte unique_id sur table personne à la
 place.
```

Le principe est le même pour les clés primaires.

Indexation avancée :

Il faut aussi savoir que PostgreSQL permet de créer des index B-tree :

- sur plusieurs colonnes ;
- sur des résultats de fonction ;
- sur une partie des valeurs indexées ;
- intégrant des champs non indexés mais souvent récupérés avec les champs indexés (index couvrants).

D'autres types d'index que B-tree existent, destinés à certains types de données ou certains cas d'optimisation précis.

1.2 FONCTIONNEMENT D'UN INDEX

- Anatomie d'un index
- Les index « simples »
- Méthodologie
- Indexation avancée
- Outillage

1.2.1 STRUCTURE D'UN INDEX

- Analogie : index dans une publication scientifique
  - structure séparée, associant des clés (termes) à des localisations (pages)
  - même principe pour un index dans un SGBD
- Structure de données spécialisée, plusieurs types
- Existe en dehors de la table

Pour comprendre ce qu'est un index, l'index dans une publication scientifique au format papier offre une analogie simple.

Lorsque l'on recherche un terme particulier dans un ouvrage, il est possible de parcourir l'intégralité de l'ouvrage pour chercher les termes qui nous intéressent. Ceci prend énormément de temps, variable selon la taille de l'ouvrage. Ce type de recherche trouve son analogie sous la forme du parcours complet d'une table (*Seq Scan*).

Une deuxième méthode pour localiser ces différents termes consiste, si l'ouvrage en dispose, à utiliser l'index de celui-ci. Un tel index associe un terme à un ensemble de pages où celui-ci est présent. Ainsi, pour trouver le terme recherché, il est uniquement nécessaire de parcourir l'index (qui ne dépasse généralement pas quelques pages) à la recherche du terme, puis d'aller visiter les pages listées dans l'index pour extraire les informations nécessaires.

Dans un SGBD, le fonctionnement d'un index est très similaire à celui décrit ici. En effet, comme dans une publication, l'index est une structure de données à part, qui n'est pas strictement nécessaire à l'exploitation des informations, et qui est utilisée pour faciliter la recherche dans l'ensemble de données. Cette structure de données possède un coût de maintenance, dans les deux cas : toute modification des données peut entraîner des modifications afin de maintenir l'index à jour.

1.2.2 UN INDEX N'EST PAS MAGIQUE…

- Un index ne résout pas tout
- Importance de la conception du schéma de données
- Importance de l'écriture de requêtes SQL correctes

Bien souvent, la création d'index est vue comme le remède à tous les maux de performance subis par une application. Il ne faut pas perdre de vue que les facteurs principaux affectant les performances vont être liés à la conception du schéma de données, et à l'écriture des requêtes SQL.

Pour prendre un exemple caricatural, un schéma EAV (*Entity-Attribute-Value*, ou *entité-clé-valeur*) ne pourra jamais être performant, de part sa conception. Bien sûr, dans certains cas, une méthodologie pertinente d'indexation permettra d'améliorer un peu les performances, mais le problème réside là dans la conception même du schéma. Il est donc important dans cette phase de considérer la manière dont le modèle va influer sur les méthodes d'accès aux données, et les implications sur les performances.

De même, l'écriture des requêtes elles-mêmes conditionnera en grande partie les performances observées sur l'application. Par exemple, la mauvaise pratique (souvent mise en œuvre accidentellement via un ORM) dite du « N+1 » ne pourra être corrigée par une indexation correcte : celle-ci consiste à récupérer une collection d'enregistrement (une requête) puis d'effectuer une requête pour chaque enregistrement afin de récupérer les enregistrements liés (N requêtes). Dans ce type de cas, une jointure est bien plus performante. Ce type de comportement doit encore une fois être connu de l'équipe de développement, car il est plutôt difficile à détecter par une équipe d'exploitation.

De manière générale, avant d'envisager la création d'index supplémentaires, il convient de s'interroger sur les possibilités de réécriture des requêtes, voire du schéma.

1.2.3 INDEX B-TREE

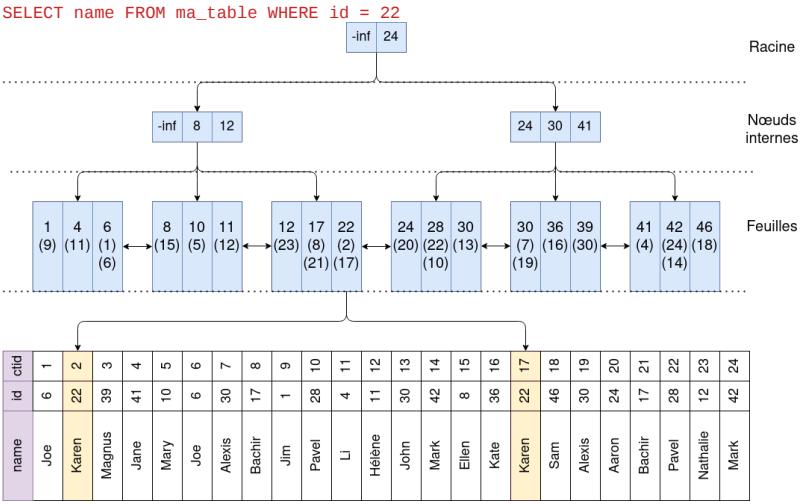
- Type d'index le plus courant
  - et le plus simple
- Utilisable pour les contraintes d'unicité
- Supporte les opérateurs : `<`, `<=`, `=`, `>=`, `>`
- Supporte le tri
- Ne peut pas indexer des colonnes de plus de 2,6 ko

L'index B-tree est le plus simple conceptuellement parlant. Sans entrer dans les détails, un index B-tree est par définition équilibré : ainsi, quelle que soit la valeur recherchée,

le coût est le même lors du parcours d'index. Ceci ne veut pas dire que toute requête impliquant l'index mettra le même temps ! En effet, si chaque clé n'est présente qu'une fois dans l'index, celle-ci peut être associée à une multitude de valeurs, qui devront alors être cherchées dans la table.

L'algorithme utilisé par PostgreSQL pour ce type d'index suppose que chaque page peut contenir au moins trois valeurs. Par conséquent, chaque valeur ne peut excéder un peu moins d'1/3 de bloc, soit environ 2,6 ko. La valeur en question correspond donc à la totalité des données de toutes les colonnes de l'index pour une seule ligne. Si l'on tente de créer ou maintenir un index sur une table ne satisfaisant pas ces prérequis, une erreur sera reportée, et la création de l'index (ou l'insertion/mise à jour de la ligne) échouera. Si un index de type B-tree est tout de même nécessaire sur les colonnes en question, il est possible de créer un index fonctionnel sur une fonction de hachage des valeurs. Dans un tel cas, seul l'opérateur `=` pourra bénéficier d'un parcours d'index.

1.2.4 CONCRÈTEMENT…



Ce schéma présente une vue simplifiée d'une table (en blanc, avec ses champs `id` et `name`) et d'un index B-tree sur `id` (en bleu), tel que le créerait :

