ASBD : TP4

Auteurs:
CONTRERAS Baptiste p1809436
CLEMENT Florent p1601511

Pourquoi faut-il toujours lancer une commande vacuum analyze après un import de données ? Cette commande va avoir deux actions sur la base de données : dans un premier temps la visibility map va être vidée (les lignes marquées comme masquées vont être physiquement supprimées). Après la partie analyse va permettre de construire des statistiques sur les tables, qui seront utilisées par l'optimiseur de requête. Pour résumer, cette commande va permettre un gain de performance sur les tables fraîchement remplie

Comment peut-on analyser les plans d'exécutions avec PostgreSQL ?
Il est possible d'analyser une requête avec un EXPLAIN. le résultat sera un plan d'exécution qui pourra être visualisé sur des sites web comme https://explain.dalibo.com

Que signifient les différentes données fournies dans le plan d'exécution ?
Parmi les données que l'on retrouve dans le plan d'exécution, on retrouve les opérations qui vont être jouées, le coût, basé sur une estimation obtenue à partir des statistiques sur la table (d'où l'importance de lancer un vacuum analyze pour alimenter cette table). On peut aussi obtenir une estimation du nombre de lignes impactées (ce n'est pas toujours précis mais cela peut donner au moins un ordre de grandeur)

Que pouvez-vous dire sur l'impact des index sur les requêtes d'écriture ? Comment l'expliquez-vous ?

Les index impactent négativement les performances sur les requêtes d'écriture. En effet, il faut recalculer les index après une insertion ou une modification. C'est pour cela que les index doivent être choisis judicieusement.

Remarques

Nous avons fait le choix d'utiliser Docker sur nos machines personnelles, avec l'image postgres officielle : postgres:13

Création du schéma

```
psql -U postgres -c "CREATE SCHEMA petasky;"
cd /data/  && psql -U postgres  < obj.sql
cd /data/ && psql -U postgres  < source.sql
```

Import de données

Après avoir unzip les données dans le répertoir **/data/tmp** :

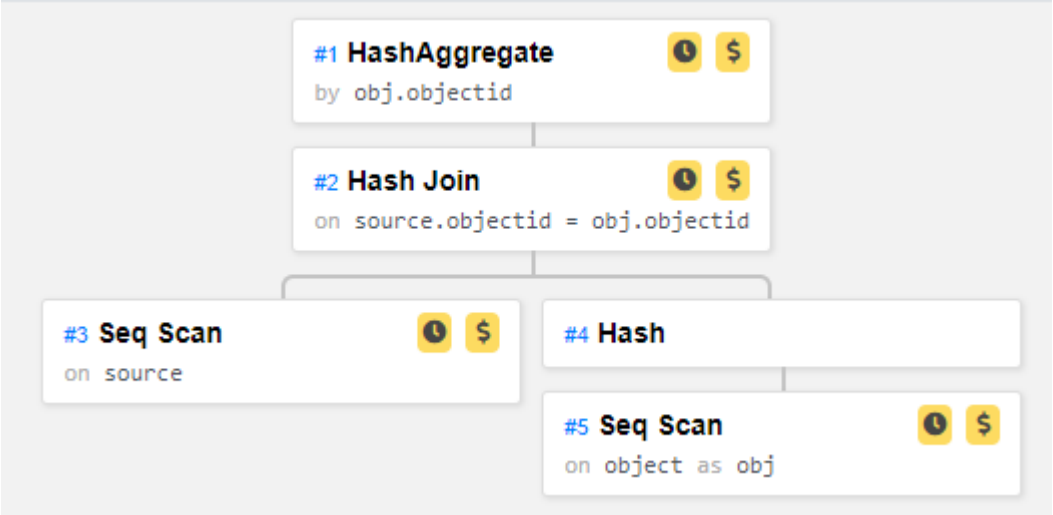
```
for i in $(ls /data/tmp/Object-*)
do
      echo "Traitement de $i"
  psql -U a -c "COPY petasky.Object FROM '${i}' delimiter ',' csv NULL
as 'NULL';"
done

for i in $(ls /data/tmp/Source-*)
do
      echo "Traitement de $i"
  psql -U a -c "COPY petasky.Source FROM '${i}' delimiter ',' csv NULL
as 'NULL';"
done
```

