1 8.1での新機能解説

本稿ではPostgreSQL 8.1の新機能を紹介します．ビットマップスキャンの導入，min／maxの最適化，マルチカラムインデックスの有効利用，集約関数の性能向上，バッファ管理の改良，テーブルパーティショニング，ロールなどを取り上げます．

SRA OSS, Inc. Japan
石井達夫 ISHII Tatsuo
ishii@sraoss.co.jp

はじめに

PostgreSQLは安定性，機能，日本語による情報の充実などもあって，オープンソースデータベースの中でもとくに高い人気を誇っています．PostgreSQLは使いやすさにも定評がありますが，最近のリリースではエンタープライズ指向の機能が次々に追加され，商用データベースに迫る実用性を備えるようになっています．

PostgreSQLと並んで人気のあるオープンソースデータベースMySQLでは，このあたりの開発姿勢にかなり温度差があり，「主要商用データベースとは競合しない」とMySQL開発者が言明しているのに対し，PostgreSQLははっきりと商用データベースの領域まで踏み込もうとしているのがおもしろいところです．

本稿では，PostgreSQL 8.1の機能や性能面での改善に焦点をあて，PostgreSQLが商用データベースに比してどの位の地平まで到達しているのかを探っていきたいと思います．なお，一部の機能については特集の中で詳細に解説しているので，そちらに譲ります．

前バージョンPostgreSQL 8.0の到達地点

PostgreSQL 8.1について解説する前に，1つ前のバージョンである8.0でPostgreSQLがどこまで到達していたかを簡単に確認しておきましょう．

PostgreSQL 8.0の1つ前のバージョンは7.4でしたので，8.0ではいきなり7.x台から8.x台にバージョン番号が上がったことになります．これは，今まで待望されていた多くの機能が1度に実現されたからです．その意味で，PostgreSQL 8.0は間違いなくエポックメイキングなリリースであったと言えるでしょう．主要な機能だけでも，次のような大物がそろっています．

①Windows対応
②PITR（Point In Time Recovery）
③バッファマネージャの改善
④セーブポイント
⑤テーブルスペース

①で膨大なWindowsユーザがPostgreSQLの世界に取り込まれ，今までPostgreSQLのユーザ層で薄かった「ローエンド」の層が厚くなりました．

一方，②〜⑤はミッションクリティカルなサーバ用途で切望されていた機能です．とくに②は重要で，ディスククラッシュのような事故の際に直近までできちんとデータをリカバリできるようになり，PostgreSQLはミッションクリティカルな用途にもじゅうぶんに耐える品質を持っていると胸を張って言えるようになりました．

まとめると，PostgreSQLは8.0でユーザ層を広げ

ると同時に，商用データベースの領域にいよいよ踏み込んでいったと言えるでしょう．

8.0 で到達した地点に磨きをかけた PostgreSQL 8.1

それでは**PostgreSQL 8.1**のリリースはどうだったでしょうか？

結論から言うと，**PostgreSQL 8.0**の到達した地平に磨きをかけたのが**PostgreSQL 8.1**であると言えます．つまり，**PostgreSQL 8.0**のときのようなユーザにもわかりやすい機能追加こそ少ないものの，内部的な改善が著しく，はじめてトランザクション機能が導入された**PostgreSQL 7.1**以来の画期的な性能改善がなされています．

それではさっそく**PorsgreSQL 8.1**での機能／性能改善を見ていきましょう．

なお，**2**相コミットについては本特集**4**章で詳しく解説しているので，本稿では割愛します．

ビットマップスキャンの導入

PostgreSQLでは，問い合わせを実行する方法（プランタイプ）を複数持っており，その中から問い合わせオプティマイザが最適と思われる方法を選択するしくみになっています．