```
CREATE INDEX mon_index ON ma_table (id) ;
```

Un index B-tree peuvent contenir trois types de nœuds :

- la racine : elle est unique c'est la base de l'arbre ;
- des nœuds internes : il peut y en avoir plusieurs niveaux ;
- des feuilles : elles contiennent :
  - les valeurs indexées (triées !) ;
  - les valeurs incluses (si applicable) ;
  - les positions physiques (`ctid`), ici entre parenthèses et sous forme abrégée, car la forme réelle est (numéro de bloc, position de la ligne dans le bloc) ;
  - l'adresse de la feuille précédente et de la feuille suivante.

La racine et les nœuds internes contiennent des enregistrements qui décrivent la valeur minimale de chaque bloc du niveau inférieur et leur adresse (`ctid`).

Lors de la création de l'index, il ne contient qu'une feuille. Lorsque cette feuille se remplit, elle se divise en deux et un nœud racine est créé en dessous. Les feuilles se remplissent ensuite progressivement et se séparent en deux quand elles sont pleines. Ce processus remplit progressivement la racine. Lorsque la racine est pleine, elle se divise en deux

nœuds internes, et une nouvelle racine est crée en dessous. Ce processus permet de garder un arbre équilibré.

Recherchons le résultat de :

```
SELECT name FROM ma_table WHERE id = 22
```

en passant par l'index.

- En parcourant la racine, on cherche un enregistrement dont la valeur est strictement supérieure à la valeur que l'on recherche. Ici, 22 est plus petit que 24 : on explore donc le nœud de gauche.
- Ce nœud référence trois nœuds inférieurs (ici des feuilles). On compare de nouveau la valeur recherchée aux différentes valeurs (triées) du nœud : pour chaque intervalle de valeur, il existe un pointeur vers un autre nœud de l'arbre. Ici, 22 est plus grand que 12, on explore donc le nœud de droite au niveau inférieur.
- Un arbre B-tree peut bien évidemment avoir une profondeur plus grande, auquel cas l'étape précédente est répétée.
- Une fois arrivé sur une feuille, il suffit de la parcourir pour récupérer l'ensemble des positions physiques des lignes correspondants au critère. Ici, la feuille nous indique qu'à la valeur 22 correspondent deux lignes aux positions 2 et 17. Lorsque la valeur recherchée est supérieure ou égale à la plus grande valeur du bloc, PostgreSQL va également lire le bloc suivant. Ce cas de figure peut se produire si PostgreSQL a divisé une feuille en deux avant ou même pendant la recherche que nous exécutons. Ce serait par exemple le cas si on cherchait la valeur 30.
- Pour trouver les valeurs de `name`, il faut aller chercher dans la table même les lignes aux positions trouvées dans l'index. D'autre part, les informations de visibilité des lignes doivent aussi être trouvées dans la table. (Il existe des cas où la recherche peut éviter cette dernière étape : ce sont les *Index Only Scan*.)

Même en parcourant les deux structures de données, si la valeur recherchée représente une assez petite fraction des lignes totales, le nombre d'accès disques sera donc fortement réduit. En revanche, au lieu d'effectuer des accès séquentiels (pour lesquels les disques durs classiques sont relativement performants), il faudra effectuer des accès aléatoires, en *sautant* d'une position sur le disque à une autre. Le choix est fait par l'optimiseur.

Supposons désormais que nous souhaitions exécuter une requête sans filtre, mais exigeant un tri, du type :

```
SELECT id FROM ma_table ORDER BY id ;
```

L'index peut nous aider à répondre à cette requête. En effet, toutes les feuilles sont liées entre elles, et permettent ainsi un parcours ordonné. Il nous suffit donc de localiser la pre-

mière feuille (la plus à gauche), et pour chaque clé, récupérer les lignes correspondantes. Une fois les clés de la feuille traitées, il suffit de suivre le pointeur vers la feuille suivante et de recommencer.

L'alternative consisterait à parcourir l'ensemble de la table, et trier toutes les lignes afin de les obtenir dans le bon ordre. Un tel tri peut être très coûteux, en mémoire comme en temps CPU. D'ailleurs, de tels tris débordent très souvent sur disque (via des fichiers temporaires) afin de ne pas garder l'intégralité des données en mémoire.

Pour les requêtes utilisant des opérateurs d'inégalité, on voit bien comment l'index peut là aussi être utilisé. Par exemple, pour la requête suivante :

```
SELECT * FROM ma_table WHERE id <= 10 AND id >= 4 ;
```

Il suffit d'utiliser la propriété de tri de l'index pour parcourir les feuilles, en partant de la borne inférieure, jusqu'à la borne supérieure.

Dernière remarque : ce schéma ne montre qu'une entrée d'index pour 22, bien qu'il pointe vers deux lignes. En fait, il y avait bien deux entrées pour 22 avant PostgreSQL 13. Depuis cette version, PostgreSQL sait dédupliquer les entrées.

1.2.5 INDEX MULTICOLONNES

- Possibilité d'indexer plusieurs colonnes :
  ```
  CREATE INDEX ON ma_table (id, name) ;
  ```
- Ordre des colonnes **primordial**
  - accès direct aux premières colonnes de l'index
  - pour les autres, PostgreSQL lira tout l'index ou ignorera l'index

Il est possible de créer un index sur plusieurs colonnes. Il faut néanmoins être conscient des requêtes supportées par un tel index. Admettons que l'on crée une table d'un million de lignes avec un index sur trois champs :

```
CREATE TABLE t1 (c1 int, c2 int, c3 int, c4 text);

INSERT INTO t1 (c1, c2, c3, c4)
SELECT i*10,j*5,k*20, 'text'||i||j||k
FROM generate_series (1,100) i
CROSS JOIN generate_series(1,100) j
CROSS JOIN generate_series(1,100) k ;

CREATE INDEX ON t1 (c1, c2, c3) ;

VACUUM ANALYZE t1 ;
```

```
-- Figer des paramètres pour l'exemple
SET max_parallel_workers_per_gather to 0;
SET seq_page_cost TO 1 ;
SET random_page_cost TO 4 ;
```

L'index est optimal pour répondre aux requêtes portant sur les premières colonnes de l'index :

```
EXPLAIN SELECT * FROM t1 WHERE c1 = 1000 and c2=500 and c3=2000 ;

                                QUERY PLAN
-------------------------------------------------------------------------
 Index Scan using t1_c1_c2_c3_idx on t1  (cost=0.42..8.45 rows=1 width=22)
   Index Cond: ((c1 = 1000) AND (c2 = 500) AND (c3 = 2000))
```

Et encore plus quand l'index permet de répondre intégralement au contenu de la requête :

```
EXPLAIN SELECT c1,c2,c3 FROM t1 WHERE c1 = 1000 and c2=500 ;

                                  QUERY PLAN
-------------------------------------------------------------------------------
 Index Only Scan using t1_c1_c2_c3_idx on t1  (cost=0.42..6.33 rows=95 width=12)
   Index Cond: ((c1 = 1000) AND (c2 = 500))
```

Mais si les premières colonnes de l'index ne sont pas spécifiées, alors l'index devra être parcouru en grande partie.

Cela reste plus intéressant que parcourir toute la table, surtout si l'index est petit et contient toutes les données du `SELECT`. Mais le comportement dépend alors de nombreux paramètres, comme les statistiques, les estimations du nombre de lignes ramenées et les valeurs relatives de `seq_page_cost` et `random_page_cost` :

```
SET random_page_cost TO 0.1 ; SET seq_page_cost TO 0.1 ;  -- SSD

EXPLAIN (ANALYZE,BUFFERS) SELECT * FROM t1 WHERE c3 = 2000 ;

