PostgreSQL インサイド

パーティショニングの概要

技術を知る

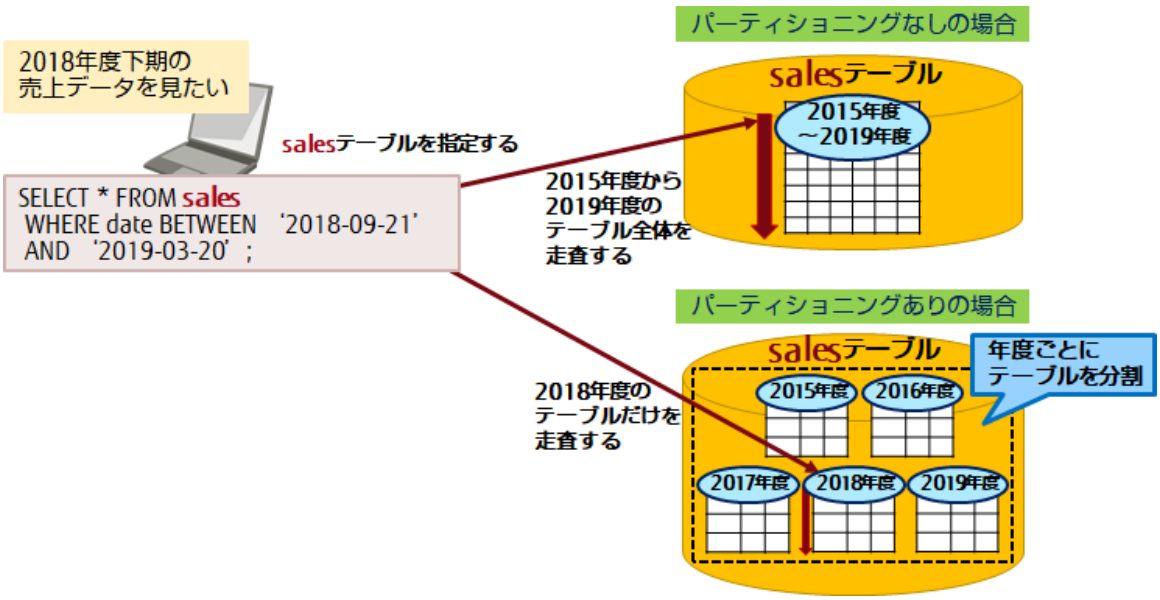
FUJITSU

- [ ] 導入／環境設定
- [ ] 移行
- [x] 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

パーティショニングは、データベースにおけるテーブル内のデータを分割して保持する機能です。PostgreSQL では、PostgreSQL 10 から『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』が追加されました。なお、この記事は、PostgreSQL 11.1 をベースに解説しています。

1. パーティショニングとは

パーティショニングは、データの特性や利用目的に応じて分割条件の設計が必要です。データはその条件に従って分割したテーブルに格納されますが、アプリケーションからは 1 つのテーブルとして扱うことができます。パーティショニングのイメージを以下に示します。



1.1 パーティショニングのメリット

パーティショニングを利用するメリットには、「性能の向上」と「メンテナンス性の向上」があります。これらのメリットについて説明します。

性能の向上

テーブルの分割により、アプリケーションからの SQL アクセスにおいて、検索性能の向上が見込まれます。性能向上を実現する要因には以下があります。

- **テーブル分割による I/O 削減**
  絞り込み条件を SQL に指定することで、アクセスする範囲を特定のパーティションのみに絞り込むことができます。また、パーティション内でよく使用される部分がメモリー上にキャッシュされやすくなります。これにより、ディスク I/O が減り、アクセス性能が向上します。

- **テーブル空間を分けることによる I/O 分散**
  分割したテーブルを物理ディスクの異なるテーブル空間に配置します。これにより、別々の物理ディスクに対して並行で読み書きが行われることでディスク I/O が分散し、処理性能が向上します。

ポイント

PostgreSQL 11 以前のパーティショニングでは、テーブルの分割数が多すぎると逆に性能が劣化する可能性があります。そのため、分割数は 100 以下を推奨しています。なお、PostgreSQL 12 では、パーティショニングの性能が改善されたことにより、数千の分割数でも効率的に処理できるようになりました。