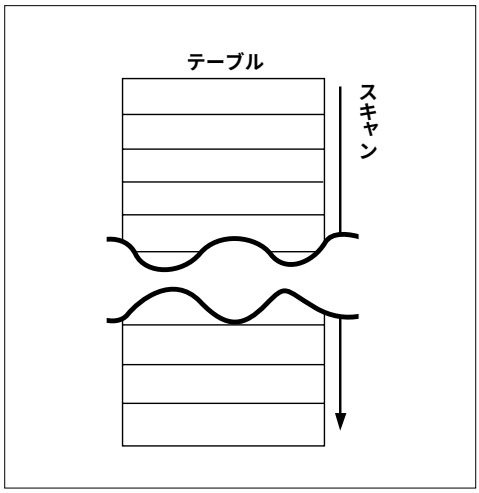
問い合わせのときに参照されるオブジェクトは，おもにテーブルとインデックスです．

PostgreSQLでは，簡単に言うと，テーブルはファイルの中に頭から順に行が詰め込まれた構造になっています．実際にはブロックやページという構造を使ってもう少し複雑な管理をしていますが，論理的には行がファイルの中に順に格納されていると考えてかまいません．

順スキャン

最も単純なプランタイプである「順スキャン」は，テーブルを頭から順に読んで必要な行を取り出します（図**1**）．容易に想像できるように，順スキャンは決して高速ではなく，テーブルの大きさに比例して実行時間が増えてしまいます．

図1●順スキャン



リスト1●複数の条件を使って検索

```
WHERE a = 1 AND b = 1;
```

インデックススキャン

順スキャンに対してインデックススキャンでは，あらかじめインデックス（索引）を作っておいて，まずインデックスを検索し，次にテーブルを検索するという**2**段構えの方法を採ります．

PostgreSQLでよく使われるインデックスは「**Btree**」というタイプで，大量のデータの中から目的のデータを高速に検索できる特徴があります．図**2**に**Btree**インデックスの構造を示します．検索は**1**番の上の箱（「ルートノード」と呼ばれます）から順に行います．たとえば目的の値が**112**であれば，ルートノードから**105**の入っている箱，そして**112**の入っている箱というように，値を順に比較することによって素早く目的のデータを見つけることができます．

PostgreSQLでは，インデックスがあり，かつ検索結果の件数が比較的少ない場合にインデックススキャンを使用します．逆に言うと，検索結果が多いと必ず順スキャンになってしまっていました．

また，リスト**1**のように，複数の条件を使って検索する場合，**a**列と**b**列にインデックスが設定されていても，どちらか一方のインデックスしか使用

されないという問題もありました.

こうした問題を解決するのがビットマップスキャンです.

メモリ上にビットマップを作成

ビットマップスキャンでは，まずインデックスを検索し，得られた結果を元にメモリ上に動的にビットマップを作成します．たとえば，先ほどのリスト1の例で言えば，まず「a = 1」に対応するビットマップ，次に「b = 1」に対応するビットマップを作成します．そして次にそのビットマップ上でANDを取ります．得られたビットマップがテーブル上の目的の行を示しているわけです.

図3にビットマップスキャンを使った問い合わせプランの例を示します.

テーブルアクセスの高速化

ビットマップスキャンのもう1つのメリットは，テーブルへのアクセスが効率的になることです．従来のインデックススキャンでは，インデックス上で目的のデータを見つけるたびにテーブルを見に行き，結果としてテーブルファイルを何度も行ったりきたりすることになります．ビットマップスキャンのビットはテーブル上の行の物理的な並び（TID）順になっていますから，自然とテーブルへのアクセスが順番どおりになり，ディスクのヘッドの移動が最小限に押さえられる結果，テーブルアクセスが高速化されます.

OR検索にも効果を発揮

従来のPostgreSQLでは，リスト2のようなOR条件での検索ではインデックスを使用することができませんでした.

ビットマップスキャンでは，前述のリスト1の検索の例で最後にビットマップ上でANDを取るところをORに置き換えるだけで，簡単にOR検索を実現できます.

