Performance (logique) en PostgreSQL

BDAV – Bases de Données AVancées

M1 MIAGE UNC 2023-2024

- Performance (logique) en PostgreSQL
  - Introduction
    - Références
  - Les algorithmes de jointure
    - Exercice sur les algorithmes de jointure
  - Les plans d'exécution
    - Démonstration
    - Exercice sur les plans
  - Les structures d'index
    - Recherche dans un *B-Tree*
    - Plan d'exécution de requête
    - Les jointures en présence d'index
    - Le poid des index
    - Les index avancés
  - Conclusion

1 Introduction

On va s'intéresser à *la performance de l'évaluation des requêtes* ***`SELECT`***. On va étudier les principaux algorithmes de jointures, à la notion d'index et comment les utiliser. L'évaluation expérimentale des performances d'une requête, à la différence de l'évaluation théorique de sa complexité, est sensible :

- au matériel utilisé (processeur, RAM, disque dur);
- au système d'exploitation et la version utilisée;
- à la configuration du système d'exploitation (limites nombre de fichiers ou processus ouverts, taille des pages mémoire et *big pages*);
- au SGBD et la version utilisée;
- à la configuration du SGBD (mémoire *shared buffers* réservée, configuration des coûts de l'optimiseur, activation du parallélisme);
- à la charge courante du système;
- à la requête, un `LIMIT` ou un `ORDER BY` **change** la requête et son plan;
- aux instances, dont leurs tailles et les distributions des valeurs.

Autrement dit, il est très difficile d'estimer la performance d'un système autrement qu'en évaluant le système lui-même dans ses conditions opérationnelles (sur la *production*).

Ici, on restera essentiellement au niveau **logique**, sans *tuning* de PostgreSQL ou de l'OS, c'est-à-dire on se limite à ce qu'on peut faire en SQL, dans l'espace utilisateur (sans accès *postgres* ou *root*).

1.1 Références

- Documentation PostgreSQL de référence :
  - Le chapitre 11 *Indexes*;
  - La commande `CREATE INDEX`;
  - Le chapitre 51.1 *The Path of a Query*.
- Recommandées :
  - La section 2.7 de ~~*Not*~~ *Only SQL* pour une introduction au problème de la recherche d'un élément dans une collection;
  - *Demystifying JOIN Algorithms* http://blog.felipe.rs/2019/01/29/demystifying-join-algorithms/ par Felipe Oliveira Carvalho, dont je me suis inspiré;
  - *Join strategies and performance in PostgreSQL* sur le blog de CyberTec.
- Pour aller plus loin :
  - Le site *Use The Index Luke* de Markus Winand, *un incontournable*;
  - *Indexes in PostgreSQL — 4 (Btree)* sur le blog de Postgres Pro et le reste de l'excellente série *Indexes in PostgreSQL*;
  - Bruce Momjian sur la performance, dont notamment *Postgres Performance Tuning*;
  - Le chapitre 3 *Query Processing* de Hironobu Suzuki (InterDB), la section 3.5 sur les algortihmes de jointures (très technique);
  - Le `README` du code source des *B-tree* de PostgreSQL.
- Litérature scientifique
  - 1981, *Efficient Locking for Concurrent Operations on B-Trees* https://www.csd.uoc.gr/~hy460/pdf/p650-lehman.pdf
  - 1970, *Organization and maintenance of large ordered indexes* https://infolab.usc.edu/csci585/Spring2010/den_ar/indexing.pdf

2 Les algorithmes de jointure

On va d'abord s'intéresser **aux algorithmes de jointures**, c'est-à-dire aux algorithmes *choisis* puis exécutés par les moteurs des SGBDs quand ils doivent calculer des jointures entre deux tables (ou plus), comme dans l'exemple simple suivant :

```sql
CREATE TABLE table_1(
    id_1 INTEGER,
    val INTEGER
);

CREATE TABLE table_2(
    id_2 INTEGER,
    val INTEGER

);

SELECT *
FROM table_1 JOIN table_2 ON table_1.val = table_2.val;
```

On fournit le programme join_algorithms.py qui contient l'implémentation directe de trois algorithmes de jointure classiques que l'on explique à l'aide de l'implantation Python (voir les commentaires) :

- *nested loop*
  - l'algorithme naïf avec une double boucle
- *sort-merge join*
  - basé sur le principe du *merge* de l'algorithme de tri *merge sort*

- *hash join*
  - basé sur une indexation de la première table

On considère ici d'abord l'algorithmique des jointures :

- *en mémoire* supposée non limitée, sans considérer les accès disques
- dans un langage interprété peu performant.

Les implantation PostgreSQL sont faites en C et utilisent des variantes qui travaillent par blocs pour éviter de tout charger en mémoire. Ici ce qui nous intéresse ce sont donc les ordres de grandeurs *entre les algorithmes* et les choix des instances. Ci-dessous la documentation de l'outil join_algorithms.py pour cette étude, voir `python join_algorithms.py --demo` pour un jeu d'essai:

```
python .\join_algorithms.py --help
usage: python join_algorithms.py [-h] [--verbose] [--demo] [--size-1 SIZE_1] [--size-2 SIZE_2] [--vals-

Comparaison des performances des algorithmes de jointure (nested loop, sort-merge et hash join).

options:
  -h, --help            show this help message and exit
  --verbose, -v         niveau de verbosité, -v pour INFO, -vv pour DEBUG. WARNING par défaut (default:
  --demo, -d            affiche une démonstration au lieu du bench (default: False)
  --size-1 SIZE_1, -s1 SIZE_1
                        nombre de tuples de la première table (default: 1000)
  --size-2 SIZE_2, -s2 SIZE_2
                        nombre de tuples de la deuxième table (default: 1000)
  --vals-1 VALS_1, -v1 VALS_1
                        nombre de valeurs sur l'attribut de jointure de la première table (default: 100)
  --vals-2 VALS_2, -v2 VALS_2
                        nombre de valeurs sur l'attribut de jointure de la deuxième table (default: 100)
  --repeat REPEAT, -r REPEAT
                        nombre de répétitions (default: 20)
```

