パーティショニングにおける性能向上のしくみ

FUJITSU

技術を知る

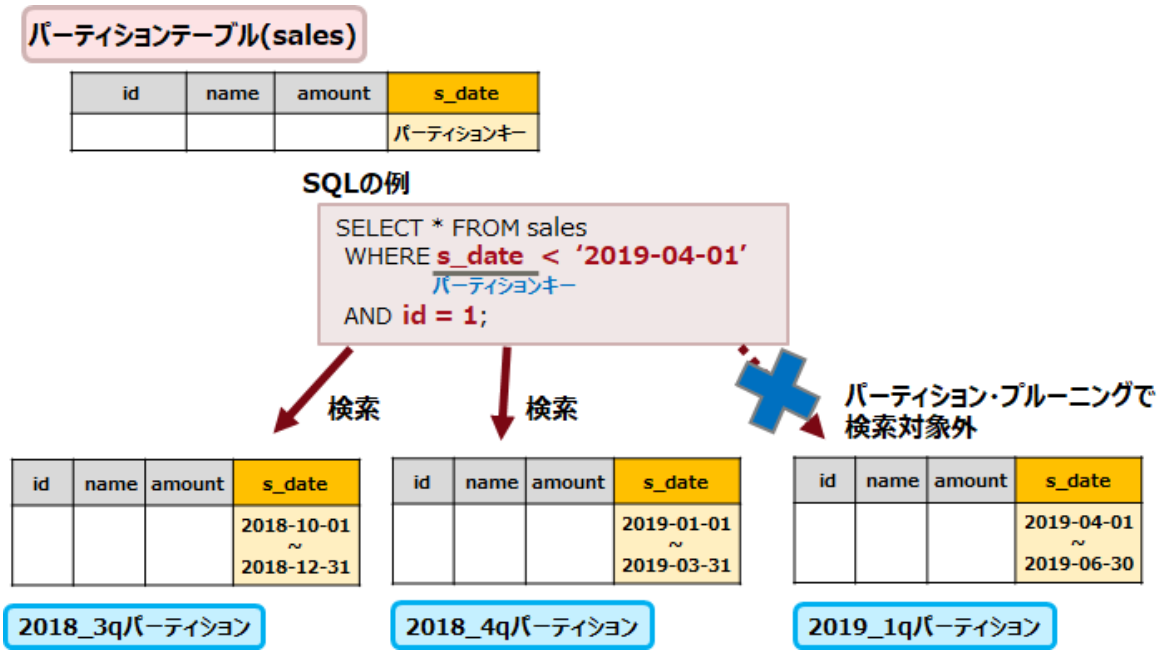
- ☐ 導入／環境設定
- ☐ 移行
- ☑ 性能
- ☐ チューニング
- ☐ バックアップ／リカバリー
- ☐ 冗長化／負荷分散
- ☐ 監視
- ☐ データ連携
- ☐ 災害対策
- ☐ 豆知識

PostgreSQL では、宣言的パーティショニング（以降、「パーティショニング」と呼びます）を利用することで性能の向上が期待できます。しかし、性能向上のしくみを理解した上で適切に使用しないと、逆に性能が劣化してしまうこともあります。ここでは、パーティショニングにおける性能向上のしくみを PostgreSQL 11.1 をベースに解説します。なお、パーティショニングの概要については、「パーティショニングの概要」を参照してください。
パーティショニングにおける性能向上は、以下の機能を利用することで実現できます。

- パーティション・プルーニング（パーティション除去）
- パーティション・ワイズ結合（パーティション同士の結合）
- パーティション・ワイズ集約（パーティション同士の集約）

1. パーティション・プルーニング（パーティション除去）

パーティション・プルーニングは、SQL からアクセスするパーティションを絞り込む機能です。「パーティショニングの概要」で、「検索条件にパーティションキーを指定すると検索範囲の絞り込みが行われる」ことを紹介しました。これは、「パーティション・プルーニング」が機能していたためです。
例えば、以下の図では、s_date 列がパーティションキーとなっています。SELECT 文の WHERE 句にパーティションキーを指定することで、アクセスするパーティションが絞り込まれます。絞り込んだパーティション内で検索が行われるので、効率的な検索処理が可能です。