リスト2●ORを使って検索

```
WHERE a = 1 OR b = 1;
```

図2●Btreeインデックスの構造

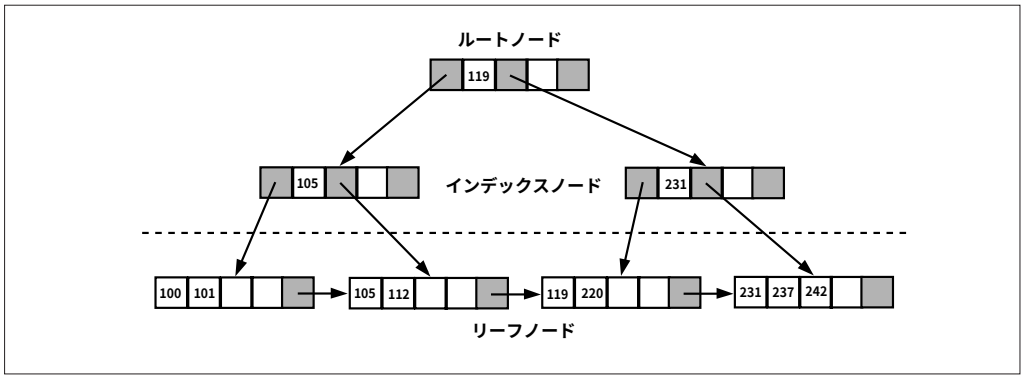


図3● ビットマップスキャンの問い合わせプランの例

```
test=# EXPLAIN SELECT * FROM t1 WHERE aid1 BETWEEN 1 AND 10000 AND aid2 BETWEEN 1 AND 100;
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Bitmap Heap Scan on t1  (cost=88.16..322.03 rows=92 width=8)
   Recheck Cond: ((aid2 >= 1) AND (aid2 <= 100) AND (aid1 >= 1) AND (aid1 <= 10000))
   ->  BitmapAnd  (cost=88.16..88.16 rows=92 width=0)
         ->  Bitmap Index Scan on aid2index  (cost=0.00..9.92 rows=986 width=0)
               Index Cond: ((aid2 >= 1) AND (aid2 <= 100))
         ->  Bitmap Index Scan on aid1index  (cost=0.00..78.00 rows=9333 width=0)
               Index Cond: ((aid1 >= 1) AND (aid1 <= 10000))
(7 rows)
```

図4にこのような問い合わせプランの例を示します．図3のプランと比べると，「BitmapAnd」が「BitmapOr」に変わっただけであることがわかります．

ビットマップスキャンの効果

ビットマップスキャンの実装により，従来順スキャンを使うしかなかったケースでもインデックスが使えるようになり，場合によっては3～5倍以上の性能向上が望めます．

min／maxの最適化

従来インデックスの設定された列に対してmin（最小値）／max（最大値）を求めるテクニックとして，LIMITを使う方法が知られていました．

たとえば，リスト3でaid1列にBtreeインデックスが設定されていれば，リスト4と書き換えることによってminと同等の結果を得ることができます．これは，Btreeインデックスがデータの大きさ順にソートされているという特性があるからで，ソートなしにminを得ることができるわけです．

PostgreSQL 8.1ではこのような小細工をしなくても，自動的に問い合わせの書き換えが行われ，min／maxが高速実行されます（図5）．

マルチカラムインデックスの有効利用

マルチカラムインデックスとは，複数の列を連結したインデックスです．たとえば，リスト5のような検索を高速に行いたい場合に有効です．

ただし，従来のPostgreSQLではこのインデックスは，リスト6のような場合にしか利用できませんでした．つまり，マルチカラムインデックスの先頭列から連続したn列を構成するAND条件にのみマルチカラムインデックスが使えたのです．

リスト3●minを求める

```
SELECT min(aid1) FROM t1;
```

リスト4●minと同等の結果を求める

```
SELECT aid1 FROM t1 ORDER BY aid1 LIMIT 1;
```

リスト5●マルチカラムインデックスの使用例

```
WHERE a = 1 AND b = 2 AND c = 3;
```

リスト6●マルチカラムインデックスを使用できる例

```
WHERE a = 1;
WHERE a = 1 AND b = 2;
WHERE a = 1 AND b = 2 AND c = 3;
```

図4● ORを使用したビットマップスキャンの問い合わせプランの例

```
test=# EXPLAIN SELECT * FROM t1 WHERE aid1 BETWEEN 1 AND 10000 OR aid2 BETWEEN 1 AND 100;
                                    QUERY PLAN
-------------------------------------------------------------------------------------
 Bitmap Heap Scan on t1  (cost=87.91..785.29 rows=10227 width=8)
   Recheck Cond: (((aid1 >= 1) AND (aid1 <= 10000)) OR ((aid2 >= 1) AND (aid2 <= 100)))
   ->  BitmapOr  (cost=87.91..87.91 rows=10319 width=0)
         ->  Bitmap Index Scan on aid1index  (cost=0.00..78.00 rows=9333 width=0)
               Index Cond: ((aid1 >= 1) AND (aid1 <= 10000))
         ->  Bitmap Index Scan on aid2index  (cost=0.00..9.92 rows=986 width=0)
               Index Cond: ((aid2 >= 1) AND (aid2 <= 100))
(7 rows)
```