Avec les paramètres par défaut et l'option `--verbose`, voici un exemple d'exécution. Les tables sont générées au hasard et les temps dépendent de la machine sur laquelle on exécute (pour information, AMD Ryzen 7 3700X 8-Core 3.6 GHz, 16.0 GB 3.4 GHz, SSD NVMe) :

```
INFO:JOINS:building 1000 tuples with 100 distinct values for table #1
INFO:JOINS:building 1000 tuples with 100 distinct values for table #2
INFO:JOINS:running each algorithm 20 times
INFO:JOINS:result contains 10007 tuples
nested loop :  35.91 ms / loop (total 0.72 s)
hash        :   0.89 ms / loop (total 0.02 s)
sort        :   0.19 ms / loop (total 0.00 s)
merge       :   2.45 ms / loop (total 0.05 s)
```

2.1 Exercice sur les algorithmes de jointure

- Pour l'algorithme *sort-merge join* on compte séparément le temps pris pour le tris des tables. Pourquoi ?
- L'algorithme *nested loop* a un intérêt si on change la condition de jointure. Lequel ?
- Comment modifier `Sort-Merge` pour calculer une jointure ouverte `LEFT OUTER`, `RIGHT OUTER` ou `FULL OUTER` ?

Jouer avec les paramètres de l'outil pour trouver des cas :

- qui soient *favorables* à `join_merge` et *défavorables* à `join_hash`. *Indice* : remarquer que les rôles de `table1` et `table2` sont asymétriques dans `join_hash`.
- qui soient *favorables* à `join_nested_loop` et *défavorables* aux deux autres. *Indice* faire en sorte que la jointure soit un produit cartésien.

3 Les plans d'exécution

L'étape du *planner*, dit aussi *optimizer*, est celle où PostgreSQL construit *un* plan d'exécution de requête parmi ceux qui permettent de calculer le résultat demandé. Ce plan est ensuite transmis à l'*executor* qui va réellement exécuter les calculs.

L'optimiseur commence par préparer l'arbre de syntaxe de la requête (comme aplatir les expressions booléennes, simplifier les constantes, déplier les CTE et les sous-requêtes) puis il choisit *le moins cher des plans* parmi toutes les combinaisons de choix d'index et d'algorithmes de jointures selon *un système de coûts*. Comme l'espace de recherche croit exponentiellement avec le nombre de jointures mais qu'il faut choisir *rapidement* le plan :

- si le nombre de tables jointes est inférieur au seuil `geqo_threshold` (voir 20.7.3. Genetic Query Optimizer) qui vaut 12 par défaut, la recherche est exhaustive et le plan optimal par rapport au système de coût;
- au delà du seuil, un algorithme génétique est employé avec l'espoir d'avoir un *bon plan*.

L'optimalité est évaluée selon un critère de coût assez complexe intégrant entre autres :

1. l'estimation de la *sélectivité des conditions* sur la base de statistiques sur les tables, ceci permet d'estimer le nombre de tuples accédés. C'est pour ça qu'il est important d'exécuter `VACUUM ANALYZE` avant de faire de l'évaluation empirique des performances;
2. le coût des algorithmes en fonction de la taille estimée des entrées;
3. la présence d'index existant sur les tables visitées;
4. le coût des accès aux disques et autres paramètres système (voir 20.7.2. Planner Cost Constants).

On accède aux plan avec la commande `EXPLAIN` et ses variantes. **Attention** à l'option `ANALYZE` qui exécute *réellement* la requête !

3.1 Démonstration

Quelques exemples de plans sur le jeu de données du début mais cette fois ci en PostgreSQL que l'on peut générer avec `dataset.sql` avec quatre paramètres de réglage. Tout d'abord l'opération `Seq Scan` où on parcourt **tout** (ici avec un filtrage) :

```sql
EXPLAIN SELECT *
FROM table_1
WHERE id_1 BETWEEN 100 AND 200;
-- -----------------------------------------------------------
-- |                        QUERY PLAN                        |
-- |----------------------------------------------------------|
-- | Seq Scan on table_1  (cost=0.00..20.00 rows=117 width=8) |
-- |   Filter: ((id_1 >= 100) AND (id_1 <= 200))              |
-- -----------------------------------------------------------
```

La requête d'exemple qui utilise un `Hash Join` avec `table_1` sur laquelle est construit le *hash index* (première phase) et `table_2` qui est parcourue séquentiellement (deuxième phase) :

```sql
EXPLAIN SELECT *
FROM table_1 JOIN table_2 ON table_1.val = table_2.val;
```

