PostgreSQL インサイド

チューニング

SQL チューニングを実施する技術を知る

FUJITSU

- [ ] 導入／環境設定
- [ ] 移行
- [ ] 性能
- [x] チューニング
- [ ] バックアップ／リカバリー
- [ ] 冗長化／負荷分散
- [ ] 監視
- [ ] データ連携
- [ ] 災害対策
- [ ] 豆知識

SQL チューニングは、SQL 実行において、処理に時間がかかっている SQL を対象に内部処理を解析し、最適な動作に改善していくことを目的としています。ここでは、サンプルを使用して SQL チューニングを実施し、その流れを解説します。

1. テーブルのスキャン方法と結合方法

SQL チューニングでは、SQL の実行計画を解析する必要があります。SQL チューニングを実施する前に、実行計画の解析で前提となるテーブルのスキャン方法と結合方法を簡単に説明します。

スキャン方法

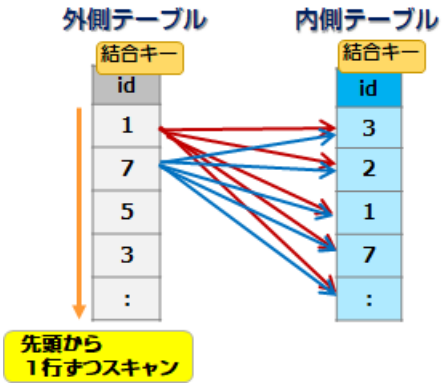
テーブルへのアクセス方法です。代表的なスキャン方法には、以下があります。

- **Seq Scan（シーケンシャルスキャン）**
  テーブルを最初から最後までシーケンシャルにアクセスします。取り出す件数が多い場合に有効な方法です。
- **Index Scan（インデックススキャン）**
  インデックスとテーブルを交互にランダムアクセスします。WHERE 句による絞り込みにより取り出す件数が少ない場合や目的のデータにピンポイントでアクセスしたい場合に有効な方法です。

結合方法

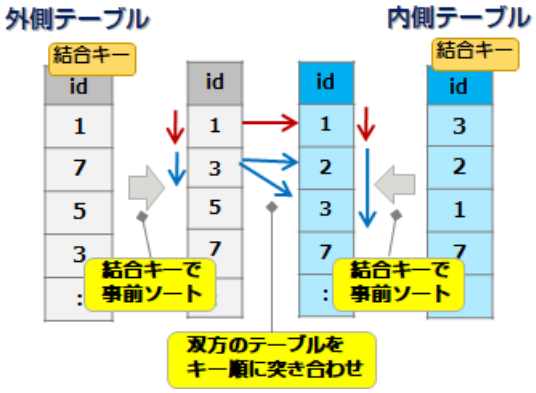
2 つのテーブルを結合する場合の結合方法です。2 つのテーブルのスキャン結果を入力として、1 つの結果を出力します。代表的な結合方法には、以下があります。ここでは、結合対象の 2 つのテーブルを区別するため「外側テーブル」と「内側テーブル」と呼びます。

- Nested Loop

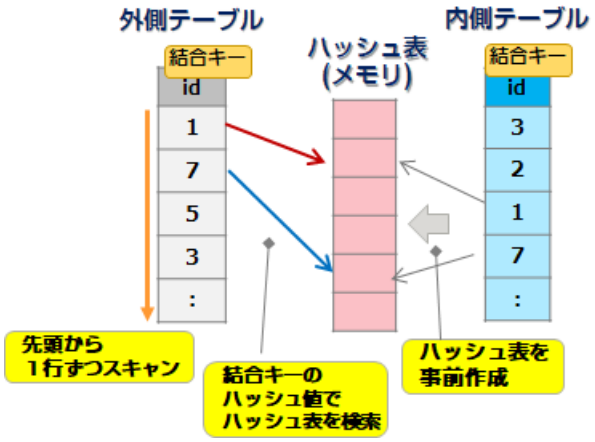


© 2021 Fujitsu Limited

- Merge Join



- Hash Join



- **Nested Loop（ネステッドループ結合）**
  外側テーブル 1 行ごとに内側テーブルを 1 周ループしながら結合する方法です。外側テーブルの行数が少なく、内側テーブルにインデックスがある場合に処理が高速になります。
- **Merge Join（マージ結合）**
  2 つのテーブルを結合キーでソート後、順番に突き合わせて結合する方法です。ソートに時間がかかる場合は有効な方法ではありません。結合キーが主キーである、または結合キーにインデックスを定義することでソート済みの状態を突き合わせることになるため、処理が高速になります。大きなテーブル同士を結合する場合に有効な方法です。
- **Hash Join（ハッシュ結合）**
  内側テーブルの結合キーでハッシュを作成し、ハッシュと外側テーブルの行を突き合わせて結合する方法です。ハッシュはメモリーに作成するため、一度作成してしまえば、高速に結合できます。ただし、ハッシュがメモリーサイズより大きくなってしまう場合は、ファイルアクセスが発生するため処理が遅くなります。比較的小さなテーブルと大きなテーブルを結合する場合に有効な方法です。