パーティション・プルーニングを利用するためには、postgresql.conf の enable_partition_pruning パラメーターが on（デフォルト値：on）に設定されている必要があります。

1.1 パーティション・プルーニングの例

サンプルを使用し、パーティションテーブルとパーティションテーブルでないテーブルに対する検索を比較することで、パーティション・プルーニングの効果を確認してみましょう。サンプルで使用する sales テーブルは、sale_date 列をパーティションキーとするレンジパーティションです。3 つのパーティションに分割されています。また、nonpartition_sales テーブルはパーティションテーブルではありません。

sales テーブル

```
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 | ストレージ | 統計の対象 | 説明
-------------+---------+----------+---------------+------------+------------+------------+------
 emp_id      | integer |          |               |            | plain      |            |
 p_name      | text    |          |               |            | extended   |            |
 sales_amount| integer |          |               |            | plain      |            |
 sales_date  | date    |          |               |            | plain      |            |
パーティションキー: RANGE (sales_date)

パーティション: sales_2018_3q FOR VALUES FROM ('2018-10-01') TO ('2019-01-01'),
                sales_2018_4q FOR VALUES FROM ('2019-01-01') TO ('2019-04-01'),
                sales_2019_1q FOR VALUES FROM ('2019-04-01') TO ('2019-07-01')

mydb=# SELECT COUNT(*) FROM sales;
  count
----------
 3000000
(1 行)
```

nonpartition_sales テーブル

```
mydb=# ¥d+ nonpartition_sales
                                 テーブル "public.nonpartition_sales"
     列      |   型    | 照合順序 | Null 値を許容 | デフォルト | ストレージ |統計の対象 | 説明
-------------+---------+----------+---------------+------------+------------+-----------+------
 emp_id      | integer |          |               |            | plain      |           |
 p_name      | text    |          |               |            | extended   |           |
 sales_amount| integer |          |               |            | plain      |           |
 sales_date  | date    |          |               |            | plain      |           |


mydb=# SELECT COUNT(*) FROM nonpartition_sales;
  count
---------
 3000000
(1 行)
```

1. nonpartition_sales テーブルに対し、sales テーブルのパーティションキーである sales_date を検索条件に指定した SQL を実行し、実行計画を参照します。

```
mydb=# EXPLAIN ANALYZE SELECT * FROM nonpartition_sales WHERE sales_date < '2019-04-01' AND emp_id=1;
                                                    QUERY PLAN
                                                                  salesテーブルのパーティションキー
------------------------------------------------------------------------------------------------------------------
 Gather  (cost=1000.00..170222.74 rows=38753 width=18)
    (actual time=1104.962..3083.638 rows=39608 loops=1)
   Workers Planned: 2
   Workers Launched: 2                                (1)
   ->  Parallel Seq Scan on nonpartition_sales (cost=0.00..165347.44 rows=16147 width=18)
         (actual time=0.061..859.472 rows=13203 loops=3)
         Filter: ((sales_date < '2019-04-01'::date) AND (emp_id = 1))
         Rows Removed by Filter: 986797
 Planning Time: 0.249 ms   (2)
 Execution Time: 3156.240 ms
(8 行)
```

テーブル全体に対する「Seq Scan」が実行されており **(1)**、実行時間は、3156.240ms **(2)** となっています。

2. sales テーブルに対し、手順 1 と同じ SQL を実行し、実行計画を参照します。

```
mydb=# EXPLAIN ANALYZE SELECT * FROM sales WHERE sales_date < '2019-04-01' AND emp_id=1;
                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------------
 Gather  (cost=1000.00..68544.82 rows=40375 width=18)
    (actual time=78.527..331.082 rows=39443 loops=1)
   Workers Planned: 2
   Workers Launched: 2
   ->  Parallel Append  (cost=0.00..63507.31 rows=16823 width=18)
          (actual time=65.637..284.371 rows=13148 loops=3)
         ->  Parallel Seq Scan on sales_2018_3q  (cost=0.00..32829.38 rows=7596 width=18)
               (actual time=37.076..101.363 rows=5939 loops=3)
             Filter: ((sales_date < '2019-04-01'::date) AND (emp_id = 1))
             Rows Removed by Filter: 292428
         ->  Parallel Seq Scan on sales_2018_4q  (cost=0.00..30593.83 rows=9227 width=18)
               (actual time=46.664..198.292 rows=10813 loops=2)
             Filter: ((sales_date < '2019-04-01'::date) AND (emp_id = 1))
             Rows Removed by Filter: 539199
 Planning Time: 0.642 ms
 Execution Time: 423.267 ms
(12 行)
```