メンテナンス性の向上

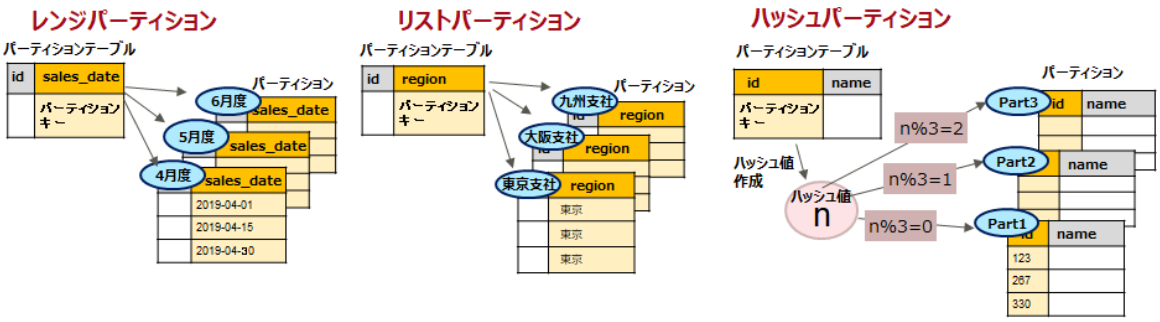
分割したテーブルの単位でデータの追加や削除、データ更新を行うことができるため、運用上のメンテナンス性が向上します。たとえば、5 年間の売上データを月ごとのパーティションで保持するシステムの場合、月が変わった際に、新たな月のパーティションを作成し、かつ、5 年前の同じ月のパーティションを削除するといった運用が可能です。また、この運用では、パーティション単位でのテーブル削除（DROP TABLE）や TRUNCATE によるデータ削除が行えるため、DELETE によるデータ削除と比較して、処理が高速化でき、VACUUM の負荷も低減できます。

1.2 PostgreSQL がサポートするパーティションの種類

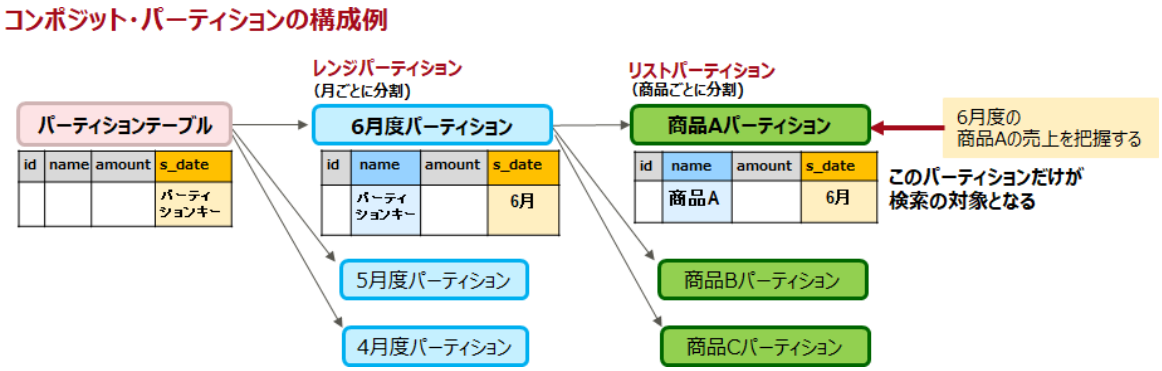
パーティショニングでは、データを分割する際にキーとなる列（「パーティションキー」と呼びます）を設定します。このパーティションキーの値をどのように分割するかでパーティションの種類が決定します。PostgreSQL では以下の種類のパーティションをサポートしています。業務や運用にあわせて適切な分割方法を選択してください。

種類	分割方法 / 特長	分割例
レンジ（範囲）パーティション	• パーティションキーの値を期間や範囲などで分割する • 時系列のデータに対して、年や月など日付を条件にアクセスしたい場合に有効	売上日など日付データで分割 • 4 月度（2019/04/01～2019/04/30） • 5 月度（2019/05/01～2019/05/31） • 6 月度（2019/06/01～2019/06/30）
リストパーティション	• パーティションキーの値をあらかじめ決められた値で分割する • 地域、部署など不連続なデータを任意の値でグループ化したい場合に有効	各支社のパーティションで分割 • 東京支社 • 大阪支社 • 九州支社
ハッシュパーティション（PostgreSQL 11 以降で利用可能）	• パーティションキーの値に対してハッシュ値を作成し、そのハッシュ値を分割数で割った余りで分割する • データをほぼ均等に振り分けることで、特定テーブルへのアクセス集中を回避したい場合に有効	3 つのパーティションに分割（"n"はパーティションキーのハッシュ値） • n % 3 =0 ・・・・ パーティション 1 に振り分け • n % 3 =1 ・・・・ パーティション 2 に振り分け • n % 3 =2 ・・・・ パーティション 3 に振り分け