                                  QUERY PLAN
-------------------------------------------------------------------------------
 Index Scan using t1_c1_c2_c3_idx on t1  (...) (...)
   Index Cond: (c3 = 2000)
   Buffers: shared hit=3899
 Planning:
   Buffers: shared hit=15
 Planning Time: 0.218 ms
 Execution Time: 67.081 ms
```

Noter que tout l'index a été lu.

Mais pour limiter les aller-retours entre index et table, PostgreSQL peut aussi décider d'ignorer l'index et de parcourir directement la table :

```
SET random_page_cost TO 4 ; SET seq_page_cost TO 1 ;  -- défaut (disque mécanique)

EXPLAIN (ANALYZE,BUFFERS) SELECT * FROM t1 WHERE c3 = 2000 ;

                                  QUERY PLAN
-------------------------------------------------------------------------------
Seq Scan on t1  (cost=0.00..18871.00 rows=9600 width=22) (...)
   Filter: (c3 = 2000)
   Rows Removed by Filter: 990000
   Buffers: shared hit=6371
 Planning Time: 0.178 ms
 Execution Time: 114.572 ms
```

Concernant les *range scans* (requêtes impliquant des opérateurs d'inégalité, tels que <, <=, >=, >), celles-ci pourront être satisfaites par l'index de manière quasi optimale si les opérateurs d'inégalité sont appliqués sur la dernière colonne requêtée, et de manière sub-optimale s'ils portent sur les premières colonnes.

Cet index pourra être utilisé pour répondre aux requêtes suivantes de manière optimale :

```
SELECT * FROM t1 WHERE c1 = 20 ;
SELECT * FROM t1 WHERE c1 = 20 AND c2 = 50 AND c3 = 400 ;
SELECT * FROM t1 WHERE c1 = 10 AND c2 <= 4 ;
```

Il pourra aussi être utilisé, mais de manière bien moins efficace, pour les requêtes suivantes, qui bénéficieraient d'un index sur un ordre alternatif des colonnes :

```
SELECT * FROM t1 WHERE c1 = 100 AND c2 >= 80 AND c3 = 40 ;
SELECT * FROM t1 WHERE c1 < 100 AND c2 = 100 ;
```

Le plan de cette dernière requête est :

```
 Bitmap Heap Scan on t1  (cost=2275.98..4777.17 rows=919 width=22) (...)
   Recheck Cond: ((c1 < 100) AND (c2 = 100))
   Heap Blocks: exact=609
   Buffers: shared hit=956
   ->  Bitmap Index Scan on t1_c1_c2_c3_idx  (cost=0.00..2275.76 rows=919 width=0) (...)
         Index Cond: ((c1 < 100) AND (c2 = 100))
         Buffers: shared hit=347
 Planning Time: 0.227 ms
 Execution Time: 15.596 ms
```

Les index multicolonnes peuvent aussi être utilisés pour le tri comme dans les exemples suivants. Il n'y a pas besoin de trier (ce peut être très coûteux) puisque les données de

l'index sont triées. Ici le cas est optimal puisque l'index contient toutes les données nécessaires :

```
SELECT * FROM t1 ORDER BY c1 ;
SELECT * FROM t1 ORDER BY c1, c2 ;
SELECT * FROM t1 ORDER BY c1, c2, c3 ;
```

Le plan de cette dernière requête est :

```
 Index Scan using t1_c1_c2_c3_idx on t1  (cost=0.42..55893.66 rows=1000000 width=22) (...)
   Buffers: shared hit=1003834
 Planning Time: 0.282 ms
 Execution Time: 425.520 ms
```

Il est donc nécessaire d'avoir une bonne connaissance de l'application (ou de passer du temps à observer les requêtes consommatrices) pour déterminer comment créer des index multicolonnes pertinents pour un nombre maximum de requêtes.

1.3 MÉTHODOLOGIE DE CRÉATION D'INDEX

- On indexe pour une requête, ou idéalement une collection de requêtes
- On n'indexe pas « une table »

La première chose à garder en tête est que l'on indexe pas le schéma de données, c'est-à-dire les tables, mais en fonction de la charge de travail supportée par la base, c'est-à-dire les requêtes. En effet, comme nous l'avons vu précédemment, tout index superflu a un coût global pour la base de données, notamment pour les opérations DML.

1.3.1 L'INDEX ? QUEL INDEX ?

- Identifier les requêtes nécessitant un index
- Créer les index permettant de répondre à ces requêtes
- Valider le fonctionnement, en rejouant la requête avec :
  `EXPLAIN (ANALYZE, BUFFERS)`

La méthodologie elle-même est assez simple. Selon le principe qu'un index sert à une (ou des) requête(s), la première chose à faire consiste à identifier celles-ci. L'équipe de développement est dans une position idéale pour réaliser ce travail : elle seule peut connaître le fonctionnement global de l'application, et donc les colonnes qui vont être utilisées, ensemble ou non, comme cible de filtres ou de tris. Au delà de la connaissance de l'application, il est possible d'utiliser des outils tels que pgBadger, pg_stat_statements

et PoWA pour identifier les requêtes particulièrement consommatrices, et qui pourraient donc potentiellement nécessiter un index. Ces outils seront présentés plus loin dans cette formation.

Une fois les requêtes identifiées, il est nécessaire de trouver les index permettant d'améliorer celles-ci. Ils peuvent être utilisés pour les opérations de filtrage (clause `WHERE`), de tri (clauses `ORDER BY`, `GROUP BY`) ou de jointures. Idéalement, l'étude portera sur l'ensemble des requêtes, afin notamment de pouvoir décider d'index multi-colonnes pertinents pour le plus grand nombre de requêtes, et éviter ainsi de créer des index redondants.

1.3.2 INDEX ET CLÉS ÉTRANGÈRES

- Indexation des colonnes faisant référence à une autre
- Performances des DML
- Performances des jointures

De manière générale, l'ensemble des colonnes étant la source d'une clé étrangère devraient être indexées, et ce pour deux raisons.

La première concerne les jointures. Généralement, lorsque deux tables sont liées par des clés étrangères, il existe au moins certaines requêtes dans l'application joignant ces tables. La colonne « cible » de la clé étrangère est nécessairement indexée, c'est un prérequis dû à la contrainte unique nécessaire à celle-ci. Il est donc possible de la parcourir de manière triée.

La colonne source devrait être indexée elle aussi : en effet, il est alors possible de la parcourir de manière ordonnée, et donc de réaliser la jointure selon l'algorithme *Merge Join* (comme vu lors du module sur les plans d'exécution[6]), et donc d'être beaucoup plus rapide. Un tel index accélérera de la même manière les *Nested Loop*, en permettant de parcourir l'index une fois par ligne de la relation externe au lieu de parcourir l'intégralité de la table.

De la même manière, pour les DML sur la table cible, cet index sera d'une grande aide : pour chaque ligne modifiée ou supprimée, il convient de vérifier, soit pour interdire soit pour « cascader » la modification, la présence de lignes faisant référence à celle touchée.

S'il n'y a qu'une règle à suivre aveuglément ou presque, c'est bien celle-ci : les colonnes faisant partie d'une clé étrangère doivent être indexées !

[6] https://dali.bo/j0_html

Deux exceptions : les champs ayant une cardinalité très faible et homogène (par exemple, un champ homme/femme dans une population équilibrée) ; et ceux dont on constate l'inutilité après un certain temps, par des valeurs à zéro dans `pg_stat_user_indexes`.

1.4 INDEX INUTILISÉ

- C'est souvent tout à fait normal
- Utiliser l'index est-il rentable ?
- La requête est-elle compatible ?
- Bug de l'optimiseur : rare

C'est l'optimiseur SQL qui choisit si un index doit ou non être utilisé. Il est tout à fait possible que PostgreSQL décide qu'utiliser un index donné n'en vaut pas la peine par rapport à d'autres chemins. Il faut aussi savoir identifier les cas où l'index ne peut *pas* être utilisé.

L'optimiseur possède forcément quelques limitations. Certaines sont un compromis par rapport au temps que prendrait la recherche systématique de toutes les optimisations imaginables. Il y aussi le problème des estimations de volumétries, qui sont d'autant plus difficiles que la requête est complexe.

Quant à un vrai bug, si le cas peut être reproduit, il doit être remonté aux développeurs de PostgreSQL. D'expérience, c'est rarissime.

1.4.1 INDEX UTILISABLE MAIS NON UTILISÉ

- L'optimiseur pense qu'il n'est pas rentable
  - sélectivité trop faible
  - meilleur chemin pour remplir d'autres critères
  - index redondant
  - *Index Only Scan* nécessite un `VACUUM` fréquent
- Les estimations de volumétries doivent être assez bonnes !
  - statistiques récentes, précises

Il existe plusieurs raisons pour que PostgreSQL néglige un index.

Sélectivité trop faible, trop de lignes :

Comme vu précédemment, le parcours d'un index implique à la fois des lectures sur l'index, et des lectures sur la table. Au contraire d'une lecture séquentielle de la table (*Seq Scan*),

l'accès aux données via l'index nécessite des lectures aléatoires. Ainsi, si l'optimiseur estime que la requête nécessitera de parcourir une grande partie de la table, il peut décider de ne pas utiliser l'index : l'utilisation de celui-ci serait alors trop coûteux.

Autrement dit, l'index n'est pas assez discriminant pour que ce soit la peine de faire des allers-retours entre lui et la table. Le seuil dépend entre autres des volumétries de la table et de l'index et du rapport entre les paramètres `random_page_cost` et `seq_page_cost` (respectivement 4 et 1 pour un disque dur classique peu rapide, et souvent 1 et 1 pour du SSD, voire moins).

Il y a un meilleur chemin :

Un index sur un champ n'est qu'un chemin parmi d'autres, en aucun cas une obligation, et une requête contient souvent plusieurs critères sur des tables différentes. Par exemple, un index sur un filtre peut être ignoré si un autre index permet d'éviter un tri coûteux, ou si l'optimiseur juge que faire une jointure avant de filtrer le résultat est plus performant.

Index redondant :

Il existe un autre index doublant la fonctionnalité de celui considéré. PostgreSQL favorise naturellement un index plus petit, plus rapide à parcourir. À l'inverse, un index plus complet peut favoriser plusieurs filtres, des tris, devenir couvrant…

VACUUM trop ancien :

Dans le cas précis des *Index Only Scan*, si la table n'a pas été récemment nettoyée, il y aura trop d'allers-retours avec la table pour vérifier les informations de visibilité (*heap fetches*). Un `VACUUM` permet de mettre à jour la *Visibility Map* pour éviter cela.

Statistiques périmées :

Il peut arriver que l'optimiseur se trompe quand il ignore un index. Des statistiques périmées sont une cause fréquente. Pour les rafraîchir :