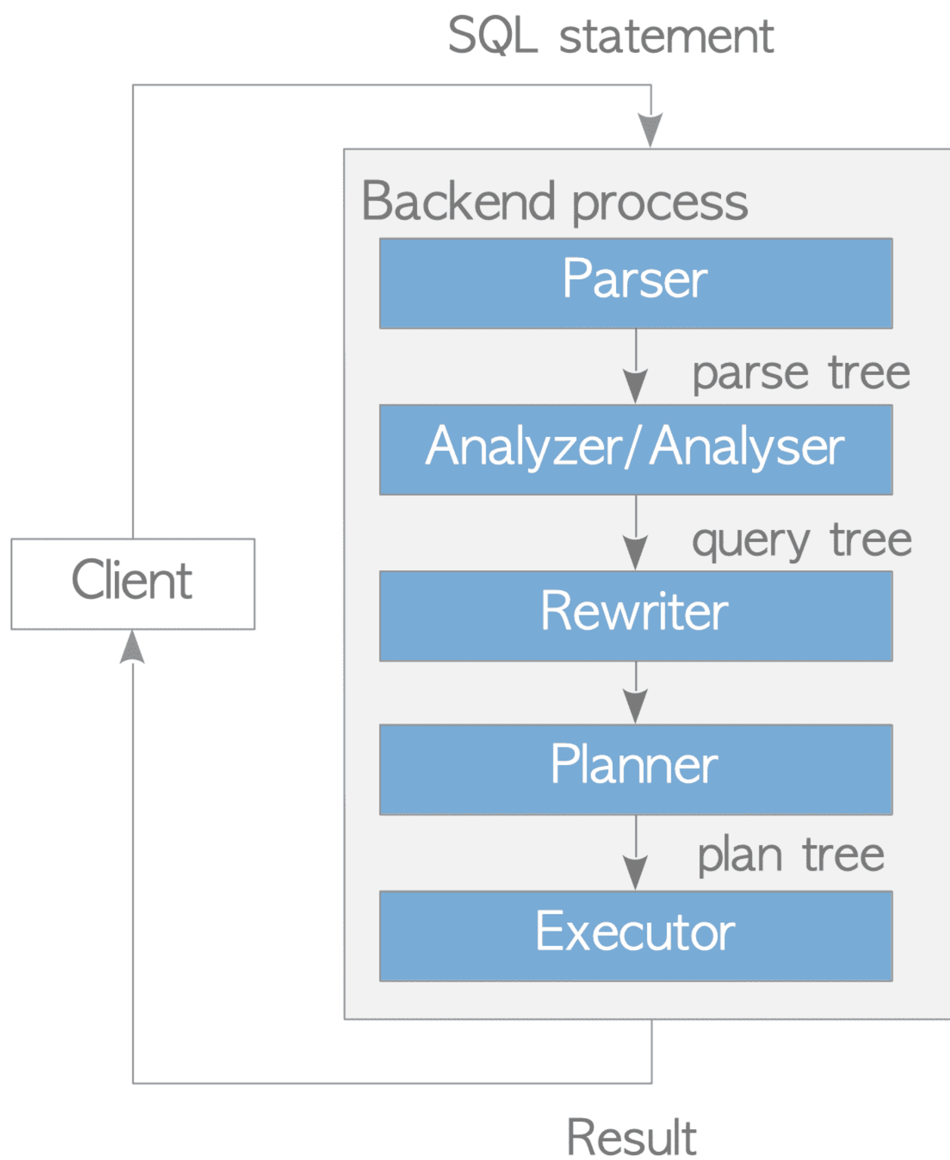

Figure 1: Les étapes de l'évaluation d'une requête source

```
-- -----------------------------------------------------------------------
-- |                             QUERY PLAN                              |
-- |---------------------------------------------------------------------|
-- | Hash Join  (cost=27.50..170.14 rows=10139 width=16)                 |
-- |   Hash Cond: (table_1.val = table_2.val)                            |
-- |   ->  Seq Scan on table_1  (cost=0.00..15.00 rows=1000 width=8)     |
-- |   ->  Hash  (cost=15.00..15.00 rows=1000 width=8)                   |
-- |         ->  Seq Scan on table_2  (cost=0.00..15.00 rows=1000 width=8) |
-- -----------------------------------------------------------------------
```

On remarque que l'estimation de la cardinalité du résultat est *excellente* :

```
SELECT count(*)
FROM table_1 JOIN table_2 ON table_1.val = table_2.val;
-- --------
-- | count |
-- |-------|
-- | 10139 |
-- --------
```

L'opération de jointure (sur test d'égalité) est commutative. En changeant le nombre de tuples de `dataset.sql`, avec par exemple `\set size1 10000` et `\set size2 100`, on va voir que PostgreSQL va choisir d'indexer sur la table qui l'arrange, à savoir celle qui contient *le moins de valeurs* pour que le *hash index* soit le plus petit possible. Ici c'est donc `table_2` qui est indexée, quand bien même l'ordre dans le `FROM` soit toujours le même :

```
EXPLAIN SELECT *
FROM table_1 JOIN table_2 ON table_1.val = table_2.val;
-- -----------------------------------------------------------------------
-- |                             QUERY PLAN                              |
-- |---------------------------------------------------------------------|
-- | Hash Join  (cost=60.85..1482.48 rows=88084 width=16)                |
-- |   Hash Cond: (table_1.val = table_2.val)                            |
-- |   ->  Seq Scan on table_1  (cost=0.00..146.70 rows=10170 width=8)   |
-- |   ->  Hash  (cost=32.60..32.60 rows=2260 width=8)                   |
-- |         ->  Seq Scan on table_2  (cost=0.00..32.60 rows=2260 width=8) |
-- -----------------------------------------------------------------------
```

On peut apprécier le comportement de PostgreSQL en lui interdisant d'utiliser certains algorithmes, voir les options chapitre 20.7.1 *Planner Method Configuration*. On commence par désactiver `Hash Join` avec `set enable_hashjoin=off;` qui était son algorithme préféré. On passe alors en `Merge Join` où on voit les étapes de tri. On voit aussi que le coût estimé est passé d'environ 170 à 287, soit 65% d'augmentation pour ce second choix.

```
set enable_hashjoin=off;
EXPLAIN SELECT *
FROM table_1 JOIN table_2 ON table_1.val = table_2.val;
-- -----------------------------------------------------------------------
-- |                             QUERY PLAN                              |
-- |---------------------------------------------------------------------|
-- | Merge Join  (cost=129.66..286.74 rows=10139 width=16)               |
-- |   Merge Cond: (table_1.val = table_2.val)                           |
-- |   ->  Sort  (cost=64.83..67.33 rows=1000 width=8)                   |
-- |         Sort Key: table_1.val                                       |
```

```
-- |         ->  Seq Scan on table_1  (cost=0.00..15.00 rows=1000 width=8) |
-- |   ->  Sort  (cost=64.83..67.33 rows=1000 width=8)                       |
-- |         Sort Key: table_2.val                                           |
-- |         ->  Seq Scan on table_2  (cost=0.00..15.00 rows=1000 width=8) |
-- -------------------------------------------------------------------------
```