以下に 3 種類のパーティションのイメージを示します。なお、本記事では、親テーブルを「パーティションテーブル」、子テーブルを「パーティション」と呼びます。



さらに、以下のようにパーティションの配下にさらにパーティションを作成し、より細分化したパーティション構成とすることも可能です。この構成は「コンポジット・パーティション（サブパーティション）」と呼ばれます。たとえば、製品売上テーブル（パーティションテーブル）に対して、月で分割したレンジパーティションを作成し、その配下に商品別に分割したリストパーティションを作成しておきます。「6 月度の商品 A の売上を把握する」場合など、検索対象となるパーティションを絞り込むことができるためアクセス性能が向上します。



2. パーティショニングの使い方

簡単なサンプルを使用してパーティションを使ってみましょう。サンプルで使用するデータベースは「mydb」とします。

2.1 作成方法

パーティションテーブルは CREATE TABLE を使用し、パーティションの種類と分割のキーとなるパーティションキーを指定して作成します。また、各パーティションは、CREATE TABLE を使用し、分割の範囲や値を指定して作成します。

2.1.1 レンジパーティションの作成

日々の売上を管理するテーブルを 3 か月ごとに分割するパーティションを作成します。

1. sales_date 列をパーティションキーとしてパーティションテーブル（sales）を作成します。その際、レンジパーティションを示す「RANGE」を指定します。

```
mydb=# CREATE TABLE sales (emp_id int, p_name text, sales_amount int, sales_date date) PARTITION BY RANGE (sales_date);
```

2. パーティションを作成します。レンジパーティションでは、各パーティションが取りうるパーティションキーの値の範囲を指定します。範囲については、下限値は含まれ、上限値は含まれません。下記の例で、「FROM ('2018-10-01') TO ('2019-01-01')」の範囲は「2018-10-01 から 2018-12-31 まで」 ということになります。

```
mydb=# CREATE TABLE sales_2018_3q PARTITION OF sales FOR VALUES FROM ('2018-10-01') TO ('2019-01-01');
mydb=# CREATE TABLE sales_2018_4q PARTITION OF sales FOR VALUES FROM ('2019-01-01') TO ('2019-04-01');
```



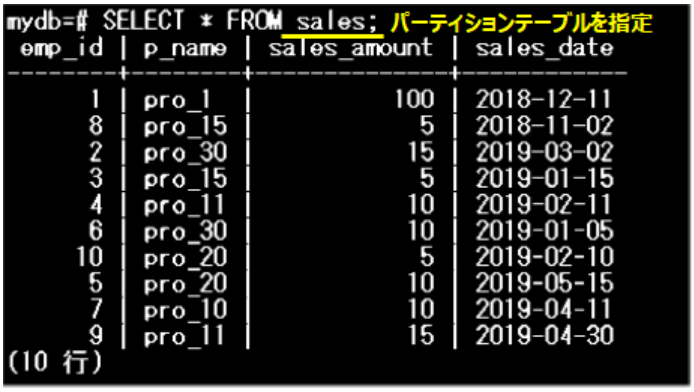
```
mydb=# CREATE TABLE sales_2019_1q PARTITION OF sales FOR VALUES FROM ('2019-04-01') TO ('2019-07-01');
```

3. パーティションテーブル（sales）を指定してデータを挿入します。ここでは、INSERT を使用します。

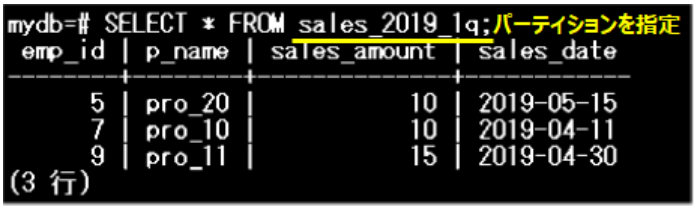
```
mydb=# INSERT INTO sales VALUES(1,'pro_1',100,'2018-12-11');
```