図5● min関数の最適化

```
test=# EXPLAIN SELECT min(aid1) FROM t1;
                                    QUERY PLAN
-----------------------------------------------------------------------------------------
 Result  (cost=0.02..0.03 rows=1 width=0)
   InitPlan
     ->  Limit  (cost=0.00..0.02 rows=1 width=4)
           ->  Index Scan using aid1index on t1  (cost=0.00..1963.32 rows=100000 width=4)
                 Filter: (aid1 IS NOT NULL)
(5 rows)
```

したがって，リスト7のような検索にはこのマルチカラムインデックスは使えませんでした．もちろん，b列に単独のインデックスを設定すればインデックスが使えるようになりますが，インデックスを増やせばそれだけ更新時の負荷が増えるので，あまり好ましいことではありません．

PostgreSQL 8.1では，そのような制限はなくなり，マルチカラムインデックスに含まれるすべての列のどのような組み合わせに対しても，インデックスが使えるようになりました．

集約関数の性能向上

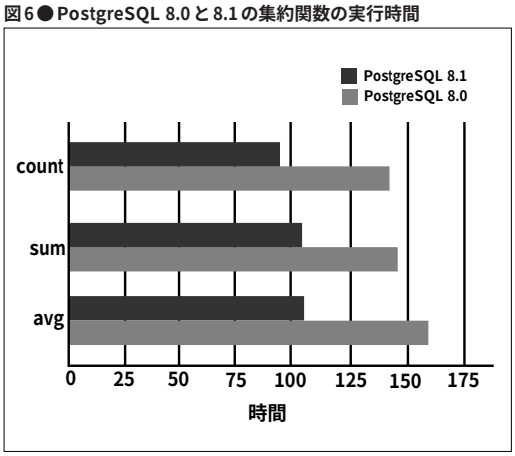
count( )，sum( )，avg( )といった集約関数の性能が向上しました．

この向上は，おもに内部的なメモリ獲得のオーバーヘッドを減らすことによって達成されています．実際にpgbenchで作成した10万件のデータでcount( )，sum( )，avg( )の実行時間を計測してみたところ図6のような結果となり，3割程度の高速化が達成されていることがわかりました．

リスト7●マルチカラムインデックスが使用できない例

```
WHERE b = 1;
```

図6● PostgreSQL 8.0と8.1の集約関数の実行時間



バッファ管理の改良

PostgreSQLでは，共有メモリ上にバッファキャッシュを持ってデータベースアクセスを高速化しています．すべてのデータベース処理がこの共有バッファを使用するため，このバッファ管理の実装によって性能が大きく変化します．

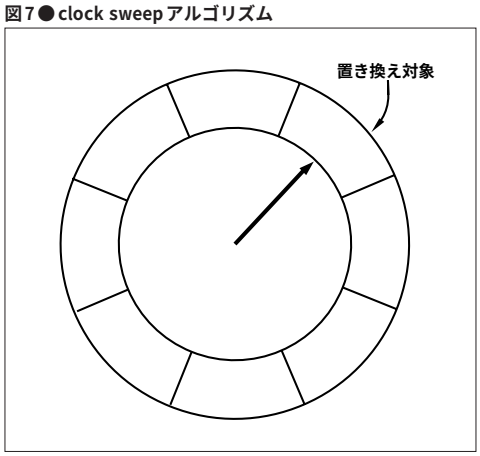
clock sweepアルゴリズムの採用

バッファ管理にはいくつか知られた手法がありますが，PostgreSQL 8.1で採用されたのは「clock sweep」というアルゴリズムです．clock sweepでは，バッファがリング上に連結されており，その上を「時計の針」に相当するポインタが一定方向に回転しています（図7）．

時計の針の指すバッファがちょうど空いていれば，そのバッファを利用します．空いていなければ次のバッファへと針を進めます．こうして，古いバッファから利用されていくことになります．

LRUなどのアルゴリズムに比べると厳密さには欠けますが，LRUではバッファを管理するグローバルな構造体をロックする必要があるのに比べ，clock sweepではロックの範囲を狭くできるので，たくさんのトランザクションが並列して実行している環境では効率の良い管理が可能です．

