10

Planification des requêtes

10.1. Introduction à l'optimiseur de requêtes

L'exécution d'une requête n'est pas une opération simple. Il y a de nombreuses décisions à prendre afin d'obtenir le résultat de la manière la plus efficace possible. Prenons la requête suivante :

```
SELECT * FROM t1 WHERE c1<10 ORDER BY c2
```

Cette requête doit faire un filtre sur les données de la table `t1` pour ne récupérer que les lignes pour lesquelles la colonne `c1` a une valeur inférieure à 10 et trier les lignes restantes suivant la valeur de la colonne `c2`. Il existe plusieurs façons de récupérer les lignes satisfaisant le prédicat `c1<10` tout en triant sur `c2`. Il est possible de lire la table bloc par bloc, de récupérer chaque ligne de chaque bloc et de ne conserver que celles satisfaisant le prédicat. Il est aussi possible de récupérer les lignes intéressantes directement dans un index de cette table et de ne lire que les blocs de la table pointés depuis l'index. Pour le tri, il est possible de le faire à l'exécution en stockant les données intermédiaires du tri soit en mémoire, soit sur disque, mais il est aussi possible d'utiliser un index B-tree pour cela.

Autrement dit, même avec une requête aussi simple, le nombre de façons de l'exécuter est très grand. Le but de l'optimiseur de requêtes est de répertorier ces différentes façons de faire, de calculer leur coût et de choisir la moins coûteuse. La façon d'exécuter une requête est appelée *plan d'exécution*. Une fois que l'optimiseur a trouvé le plan qu'il considère le plus efficace pour exécuter la requête, ce plan est fourni à l'exécuteur, qui se contente de suivre les différentes étapes décrites dans ce plan pour fournir le résultat.

Fonctionnement de l'optimiseur

Le langage SQL est un langage déclaratif. Il permet d'exprimer le résultat attendu, mais pas la façon dont on souhaite l'obtenir. La requête ci-dessus, en langage plus humain, veut dire : *récupère toutes les lignes pour lesquelles la valeur de `c1` est inférieure à 10 et trie-les suivant la valeur de la colonne `c2`*. Elle n'indique pas comment le faire. C'est

le travail de l'optimiseur de savoir comment exécuter une requête le plus rapidement possible tout en renvoyant le résultat attendu.

Seules les requêtes DML (`SELECT`, `INSERT`...) sont traitées par l'optimiseur. En effet, les autres types de requêtes ne peuvent pas s'exécuter de plusieurs façons, et la phase de planification est donc superflue. Ainsi, ces différents types de requêtes ne sont pas gérés par l'optimiseur :

- les requêtes DDL ;
- l'instruction `COPY` (si cette instruction `COPY` intègre un appel à une requête `SELECT`, cette dernière est traitée par l'optimiseur mais pas la requête appelante) ;
- l'instruction `TRUNCATE` ;
- les opérations de maintenance telles que `VACUUM` et `ANALYZE` ;
- les différents ordres SQL ne gérant pas d'accès aux données (`BEGIN`, `LISTEN`, etc.).

Lorsqu'un client soumet une requête à exécuter, l'optimiseur tente de générer tous les plans d'exécution possibles, sauf si cela représente un trop grand nombre de plans d'exécution. Par exemple, lorsque le nombre de jointures augmente fortement, le nombre de plans d'exécution à calculer serait trop important, et seul un sous-ensemble des plans possibles est calculé.

Le planificateur effectue une optimisation par les coûts. À chaque opération de base (récupération d'un bloc en mémoire, application d'un opérateur, etc.) est associé un coût spécifique (et configurable afin de refléter les performances du serveur). Les coûts de ces opérations de base sont ensuite corrélés à un certain nombre de statistiques sur les données (comme le nombre de lignes d'une table, son nombre de blocs, un histogramme de valeurs, etc.). Le coût total d'un plan est donc le cumul des coûts de chacune de ces opérations sur la volumétrie estimée. L'unité étant totalement arbitraire, il n'est pas possible de déduire du coût d'un plan sa durée d'exécution, mais on peut déduire qu'un plan ayant un coût inférieur à un autre plan s'exécutera plus rapidement, pour peu que la configuration et que les statistiques soient correctes.

Afin de gagner en efficacité, l'optimiseur annule la génération des plans en cours de traitement s'il se rend compte que ces derniers sont déjà bien plus coûteux qu'un plan généré précédemment.