パーティションキー

(1)

(1)

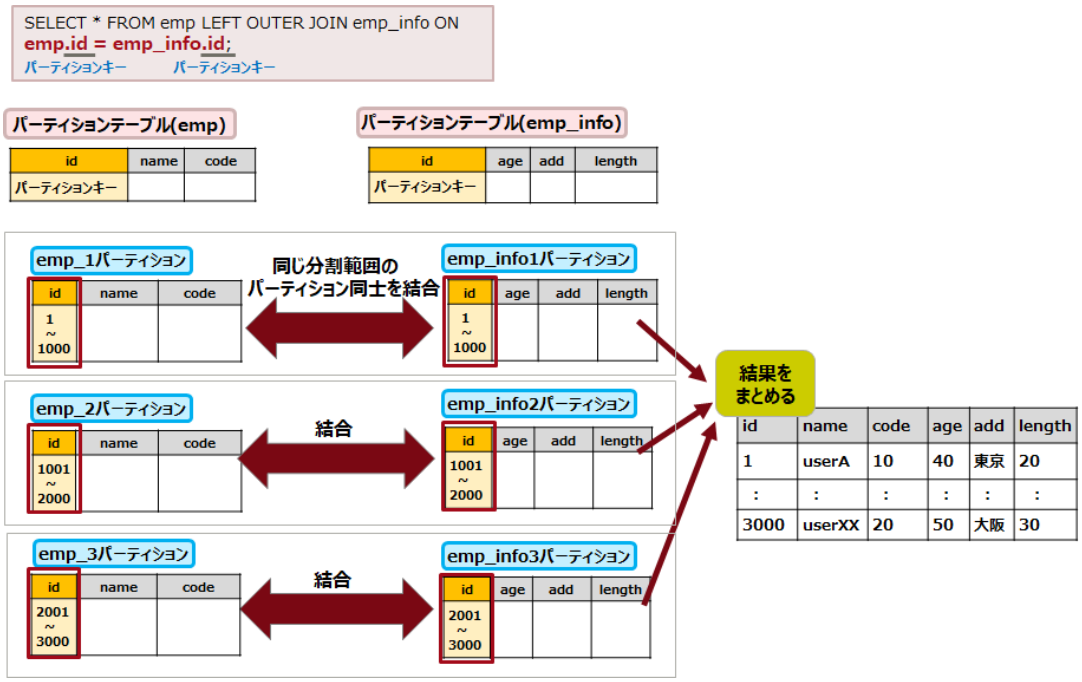
(2)

パーティション・プルーニングにより、スキャンの対象が sales_2018_3q パーティションと sales_2019_4q パーティションに絞り込まれ、絞り込まれた各パーティション対して「Seq Scan」が実行されています**（1）**。手順 1 と比較すると、以下のように実行時間**（2）**は約 1/8 に短縮されており、パーティション・プルーニングの効果が確認できます。

テーブルの種類	実行時間
パーティションテーブルでない場合	3156.240ms
パーティションテーブルの場合	423.267ms

2. パーティション・ワイズ結合（パーティション同士の結合）

パーティション・ワイズ結合は、パーティショニングされたテーブルに対する結合処理において、パーティション同士を結合する機能です。同じ範囲や値を持つパーティション同士を結合することで無駄な結合処理が省略され、効率的な結合処理を行うことができます。パーティション・ワイズ結合は、PostgreSQL 11 以降で利用可能です。
例えば、以下の図では、emp テーブルと emp_info テーブルは、id 列をパーティションキーとするパーティションテーブルで、それぞれ id 列の値が「1 から 1000」、「1001 から 2000」、「2001 から 3000」の範囲でパーティションが作成されています。「emp.id＝emp_info.id」の条件で結合する際、emp_1 パーティションと emp_info1 パーティションは、ともに id 列が「1 から 1000」の値を取りうるため結合条件に合致し、パーティション同士の結合を行います。しかし、emp_1 パーティションと emp_info2 パーティションでは、id 列が取りうる値の範囲が異なり結合条件には合致しないため、パーティション同士の結合を行いません。