図7● clock sweepアルゴリズム



ジャイアントロックの回避

PostgreSQL 8.1ではバッファ管理の効率をさらに高めるため，バッファ管理のためのロックを最小限にしています．PostgreSQL 8.0以前では，バッファ管理全体をロックするジャイアントロック（すべてをロックする大きなロック）があり，トランザクションの同時実行を妨げていました．PostgreSQL 8.1では，これがいくつかのロックに細分化され，必要最低限のロックだけを取得するようにしています．

バッファ管理の改良の効果

これらの改良により，従来PostgreSQLの弱点だった次の問題が解決しました．

●SMPシステムでの性能向上

従来PostgreSQLではCPU数が4以上になっても性能が向上しなかった．

●大量メモリの有効利用

物理メモリを大量に搭載したマシンでは，共有メモリバッファも大量に設定することが可能だが，従来のPostgreSQLではバッファ管理のオーバーヘッドにより，共有メモリバッファを増やしすぎるとかえって性能が低下した．

では，実際にどの程度性能が改善されたかですが，筆者が『日経IT Pro』に連載中の「PostgreSQLウオッチ　第23回」[注1]で発表したデータの中からいくつかお見せします．

検証に使ったマシンはAMDのOpteron/2.2GHzを4個搭載し，64ビット版のLinuxを動作させているシステムです．メモリは8Gバイト積んでいます．データはpgbenchで作成した10万件のデータを使いました．

まず検索性能です．図8をご覧ください．縦軸が1秒あたりのトランザクション実行回数，横軸は同時接続ユーザ数です．したがって，縦軸は上に行くほど性能が高く，横軸は負荷の大きさと思ってください．ご覧のように，PostgreSQL 8.1は8.0の2倍近い性能が出ています．

更新処理を含む測定でも傾向は変わりません（図9）．

テーブルパーティショニング

テーブルパーティショニングとは，大きなテーブルを物理的に分割する機能です．テーブルパーティショニングを使用することにより，次のような場合に性能的なメリットがあります．

●テーブルの一部のデータだけを欲しい場合

テーブルをうまく分割すれば，物理的にも欲し

図8●PostgreSQL 8.0と8.1の検索性能の比較

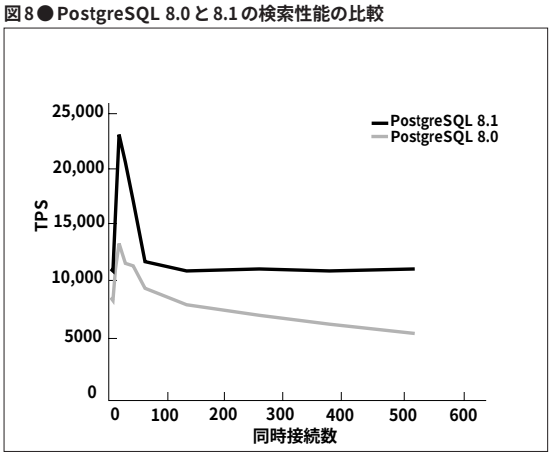
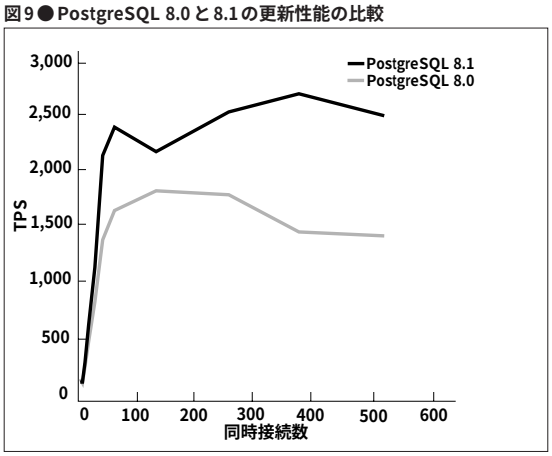


図9●PostgreSQL 8.0と8.1の更新性能の比較



注1　http://itpro.nikkeibp.co.jp/article/COLUMN/20051213/226148/．このWebサイトは会員登録が必要です（会費は無料）．