4. 10 件のデータを挿入後、テーブルを全件検索し、データを参照してみます。

   - **【パーティションテーブルの場合】**
     パーティションテーブルを対象とすると、全パーティションのデータが参照できます。



   - **【パーティションの場合】**
     1 つのパーティションを対象とすると、そのパーティション内に格納されているデータが参照できます。以下は、sales_2019_1q パーティションを対象とした場合の例です。



2.1.2 リストパーティションの作成

支社ごとの売上を管理するテーブルについて、地域ごとにテーブルを分割するリストパーティションを作成します。

1. region 列をパーティションキーとしてパーティションテーブル（sales_region）を作成します。その際、リストパーティションを示す「LIST」を指定します。

```
mydb=# CREATE TABLE sales_region (emp_id int, sales_amount int, branch text, region text) PARTITION BY LIST (region);
```

2. パーティションを作成します。リストパーティションでは、パーティションキーの値を指定します。下記の例では、「札幌」、「東京」、「名古屋」、「大阪」がパーティションキーの値となります。

```
mydb=# CREATE TABLE Sapporo PARTITION OF sales_region FOR VALUES IN('札幌');
mydb=# CREATE TABLE Tokyo PARTITION OF sales_region FOR VALUES IN('東京');
mydb=# CREATE TABLE Nagoya PARTITION OF sales_region FOR VALUES IN('名古屋');
mydb=# CREATE TABLE Osaka PARTITION OF sales_region FOR VALUES IN('大阪');
```

3. パーティションテーブル（sales_region）を指定してデータを挿入します。ここでは、COPY を使用します。


```
mydb=# COPY sales_region FROM '/home/tmp/listpart_1.sql';
```

4. Tokyo パーティションのデータを参照してみます。

```
mydb=# SELECT * FROM Tokyo;
 emp_id | sales_amount | branch | region
--------+--------------+--------+--------
      1 |          100 | 横浜   | 東京
      7 |            5 | 品川   | 東京
     10 |           10 | 長野   | 東京
(3 行)
```

region 列が「東京」のデータが格納されていることがわかります。

2.1.3 ハッシュパーティションの作成

1000 件のデータを 3 つのパーティションに均等に分割するハッシュパーティションを作成します。ハッシュパーティションでは、パーティションキーの列の値を元に作成したハッシュ値を分割数で割った剰余によって格納するテーブルが決定するため、パーティション作成時に、分割数と剰余を指定します。

1. emp_id 列をパーティションキーとしてパーティションテーブル（emp）を作成します。その際、ハッシュパーティションを示す「HASH」を指定します。

```
mydb=# CREATE TABLE emp (emp_id int, emp_name text, dep_code int) PARTITION BY HASH (emp_id);
```

2. 分割数を 3、剰余を 0,1,2 とする 3 つのパーティションを作成します。

```
mydb=# CREATE TABLE emp_0 PARTITION OF emp FOR VALUES WITH(MODULUS 3,REMAINDER 0);
mydb=# CREATE TABLE emp_1 PARTITION OF emp FOR VALUES WITH(MODULUS 3,REMAINDER 1);
mydb=# CREATE TABLE emp_2 PARTITION OF emp FOR VALUES WITH(MODULUS 3,REMAINDER 2);
```

3. 1000 件のデータを挿入します。この例では、検証用として、generate_series 関数を使用してデータを挿入しています。

```
mydb=# INSERT INTO emp SELECT num ,'user_' || num , (RANDOM()*50)::INTEGER FROM generate_series(1,1000) AS num;
```

4. 各パーティション内のデータ件数を確認してみます。pg_class カタログを参照することで各テーブルの行数を参照することができます。

```
mydb=# SELECT relname,reltuples as rows FROM pg_class
 WHERE relname IN ('emp','emp_0','emp_1','emp_2') ORDER BY relname;
 relname | rows
---------+------
 emp     |    0
 emp_0   |  324
 emp_1   |  333
 emp_2   |  343
(4 行)
```