パーティション・ワイズ結合を利用するためには、postgresql.conf の enable_partitionwise_join パラメーターを on（デフォルト値：off）に設定する必要があります。

ポイント

- パーティション・ワイズ結合は、実行計画の作成時に CPU やメモリーを多く消費することがあるため、デフォルトでは無効になっています。実際に運用を始める前に機能の利用が有効かどうか（性能向上が見込まれるか）を検証することをお勧めします。
- postgresql.conf ファイルを編集する、または、SET 文を使用することで enable_partitionwise_join パラメーターの on / off を切り替えることができます。SET 文の使用については、「2.1 パーティション・ワイズ結合の例」の「手順 3.」を参照してください。
- パラレルクエリを併用することで処理の高速化が期待できます。「【付録】パーティションテーブルに対するパラレルクエリについて」を参照してください。

2.1 パーティション・ワイズ結合の例

サンプルを使用し、パーティション・ワイズ結合が無効化されている場合と有効化されている場合を比較することで、その効果を確認してみましょう。サンプルで使用する emp テーブルと emp_info テーブルは、emp_id 列をパーティションキーとするハッシュパーティションです。同じ条件でそれぞれ 3 つのパーティションに分割されています。

emp テーブル

```
mydb=# ¥d+ emp
                                    テーブル "public.emp"
   列    |   型    | 照合順序 | Null 値を許容 | デフォルト | ストレージ | 統計の対象 | 説明
---------+---------+----------+---------------+------------+------------+------------+------
 emp_id   | integer |          | not null      |            | plain      |            |
 emp_name | text    |          |               |            | extended   |            |
 dep_code | integer |          |               |            | plain      |            |
パーティションキー: HASH (emp_id)
インデックス:
    "emp_pkey" PRIMARY KEY, btree (emp_id)
パーティション: emp_0 FOR VALUES WITH (modulus 3, remainder 0),
                emp_1 FOR VALUES WITH (modulus 3, remainder 1),
                emp_2 FOR VALUES WITH (modulus 3, remainder 2)

mydb=# SELECT COUNT(*) FROM emp;
  count
---------
 3000000
(1 行)
```