```
ANALYZE (VERBOSE) nom_table;
```

Si cela résout le problème, ce peut être un indice que l'autovacuum ne passe pas assez souvent (voir `pg_stat_user_tables.last_autoanalyze`). Il faudra peut-être ajuster les paramètres `autovacuum_analyze_scale_factor` ou `autovacuum_analyze_threshold` sur les tables.

Statistiques pas assez fines :

Les statistiques sur les données peuvent être trop imprécises. Le défaut est un histogramme de 100 valeurs, basé sur 300 fois plus de lignes. Pour les grosses tables, augmenter l'échantillonnage sur les champs aux valeurs peu homogènes est possible :

```
ALTER TABLE ma_table ALTER ma_colonne SET STATISTICS 500 ;
```

La valeur 500 n'est qu'un exemple. Monter beaucoup plus haut peut pénaliser les temps de planification. Ce sera d'autant plus vrai si on applique cette nouvelle valeur globalement, donc à tous les champs de toutes les tables (ce qui est certes le plus facile).

Estimations de volumétries trompeuses :

Par exemple, une clause `WHERE` sur deux colonnes corrélées (ville et code postal par exemple), mène à une sous-estimation de la volumétrie résultante par l'optimiseur, car celui-ci ignore le lien entre les deux champs.

À partir de la version 10, vous pouvez indiquer explicitement cette corrélation à PostgreSQL avec l'ordre `CREATE STATISTICS` (voir https://docs.postgresql.fr/current/sql-createstatistics.html).

1.4.2 INDEX INUTILISABLE PAR LA REQUÊTE

- Pas le bon type (`CAST` plus ou moins explicite)
- Utilisation de fonctions, comme :

```
SELECT * FROM ma_table WHERE to_char(ma_date, 'YYYY')='2014' ;
```

- Pas les bons opérateurs
  - ex : `LIKE 'critère%'`
- Index invalide

Il faut toujours s'assurer que la requête est écrite correctement et permet l'utilisation de l'index.

Un index peut être inutilisable à cause d'une fonction plus ou moins explicite, ou encore d'un mauvais typage. Il arrive que le critère de filtrage ne peut remonter sur la table indexée à cause d'un CTE matérialisé, d'un `DISTINCT`, ou d'une vue complexe.

Voici quelques exemples d'index incompatible avec la clause `WHERE` :

Mauvais type :

Cela peut paraître contre-intuitif, mais certains transtypages ne permettent pas de garantir que les résultats d'un opérateur (par exemple l'égalité) seront les mêmes si les arguments sont convertis dans un type ou dans l'autre.

```
sql=# EXPLAIN SELECT * FROM clients WHERE client_id = 3::numeric;

                        QUERY PLAN
-------------------------------------------------------------
```

```
 Seq Scan on clients  (cost=0.00..2525.00 rows=500 width=51)
   Filter: ((client_id)::numeric = 3::numeric)

sql=# EXPLAIN SELECT * FROM clients WHERE client_id = 3;

                                QUERY PLAN
-------------------------------------------------------------------------
 Index Scan using clients_pkey on clients  (cost=0.29..8.31 rows=1 width=51)
   Index Cond: (client_id = 3)
```

Aure exemple : vous avez créé un index B-tree sur un tableau ou un JSON, et vous exécutez une recherche sur un de ses éléments. Il faudra s'orienter vers un index de type GIN, par exemple.

Utilisation de fonction :

Une fonction est appliquée sur la colonne à indexer, comme dans cet exemple classique :

```
SELECT * FROM ma_table WHERE to_char(ma_date, 'YYYY')='2014' ;
```

PostgreSQL n'utilisera pas l'index sur `ma_date`. Il faut réécrire la requête ainsi :

```
SELECT * FROM ma_table WHERE ma_date >='2014-01-01' AND ma_date<'2015-01-01' ;
```

Dans cet autre exemple, on cherche les commandes dont la date tronquée au mois correspond au 1er janvier, c'est-à-dire aux commandes dont la date est entre le 1er et le 31 janvier. Pour un humain, la logique est évidente, mais l'optimiseur n'en a pas connaissance.