いデータだけをアクセスできますから，効率が向上します．

●テーブルスペースの併用

大きなテーブルをテーブルスペースを使って複数のディスクドライブに分けることにより，**I/O**スループットが向上します．

CEを使って テーブルパーティショニングを実装

簡単に言うと，**PostgreSQL**のテーブルパーティショニングは**CE**（**Constraint Exclusion**）という機能で実装されています．以降，テーブルパーティショニングを**CE**と呼ぶことにします．

製品マスタをCEしてみる

CEの適用例として，リスト**8**のような「製品マスタ」テーブルを考えます．

ここでは，製品**ID**を基準にしてテーブルをパーテショニングする方針で**CE**を適用してみましょう．

分割先のテーブルを作成

今回は，製品番号**1**～**1,000**，**1,001**～**2,000**，**2,001**～**3,000**の**3**つに製品マスタを分割することにします（リスト**9**）．分割先のテーブルはすべて**master**を継承します．そして，**CHECK**制約で分割する条件を指定します．

最後に，**postgresql.conf**を設定して**CE**を有効にします（リスト**10**）．

CEの効果を検証

これで準備ができました．試しに**master**を検索してみましょう（図**10**）．ご覧のように，**master**への検索は，実際には**master1**への検索に置き換えられていることがわかります．

リスト8●製品マスタテーブル

```
CREATE TABLE master (
       pid INTEGER;          製品ID
       pdate DATE;           製品開発日
       price BIGINT;         標準価格
);
```

リスト10● postgresql.confを設定

```
constraint_exclusion = on
```

リスト9●製品番号1～1,000，1,001～2,000，2,001～3,000の3つに製品マスタを分割

```
CREATE TABLE master1(CHECK (pid >= 1 AND pid < 1000)) INHERITS(master);
CREATE TABLE master2(CHECK (pid >= 1000 AND pid < 2000)) INHERITS(master);
CREATE TABLE master1(CHECK (pid >= 2000 AND pid < 3000)) INHERITS(master);
```

図10● masterを検索

```
test=# EXPLAIN SELECT * FROM master WHERE pid = 1;
                                QUERY PLAN
---------------------------------------------------------------------------
 Result  (cost=0.00..60.75 rows=16 width=16)
   ->  Append  (cost=0.00..60.75 rows=16 width=16)
         ->  Seq Scan on master  (cost=0.00..30.38 rows=8 width=16)
               Filter: (pid = 1)
         ->  Seq Scan on master1 master  (cost=0.00..30.38 rows=8 width=16)
               Filter: (pid = 1)
(6 rows)
```

図11● master1とmaster2にまたがる検索

```
test=# EXPLAIN SELECT * FROM master WHERE pid BETWEEN 999 AND 1001;
                                QUERY PLAN
---------------------------------------------------------------------------
 Result  (cost=0.00..103.35 rows=24 width=16)
   ->  Append  (cost=0.00..103.35 rows=24 width=16)
         ->  Seq Scan on master  (cost=0.00..34.45 rows=8 width=16)
               Filter: ((pid >= 999) AND (pid <= 1001))
         ->  Seq Scan on master1 master  (cost=0.00..34.45 rows=8 width=16)
               Filter: ((pid >= 999) AND (pid <= 1001))
         ->  Seq Scan on master2 master  (cost=0.00..34.45 rows=8 width=16)
               Filter: ((pid >= 999) AND (pid <= 1001))
(8 rows)
```

それでは，**master1**と**master2**にまたがるような検索条件ではどうでしょう（図**11**）？ **master1**に加え，**master2**が検索されました．

このように，**CE**では検索範囲に応じて自動的に必要なテーブルのみを選択していることがわかります．

UPDATEやDELETEはCEに未対応

では，**UPDATE**や**DELETE**ではどうでしょう（図**12**，**13**）？

おやおや，**master1**～**3**まで全部を見に行ってしまっています．残念ながら，今のところ**CE**は**UPDATE**や**DELETE**には対応していないのです．

この問題は，次期バージョンの**PostgreSQL 8.2**で改善されることになっています．

INSERTへの対応

今のままでは，**master**に**INSERT**するとそのまま**master**にデータが登録されてしまいます．この問題には，ルールで対応します．すなわち，**master**への**INSERT**を，分割先の**master1**～**3**までに振り分けるルールを作成します．テーブルが**3**つあるので，ルールも**3**つ作ります（リスト**11**）．