Analyse de requête en lecture

Requête 1

```
SELECT obj.objectId, COUNT(*) FROM petasky.Object as obj JOIN
petasky.Source as source ON obj.objectId = source.objectId GROUP BY
obj.objectId;
```

Avant optimisation



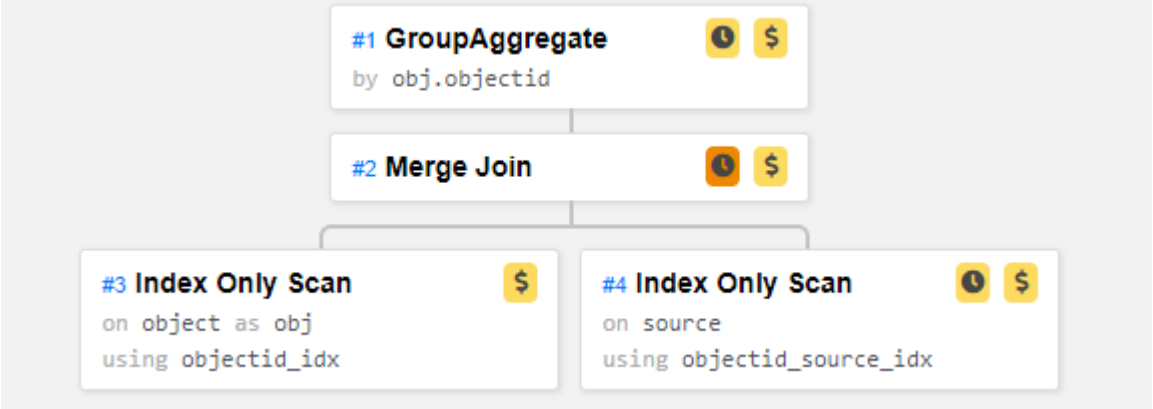

```
HashAggregate  (cost=317875.03..337530.56 rows=829795 width=16) (actual
time=31522.606..31949.565 rows=88740 loops=1)
   Group Key: obj.objectid
   Planned Partitions: 32  Batches: 33  Memory Usage: 4113kB  Disk Usage:
30632kB
   ->  Hash Join  (cost=84609.39..236100.41 rows=1453771 width=8) (actual
time=12538.059..28208.665 rows=1482313 loops=1
)
         Hash Cond: (source.objectid = obj.objectid)
         ->  Seq Scan on source  (cost=0.00..112867.68 rows=1802568 width=8)
(actual time=134.662..6062.189 rows=1800000
 loops=1)
         ->  Hash  (cost=70994.95..70994.95 rows=829795 width=8) (actual
time=12115.459..12115.465 rows=830139 loops=1)
               Buckets: 131072  Batches: 16  Memory Usage: 3053kB
               ->  Seq Scan on object obj  (cost=0.00..70994.95 rows=829795
width=8) (actual time=17.663..10326.971 rows
=830139 loops=1)
```

Temps d'éxecution: 32895.799 ms

Création des index :

```
CREATE UNIQUE INDEX objectId_idx ON petasky.Object(objectId);
CREATE INDEX objectId_source_idx ON petasky.Source(objectId);
```