```
# EXPLAIN ANALYZE
  SELECT * FROM commandes
  WHERE date_trunc('month', date_commande) = '2015-01-01';

                              QUERY PLAN
-----------------------------------------------------------------------
Gather  (cost=1000.00..8160.96 rows=5000 width=51)
        (actual time=17.282..192.131 rows=4882 loops=1)
 Workers Planned: 3
 Workers Launched: 3
 ->  Parallel Seq Scan on commandes (cost=0.00..6660.96 rows=1613 width=51)
                          (actual time=17.338..177.896 rows=1220 loops=4)
     Filter: (date_trunc('month'::text,
                                  (date_commande)::timestamp with time zone)
                       = '2015-01-01 00:00:00+01'::timestamp with time zone)
     Rows Removed by Filter: 248780
 Planning time: 0.215 ms
 Execution time: 196.930 ms
```

Il faut plutôt écrire :

```
# EXPLAIN ANALYZE
  SELECT * FROM commandes
  WHERE date_commande BETWEEN '2015-01-01' AND '2015-01-31' ;

                          QUERY PLAN
-----------------------------------------------------------
 Index Scan using commandes_date_commande_idx on commandes
                                    (cost=0.42..118.82 rows=5554 width=51)
                               (actual time=0.019..0.915 rows=4882 loops=1)
   Index Cond: ((date_commande >= '2015-01-01'::date)
           AND (date_commande <= '2015-01-31'::date))
 Planning time: 0.074 ms
 Execution time: 1.098 ms
```

Dans certains cas, la réécriture est impossible (fonction complexe, code non modifiable…) et un index fonctionnel sera nécessaire.

Ces exemples semblent évidents, mais il peut être plus compliqué de trouver la cause du problème dans une grande requête d'un schéma mal connu dans l'urgence.

LIKE :

Si vous avez un index « normal » sur une chaîne texte, certaines recherches de type `LIKE` n'utiliseront pas l'index. En effet, il faut bien garder à l'esprit qu'un index ne sert qu'à certains opérateurs. Ceci est généralement indiqué correctement dans la documentation, mais pas forcément très intuitif. Pour plus de détails à ce sujet, se référer à la section correspondant aux classes d'opérateurs[7] . Si un opérateur non supporté est utilisé, l'index ne servira à rien :

```
sql=# CREATE INDEX ON fournisseurs (commentaire);
CREATE INDEX

sql=# EXPLAIN ANALYZE SELECT * FROM fournisseurs WHERE commentaire LIKE 'ipsum%';

                              QUERY PLAN
-----------------------------------------------------------------------
 Seq Scan on fournisseurs  (cost=0.00..225.00 rows=1 width=45)
                           (actual time=0.045..1.477 rows=47 loops=1)
   Filter: (commentaire ~~ 'ipsum%'::text)
   Rows Removed by Filter: 9953
 Planning time: 0.085 ms
 Execution time: 1.509 ms
```

Nous verrons qu'il existe d'autre classes d'opérateurs, permettant d'indexer correctement la requête précédente.

[7] https://www.postgresql.org/docs/current/static/indexes-opclass.html

Index invalide :

Dans le cas où un index a été construit ou ré-indexé avec la clause `CONCURRENTLY`, il peut arriver que l'opération échoue et l'index existe mais reste invalide. De tels index se repèrent :

```
SELECT indexrelid::regclass
FROM pg_index
WHERE indisvalid IS FALSE ;
```

Si cette requête trouve un index invalide, il doit être supprimé ou reconstruit.

1.5 INDEXATION B-TREE AVANCÉE

De nombreuses possibilités d'indexation avancée :

- Index partiels
- Index fonctionnels
- Index couvrants
- Classes d'opérateur

1.5.1 INDEX PARTIELS

- N'indexe qu'une partie des données :

```
CREATE INDEX on evenements (type) WHERE  traite IS FALSE ;
```

- Ne sert que si la clause exacte est respectée !
- Intérêt : index beaucoup plus petit

Un index partiel est un index ne couvrant qu'une partie des enregistrements. Ainsi, l'index est beaucoup plus petit. En contrepartie, il ne pourra être utilisé que si sa condition est définie dans la requête.

Pour prendre un exemple simple, imaginons un système de « queue », dans lequel des événements sont entrés, et qui disposent d'une colonne `traite` indiquant si oui ou non l'événement a été traité. Dans le fonctionnement normal de l'application, la plupart des requêtes ne s'intéressent qu'aux événements non traités :

```
sql=# CREATE TABLE evenements (
    id int primary key,
    traite bool NOT NULL,
    type text NOT NULL,
    payload text
);
```

```
CREATE TABLE
```

```
sql=# INSERT INTO evenements (id, traite, type) (
    SELECT i, true,
        CASE WHEN i % 3 = 0 THEN 'FACTURATION'
             WHEN i % 3 = 1 THEN 'EXPEDITION'
             ELSE 'COMMANDE'
        END
    FROM generate_series(1, 10000) as i);
INSERT 0 10000

sql=# INSERT INTO evenements (id, traite, type) (
    SELECT i, false,
        CASE WHEN i % 3 = 0 THEN 'FACTURATION'
             WHEN i % 3 = 1 THEN 'EXPEDITION'
             ELSE 'COMMANDE'
        END
    FROM generate_series(10001, 10010) as i);
INSERT 0 10

sql=# \d evenements

                Table « public.evenements »
 Colonne |  Type   | Collationnement | NULL-able | Par défaut
---------+---------+-----------------+-----------+------------
 id      | integer |                 | not null  |
 traite  | boolean |                 | not null  |
 type    | text    |                 | not null  |
 payload | text    |                 |           |
Index :
    "evenements_pkey" PRIMARY KEY, btree (id)
```

Typiquement, différents applicatifs vont être intéressés par des événements d'un certain type, mais les événements déjà traités ne sont quasiment jamais accédés, du moins via leur état (une requête portant sur `traite IS true` sera exceptionnelle et ramènera l'essentiel de la table : un index est inutile).

Ainsi, on peut souhaiter indexer le type d'événement, mais uniquement pour les événements non traités :

```
sql=# CREATE INDEX index_partiel on evenements (type) WHERE NOT traite;
CREATE INDEX
```

Si on recherche les événements dont le type est « FACTURATION », sans plus de précision, l'index ne peut évidemment pas être utilisé :

```
sql=# EXPLAIN SELECT * FROM evenements WHERE type = 'FACTURATION';
```

```
                            QUERY PLAN
------------------------------------------------------------------
 Seq Scan on evenements  (cost=0.00..183.12 rows=3336 width=47)
   Filter: (type = 'FACTURATION'::text)
```

En revanche, si la condition sur l'état de l'événement est précisée, l'index sera utilisé :

```
sql=# EXPLAIN SELECT * FROM evenements
      WHERE type = 'FACTURATION' AND NOT traite;

                                QUERY PLAN
---------------------------------------------------------------------------
 Bitmap Heap Scan on evenements  (cost=4.13..14.12 rows=3 width=47)
   Recheck Cond: ((type = 'FACTURATION'::text) AND (NOT traite))
   ->  Bitmap Index Scan on index_partiel  (cost=0.00..4.13 rows=3 width=0)
         Index Cond: (type = 'FACTURATION'::text)
(4 rows)
```

Attention ! La clause `NOT traite` dans le `WHERE` doit être utilisée de manière strictement identique : un critère `traite IS FALSE` à la place de `NOT false` n'utilise pas l'index !

Sur ce jeu de données, on peut comparer la taille de deux index, partiels ou non :