3分割

データがほぼ 3 分割されていることがわかります。

2.1.4 パーティション範囲外のデータの扱いについて

パーティショニングでは、分割の範囲が定義されているパーティションとは別に、範囲外のデータを格納するためのパーティション（「デフォルトパーティション」と呼びます）を作成することができます。デフォルトパーティションを使用すると、たとえば、範囲外のデータを一時的にデフォルトパーティションに蓄積しておき、新たにパーティションを追加してデフォルトパーティションのデータを移動するという運用も可能です。

ポイント

デフォルトパーティションは、PostgreSQL 11 以降のレンジパーティションとリストパーティションで使用できます。PostgreSQL 11 でデフォルトパーティションが未使用の場合や PostgreSQL 10 の場合に範囲外のデータを挿入しようとすると、格納するパーティションが存在しないためエラーとなります。

ここでは、「2.1.2 リストパーティションの作成」で、以下のように作成したパーティションテーブル（sales_region）に、デフォルトパーティションを追加してみます。

```
mydb=# ¥d+ sales_region
                                テーブル "public.sales_region"
     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      |           |
 sales_amount | integer |          |               |            | plain      |           |
 branch      | text    |          |               |            | extended   |           |
 region      | text    |          |               |            | extended   |           |
パーティションキー: LIST (region)
パーティション: nagoya FOR VALUES IN ('名古屋'),
              osaka FOR VALUES IN ('大阪'),
              sapporo FOR VALUES IN ('札幌'),
              tokyo FOR VALUES IN ('東京')
```

ポイント

「¥d+ パーティションテーブル名」を使用すると、パーティションテーブルとパーティションの情報が確認できます。

1. デフォルトパーティション（region_default）を作成します。デフォルトパーティションの作成では、パーティションキーの代わりに DEFAULT 句を指定します。

```
mydb=# CREATE TABLE region_default PARTITION OF sales_region DEFAULT;
```


2. パーティションテーブルの情報を参照します。

```
mydb=# ¥d+ sales_region
                                テーブル "public.sales_region"
     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      |            |
 sales_amount| integer |          |               |            | plain      |            |
 branch      | text    |          |               |            | extended   |            |
 region      | text    |          |               |            | extended   |            |
パーティションキー: LIST (region)
パーティション: nagoya FOR VALUES IN ('名古屋'),
                osaka FOR VALUES IN ('大阪'),
                sapporo FOR VALUES IN ('札幌'),
                tokyo FOR VALUES IN ('東京'),
                region_default DEFAULT
                  デフォルトパーティション
```

デフォルトパーティションとして region_default パーティションが追加されました。

3. パーティションテーブルに範囲外のデータを挿入してみます。

```
mydb=# INSERT INTO sales_region VALUES(11,10,'博多','九州');
INSERT 0 1                                          範囲外のデータ
```

エラーとならずに INSERT が成功しました。

4. region_default パーティションを参照します。

```
mydb=# SELECT * FROM region_default;
 emp_id | sales_amount | branch | region
--------+--------------+--------+--------
     11 |           10 | 博多   | 九州
(1 行)
```

region 列が「九州」の範囲外データが格納されていることがわかります。

2.2 パーティショニングの動作

2.1 で作成したパーティションを使用し、パーティショニングの動作について説明します。

2.2.1 パーティションテーブルの検索

「2.1.3 ハッシュパーティションの作成」で作成したハッシュパーティションを使用し、検索処理を実行してみます。

1. emp_id=1000 の従業員情報を検索します。パーティションテーブルを指定することで、条件に合致するデータが参照できます。どのパーティションにデータが格納されているかを意識する必要はありません。

```
mydb=# SELECT * FROM emp WHERE emp_id=1000;
                    パーティションテーブルを指定
 emp_id | emp_name  | dep_code
--------+-----------+----------
   1000 | user_1000 |        7
(1 行)
```


2. 次に、検索条件に emp_id=1000 を指定した実行計画を表示してみましょう。

```
mydb=# EXPLAIN ANALYZE SELECT * FROM emp WHERE emp_id=1000;
                                  QUERY PLAN
------------------------------------------------------------------------------
 Append  (cost=0.00..7.29 rows=1 width=16)
         (actual time=0.047..0.053 rows=1 loops=1)
   ->  Seq Scan on emp_2  (cost=0.00..7.29 rows=1 width=16)
                          (actual time=0.042..0.045 rows=1 loops=1)
         Filter: (emp_id = 1000)
         Rows Removed by Filter: 342
 Planning Time: 0.293 ms
 Execution Time: 0.101 ms
(6 行)
```