Après optimisation

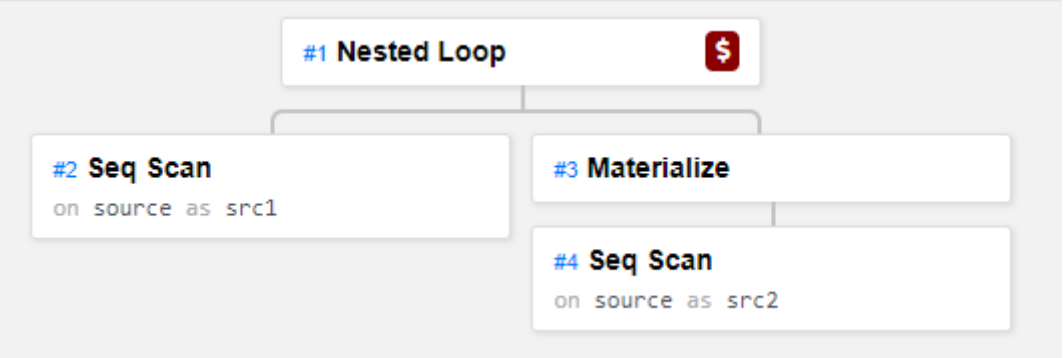



```
 GroupAggregate  (cost=3.21..86391.80 rows=830223 width=16) (actual
time=0.077..12952.667 rows=88740 loops=1)
   Group Key: obj.objectid
   ->  Merge Join  (cost=3.21..70609.28 rows=1496057 width=8) (actual
time=0.041..10163.157 rows=1482313 loops=1)
         Merge Cond: (obj.objectid = source.objectid)
         ->  Index Only Scan using objectid_idx on object obj
(cost=0.42..21569.77 rows=830223 width=8) (actual time=0.
017..1319.707 rows=748136 loops=1)
               Heap Fetches: 0
         ->  Index Only Scan using objectid_source_idx on source
(cost=0.43..34001.49 rows=1799804 width=8) (actual time=0.013..2654.316
rows=1482314 loops=1)
               Heap Fetches: 0
```

Temps d'exécution : 13097.152 ms

Requête 2

```
SELECT src1.sourceId, src2.sourceId FROM petasky.Source as src1 INNER
JOIN petasky.Source as src2 on src1.sourceId <> src2.sourceId AND
abs(src1.ra - src2.ra) <= 0.1;
```




```
 Nested Loop  (cost=0.00..88702832399.95 rows=1079764212871 width=16)
   Join Filter: ((src1.sourceid <> src2.sourceid) AND (abs((src1.ra -
src2.ra)) <= '0.1'::double precision))
   ->  Seq Scan on source src1  (cost=0.00..112840.04 rows=1799804
width=12)
   ->  Materialize  (cost=0.00..130628.06 rows=1799804 width=12)
         ->  Seq Scan on source src2  (cost=0.00..112840.04 rows=1799804
width=12)
```

Optimisation:

```
SELECT src1.sourceId, src2.sourceId FROM petasky.Source as src1 INNER
JOIN petasky.Source as src2 on src1.sourceId > src2.sourceId AND
src1.ra_01 = src2.ra_01;
```

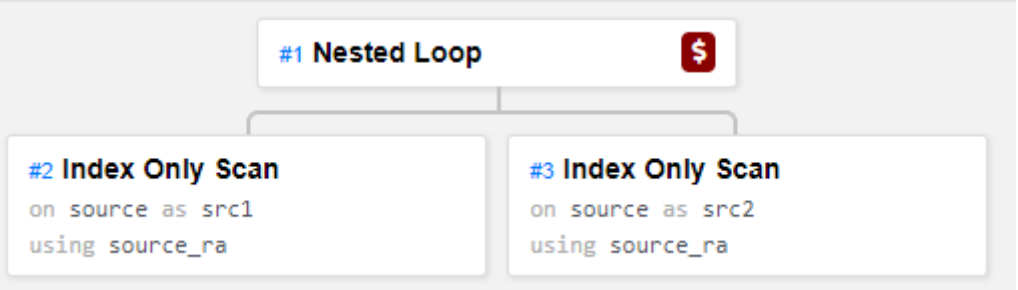
Ajout de colonne

```
ALTER TABLE petasky.Source ADD column ra_01 NUMERIC GENERATED ALWAYS AS
(round(ra::numeric, 1)) stored;
```

Création des index

```
CREATE UNIQUE INDEX sourceId_idx ON petasky.Source(sourceId);
CREATE INDEX source_ra on petasky.Source(ra_01, sourceId);
```