```
CREATE INDEX index_complet ON evenements(type);

SELECT idxname, pg_size_pretty(pg_total_relation_size(idxname::text))
      FROM (VALUES ('index_complet'), ('index_partiel')) as a(idxname);

    idxname    | pg_size_pretty
---------------+----------------
 index_complet | 88 kB
 index_partiel | 16 kB
```

Un index composé sur `(is_traite,type)` serait efficace, mais inutilement gros.

1.5.2 INDEX PARTIELS : CAS D'USAGE

- Données *chaudes* et *froides*
- Index pour une requête ayant une condition fixe
- Colonne indexée et `WHERE` peuvent porter sur des colonnes différentes
- Éviter les index de type :

```
CREATE INDEX ON matable( champ1 ) WHERE champ1 = …
```

- Préférer :

```
CREATE INDEX ON matable ( champ_retourné ) WHERE champ1 = …
```

Le cas typique d'utilisation d'un index partiel correspond, comme dans l'exemple précédent, aux applications contenant des données *chaudes*, fréquemment accédées et traitées, et des données *froides*, qui sont plus destinées à de l'historisation ou de l'archivage.

Par exemple, un système de vente en ligne aura probablement intérêt à disposer d'index sur les commandes dont l'état est différent de clôturé : en effet, un tel système effectuera probablement des requêtes fréquemment sur les commandes qui sont en cours de traitement, en attente d'expédition, en cours de livraison mais très peu sur des commandes déjà livrées, qui ne serviront alors plus qu'à de l'analyse statistique. De manière générale, tout système ayant à gérer des données ayant un état, et dont seul un sous-ensemble des états est activement exploité par l'application.

Nous avons mentionné précédemment qu'un index est destiné à satisfaire une requête ou un ensemble de requêtes. De cette manière, si une requête présente fréquemment des critères de ce type :

```
WHERE une_colonne = un_parametre_variable
  AND une_autre_colonne = une_valeur_fixe
```

alors il peut être intéressant de créer un index partiel pour les lignes satisfaisant le critère :

```
WHERE une_autre_colonne = une_valeur_fixe
```

Ces critères sont généralement très liés au fonctionnel de l'application : du point de vue de l'exploitation, il est souvent difficile d'identifier des requêtes dont une valeur est toujours fixe. Encore une fois, l'appropriation des techniques d'indexation par l'équipe de développement permet d'améliorer grandement les performances de l'application.

En général, un index partiel doit indexer une colonne différente de celle qui est filtrée : en effet, le contenu de l'index est alors peu utile, puisqu'il sert à filtrer. Il est plus intéressant d'y associer une colonne sur laquelle un autre filtre ou un tri est effectué. Ainsi, dans l'exemple précédent, la colonne indexée (`type`) n'est pas celle de la clause `WHERE`. On pose un critère, mais on s'intéresse aux types d'événements ramenés. Un autre index partiel pourrait porter sur `id WHERE NOT traite` pour simplement récupérer une liste des identifiants non traités de tous types.

L'intérêt est d'obtenir un index très ciblé et compact, et aussi d'économiser la place disque et la charge CPU de maintenance.

1.5.3 INDEX FONCTIONNELS

- Un index sur `a` est inutilisable pour :

```
WHERE upper(a)='DUPOND'
```

- Indexer le résultat de la fonction :

```
CREATE INDEX mon_idx ON ma_table (upper(a)) ;
```

- Fonction impérativement `IMMUTABLE` !
- Ne pas oublier `ANALYZE` !
- La fonction ne doit jamais tomber en erreur !
- Modification de la fonction : réindexation !

À partir du moment où une clause `WHERE` applique une fonction sur une colonne, un index sur la colonne ne permet plus un accès à l'enregistrement.

C'est comme demander à un dictionnaire Anglais vers Français : « Quels sont les mots dont la traduction en français est 'fenêtre' ? ». Le tri du dictionnaire ne correspond pas à la question posée. Il nous faudrait un index non plus sur les mots anglais, mais sur leur traduction en français.

C'est exactement ce que font les index fonctionnels : ils indexent le résultat d'une fonction appliquée à l'enregistrement.

L'exemple classique est l'indexation insensible à la casse : on crée un index sur `UPPER` (ou `LOWER`) de la chaîne à indexer, et on recherche les mots convertis à la casse souhaitée.

Il est facile de tomber involontairement dans ce cas, notamment avec des manipulations de dates. En général, il est résolu en plaçant la transformation du côté de la constante. Par exemple, la requête suivante retourne toutes les commandes de l'année 2011, mais la fonction `extract` est appliquée à la colonne `date_commande`. L'optimiseur ne peut donc pas utiliser un index :

```
tpc=# EXPLAIN SELECT * FROM commandes
WHERE extract('year' from date_commande) = 2011;

                                QUERY PLAN
-------------------------------------------------------------------------------
 Seq Scan on commandes  (cost=0.00..5364.12 rows=844 width=77)
   Filter: (date_part('year'::text,
     (date_commande)::timestamp without time zone) = 2011::double precision)
```

En réécrivant le prédicat, l'index est bien utilisé :

```
tpc=# EXPLAIN SELECT * FROM commandes
WHERE date_commande BETWEEN '01-01-2011'::date AND '31-12-2011'::date;

                                QUERY PLAN
-------------------------------------------------------------------------------
```

```
Bitmap Heap Scan on commandes  (cost=523.85..3302.80 rows=24530 width=77)
  Recheck Cond: ((date_commande >= '2011-01-01'::date)
                 AND (date_commande <= '2011-12-31'::date))
  ->  Bitmap Index Scan on idx_commandes_date_commande
          (cost=0.00..517.72 rows=24530 width=0)
        Index Cond: ((date_commande >= '2011-01-01'::date)
                     AND (date_commande <= '2011-12-31'::date))
```

Mais dans d'autres cas, une telle réécriture de la requête sera impossible. L'index fonctionnel associé doit être strictement celui utilisé dans le `WHERE` :

```
CREATE INDEX ON commandes( extract('year' from date_commande) ) ;
```

La fonction d'indexation utilisée doit être notée `IMMUTABLE`, indiquant que la fonction retournera toujours le même résultat quand elle est appelée avec les mêmes arguments (elle ne dépend pas du contenu de la base, de la configuration et n'a pas de comportement non-déterministe comme `random` ou `clock_timestamp()`). Dans le cas contraire, l'endroit dans lequel la donnée devrait être insérée dans l'index serait potentiellement différent à chaque exécution, ce qui est évidemment incompatible avec la notion d'indexation.

Dans un exemple cité plus haut, il aurait été commode d'appliquer un index avec la fonction `to_char`, cependant cette dernière n'est pas notée `IMMUTABLE` dans sa définition. Au moment de la création d'un tel index, PostgreSQL renvoie l'erreur suivante :

```
CREATE INDEX ON ma_table ( to_char(ma_date,'YYYY') );