パーティションキーを指定

検索条件にパーティションキー（emp_id）を指定することで検索範囲の絞り込みが行われ、3 つのパーティションのうち、emp_2 パーティションのみがスキャンの対象となっていることがわかります。

2.2.2 パーティション間のデータ更新

パーティショニングでは、パーティションキーの列の値を更新すると、自動的に適切なパーティションに値が移動します。「2.1.2 リストパーティションの作成」で作成したリストパーティションを使用し、この動作を確認してみましょう。以下の例では、Nagoya パーティションにある静岡支社のデータが Tokyo パーティションに移動します。

ポイント

PostgreSQL 10 では、パーティションをまたがる UPDATE は実行できないため注意が必要です。PostgreSQL 10 では、対象のデータを削除し、再度、更新したデータを挿入する必要があります。

1. Nagoya パーティションのデータを確認します。静岡支社（branch='静岡'）の情報がこのパーティション内に存在していることがわかります。

```
mydb=# SELECT * FROM Nagoya;
 emp_id | sales_amount | branch | region
--------+--------------+--------+--------
      8 |           10 | 名古屋 | 名古屋
      9 |           15 | 名古屋 | 名古屋
      4 |           10 | 静岡   | 名古屋
(3 行)
```

パーティションキー

2. branch='静岡'の region を名古屋から東京に変更します。

```
UPDATE sales_region SET region = '東京' WHERE branch = '静岡';
```

1. Tokyo パーティションのデータを確認します。

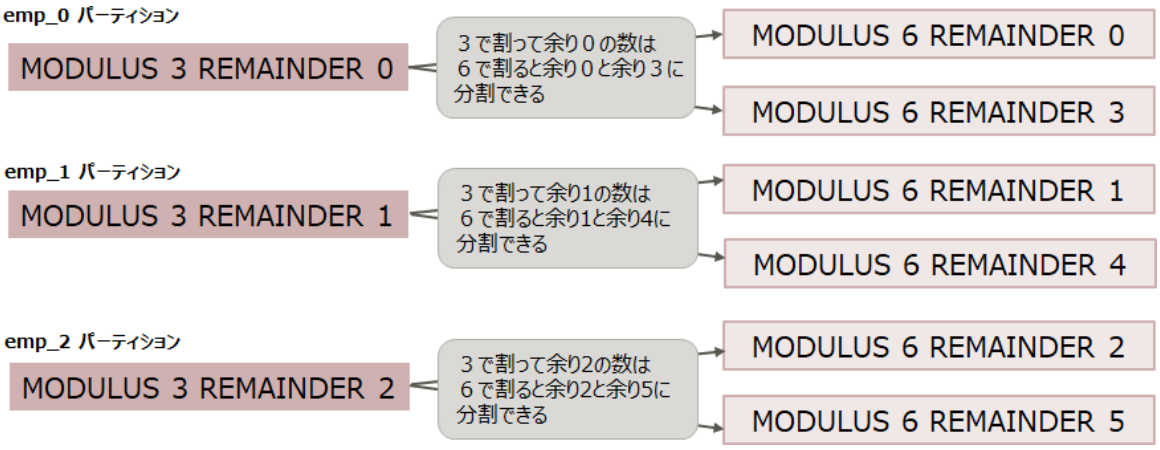
```
mydb=# SELECT * FROM Tokyo;
 emp_id | sales_amount | branch | region
--------+--------------+--------+--------
      1 |          100 | 横浜   | 東京
      7 |            5 | 品川   | 東京
     10 |           10 | 長野   | 東京
      4 |           10 | 静岡   | 東京
(4 行)
```

branch='静岡'の情報が、Nagoya パーティションから Tokyo パーティションに移動していることがわかります。

2.2.3 パーティションの追加

データ量の増加により現状のパーティション数では性能向上が見込めなくなったとき、パーティションの数を増やすことで、さらに細分化したパーティション構成とし、性能向上を図りたい場合があります。レンジパーティションやリストパーティション