emp_info テーブル

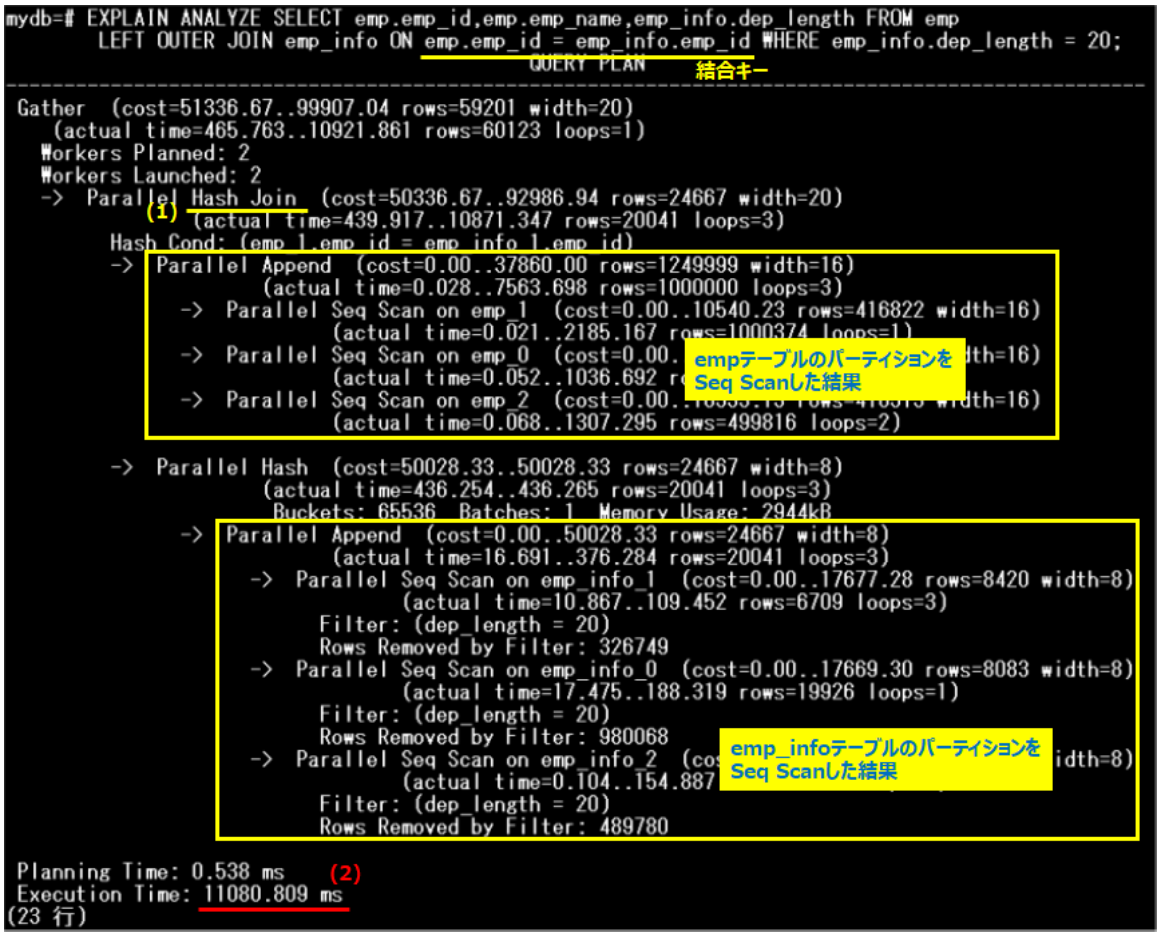
```
mydb=# ¥d+ emp_info
                                    テーブル "public.emp_info"
    列     |   型    | 照合順序 | Null 値を許容 | デフォルト | ストレージ | 統計の対象 | 説明
-----------+---------+----------+---------------+------------+------------+------------+------
 emp_id     | integer |          | not null      |            | plain      |            |
 emp_age    | integer |          |               |            | plain      |            |
 emp_add    | text    |          |               |            | extended   |            |
 dep_length | integer |          |               |            | plain      |            |
パーティションキー: HASH (emp_id)
インデックス:
    "emp_info_pkey" PRIMARY KEY, btree (emp_id)
パーティション: emp_info_0 FOR VALUES WITH (modulus 3, remainder 0),
                emp_info_1 FOR VALUES WITH (modulus 3, remainder 1),
                emp_info_2 FOR VALUES WITH (modulus 3, remainder 2)

mydb=# SELECT COUNT(*) FROM emp_info;
  count
---------
 3000000
(1 行)
```

1. パーティション・ワイズ結合が無効化されている(enable_partitionwise_join パラメーターが off)ことを確認します。

```
mydb=# SHOW enable_partitionwise_join;
 enable_partitionwise_join
---------------------------
 off
(1 行)
```


2. emp_id を結合キーとしてテーブルを結合する SQL を実行し、実行計画を参照します。



emp テーブルのパーティションをスキャンした結果と emp_info テーブルのパーティションをスキャンした結果を別々に取得し、最後に結合処理が行われています（**1**）。また、実行時間は、11080.809ms（**2**）となっています。次に、パーティション・ワイズ結合を有効化して、処理を比較してみましょう。

3. パーティション・ワイズ結合を有効化するため、enable_partitionwise_join パラメーターを on に設定します。

```
mydb=# SET enable_partitionwise_join TO on;
SET
mydb=# SHOW enable_partitionwise_join;
 enable_partitionwise_join
---------------------------
 on
(1 行)
```