2. 処理が遅い SQL を検出

統計情報ビューとサーバーログから、処理が遅い SQL を検出します。

統計情報ビューを利用して検出する

pg_stat_statements ビューに対して以下の SQL を実行し、実行時間が長い SQL を、時間がかかっている順に 3 件出力します。なお、PostgreSQL 13 からは、total_time を total_exec_time に置き換えて実行してください。

 © 2021 Fujitsu Limited

```
SELECT query, calls, CAST(total_time as numeric(10,3)), CAST(total_time/calls AS numeric(10,3)) AS avg_time FROM
pg_stat_statements ORDER BY avg_time DESC,calls LIMIT 3;
```

出力内容

```
                                                query                                                 | calls | total_time | avg_time
------------------------------------------------------------------------------------------------------+-------+------------+----------
 SELECT sales.name,sum(sales.num),(sum(sales.num)) * price.price                                      |     5 |  31000.071 | 6200.014
  AS total_price FROM sales,price  WHERE sales.name = price.name                                      |       |            |
  GROUP BY sales.name,price.price ORDER BY sales.name ASC                                             |       |            |
 SELECT sales.name,sum(sales.num),(sum(sales.num)) * price.price                                      |     2 |   5886.436 | 2943.218
  AS total_price FROM sales,price  WHERE sales.name = price.name                                      |       |            |
  AND sales.datetime BETWEEN $1 AND $2 GROUP BY sales.name,price.price                                |       |            |
 SELECT sales.name,sum(sales.num) AS sum FROM sales GROUP BY sales.name                               |     6 |   6793.545 | 1132.257
(3 行)
```

pg_stat_statements ビューの内容から、sales テーブルと price テーブルを検索する SQL の実行で時間がかかっていることがわかります。

参考

クエリ統計情報（pg_stat_statements）以外にも、プランナ統計情報（pg_statistic）や関数統計情報（pg_stat_user_functions）を定期的に取得しておくことで、性能劣化が発生した際の分析に利用できます。

サーバーログを確認する

統計情報ビューで平均 6 秒かかっている SQL が検出できたため、次に、サーバーログを利用して 3 秒以上かかっている SQL を確認してみます（postgresql.conf の log_min_duration_statement パラメーターを 3 秒に設定）。

```
00000: 2019-03-05 11:18:08 JST [11408]: [3-1] user = fsep,db = mydb,remote = ::1(50650) app = psql
 LOG:  duration: 6518.096 ms  statement: SELECT sales.name,sum(sales.num),(sum(sales.num)) * price. price
 AS total_price FROM sales,price  WHERE sales.name = price.name GROUP BY sales.name,price.price
 ORDER BY sales.name ASC;
00000: 2019-03-05 11:21:30 JST [7184]: [7-1] user = ,db = ,remote =  app =  LOG:  checkpoint starting: time
00000: 2019-03-05 11:21:32 JST [7184]: [8-1] user = ,db = ,remote =  app =  LOG:  checkpoint complete: wrote
 14 buffers (0.0%); 0 WAL file(s) added, 0 removed, 6 recycled; write=1.556 s, sync=0.014 s, total=1.693 s;
 sync files=13, longest=0.001 s, average=0.001 s; distance=199525 kB, estimate=199525 kB
```

統計情報ビューを利用した場合と同様に、sales テーブルと price テーブルを検索する SQL の実行で時間がかかっていることがわかります。

参考

サーバーログでは、個々の SQL について、実際にどれくらいの時間（duration）がかかっているか、また、呼び出し元のユーザー（user）、データベース（db）、アプリケーション（app）などを確認することができます。ただし、これらの情報がログに出力されるよう、事前に postgresql.conf の log_min_duration_statement パラメーターや log_line_prefix パラメーターを設定する必要があります。