ERROR:  functions in index expression must be marked IMMUTABLE
```

Après la création de l'index fonctionnel, un `ANALYZE nom_table` est nécessaire : en effet, l'optimiseur ne peut utiliser les statistiques déjà connues sur les valeurs dans la table pour un index fonctionnel. Il faut donc indexer les valeurs calculées. Ces statistiques seront visibles dans la vue système `pg_stats` avec comme `tablename` le nom de l'index (et non celui de la table).

ATTENTION : la fonction ne doit jamais tomber en erreur ! Il ne faut pas tester que les données en place mais toutes celles susceptibles de se trouver dans la champ concerné. Sinon, des `ANALYZE` ou `VACUUM` pourraient échouer, avec de gros problèmes sur le long terme.

ATTENTION : si le contenu de la fonction est modifié avec `CREATE OR REPLACE FUNCTION`, il faudra impérativement réindexer. Sinon, les résultats des requêtes différeront selon qu'elles utiliseront ou non l'index !

1.5.4 INDEX COUVRANTS

```
CREATE UNIQUE INDEX clients_idx1 ON clients (id_client) INCLUDE (nom_client) ;
```

- Répondent à la clause `WHERE`
- **ET** contiennent toutes les colonnes demandées par la requête :

```
SELECT id_client,nom_client FROM clients WHERE id_client > 100 ;
```

- Limite les visites à la table

Un parcours d'index classique (*Index Scan*) est en fait un aller/retour entre l'index et la table : on va chercher un enregistrement dans l'index, qui nous donne son adresse dans la table, on accède à cet enregistrement dans la table, puis on passe à l'entrée d'index suivante. Le coût en entrées-sorties peut être énorme : les données de la table sont habituellement éparpillées dans tous les blocs.

Un index couvrant (*covering index*) évite cela en plaçant dans l'index non seulement les champs servant de critères de recherche, mais aussi les champs résultats.

Un index couvrant peut ainsi permettre un *Index Only Scan* car il n'a plus besoin d'interroger la table. Pour pouvoir en bénéficier, il faut que toutes les colonnes retournées par la requête soient présentes dans l'index. De plus, les enregistrements cherchés étant contigus dans l'index (puisqu'il est trié), le nombre d'accès disque est bien plus faible, ce qui peut apporter des gains de performances énormes en sélection. Il est tout à fait possible d'obtenir dans des cas extrêmes des gains de l'ordre d'un facteur 10 000.

Les index couvrants peuvent être explicitement déclarés à partir de la version 11 avec la clause `INCLUDE` :

```
CREATE TABLE t (id int NOT NULL, valeur int) ;

INSERT INTO t SELECT i, i*50 FROM generate_series(1,1000000) i;

CREATE UNIQUE INDEX t_pk ON t (id) INCLUDE (valeur) ;

VACUUM t ;

EXPLAIN ANALYZE SELECT valeur FROM t WHERE id = 555555 ;

                                QUERY PLAN
-------------------------------------------------------------------------------
 Index Only Scan using t_pk on t  (cost=0.42..1.44 rows=1 width=4)
                                  (actual time=0.034..0.035 rows=1 loops=1)
   Index Cond: (id = 555555)
   Heap Fetches: 0
 Planning Time: 0.084 ms
 Execution Time: 0.065 ms
```

Dans cet exemple, il n'y a pas eu d'accès à la table. L'index est unique mais contient aussi la colonne `valeur`.

Noter le `VACUUM`, nécessaire pour garantir que la *visibility map* de la table est à jour et permet ainsi un *Index Only Scan* sans aucun accès à la table (clause *Heap Fetches* à 0).

Dans les versions antérieures à la 11, le principe reste valable : il suffit de déclarer les colonnes dans des index (`CREATE INDEX t_idx ON t (id, valeur)`). La clause `INCLUDE` a l'avantage de pouvoir se greffer sur des index uniques ou de clés et ainsi d'économiser des créations d'index, ainsi que d'éviter le tri des champs dans la clause `INCLUDE`.

1.5.5 INDEX COUVRANTS : INCONVÉNIENTS ET COMPATIBILITÉ

- Inconvénients :
  - index plus gros
  - limite d'enregistrement (2,6 ko)
  - pas de déduplication
- Compatibilité : B-tree (v11), GiST (v12), SP-GiST (v14), principe valable avant v11

Il faut garder à l'esprit que l'ajout de colonnes à un index augmente sa taille. Cela peut avoir un impact sur les performances des requêtes qui n'utilisent pas la colonne qui a été ajoutée. Il faut également être vigilant à ce que la taille des enregistrements avec les colonnes incluses ne dépassent pas 2,6 ko. Au-delà de cette valeur, les insertions ou mises à jour échouent. Enfin, la déduplication (apparue en version 13) n'est pas active sur les index couvrants.

Les méthodes d'accès aux index doivent inclure le support de cette fonctionnalité. C'est le cas pour le B-tree en version 11, pour le GiST en version 12 et pour le SP-GiST en version 14.

1.5.6 CLASSES D'OPÉRATEURS

- Un index utilise des opérateurs de comparaison
- Texte : différentes collations = différents tris
  - `WHERE col_varchar LIKE 'chaine%'` : quel tri ?
  - Solution : opérateur `varchar_pattern_ops` :

```
CREATE INDEX idx1
ON ma_table (col_varchar varchar_pattern_ops)
```

- Autres opérateurs pour d'autres types d'index

Il est tout à fait possible d'utiliser un jeu « alternatif » d'opérateurs de comparaison pour l'indexation, dans des cas particuliers.

Le cas d'utilisation le plus fréquent d'utilisation dans PostgreSQL est la comparaison de chaîne `LIKE 'chaine%'`. L'indexation texte « classique » utilise la collation par défaut de la base (en France, généralement `fr_FR.UTF-8` ou `en_US.UTF-8`) ou la collation de la colonne de la table si elle diffère. Cette collation contient des notions de tri. Les règles sont différentes pour chaque collation. Les nouvelles collations sont à déclarer séparément dans chaque base.

Par exemple, le **ß** allemand se place entre **ss** et **t** (et ce, même en français). En danois, le tri est très particulier car le **å** et le **aa** apparaissent après le **z**.

```
-- Cette collation doit exister sur le système
CREATE COLLATION IF NOT EXISTS "da_DK" (locale='da_DK.utf8');

WITH ls(x) AS (VALUES ('aa'),('å'),('t'),('s'),('ss'),('ß'), ('zz') )
SELECT * FROM ls ORDER BY x COLLATE "da_DK";
```

```
 x
----
 s
 ss
 ß
 t
 zz
 å
 aa
```

Il faut être conscient que cela a une influence sur le résultat d'un filtrage :

```
WITH ls(x) AS (VALUES ('aa'),('å'),('t'),('s'),('ss'),('ß'), ('zz') )
SELECT * FROM ls
WHERE x > 'z' COLLATE "da_DK" ;
```

```
 x
----
 aa
 å
 zz
```

Il serait donc très complexe de réécrire le `LIKE` en un `BETWEEN`, comme le font habituellement tous les SGBD : `col_texte LIKE 'toto%'` peut être réécrit comme `coltexte >= 'toto' and coltexte < 'totp'` en ASCII, mais la réécriture est bien plus complexe en tri linguistique sur Unicode par exemple. Même si l'index est dans la bonne collation, il n'est pas facilement utilisable :

```
CREATE INDEX ON textes (livre) ;
EXPLAIN SELECT * FROM textes WHERE livre LIKE 'Les misérables%';

                                QUERY PLAN
---------------------------------------------------------------------------
 Gather  (cost=1000.00..525328.76 rows=75173 width=123)
   Workers Planned: 2
   ->  Parallel Seq Scan on textes  (cost=0.00..516811.46 rows=31322 width=123)
         Filter: (livre ~~ 'Les misérables%'::text)
```

La classe d'opérateurs `varchar_pattern_ops` sert à forcer ce comportement : l'index est construit sur la comparaison brute des valeurs octales de tous les caractères qu'elle contient.