Note *› Une fonctionnalité fréquemment demandée est une commande permettant d'avoir la liste des plans considérés par PostgreSQL pour exécuter une requête, ce qui permettrait de mieux se rendre compte de la raison pour laquelle il a choisi ce plan plutôt qu'un autre. Cette fonctionnalité ne peut pas être proposée, étant donné que PostgreSQL ne génère pas forcément l'intégralité des plans, s'il se rend compte que certains sont déjà plus coûteux que d'autres.*

Le plan retenu, qui sera donc exécuté, est celui qui a le coût le moins élevé. Deux types de coûts peuvent être demandés lors de la génération d'un plan. Cela peut être soit le coût de la récupération de la première ligne (dans le cas de l'utilisation de curseurs ou d'une clause `LIMIT`) soit de la récupération de toutes les lignes (dans la majorité des cas).

PostgreSQL ne met pas les plans en cache. Ils sont calculés à chaque exécution de requêtes, sauf dans des cas spécifiques (notamment les requêtes préparées ou les requêtes au sein d'une routine en PL/pgSQL). Dans ce cas, les plans sont conservés en cache uniquement pour la session qui les a préparés. La gestion de ce cache est configurable dès la version 12 avec le paramètre `plan_cache_mode`.

Enfin, la notion de *hints* dont disposent des moteurs comme Oracle et SQL Server est inconnue à PostgreSQL. Les raisons en sont multiples, et un excellent billet de Josh Berkus les explique en détail. Il est donc nativement impossible de forcer l'optimiseur de PostgreSQL à utiliser (ou interdire) un index spécifique ou un plan spécifique. Des extensions existent cependant pour apporter certaines de ces fonctionnalités, utiles dans des cas très précis. Par exemple, l'extension `plantuner` permet de forcer l'utilisation ou non d'index particuliers, ce qui peut servir notamment à vérifier en amont si la suppression d'un index poserait soucis. L'extension `pg_hint_plan` se focalise sur les nœuds du plan d'exécution et permet de forcer tel ou tel type de nœud. Ces extensions sont très peu utilisées à notre connaissance.

Plans d'exécution

L'optimiseur doit faire de nombreux choix pendant la génération des plans de requêtes. Le premier choix consiste à savoir comment parcourir les données de base : en lisant une table, en parcourant un index, etc.

Si des jointures sont utilisées, il faut qu'il sélectionne l'algorithme de jointure, mais aussi l'ordre dans lequel les jointures seront exécutées. En effet, une jointure se fait entre deux ensembles de données uniquement. Si une jointure doit être réalisée entre les tables `t1`, `t2`, `t3` et `t4`, quelles sont les deux premières tables à joindre ? À quelle autre table va-t-on faire une jointure avec le résultat précédent ?

Si la requête dispose d'un agrégat, le bon algorithme d'agrégation des données doit être sélectionné. Dans le cas d'un tri, il est possible de le faire en parcourant un index sur les données triées ou lors de l'exécution de la requête, soit en mémoire, soit sur disque. Là aussi, l'optimiseur doit savoir sélectionner la bonne méthode de tri pour que les performances soient au rendez-vous.

10.2. Nœuds d'exécution d'un plan

Accès aux données

Plusieurs objets contiennent des données : les tables, les vues matérialisées, les index et les fonctions. Pour accéder aux données, l'exécuteur parcourt ces objets et récupère l'intégralité des données ou juste une partie si un filtre est appliqué. Il peut récupérer les données prétriées ou non, suivant le type d'objet parcouru.

Parcours de tables

Note *› Cette section parle de parcours de tables, mais elle s'applique tout aussi bien aux parcours de vues matérialisées.*

Seq Scan

Un moyen de lire les données d'une table revient à parcourir séquentiellement les blocs de la table et d'en déchiffrer les lignes pour les renvoyer à l'utilisateur. D'où le titre *Sequential Scan* (raccourci en *SeqScan*) pour le dénommer.

Le parcours se fait toujours sur tous les blocs de la table, à l'exception d'un cas (l'utilisation de la clause `LIMIT`). Par contre, certaines lignes peuvent ne pas être renvoyées si un filtre est ajouté. La question qui se pose dans ce cas est de savoir si un index ne permettrait pas de gagner en performances.

Comme les blocs sont parcourus séquentiellement, les données ne sont pas forcément triées. Elles sont renvoyées dans l'ordre physique de la table, qui ne correspond pas forcément à leur ordre d'insertion ou à un quelconque autre ordre. Voici les informations renvoyées par la commande `EXPLAIN` en cas de parcours séquentiel :

```
Seq Scan on t1 [...] ❶
  Filter: (id = 10) ❷
  Rows Removed by Filter: 999999 ❸
  Buffers: shared hit=4005 read=420 ❹
```

❶ `Seq Scan` est le nom du nœud, suivi du nom de la table ou de la vue matérialisée concernée. L'exécuteur de requêtes de PostgreSQL va donc lire l'objet indiqué, bloc par bloc, y déchiffrer les lignes et ne conserver que les lignes visibles par la transaction en cours.