Après optimisation

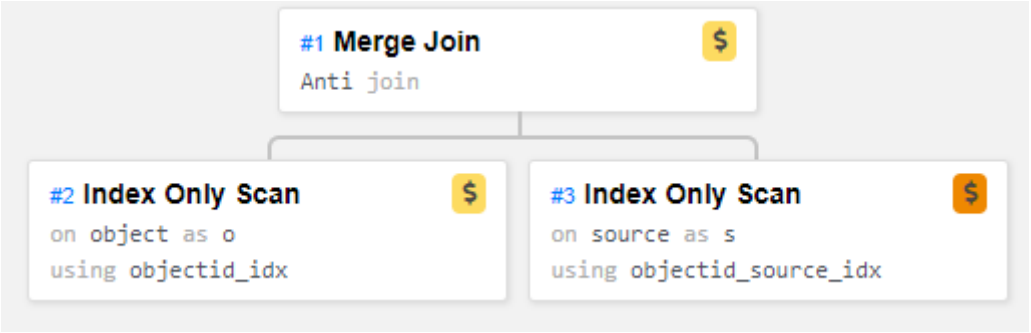



```
 Nested Loop  (cost=0.85..624288808.55 rows=54988853738 width=16)
   ->  Index Only Scan using source_ra on source src1
(cost=0.43..54739.73 rows=1800487 width=14)
   ->  Index Only Scan using source_ra on source src2
(cost=0.43..231.28 rows=11542 width=14)
         Index Cond: ((ra_01 = src1.ra_01) AND (sourceid <
src1.sourceid))
```

Requête 3

```
SELECT o.objectId FROM petasky.Object o LEFT JOIN petasky.Source s ON o.objectId
= s.objectId WHERE s.objectId IS NULL;
```

Les index précédents permettent d'avoir une optimisation complète de cette requête.




```
 Merge Anti Join  (cost=3.65..70301.39 rows=772488 width=8)
   Merge Cond: (o.objectid = s.objectid)
   ->  Index Only Scan using objectid_idx on object o  (cost=0.42..21682.52
rows=830552 width=8)
   ->  Index Only Scan using objectid_source_idx on source s
(cost=0.43..34011.73 rows=1800487 width=8)
```

Temps d'exécution avant index: 58880.666 ms
Temps d'exécution après index: 9041.885ms

Analyse de requête en écriture

La requête SQL devra mettre à jour l'attribut ra de la table source avec la valeur 2*ra²

```
UPDATE petasky.Source SET ra = 2*POWER(ra, 2);
```

Avant mise en place des index

```
Update on source  (cost=0.00..130862.50 rows=1800525 width=732) (actual
time=104539.253..104539.253 rows=0 loops=1)
   ->  Seq Scan on source  (cost=0.00..130862.50 rows=1800525 width=732) (actual
time=195.938..3138.631 rows=1800000 loo
ps=1)
 Planning Time: 0.259 ms
 JIT:
   Functions: 2
   Options: Inlining false, Optimization false, Expressions true, Deforming true
   Timing: Generation 0.656 ms, Inlining 0.000 ms, Optimization 24.661 ms,
Emission 170.509 ms, Total 195.826 ms
Execution Time: 105256.880 ms
```

Après mise à jour des index

```
Update on source  (cost=0.00..133336.80 rows=1800640 width=738) (actual
time=125515.593..125515.594 rows=0 loops=1)
   ->  Seq Scan on source  (cost=0.00..133336.80 rows=1800640 width=738) (actual
time=7.318..2059.651 rows=1800000 loops
=1)
 Planning Time: 0.373 ms
 JIT:
   Functions: 2
   Options: Inlining false, Optimization false, Expressions true, Deforming true
   Timing: Generation 0.663 ms, Inlining 0.000 ms, Optimization 0.658 ms,
Emission 6.476 ms, Total 7.797 ms
 Execution Time: 125535.375 ms
```

Avant index (en ms)	Après index (après ms)
105256.880	125535.375

Nous constatons que cette requête met plus de temps à s'exécuter quand des index sont présents sur la colonne **ra**. Ce comportement était celui qu'on attendait car il faut recalculer les index après une modification sur cette colonne.