Si maintenant on interdit `Merge Join`, on obtient alors le troisième choix `Nested Loop` qui a bien sûr un coût prohibitif :

```sql
EXPLAIN SELECT *
FROM table_1 JOIN table_2 ON table_1.val = table_2.val;
-- -------------------------------------------------------------------------
-- |                              QUERY PLAN                                |
-- |------------------------------------------------------------------------|
-- | Nested Loop  (cost=0.00..15032.50 rows=10139 width=16)                 |
-- |   Join Filter: (table_1.val = table_2.val)                             |
-- |   ->  Seq Scan on table_1  (cost=0.00..15.00 rows=1000 width=8)        |
-- |   ->  Materialize  (cost=0.00..20.00 rows=1000 width=8)                |
-- |         ->  Seq Scan on table_2  (cost=0.00..15.00 rows=1000 width=8) |
-- -------------------------------------------------------------------------
```

On remarque que si les rapports entre les coûts estimés (170, 287, 15032) diffèrent des rapports des temps d'exécution mesurés en Python (0.86, 2.67, 35.12), mais *les ordres de grandeurs* sont bien comparable sur ce cas :

- `Hash Join` est plus performant que `Merge Join`, estimé de 1.7x (PG) et mesuré à 3.1x (Python), moins qu'un ordre de grandeur,
- `Hash Join` est des dizaines voire des centaines de fois plus rapide que que `Nested Loop`, estimé à 88x et mesuré à 41x

3.2 Exercice sur les plans

On va comparer les plans d'exécution de variantes de requêtes qui calculent toutes la même chose. Ici, on prend les paramètres suivant dans `dataset.sql` :

```
\set size1 1000
\set vals1 1000
\set size2 1000
\set vals2 1000
```

Les requêtes sont dans le fichier `differences.sql`.

- Quels sont les requêtes qui sont indistinguables d'après leurs plans ?
- Classer les requêtes selon leur performance en coût et en temps d'exécution mesuré (`EXPLAIN ANALYZE`).
- Le temps mesuré est-il proportionnel à l'estimation de coût ?
- Donner une requête qui pourrait donner le plan suivant

```
-------------------------------------------------------------------------
|                              QUERY PLAN                                |
|------------------------------------------------------------------------|
| Merge Join  (cost=88.52..96.54 rows=66 width=16)                      |
|   Merge Cond: (s.id_1 = t.id_1)                                        |
|   Join Filter: (t.val > s.val)                                         |
|   ->  Sort  (cost=23.69..23.96 rows=109 width=8)                       |
|         Sort Key: s.id_1                                               |
|         ->  Seq Scan on table_1 s  (cost=0.00..20.00 rows=109 width=8)  |
```

```
|               Filter: ((val >= 100) AND (val <= 200))                    |
|   ->  Sort  (cost=64.83..67.33 rows=1000 width=8)                        |
|         Sort Key: t.id_1                                                 |
|         ->  Seq Scan on table_1 t  (cost=0.00..15.00 rows=1000 width=8) |
--------------------------------------------------------------------------
```

4 Les structures d'index

PostgreSQL propose plusieurs types d'index. D'autres peuvent être ajoutés via des extensions, comme les filtres de Bloom par exemple. La requête suivante liste les sept types d'index de base et les fonctionnalités fournies :

```sql
SELECT a.amname, p.name, pg_indexam_has_property(a.oid, p.name)
FROM pg_am a,
     unnest(array['can_order','can_unique','can_multi_col','can_exclude']) p(name)
ORDER by a.amname;
\crosstabview
-- --------------------------------------------------------------
-- | amname | can_unique | can_exclude | can_order | can_multi_col |
-- |--------|------------|-------------|-----------|---------------|
-- | brin   | f          | f           | f         | t             |
-- | btree  | t          | t           | t         | t             |
-- | gin    | f          | f           | f         | t             |
-- | gist   | f          | t           | f         | t             |
-- | hash   | f          | t           | f         | f             |
-- | heap   | Ø          | Ø           | Ø         | Ø             |
-- | spgist | f          | t           | f         | f             |
-- --------------------------------------------------------------
```

Nous nous intéressons au principal type qui est utilisé par défaut par les contraintes `PRIMARY KEY` et `UNIQUE` et la commande `CREATE INDEX` appelé **B-Tree**, notons qu'il supporte aussi toutes les opérations listées. Notons aussi que 'implémentation PostgreSQL des *B-Tree* ressemble toutefois plus à la structure appelée *B+-tree*.

Les B-Tree sont des arbres *auto-équilibrés* qui stockent les valeurs indexées dans leur nœuds intermédiaires et dont le dernier niveau des feuilles constitue une liste doublement chainée d'éléments qui référence les tuples de la base (`ctid` dans PostgreSQL).

- les niveaux supérieurs permettent une recherche dichotomique efficace sur l'attribut indexé, avec un facteur de branchement élevé (plusieurs centaines de pointeurs dans chaque nœud) : l'arbre est donc peu profond et ses pages tiennent facilement en mémoire vive.
- les feuilles permettent de parcourir les données dans l'ordre (ou l'ordre inverse) de la clef entre deux bornes (e.g., `WHERE key BETWEEN low AND high`)

Jouer avec *Data Structure Visualizations : B-Trees* pour apprécier le principe des *B-Trees* (ici, la version "classique", sans la liste chainée des feuilles).

4.1 Recherche dans un *B-Tree*

La recherche se fait selon le principe de la dichotomie dans les arbres binaires généralisé à un facteur de branchement supérieur à 2, comme 3 ou 4 dans l'illustration suivante, mais souvent plusieurs centaines en réalité :