は、パーティションキーの範囲や値を新たに指定することで、パーティションを増やすことができます。しかし、ハッシュパーティションは分割数（除算）によってパーティション数が決まるため、レンジパーティションやリストパーティションと同様の方法でパーティションを追加することができません。ハッシュパーティションについて、分割数や剰余を再定義することでパーティションを再分割する例を見てみます。ここでは、「2.1.3 ハッシュパーティションの作成」で作成したハッシュパーティションを使用します。
ハッシュパーティションの分割数は、現在設定されている値の倍数を指定します。以下は、2.1.3 のハッシュパーティションを 3 分割から 3 の倍数である 6 分割にする場合の例です。



図：ハッシュパーティションの分割

以降では、上記に基づき、6 つのパーティションに分割する手順を説明します。なお、パーティションテーブルには、100 万件のデータが挿入されており、各パーティションは以下のようにデータが分割されている状態とします。

```
mydb=# SELECT relname,reltuples as rows FROM pg_class
  WHERE relname IN ('emp','emp_0','emp_1','emp_2') ORDER BY relname;
 relname |  rows
---------+--------
 emp     |      0
 emp_0   | 333263
 emp_1   | 333497
 emp_2   | 333240
(4 行)
```

1. パーティションをパーティションテーブルから切り離し（デタッチ）します。

```
mydb=# ALTER TABLE emp DETACH PARTITION emp_0;
mydb=# ALTER TABLE emp DETACH PARTITION emp_1;
mydb=# ALTER TABLE emp DETACH PARTITION emp_2;
```

2. パーティションのデータをバックアップするため、パーティションをリネームします。

```
mydb=# ALTER TABLE emp_0 RENAME TO emp_0_back;
mydb=# ALTER TABLE emp_1 RENAME TO emp_1_back;
mydb=# ALTER TABLE emp_2 RENAME TO emp_2_back;
```


3. 「図：ハッシュパーティションの分割」に示した MODULUS と REMAINDER の値を指定して 6 つのパーティションを作成します。

```
mydb=# CREATE TABLE emp_0 PARTITION OF emp FOR VALUES WITH (MODULUS 6, REMAINDER 0);
mydb=# CREATE TABLE emp_1 PARTITION OF emp FOR VALUES WITH (MODULUS 6, REMAINDER 1);
mydb=# CREATE TABLE emp_2 PARTITION OF emp FOR VALUES WITH (MODULUS 6, REMAINDER 2);
mydb=# CREATE TABLE emp_3 PARTITION OF emp FOR VALUES WITH (MODULUS 6, REMAINDER 3);
mydb=# CREATE TABLE emp_4 PARTITION OF emp FOR VALUES WITH (MODULUS 6, REMAINDER 4);
mydb=# CREATE TABLE emp_5 PARTITION OF emp FOR VALUES WITH (MODULUS 6, REMAINDER 5);
```

4. 2.でバックアップしたデータをパーティションに戻します。

```
mydb=# INSERT INTO emp SELECT * FROM emp_0_back;
mydb=# INSERT INTO emp SELECT * FROM emp_1_back;
mydb=# INSERT INTO emp SELECT * FROM emp_2_back;
```

5. 2.でバックアップしたパーティションを削除します。

```
mydb=# DROP TABLE emp_1_back;
mydb=# DROP TABLE emp_2_back;
```

6. 以下のとおり、6 つのパーティションにデータが分割されました。

```
mydb=# SELECT relname,reltuples as rows FROM pg_class
 WHERE relname IN ('emp','emp_0','emp_1','emp_2','emp_3','emp_4','emp_5')
 ORDER  BY relname;
 relname |  rows
---------+--------
 emp     |      0
 emp_0   | 166480   6分割
 emp_1   | 166904
 emp_2   | 166302
 emp_3   | 166783
 emp_4   | 166593
 emp_5   | 166938
(7 行)
```

ポイント

データ量が多い場合など、1 度に全パーティションの分割を実施するとデータの移行に時間がかかる場合があります。このような場合は、「emp_0 パーティションを分割後、emp_1 パーティションを分割する」というように、1 つのパーティションごとに分割することもできます。データ量や運用に応じてどの単位でパーティション分割を実施するのかを決めてください。

ここでは、サンプルを使用し、パーティショニングの概要と使い方を解説しました。業務や運用に合わせて適切にテーブルを分割すると、性能の向上とメンテナンス性の向上が見込めるため、パーティショニングの使用をご検討ください。

2020 年 2 月 14 日