4. 手順 2 と同じ SQL を実行し、実行計画を参照します。

emp_1 パーティションと emp_info1 パーティションのようにパーティション同士が「Nested Loop」で結合され、最後にそれらの結果をまとめています**（1）**。手順 2 のパーティション・ワイズ結合が無効化されている場合と比較すると、以下のように実行時間**（2）**は約 10 分の 1 に短縮されています。

パーティション・ワイズ結合	実行時間
無効化	11080.809ms
有効化	1136.806ms

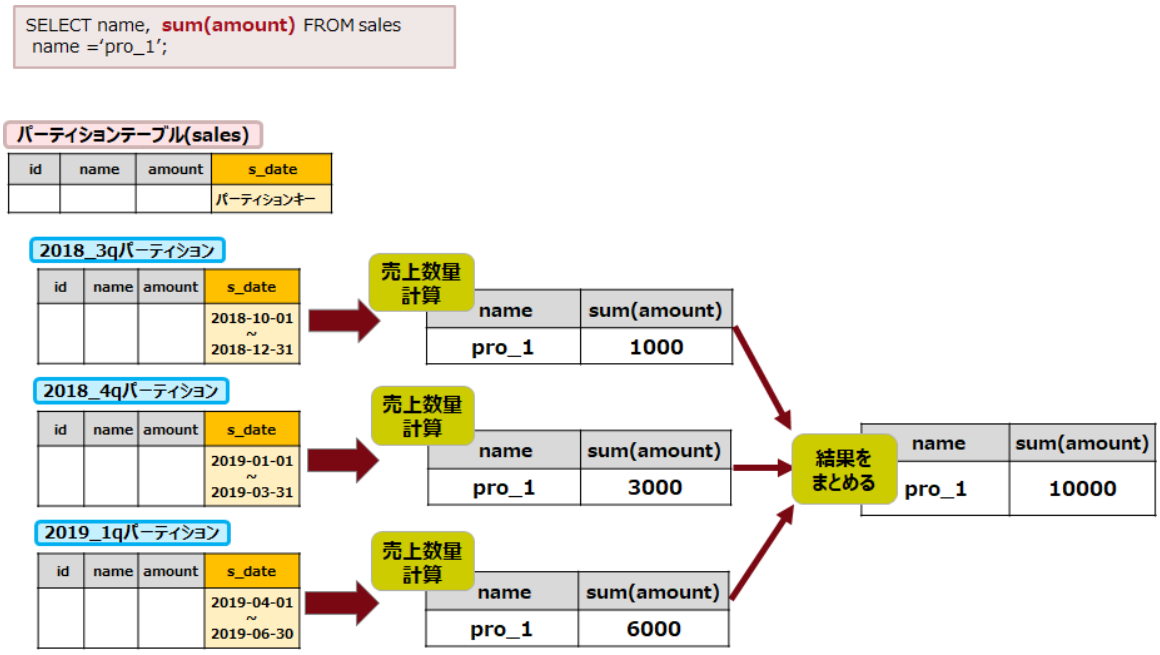
これは、パーティション同士の結合において、データの絞り込みの処理が変化し、高速な処理が実現できているためです。

ポイント

パーティション・ワイズ結合の有効化/無効化により、実行計画が変更されます。この機能により、最適な絞り込みに合わせ、結合方式も最適化されることがあります。

3. パーティション・ワイズ集約（パーティション同士の集約）

パーティション・ワイズ集約は、パーティショニングされたテーブルに対する集約処理において、パーティションごとに集約処理を行い、最後にその結果を統合する機能です。パーティション単位に集約処理を行うことで処理時間が短縮できます。パーティション・ワイズ集約は、PostgreSQL 11 以降で利用可能です。
例えば、以下の図では、各パーティションにおいて集約処理が実行され、最後に結果が統合されます。



パーティション・ワイズ集約を利用するためには、postgresql.conf の enable_partitionwise_aggregate パラメーターを on（デフォルト値：off）に設定する必要があります。