3. 原因を調査

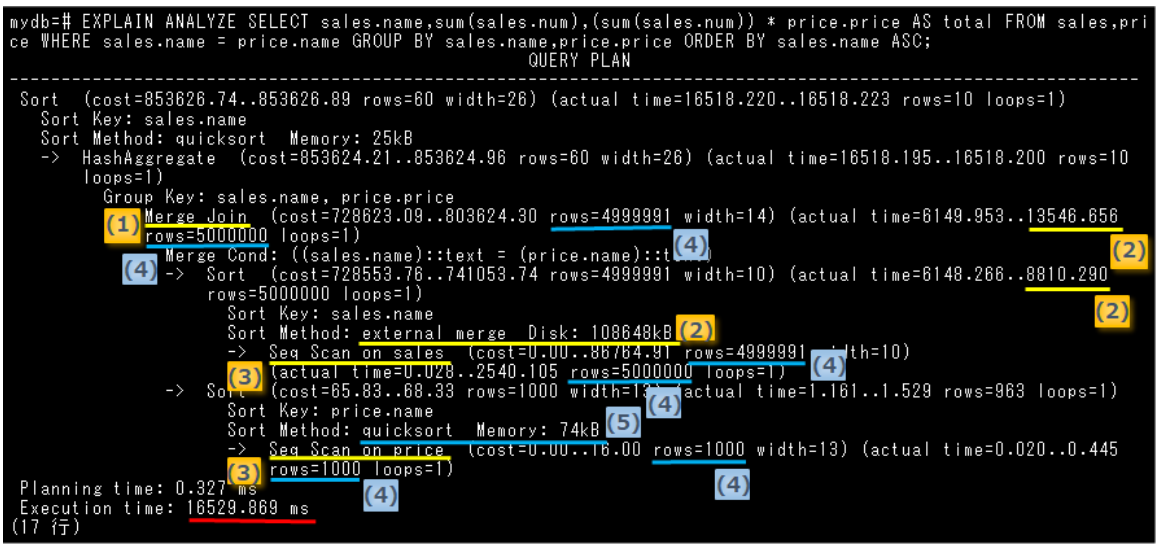
sales テーブルと price テーブルに対する検索処理で、なぜ時間がかかっているのかを調べてみましょう。

実行計画を表示する

まず、何が原因となっているのかを調査するために EXPLAIN コマンドを実行して、対象 SQL の実行計画を確認します。ANALYZE オプションを付けて実際に SQL を実行し、その実測値もあわせて確認してみます。ANALYZE オプションを使用すると実際に SQL 文が

 © 2021 Fujitsu Limited

実行されます。INSERT 文や DELETE 文などの実行計画を表示する際は、BEGIN 文と ROLLBACK 文を使用してデータに影響を与えないようにしてください。



実行計画から、以下がわかります。また、最終的な実行時間は、16529.869ms となっています。

- （1）sales テーブルと price テーブルの結合方法として「Merge Join」が選択されている。
- （2）sales テーブルのソート処理が「外部ソート（external merge：ソート結果を外部ファイルに書き出すソート）」になっており、処理に時間がかかっている。その結果、マージ結合の処理時間が長くなってしまっている。
- （3）sales テーブルと price テーブルのスキャン方法に、「Seq Scan」が選択されている。
- （4）推測値と実測値の rows がほぼ一致しているため、統計情報は最新化されている。
- （5）price テーブルは、「quicksort（メモリー上のソート）」が選択されているため、メモリー内（74KB 使用）で高速に処理されている。

まず、チューニングポイントとして、actual time を参考に、どの処理に時間がかかっているかを確認します。上記（4），（5）については特に問題がないことが確認できますが、（1），（2），（3）からは、結合処理に時間がかかっていると分析できます。そこで、マージ結合が適切な結合であるか、またその際の処理内容を改善できないか検討します。

参考

（2）では、postgesql.conf の work_mem パラメーターの値以上のメモリーをソートで使用しているために外部ソートとなっています。この場合、以下を実施し、work_mem パラメーターの値を Disk（108648KB）より大きくすることでソート方式を quicksort に変えて処理を高速化するといった対策も考えられます。

- サーバー側で postgesql.conf ファイルを変更
- クライアント側で環境変数 PGOPTIONS を使用して変更
- クライアント側で set 文を使用して変更

サーバー側の postgresql.conf ファイルを変更すると、データベースクラスタ全体に対して変更が有効になり、使用メモリー量の大幅な増加が考えられます。このため、できるだけクライアント側での変更を選択し、影響範囲を局所化してください。

 © 2021 Fujitsu Limited

データ件数を確認する

sales テーブルと price テーブルのデータ件数を確認します。

```
mydb=# SELECT COUNT(*) FROM sales;
  count
---------
 5000000
(1 行)

mydb=# SELECT COUNT(*) FROM price;
 count
-------
  1000
(1 行)
```

sales テーブルには 500 万件、price テーブルには 1000 件のデータが登録されていました。今回のケースでは、データベース環境に対して比較的大きなテーブル同士を結合しており、マージ結合が妥当な結合方法であると推測できます。