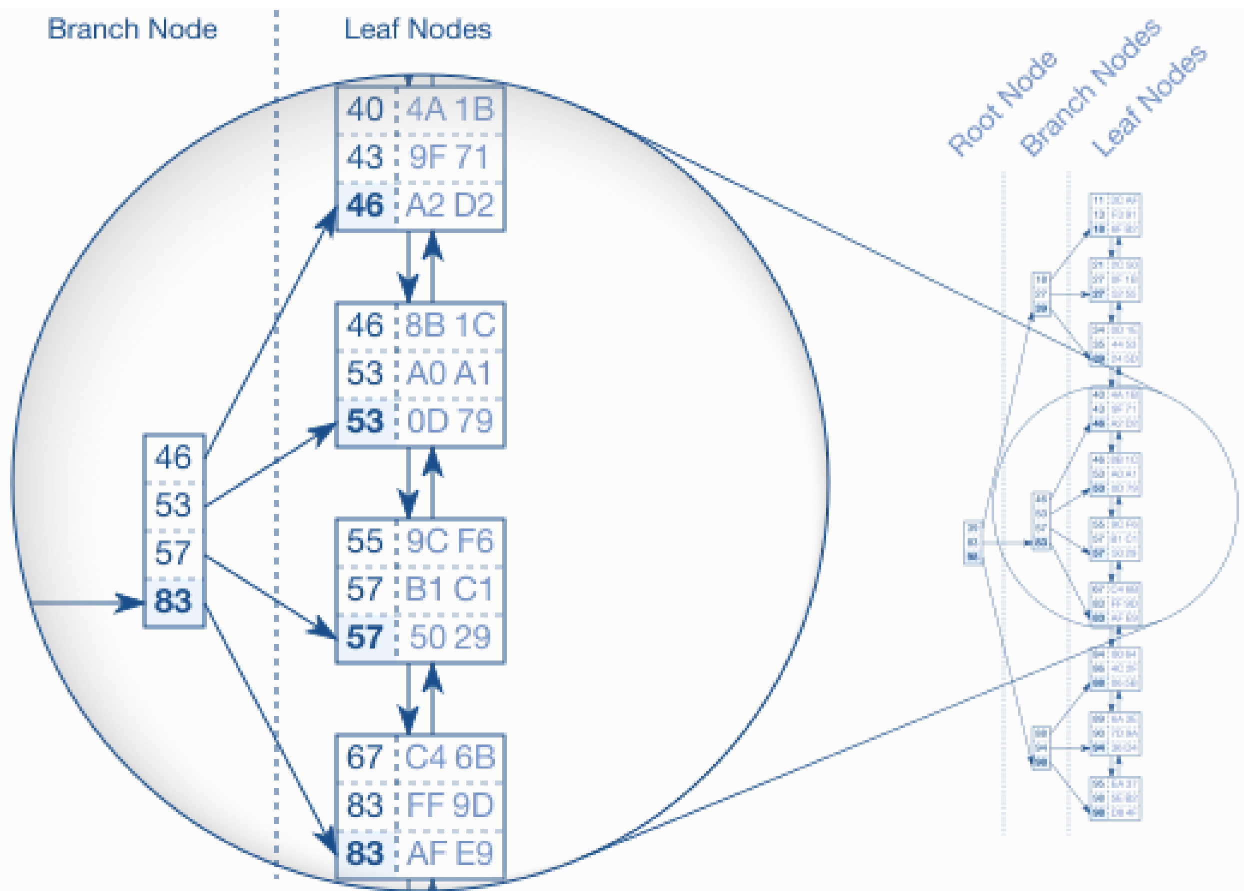

Figure 2: Vue générale d'un *B-Tree*, Source

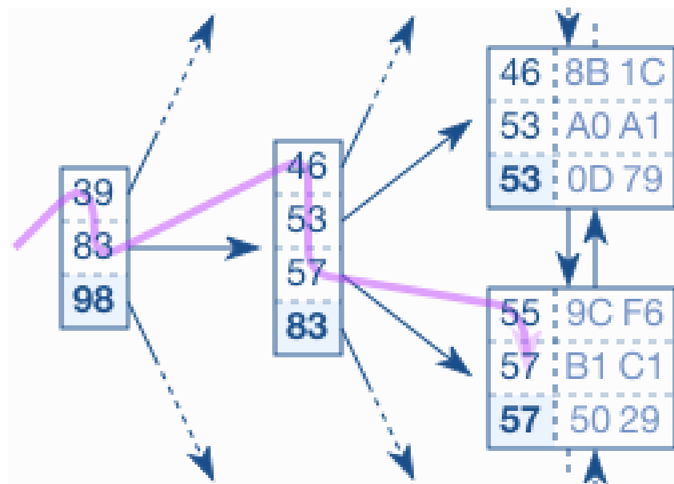

Figure 3: Recherhe dans un *B-Tree*, Source

Sur la table `emp` utilisée précédemment, avec 100 000 employés, on stocke en moyenne 367 feuilles de 16 octets par page (on laisse environ 10% d'espace libre pour éviter de réorganiser les pages). On va voir ceci avec l'outil `pageinspect` qu'il faut installer et utiliser avec l'utilisateur `postgres`.

```
bdav=# SELECT * FROM bt_metap('emp_pkey');
 magic  | version | root | level | fastroot | fastlevel | oldest_xact | last_cleanup_num_tuples | allequ
--------+---------+------+-------+----------+-----------+-------------+-------------------------+-------
 340322 |       4 |    3 |     1 |        3 |         1 |           0 |                      -1 | t

bdav=# SELECT * FROM bt_page_stats('emp_pkey', 3);
 blkno | type | live_items | dead_items | avg_item_size | page_size | free_size | btpo_prev | btpo_next
-------+------+------------+------------+---------------+-----------+-----------+-----------+----------
     3 | r    |        274 |          0 |            15 |      8192 |      2676 |         0 |         0

bdav=# SELECT * FROM bt_page_stats('emp_pkey', 275);
 blkno | type | live_items | dead_items | avg_item_size | page_size | free_size | btpo_prev | btpo_next
-------+------+------------+------------+---------------+-----------+-----------+-----------+----------
   275 | l    |         82 |          0 |            16 |      8192 |      6508 |       274 |         0
```

On va montrer le principe général sur l'exemple où on recherche l'employé numéro 666, il existe (sur cette instance-ci générée aléatoirement):

```
SELECT * FROM emp WHERE empno = 666;
--  depname | empno | salary
-- ---------+-------+--------
--  dep120  |   666 |   5091
```