ポイント

- パーティション・ワイズ集約は、実行計画の作成時に CPU やメモリーを多く消費することがあるため、デフォルトでは無効になっています。実際に運用を始める前に機能の利用が有効かどうか（性能向上が見込まれるか）を検証することをお勧めします。
- postgresql.conf ファイルを編集する、または、SET 文を使用することで、enable_partitionwise_aggregate パラメーターの on / off を切り替えることができます。SET 文の使用については、「3.1 パーティション・ワイズ集約の例」の「手順 3.」を参照してください。
- パラレルクエリを併用することで処理の高速化が期待できます。「【付録】パーティションテーブルに対するパラレルクエリについて」を参照してください。

3.1 パーティション・ワイズ集約の例

サンプルを使用し、パーティション・ワイズ集約が無効化されている場合と有効化されている場合を比較することで、その効果を確認してみましょう。サンプルで使用する sales テーブルは、sale_date をパーティションキーとするレンジパーティションです。3 つのパーティションに分割されています。

sales テーブル

```
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 | ストレージ | 統計の対象 | 説明
-------------+---------+----------+---------------+------------+------------+------------+------
 emp_id      | integer |          |               |            | plain      |            |
 p_name      | text    |          |               |            | extended   |            |
 sales_amount| integer |          |               |            | plain      |            |
 sales_date  | date    |          |               |            | plain      |            |
パーティションキー: RANGE (sales_date)
インデックス:
    "sales_emp_id_idx" btree (emp_id)
パーティション: sales_2018_3q FOR VALUES FROM ('2018-10-01') TO ('2019-01-01'),
                sales_2018_4q FOR VALUES FROM ('2019-01-01') TO ('2019-04-01'),
                sales_2019_1q FOR VALUES FROM ('2019-04-01') TO ('2019-07-01')

mydb=# SELECT COUNT(*) FROM sales;
  count
---------
 3000000
(1 行)
```

1. パーティション・ワイズ集約が無効化されている（enable_partitionwise_aggregate パラメーターが off）になっていることを確認します。

```
mydb=# SHOW enable_partitionwise_aggregate;
 enable_partitionwise_aggregate
--------------------------------
 off
(1 行)
```

2. 集約処理を行う SQL を実行し、実行計画を参照します。

```
mydb=# EXPLAIN ANALYZE SELECT p_name,sum(sales_amount) sales_total FROM sales
       WHERE p_name = 'pro_1'GROUP BY p_name;
                                                                                QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------------------
 GroupAggregate  (cost=379.48..82244.43 rows=51 width=14)
   (actual time=502.909..502.912 rows=1 loops=1)
(2) Group Key: sales_2018_3q.p_name
   ->  Append  (cost=379.48..81943.21 rows=60142 width=10)
         (actual time=2.846..412.547 rows=60141 loops=1)
     (1) ->  Bitmap Heap Scan on sales_2018_3q  (cost=379.48..25830.56 rows=17426 width=10)
               (actual time=2.843..74.142 rows=18043 loops=1)
             Recheck Cond: (p_name = 'pro_1'::text)
             Heap Blocks: exact=5483
             ->  Bitmap Index Scan on sales_2018_3q_p_name_idx
                   (cost=0.00..375.12 rows=17426 width=0)
                     (actual time=2.102..2.104 rows=18043 loops=1)
                   Index Cond: (p_name = 'pro_1'::text)
         ->  Bitmap Heap Scan on sales_2018_4q  (cost=466.39..29501.10 rows=21414 width=10)
               (actual time=3.489..91.608 rows=21935 loops=1)
             Recheck Cond: (p_name = 'pro_1'::text)
             Heap Blocks: exact=6716
             ->  Bitmap Index Scan on sales_2018_4q_p_name_idx
                   (cost=0.00..461.03 rows=21414 width=0)
                     (actual time=2.621..2.622 rows=21935 loops=1)
                   Index Cond: (p_name = 'pro_1'::text)
         ->  Bitmap Heap Scan on sales_2019_1q  (cost=465.52..26310.84 rows=21302 width=10)
               (actual time=3.087..81.406 rows=20163 loops=1)
             Recheck Cond: (p_name = 'pro_1'::text)
             Heap Blocks: exact=6120
             ->  Bitmap Index Scan on sales_2019_1q_p_name_idx
                   (cost=0.00..460.19 rows=21302 width=0)
                     (actual time=2.226..2.228 rows=20163 loops=1)
                   Index Cond: (p_name = 'pro_1'::text)

 Planning Time: 0.345 ms  (3)
 Execution Time: 503.067 ms
(20 行)
```