さっそく**INSERT**文を実行してみます（リスト**12**）．

まず**master**を確認します（図**14**）．次に**master1**も

リスト11●テーブルが3つあるので，ルールも3つ作成

```
CREATE OR REPLACE RULE r_master1 AS
         ON INSERT TO master WHERE pid >= 1 AND pid < 1000
         DO INSTEAD INSERT INTO master1 VALUES ( NEW.pid, NEW.pdate, NEW.price);

CREATE OR REPLACE RULE r_master2 AS
         ON INSERT TO master WHERE pid >= 1000 AND pid < 2000
         DO INSTEAD INSERT INTO master2 VALUES ( NEW.pid, NEW.pdate, NEW.price);

CREATE OR REPLACE RULE r_master3 AS
         ON INSERT TO master WHERE pid >= 2000 AND pid < 3000
         DO INSTEAD INSERT INTO master3 VALUES ( NEW.pid, NEW.pdate, NEW.price);
```

図12● UPDATEを試す

```
test=# EXPLAIN UPDATE master SET pid = 1 WHERE pid = 1;
                              QUERY PLAN
-----------------------------------------------------------------------
 Append  (cost=0.00..121.50 rows=32 width=18)
   ->  Seq Scan on master  (cost=0.00..30.38 rows=8 width=18)
         Filter: (pid = 1)
   ->  Seq Scan on master1 master  (cost=0.00..30.38 rows=8 width=18)
         Filter: (pid = 1)
   ->  Seq Scan on master2 master  (cost=0.00..30.38 rows=8 width=18)
         Filter: (pid = 1)
   ->  Seq Scan on master3 master  (cost=0.00..30.38 rows=8 width=18)
         Filter: (pid = 1)
(9 rows)
```

図13● DELETEを試す

```
test=# EXPLAIN DELETE FROM master WHERE pid = 1;
                              QUERY PLAN
-----------------------------------------------------------------------
 Append  (cost=0.00..121.50 rows=32 width=6)
   ->  Seq Scan on master  (cost=0.00..30.38 rows=8 width=6)
         Filter: (pid = 1)
   ->  Seq Scan on master1 master  (cost=0.00..30.38 rows=8 width=6)
         Filter: (pid = 1)
   ->  Seq Scan on master2 master  (cost=0.00..30.38 rows=8 width=6)
         Filter: (pid = 1)
   ->  Seq Scan on master3 master  (cost=0.00..30.38 rows=8 width=6)
         Filter: (pid = 1)
(9 rows)
```

見てみます（図15）．たしかに分割先のテーブルにデータがINSERTされていることが確認できました．

ロール

ロール（ROLE）は，PostgreSQL 8.1から導入された権限管理のしくみです．従来のユーザ／グループによる管理に比べ，きめが細かく柔軟な管理できる特徴があります．

以前は新しいユーザを作ることのできるユーザはスーパーユーザだけでしたが，ロールを使うとユーザを作成できる「管理者」ユーザ（ロール）を作ることができます．すべての権限チェックが適用されないスーパーユーザでは，操作を誤るとデータベースシステムに致命的な問題を引き起こすことも考えられます．ロールを使ってスーパーユーザで仕事をしなければならない場面を最小限に減らすことができます．

ロールの作成

ロールはSQLコマンドの“CREATE ROLE”を使って作成します（リスト13）．これによって“bar”という名前のロールができました．

CREATE ROLE時には図16のような形で属性を指定できます．

作成済みのロールのリストは，psqlの“\du”コマンドで表示できます．

ロールには表1のような属性があります．

リスト12●INSERT文を実行

```
INSERT INTO master VALUES(1, '2006/3/1', 1000);
INSERT 0 0
```

リスト13●ロールの作成

```
CREATE ROLE bar;
```

リスト14●LOGIN属性を持たないロールを作成

```
CREATE ROLE group1;
```

リスト15●fooというユーザ（LOGIN属性を持つロール）を所属させる

```
GRANT group1 TO foo;
```

グループロール

複数のロールをまとめて「グループロール」を定義することができます．

グループロールを作るには，まずLOGIN属性を持たないロールを作ります（リスト14）．これで「group1」という名前のグループロールができました．次にこのグループロールにfooというユーザ（LOGIN属性を持つロール）を所属させます（リスト15）．