On va chercher `hex(666) = 0x029a` dans l'index (note comme PostgreSQL adopte la convention *little endian*, on va voir `9a 02` dans le binaire). Les pages stockent les nœuds dans l'ordre consécutif de l'attribut indexé (c'est le principe !). On lit la première page qui est la racine : elle va indexer les 274 pages qui contiennent les feuilles.

```
bdav=# SELECT count(*) FROM bt_page_items('emp_pkey', 3);
 count
-------
   274


bdav=# SELECT * FROM bt_page_items('emp_pkey', 3);
 itemoffset |  ctid   | itemlen | nulls | vars |          data           | dead | htid | tids
------------+---------+---------+-------+------+-------------------------+------+------+------
          1 | (1,0)   |       8 | f     | f    |                         |      |      |
          2 | (2,1)   |      16 | f     | f    | 6f 01 00 00 00 00 00 00 |      |      |
          3 | (4,1)   |      16 | f     | f    | dd 02 00 00 00 00 00 00 |      |      |
          4 | (5,1)   |      16 | f     | f    | 4b 04 00 00 00 00 00 00 |      |      |
          ...
        273 | (274,1) |      16 | f     | f    | e1 84 01 00 00 00 00 00 |      |      |
        274 | (275,1) |      16 | f     | f    | 4f 86 01 00 00 00 00 00 |      |      |
```

On cherche séquentiellement la valeur 666 dans la liste des items de la page 3. Elle n'y est pas mais on sait qu'elle est dans la page `2` qui contient les feuilles entre `0x0016f` (367 en base 10) et `0x02dd` (733 en base 10). On va donc accéder à la page `2` et y chercher `0x029a`. On la trouve et on voit que cette valeur de `empno` est celle du tuple ayant pour `ctid` le couple `(4,38)`.

```
bdav=# SELECT * FROM bt_page_items('emp_pkey', 2);
```

```
 itemoffset |  ctid    | itemlen | nulls | vars |          data           | dead |  htid    | tids
------------+---------+---------+-------+------+-------------------------+------+---------+------
          1 | (4,1)   |      16 | f     | f    | dd 02 00 00 00 00 00 00 |      |         |
          2 | (2,53)  |      16 | f     | f    | 6f 01 00 00 00 00 00 00 | f    | (2,53)  |
          ...
        300 | (4,37)  |      16 | f     | f    | 99 02 00 00 00 00 00 00 | f    | (4,37)  |
        301 | (4,38)  |      16 | f     | f    | 9a 02 00 00 00 00 00 00 | f    | (4,38)  |
        302 | (4,39)  |      16 | f     | f    | 9b 02 00 00 00 00 00 00 | f    | (4,39)  |
         ...
```

On sait donc qu'on peut chercher toutes les informations de l'employé au 38 tuple de la page 4 de la relation `emp`.

```
bdav=# SELECT * FROM heap_page_items(get_raw_page('emp', 4));
```

```
 lp   | lp_off | lp_flags | lp_len | t_xmin | t_xmax | t_field3 | t_ctid   | t_infomask2 | t_infomask | t
-----+--------+----------+--------+--------+--------+----------+---------+-------------+------------+---
   1 |   8144 |        1 |     44 |   3835 |      0 |        0 | (4,1)    |           3 |       2306 |
   2 |   8096 |        1 |     44 |   3835 |      0 |        0 | (4,2)    |           3 |       2306 |
   ...
  37 |   6416 |        1 |     44 |   3835 |      0 |        0 | (4,37)   |           3 |       2306 |
  38 |   6368 |        1 |     44 |   3835 |      0 |        0 | (4,38)   |           3 |       2306 |
  39 |   6320 |        1 |     44 |   3835 |      0 |        0 | (4,39)   |           3 |       2306 |
```

Les données de l'employé 666 sont (en binaire) `\x0f646570313230009a0200000000000e3130000`. On voit qu'il ou elle a un salaire de `e3130000` en *little endian* via les 4 derniers octets, c'est-à-dire `0x13e3 = 5091`. C'est le bon. Pour le vérifier, on peut demander d'ajouter le `ctid` dans la requête d'origine.

```
bdav=# SELECT ctid, emp.* FROM emp WHERE empno = 666;
  ctid   | depname | empno | salary
--------+---------+-------+--------
 (4,38) | dep120  |   666 |   5091
```

4.2 Plan d'exécution de requête

Le plan de la requête `SELECT emp.* FROM emp WHERE empno = 666;` introduit un nouvel opérateur `Index Scan`. Il s'agit ni plus ni moins que de la procédure de recherche que l'on vient de suivre à la main.

```sql
EXPLAIN SELECT emp.* FROM emp WHERE empno = 666;
--                              QUERY PLAN
-- ---------------------------------------------------------------------
--  Index Scan using emp_pkey on emp  (cost=0.29..8.31 rows=1 width=24)
--    Index Cond: (empno = 666)
```

La table `emp` utilise 637 pages (calculé avec `SELECT (pg_relation_size('emp')/8192);`) et 276 pour son index. Grâce à l'`Index Scan`, on a seulement lu **3 pages** :

1. la racine de l'index;
2. un noeud feuille qui référence `(4,38)`;
3. une seule page sur les 637 de `emp`.

Une amélioration de 2 voire 3 ordres de grandeurs. On peut vérifier ce gain en supprimant l'index : le coût passe de 8.31 à 1887 (un facteur 227) :

```sql
ALTER TABLE emp DROP CONSTRAINT emp_pkey;
-- ALTER TABLE
```

```sql
EXPLAIN ANALYZE SELECT emp.* FROM emp WHERE empno = 666;
--                                             QUERY PLAN
-- ------------------------------------------------------------------------------------------------
--  Seq Scan on emp  (cost=0.00..1887.00 rows=1 width=18) (actual time=0.047..5.842 rows=1 loops=1)
--    Filter: (empno = 666)
--    Rows Removed by Filter: 99999
--  Planning Time: 0.056 ms
--  Execution Time: 5.853 ms
ALTER TABLE emp ADD PRIMARY KEY (empno);
```