```
CREATE INDEX ON ma_table (col_varchar varchar_pattern_ops)
EXPLAIN SELECT * FROM textes WHERE livre LIKE 'Les misérables%';

                                     QUERY PLAN
-------------------------------------------------------------------------------------------
 Index Scan using textes_livre_idx1 on textes  (cost=0.69..70406.87 rows=75173 width=123)
   Index Cond: ((livre ~>=~ 'Les misérables'::text) AND (livre ~<~ 'Les misérablet'::text))
   Filter: (livre ~~ 'Les misérables%'::text)
```

Il devient alors trivial pour l'optimiseur de faire la réécriture. Cela convient pour un `LIKE 'toto%'`, car le début est fixe, et l'ordre de tri n'influe pas sur le résultat. Noter la clause `Filter` qui filtre en deuxième intention ce qui a pu être trouvé dans l'index.

Il existe quelques autres cas d'utilisation d'`opclass` alternatives, notamment pour utiliser d'autres types d'index que B-tree. Deux exemples :

- indexation d'un JSON (type `jsonb`) par un index GIN :

```
CREATE INDEX ON stock_jsonb USING gin (document_jsonb jsonb_path_ops);
```

- indexation de trigrammes de textes avec le module `pg_trgm` et des index GiST :

```
CREATE INDEX ON livres USING gist (text_data gist_trgm_ops);
```

1.5.7 CONCLUSION

- Responsabilité de l'indexation
- Compréhension des mécanismes
- Différents types d'index, différentes stratégies

L'indexation d'une base de données est souvent un sujet qui est traité trop tard dans le cycle de l'application. Lorsque celle-ci est gérée à l'étape du développement, il est possible de bénéficier de l'expérience et de la connaissance des développeurs. La maîtrise de cette compétence est donc idéalement transverse entre le développement et l'exploitation.

Le fonctionnement d'un index B-tree est somme toute assez simple, mais il est important de bien l'appréhender pour comprendre les enjeux d'une bonne stratégie d'indexation.

PostgreSQL fournit aussi d'autres types d'index moins utilisés, mais très précieux dans certaines situations : BRIN, GIN, GiST, etc.

1.6 QUIZ

https://dali.bo/j4_quiz

1.7 TRAVAUX PRATIQUES

Cette série de question 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` :

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

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

1.7.1 INDEX « SIMPLES »

But : Mettre en avant un cas d'usage d'un index « simple ».

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.

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

Nous souhaitons désormais afficher les résultats à l'utilisateur par ordre de date croissante.

> Réécrire la requête par ordre de date croissante. Afficher de nouveau son plan. Que constatez-vous ?

Maintenant, nous allons essayer d'optimiser ces deux requêtes.

> Créer un index permettant de répondre à ces requêtes.

> Afficher de nouveau le plan des deux requêtes. Que constatez-vous ?

Maintenant, étudions l'impact des index pour une opération de jointure. Le besoin fonctionnel est désormais de lister toutes les commandes associées à un client (admettons, dont le `client_id` vaut 3), avec les informations du client lui-même.

> Écrire la requête listant les commandes pour `client_id = 3`. Afficher son plan. Que constatez-vous ?

> Créer un index pour accélérer cette requête.

> Afficher de nouveau son plan. Que constatez-vous ?

1.7.2 SÉLECTIVITÉ

But : Comprendre la sélectivité des index.

> Écrivez 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').

> Ajoutez un index sur la colonne `type_client`, et rejouez les requêtes précédentes.

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

1.7.3 INDEX PARTIELS

But : Mettre en avant un cas d'usage d'un index partiel.

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 à l'état suivant :

- En dépôt depuis plus de 12 h, mais non expédié.
- Expédié depuis plus de 3 jours, mais non réceptionné.

> Écrire les requêtes correspondant à ce besoin fonctionnel.

> Quel index peut-on créer pour optimiser celles-ci ?

> Affichez les plans d'exécution. Que constate-t-on ?

1.7.4 INDEX FONCTIONNELS

But : Mettre en avant un cas d'usage d'un index fonctionnel.

Pour répondre aux exigences de stockage, l'application a besoin de pouvoir trouver rapidement les produits dont le volume est compris entre certaines bornes (nous négligeons ici le facteur de forme, qui est problématique dans le cadre d'un véritable stockage en entrepôt !).

> Écrivez une requête permettant de renvoyer l'ensemble des produits dont le volume ne dépasse pas 1 L (les unités sont en mm, 1 L = 1 000 000 mm3).

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

1.7.5 CAS D'INDEX NON UTILISÉS

But : Mettre en avant des cas d'index inutilisés.

Un développeur cherche à récupérer les commandes dont le numéro d'expédition est 190774 avec cette requête :

```
select * from lignes_commandes WHERE numero_lot_expedition = '190774'::numeric;
```

> Afficher le plan de la requête.

> Créer un index pour améliorer son exécution.

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

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

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

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

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

1.8 TRAVAUX PRATIQUES (SOLUTIONS)

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

```
SET search_path = magasin;
```

1.8.1 INDEX « SIMPLES »

> 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 constatez-vous ?

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 constatez-vous ?

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 constatez-vous ?

```
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 listant les commandes pour `client_id = 3`. Afficher son plan. Que constatez-vous ?

Pour récupérer toutes les commandes associées à un client :

```
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=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 constatez-vous ?

```
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.

1.8.2 SELECTIVITÉ

> Écrivez 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 :

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

> Ajoutez un index sur la colonne `type_client`, et rejouez les requêtes précédentes.

Pour créer l'index :

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

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

Les plans d'éxécution :

```
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

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).

1.8.3 INDEX PARTIELS

> Écrire les requêtes correspondant à ce besoin fonctionnel.

Les requêtes correspondantes sont les suivantes :

```
SELECT * FROM lots
    WHERE date_expedition IS NULL
    AND date_depot < now() - '12h'::interval ;

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

> Quel index peut-on créer pour optimiser celles-ci ?

On peut donc optimiser ces requêtes à l'aide des index partiels suivants :

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

> Affichez les plans d'exécution. Que constate-t-on ?

Si l'on regarde les plans d'exécution de ces requêtes avec les nouveaux index, on voit qu'ils sont utilisés :

```
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=0.006..0.006 rows=0 loops=1)
   Index Cond: (date_expedition < (now() - '3 days'::interval))
 Planning time: 0.078 ms
 Execution time: 0.036 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.

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, l'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.

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

1.8.4 INDEX FONCTIONNELS

> Écrivez une requête permettant de renvoyer l'ensemble des produits dont le volume ne dépasse pas 1 L (les unités sont en mm, 1 L = 1 000 000 mm3).

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 ? (Indexer le résultat d'une nouvelle fonction est possible, mais pas obligatoire.)

On peut aussi tout simplement 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. Il faut que cette une fonction soit `IMMUTABLE` :

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

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.

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;
```

1.8.5 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.

> Écrivez 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
```

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

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

> Pourquoi celui-ci n'est-il pas utilisé ? (Conseil : regardez 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.

> Faites 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.

NOTES

NOTES

NOTES

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.