ロールの階層管理

グループロールと言っても通常のロールとはなんら区別があるわけではありませんから，グルー

図14● masterを確認

```
test=# EXPLAIN SELECT * FROM master;
 pid |   pdate    | price
-----+------------+-------
   1 | 2006-03-01 |  1000
(1 row)
```

図15● master1を確認

```
test=# EXPLAIN SELECT * FROM master1;
 pid |   pdate    | price
-----+------------+-------
   1 | 2006-03-01 |  1000
(1 row)
```

図16●CREATE ROLEの文法

```
CREATE ROLE ロール名 WITH 属性1, 属性2 …;
```

表1●おもなロールの属性

pg_rolesの列名	意味	初期値	CREATE／ALTER ROLEでのオプション名
rolename	ロール名	CREATE ROLEで指定した値	
rolesuper	スーパユーザかどうか	false	SUPERUSER／NOSUPERUSER
roleinherit	他のロール権限を継承する	true	INHERIT／NOINHERIT
rolecreaterole	他のロールの作成ができるか	false	CREATEROLE／NOCREATEROLE
rolecreatedb	DB作成権限	false	CREATEDB／NOCREATEDB
rolecanlogin	ログイン可能かどうか	false	LOGIN／NOLOGIN
roleconnlimit	データベースへの接続数制限	制限なし	CONNECTION LIMIT
rolepassword	パスワード	なし	PASSWORD
rolevaliduntil	パスワード有効期限	無期限	VALID UNTIL 'timestamp'

プロールにさらにグループロールを所属させ，グループロールを階層的に管理することもできます．たとえば，リスト16のようにすると，ユーザfooは，group1とgroup2の両方に所属することになります．fooが直接所属するgroup1がgroup2のメンバでもあるからです．

ロールの権限の継承

グループロールに所属したユーザは，権限をグループロールから継承します[注2]．

たとえば，リスト17のようにしてgroup1にt1テーブルへのSELECT権限を与えておくと，group1に所属するfooユーザもt1テーブルへのSELECTができるようになります．

SET ROLE コマンド

SET ROLEコマンドは，一時的に権限をあるロールのものに変更します．たとえばリスト18のようにすると，権限がgroup1のものになります．

もともとのfooの権限に戻るためには，リスト19のいずれかを実行します．

その他の新機能

autovacuum

PostgreSQL 8.1では，今までcontrib扱いだったautovacuumが正式にサポートされました．

リスト16●グループロールを階層的に管理

```
CREATE ROLE group2;
GRANT group2 TO group1;
```

リスト17●ロールの権限の継承

```
GRANT SELECT ON t1 TO group1;
```

リスト18●一時的に権限を変更

```
SET ROLE group1;
```

リスト19●もともとのfooの権限に戻す

```
SET ROLE foo;
SET ROLE NONE;
RESET ROLE;
```

SELECT FOR SHARE

行単位の共有ロックがサポートされました．従来の“SELECT FOR UPDATE”も使えます．また，行ロックで“NO WAIT”オプションが使えるようになりました．

トランザクションID周回問題の扱い

PostgreSQL 8.1では，トランザクションIDが周回してデータが見えなくなる問題に対しより確実に対応するため，危険なほどVACUUM頻度が少ない場合にpostmasterが起動できなくなるようになりました．

ユーザ単位のコネクション数の制限

“ALTER USER”コマンドにより，ユーザごとに接続数の制限ができるようになりました．

大量メモリのサポート

2Gバイト以上の共有メモリバッファや，ローカル作業メモリが使えるようになりました．

OUTパラメータのサポート

PL/pgSQLなどのユーザ定義関数で，OUTパラメータがサポートされました．

pg_dumpでのラージオブジェクトサポート

pg_dump／pg_dumpallでラージオブジェクがダンプできるようになりました．

さいごに

PostgreSQL 8.1のおもな新機能／性能改善を見てきました．誌面の関係ですべての変更項目に触れることはできませんでしたが，PostgreSQL 8.1の魅力をかいま見ていただけたと思います．PostgreSQL 8.1がリリースされて以来「8.1は速い！」と言う評判がユーザの間で広まっています．読者の皆さんもぜひご自分でPostgreSQL 8.1の実力を確かめていただけたらと思います．SD

注2　LOGIN，SUPERUSER，CREATEDB，CREATEROLEの各属性はここでの継承の対象とはなりません．