On peut aller plus loin : si on ne recherche qu'une information indexée, sans accéder aux autres attributs, alors, il n'y a pas besoin d'accéder aux données de la table : *l'index suffit*. C'est le `Index Only Scan` : l'opération que l'on cherche souvent à obtenir car la plus performante.

```sql
EXPLAIN ANALYZE SELECT empno FROM emp WHERE empno BETWEEN 256 AND 512;
--                                                        QUERY PLAN
-- ---------------------------------------------------------------------------------------------------
--  Index Only Scan using emp_pkey on emp  (cost=0.29..9.45 rows=258 width=8) (actual time=0.007..0.037
--    Index Cond: ((empno >= 256) AND (empno <= 512))
--    Heap Fetches: 0
--  Planning Time: 0.157 ms
--  Execution Time: 0.059 ms
```

4.3 Les jointures en présence d'index

C'est souvent l'opération de jointure qu'on souhaite optimiser en exploitant les index :

- en utilisant un `Index Scan` au lieu d'un `Seq Scan` dans une ou deux tables, on obtient un `Nested Loop` efficace, qui revient en quelques sortes à ne faire que la seconde étape du `Hash Join`, l'index étant déjà existant
- comme un index permet de parcourir les données triées, il permet d'éviter la première étape du `Sort-Merge Join` ou les étapes `Order By` si elles portent sur le même attribut
- pour éviter de lire plusieurs fois la même page, on peut utiliser le `Bitmap Index Scan` : l'index est utilisé pour connaitre la liste des pages dont on a besoin, et une fois qu'on la connait, on accède aux données en revérifiant la condition (car dans une page, tout ne nous intéresse pas).

Par exemple, sans index sur les tables d'exemple du début on a un plan `Nested Loop` de coût 15032 pour la requête suivante (paramètres 1000/1000 et 1000/1000) :

```sql
EXPLAIN SELECT *
FROM table_1 JOIN table_2 ON table_1.val > table_2.val;
-- ----------------------------------------------------------------------
-- |                              QUERY PLAN                            |
-- |--------------------------------------------------------------------|
-- | Nested Loop  (cost=0.00..15032.50 rows=333333 width=16)            |
-- |   Join Filter: (table_1.val > table_2.val)                         |
-- |   ->  Seq Scan on table_1  (cost=0.00..15.00 rows=1000 width=8)    |
-- |   ->  Materialize  (cost=0.00..20.00 rows=1000 width=8)            |
-- |         ->  Seq Scan on table_2  (cost=0.00..15.00 rows=1000 width=8) |
-- ----------------------------------------------------------------------
```

En présence d'un index sur l'un des attributs `val`, l'optimiseur va mettre la table indexée dans la boucle interne du `Nested Loop`. Que ce soit sur `table_1` ou sur `table_2`, le coût descend à 9 487 car on évite la lecture de la moitié des données. Toutefois, à l'exécution cela ne changera pas grand chose *sur ce cas* car on lira vraisemblablement toutes les pages des tables lors du calcul.

```sql
BEGIN;
-- BEGIN
CREATE INDEX ON table_1(val);
-- CREATE INDEX
EXPLAIN SELECT *
FROM table_1 JOIN table_2 ON table_1.val > table_2.val;
-- ---------------------------------------------------------------------------------------
-- |                                     QUERY PLAN                                      |
-- |-------------------------------------------------------------------------------------|
-- | Nested Loop  (cost=0.28..9487.50 rows=333333 width=16)                              |
-- |   ->  Seq Scan on table_2  (cost=0.00..15.00 rows=1000 width=8)                     |
-- |   ->  Index Scan using table_1_val_idx on table_1  (cost=0.28..6.14 rows=333 width=8) |
-- |         Index Cond: (val > table_2.val)                                             |
-- ---------------------------------------------------------------------------------------


ROLLBACK;
-- ROLLBACK
BEGIN;
-- BEGIN
CREATE INDEX ON table_2(val);
-- CREATE INDEX
EXPLAIN SELECT *
FROM table_1 JOIN table_2 ON table_1.val > table_2.val;
-- ---------------------------------------------------------------------------------------
-- |                                     QUERY PLAN                                      |
-- |-------------------------------------------------------------------------------------|
-- | Nested Loop  (cost=0.28..9487.50 rows=333333 width=16)                              |
-- |   ->  Seq Scan on table_1  (cost=0.00..15.00 rows=1000 width=8)                     |
-- |   ->  Index Scan using table_2_val_idx on table_2  (cost=0.28..6.14 rows=333 width=8) |
-- |         Index Cond: (val < table_1.val)                                             |
-- ---------------------------------------------------------------------------------------
ROLLBACK;
-- ROLLBACK
```

4.3.1 Exercices

- A-t'on intérêt à avoir *les deux index* sur `val` pour la requête précédente ?
- On considère maintenant la requête `SELECT * FROM table_1 JOIN table_2 ON table_1.val BETWEEN table_2.val AND table_2.val + 10;`. Quel index vaut-il mieux créer : celui sur `table_1.val` ou celui sur `table_2.val` ?
- Les requêtes de différences ne sont pas impactées par la présence d'index sur `id_1` ou `id_2`. Vérifier sur les variantes 1, 3 et 5. Pourquoi ?

4.4 Le poid des index

Attention à ne pas multiplier les index arbitrairement :

- les index prennent de la place sur le disque et en mémoire. Il peuvent augmenter substantiellement la taille de la base;
- *no free lunch* : si les sélections (lecture) sont considérablement améliorées c'est au prix d'une dégradation des écritures car il faut maintenir l'index à chaque modification. C'est pourquoi pour des insertions de

masse, il est recommandé de les désactiver *puis* de remettre les index, voir 14.4. Populating a Database
- les index n'améliorent les performance que *s'ils sont utilisés* : il doivent être donc judicieusement choisis vis-à-vis d'un problème de performance (*slow queries*).