sales_2018_3qパーティションを Index Scanした結果

sales_2018_4qパーティションを Index Scanした結果

sales_2019_1qパーティションを Index Scanした結果

各パーティションに対して「Index Scan」した処理をまとめて（1）、最後に集約処理（2）が実行されています。また、実行時間は、503.067ms（3）となっています。次に、パーティション・ワイズ集約を有効化して、処理を比較してみましょう。

3. enable_partitionwise_aggregate パラメーターを on に設定します。

```
mydb=# SET enable_partitionwise_aggregate TO on;
SET
mydb=# SHOW enable_partitionwise_aggregate;
 enable_partitionwise_aggregate
--------------------------------
 on
(1 行)
```

4. 手順 2 と同じ SQL を実行し、実行計画を参照します。

各パーティションで集約処理が実行され、最後に結果をまとめています（1）。手順 2 のパーティション・ワイズ集約が無効化されている場合と比較すると、以下のように実行時間（2）が短縮されています。

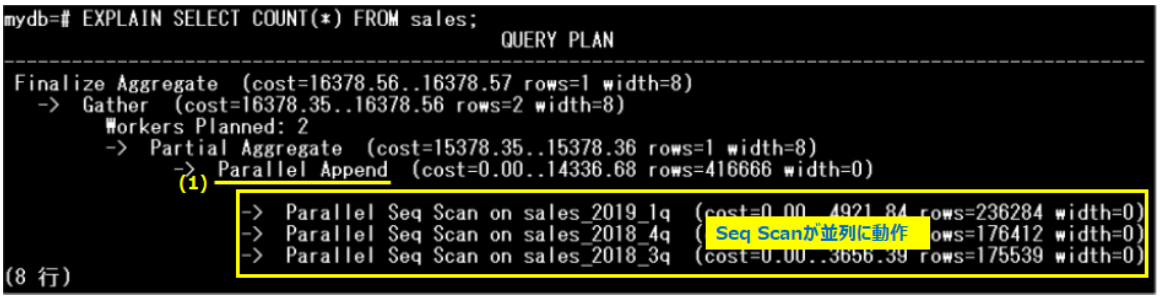
パーティション・ワイズ集約	実行時間
無効化	503.067ms
有効化	394.091ms

これは、各パーティションで集約処理が実行されることで、「Append」**(1)** の対象行数が少なくなり、Append 処理の高速化が実現できているためです。

ここでは、サンプルを使用し、パーティショニングの性能向上のしくみを解説しました。性能向上のしくみを正しく理解した上で、パーティショニングのテーブル設計や運用設計を実施してください。

【付録】 パーティションテーブルに対するパラレルクエリについて

パラレルクエリは、1 つの SQL を複数のプロセスで並列に実行する機能です。複数の CPU に処理を分散させて並列に実行することで性能向上を実現します。パラレルクエリについての詳細は、“PostgreSQL 11.1 文書”の“パラレルクエリ”を参照してください。
PostgreSQL 11 で、パーティションテーブルに対するパラレルクエリが実行可能となりました。パーティション単位に並列に処理を実行し、最後に結果をまとめることで処理時間が短縮できます。パーティションテーブルに対するパラレルクエリを使用するためには、postgresql.conf の enable_parallel_append パラメーターが on（デフォルト値：on）に設定されている必要があります。
以下は、パーティションテーブルに対するパラレルクエリの実行計画の例です。



各パーティションに対する「Seq Scan」が並列に動作し、最後に「Parallel Append」**(1)** で結果をまとめています。
本記事で解説したパーティション・ワイズ結合やパーティション・ワイズ集約は、パラレルクエリと併用することでパーティション単位での結合処理や集約処理の並列実行が可能となるため、更なる性能向上が期待できます。

2019 年 10 月 25 日