テーブル定義を確認する

次に、sales テーブルと price テーブルのテーブル定義を出力し、WHERE 句の条件で指定された結合キーである“name”の定義内容を確認します。

```
mydb=# ¥d sales
                                テーブル "public.sales"
   列   |            型             | 照合順序 | Null 値を許容 |           デフォルト
----------+---------------------------+----------+---------------+---------------------------------
 id       | integer                   |          | not null      | nextval('sales_id_seq'::regclass)
 name     | character varying(255)    |          | not null      |
 datetime | timestamp without time zone |          | not null      |
 num      | integer                   |          | not null      |

mydb=# ¥d price
                               テーブル "public.price"
  列   |          型           | 照合順序 | Null 値を許容 |           デフォルト
-------+-----------------------+----------+---------------+---------------------------------
 id    | integer               |          | not null      | nextval('price_id_seq'::regclass)
 name  | character varying(255) |          | not null      |
 price | integer               |          | not null      |

インデックス:
    "price_pkey" PRIMARY KEY, btree (name)
```

上記から、price テーブルでは、主キーである列“name”がインデックスとなっていることがわかります。しかし、sales テーブルは大きなテーブルであるにも関わらず、結合キーにインデックスが定義されていません。マージ結合の処理において、結合キーにインデックスを使用していないためソートに時間がかかっていると推測できます。

 © 2021 Fujitsu Limited

4. チューニング

「3. 原因を調査」での調査結果から、インデックスを追加し、性能が改善するかを確認します。

インデックスを追加する

sales テーブルの列“name”に対して、インデックス“sales_name_idx”を追加します。

```
mydb=# CREATE INDEX sales_name_idx ON sales(name);
CREATE INDEX
mydb=# ANALYZE;
ANALYZE

mydb=# ¥d sales
                                    テーブル "public.sales"
   列    |            型             | 照合順序 | Null 値を許容 |           デフォルト
---------+---------------------------+----------+--------------+--------------------------------
 id       | integer                   |          | not null     | nextval('sales_id_seq'::regclass)
 name     | character varying(255)    |          | not null     |
 datetime | timestamp without time zone |          | not null     |
 num      | integer                   |          | not null     |
インデックス:
    "sales_name_idx" btree (name)
```

実行計画を表示する

再度、実行計画を表示してみます。

```
mydb=# EXPLAIN ANALYZE SELECT sales.name,sum(sales.num),(sum(sales.num)) * price.price AS total FROM sales,pri
ce WHERE sales.name = price.name GROUP BY sales.name,price.price ORDER BY sales.name ASC;
                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------------
 Sort  (cost=390122.04..390122.19 rows=60 width=26) (actual time=9975.664..9975.668 rows=10 loops=1)
   Sort Key: sales.name
   Sort Method: quicksort  Memory: 25kB
   ->  HashAggregate  (cost=390119.52..390120.27 rows=60 width=26)
       (actual time=9975.640..9975.646 rows=10 loops=1)
         Group Key: sales.name, price.price
         ->  Merge Join  (cost=57.67..340119.59 rows=4999993 width=14)
             (actual time=1.504..7972.011 rows=5000000 loops=1)
               Merge Cond: ((sales.name)::text = (price.name)::text)
               ->  Index Scan using sales_name_idx on sales  (cost=0.43..277540.46 rows=4999993 width=10)
                   (actual time=0.066..3083.440 rows=5000000 loops=1)
               ->  Index Scan using price_pkey on price  (cost=0.28..79.15 rows=1000 width=13)
                   (actual time=0.009..0.933 rows=963 loops=1)
 Planning time: 0.929 ms
 Execution time: 9975.771 ms
(11 行)
```

マージ結合の事前処理として実行されていたソート処理とそれに伴う「Seq Scan」が、インデックス“sales_name_idx”を利用した「Index Scan」に変更されています。これにより、マージ結合にかかる時間が短くなり（インデックス追加前：13546.656ms、インデックス追加後：7972.011ms）、全体として、実行時間（Execution time）が短縮されました（インデックス追加前：16529.869ms、インデックス追加後：9975.771ms）。
また、EXPLAIN ANALYZE の実行時間（Execution time）は、「サーバーログを確認する」で確認した SQL の実行時間（duration）と異なる場合があります。「サーバーログを確認する」では、6518.096ms となっていましたので、チューニング後のサーバーログでも SQL の実行時間（duration）を確認してみます。（postgresql.conf の log_min_duration_statement パラメーターを 0（すべてを出力）に設定）

```
00000: 2019-03-05 15:29:44 JST [11408]: [43-1] user = fsep,db = mydb,remote = ::1(50650) app = psql
LOG:  duration: 1479.164 ms  statement: SELECT sales.name,sum(sales.num),(sum(sales.num)) * price.price
AS total FROM sales,price WHERE sales.name = price.name GROUP BY sales.name,price.price
ORDER BY sales.name ASC;
```

duration が 1479.164ms であり、サーバーログからも SQL の実行時間が短縮されたことがわかります。

 © 2021 Fujitsu Limited

ここでは、実際にチューニングを行いながら、SQL チューニングを説明しました。SQL 実行で性能に問題が発生した場合は、実行計画を分析し適切な処理となるようにチューニングを実施してください。

2021 年 1 月 22 日

© 2021 Fujitsu Limited