Sur le surcoût en espace, par exemple, sur la table `emp` avec 100 000 employés générés et 1000 départements, la taille de l'unique index `emp_pkey` crée par la `PRIMARY KEY` représente 43% de la taille de la table.

```
-- voir https://stackoverflow.com/questions/41991380/whats-the-difference-between-pg-table-size-pg-rela

SELECT pg_size_pretty(pg_total_relation_size('emp'));
-- ------------------
-- | pg_size_pretty |
-- |----------------|
-- | 7344 kB        |
-- ------------------

SELECT pg_size_pretty(pg_table_size('emp'));
-- ------------------
-- | pg_size_pretty |
-- |----------------|
-- | 5136 kB        |
-- ------------------

SELECT pg_size_pretty(pg_relation_size('emp'));
-- ------------------
-- | pg_size_pretty |
-- |----------------|
-- | 5096 kB        |
-- ------------------

SELECT pg_size_pretty(pg_indexes_size('emp'));
-- ------------------
-- | pg_size_pretty |
-- |----------------|
-- | 2208 kB        |
-- ------------------
```

Sur le cas `emp`, les index rendent les insertions 80% plus lentes.

```
ALTER TABLE emp DROP CONSTRAINT emp_depname_fkey;
ALTER TABLE emp DROP CONSTRAINT emp_pkey;
TRUNCATE emp;
VACUUM ANALYZE;

INSERT INTO emp (
    SELECT
        'dep' || ((1000 * random())::int),
        i,
        (6000 * random())::int
    FROM
        generate_series(1, 1000000) AS g (i));
-- INSERT 0 1000000
-- Time: 902.071 ms
```

```sql
ALTER TABLE emp ADD PRIMARY KEY (empno);
TRUNCATE emp;
VACUUM ANALYZE;

INSERT INTO emp (
    SELECT
        'dep' || ((1000 * random())::int),
        i,
        (6000 * random())::int
    FROM
        generate_series(1, 1000000) AS g (i));
-- INSERT 0 1000000
-- Time: 1617.218 ms (00:01.617)
```

4.5 Les index avancés

PostgreSQL offre de nombreuses possibilités à la création d'un index, voir la documentation de `CREATE INDEX`

- On peut indexer sur *plusieurs attributs* simultanément, selon l'ordre lexicographique des attributs, on appelle ceci *concatenated index*, ou alternativement *multicolumn*, *composite* ou *combined index*. La terminologie PostgreSQL étant multicolumn
- On peut demander d'indexer sur un attribut mais aussi sur le résultat d'une fonction (index dit *fonctionnel*)
- L'option `INCLUDE` dans la création d'index permet d'ajouter *d'autres attributs* dans l'index et ainsi augmenter les cas où un `Index Scan` suffit. On dira qu'on index est *couvrant* si tous les attributs demandés sont dans l'index et qu'il n'y a plus besoin d'accès aux pages de la table.
- Les *collations*, c'est-à-dire l'ordre des caractères (pour avoir `é` et `è` à côté de `e` et pas à la fin de l'alphabet)
- Les classes d'opérateurs, notamment sur les chaînes de caractères pour utiliser les index dans les condition `LIKE` et les expressions regulières.

4.5.1 Exercices sur les index avancés

On reprend la génération du jeu de données RH. On ajoute un index multi colonnes comme suit.

```sql
CREATE INDEX on emp(empno, depname);
```

- L'index est-il utile pour la requête `SELECT empno FROM emp WHERE depname = 'dep42';` ?
- Même question pour `SELECT empno FROM emp WHERE depname = 'dep42' AND empno BETWEEN 1000 AND 2000;`
- Si on supprime l'index multi colonnes et qu'on le remplace par l'index mono colonne `CREATE INDEX on emp(depname);`, les requêtes précédentes en bénéficieront-elles ?

On considère les deux requêtes suivantes :

```sql
SELECT * FROM dep WHERE lower(depname) BETWEEN 'dep100' AND 'dep200';
SELECT * FROM dep WHERE depname BETWEEN 'dep100' AND 'dep200';
```

- Une seule des deux utilise l'index `dep_pkey`. Pourquoi ?
- Ajouter l'index qui permet d'optimiser la deuxième.

On considère maintenant la requête suivante :

```sql
SELECT *
FROM emp JOIN dep ON emp.depname = dep.depname
WHERE salary / 1000 = 4;
```

- Le plan d'exécution de la requête suivante n'utilise pas les index `emp_pkey` et `dep_pkey`. Pourquoi ?
- L'index `CREATE INDEX ON emp((salary/1000));` est-il pertinent ? Justifier.
- Supprimer la jointure inutile de la requête et comparer les coûts. Le changement est-il substantiel, pourquoi ?

On considère les requêtes suivantes :

```sql
EXPLAIN SELECT emp.* FROM emp ORDER BY empno ASC NULLS LAST;
EXPLAIN SELECT emp.* FROM emp ORDER BY empno DESC NULLS LAST;
EXPLAIN SELECT emp.* FROM emp ORDER BY empno ASC NULLS FIRST;
EXPLAIN SELECT emp.* FROM emp ORDER BY empno DESC NULLS FIRST;
```

- Si on demande seulement `SELECT empno` au lieu de `SELECT emp.*`, quel parcours sera utilisé dans le plan ?
- Seulement une requête sur deux utilise l'index `emp_pkey`. Pourquoi ?
- Il est possible d'optimiser les deux autres requêtes avec un seul index. Lequel et pourquoi ?

5 Conclusion

On résume ici les avantages et inconvénient des trois algorithmes de jointure et leur utilisation des index source :

	Nested Loop Join	Hash Join	Merge Join
Algorithme	Pour chaque tuple de la relation de la boucle externe, parcourir intégralement la relation interne	Construire une table de hashage pour la première relation, puis parcourir la seconde en utilisant la table de hashage pour un accès direct	Trier les deux relations puis les fusionner en les parcourant en parallèle
Index utile	Un index sur les attributs de jointure de la *relation interne*	Aucun	Index sur les attributs de jointure dans *chaque relation*
Bonne stratégie	Si la table externe est petite ou si le résultat est proche en taille du produit cartésien	Si la table de hashage tient en mémoire vive (`work_mem`)	Si les deux relations sont grandes