PostgreSQL

Chapter 5

Tips for PostgreSQL

知っておきたいTips集

PostgreSQL

5-1 データベースのバックアップ

実際にPostgreSQLを使ったシステムを運用し始めると，まず考えなければいけないのがバックアップです．データベースに格納されているデータは代替のきかないものが多いので，コンピュータ上の他のデータ，たとえばプログラム本体やOSなどよりも，バックアップをきちんと取ることが重要になってきます．

バックアップにはいくつかの方法があります．

①データベースに格納する前のオリジナルデータが存在する場合はそれを保管し，データベース自体のバックアップは取らない
②OSに付属する汎用ツールを使ってバックアップを取る
③PostgreSQLに付属する専用ツールを使ってバックアップを取る

これらの方法はお互いに矛盾するものではなく，併用することもできます．本当に重要なデータの場合は，できればこれらのすべてを使って冗長にバックアップを取ることをお勧めします．

なお，バックアップを取っている間は，データベースの更新が行われないようにしなければなりません．このためには，①と②の方法では単にpostmasterを停止すればよいのですが，③の方法ではpostmasterを動かし続けなければなりません．対処方法はどのようにPostgreSQLを運用しているかによって異なりますが，最も確実なのは，

q postmasterを停止する
w postmasterを-pオプション付きで起動し，デフォルトの5432以外のポート番号を使用するようにする
e 後述のpg_dumpなどを使う際は②で設定したポート番号で接続するようにする

という方法です．

データベースの更新ができないのはやむを得ないが，せめてデータベースの検索だけは許可したい，という場合は，該当テーブルに対してgrant/revokeを使ってデータベースの更新権限を剥奪します．grantについては第2章を参照してください．

次に，これらのバックアップ方法の利点と欠点を検討してみましょう．

たとえば，4.1節のメール全文検索システムでは，オリジナルのメールファイルを保存しておくことによっていつでもデータベースを再構築できます．このような場合，①の方法が有効です．ただし，create table文などを別途保存しておく必要があります．

PostgreSQLのデータベースは，普通のUNIXファイルなので，tarやdumpなどの汎用ツールでバックアップを取ることができます．たとえば，

```
$ cd /usr/local/pgsql
$ tar cfz data.tar.gz data
```

これでPostgreSQLのデータベース領域全体をバックアップできます．これが②の方法です．この方法の欠点は，特定のテーブルだけを復元するなど，部分的にデータベースを回復することができない点です．また，PostgreSQLはバージョンが変わると物理的なデータベースファイルの互換性がないため，バージョンアップに対応できないことも問題です．利点としては，実績のある汎用ツールを使うので，バックアップの信頼性が高いことが挙げられます．

③のPostgreSQLの専用ツールを使う方法ですが，これにもいくつか方法があります．

5.1.1 copyを使う方法

psqlのcopyを使って，データベースの内容をテキスト形式のファイルにして外部に保存します．ただし，保存されるのはテーブルの内容だけですから，create tableなどのスキーマ定義情報は別途管理する必要があります．この方法は，テーブルの数がそれほど多くない場合に適しています．また，PostgreSQLのバージョンアップの際にも使えます．

5.1.2 pg_dumpを使う方法

PostgreSQLには，pg_dumpというバックアップ専用ツールが付属しています．pg_dumpはデータベース単位やテーブル単位でバックアップを取ることができます．pg_dumpを使えば，スキーマ定義やユーザ関数定義も含めてデータベースの内容をほとんどそのままバックアップし，復元することができます．

pg_dumpの基本的な使い方は以下のようになります．

```
$ pg_dump dbname > db.out
```

dbnameはデータベース名です．db_dumpの出力は，データベースを構築するのに必要なSQL文です．たとえば，新しくtestという名前のデータベースを作り，図5.1.1のようにしてt1というテーブルとインデックスを作ったとします．この状態でpg_dump testを実行すると，図5.1.2のようなSQL文が出力されます．この出力をdb.outというファイルに保存しておけば，testデータベースを誤って消去してしまっても

```
4 createdb test
$ psql -e < db.out
```

のようにしてデータを復元できます．また，PostgreSQLのバージョンが変わった際にもデータを移行できます[注1]．

表5.1.1に，pg_dumpのオプション引数を示します．

●注1
6.4では，char4などのデータ型がなくなりました．このようにサポートされなくなったデータ型を使用している場合は，別のデータ型に修正するなどしてからデータを移行する必要があります．

●図 5.1.1　新しくテーブルとインデックスを作る

```
test=> create table t1(i int);
CREATE
test=> insert into t1 values(1);
INSERT 343657 1
test=> insert into t1 values(2);
INSERT 343658 1
test=> create index t1index on t1(i);
CREATE
```

●図 5.1.2　pg_dump test実行後に出力されたSQL文

```
CREATE TABLE "t1" ("i" "int4");
COPY "t1" FROM stdin;
1
2
\.
CREATE  INDEX "t1index" on "t1" using btree ( "i" "int4_ops" );
```

●表 5.1.1　pg_dumpの引数

-a	データだけをダンプする．スキーマ定義は出力しない
-d	デフォルトではpg_dumpはデータをcopy文としてダンプするが，-dを指定するとinsert文としてダンプする
-D	-dと似ているが“insert into t1(i) values(1)”のように，カラム名付きのinsert文を生成する
-f filename	filenameにダンプ出力する（デフォルトでは標準出力にダンプ）
-h hostname	バックエンドのホスト名を指定
-n	テーブル名やカラム名などのアイデンティファイヤに"（ダブルクォーティション）を付けない（この引数は6.3.2にはない）
-o	object id（oid）情報も一緒に出力する．PostgreSQLではすべてのデータベースオブジェクトはoidという一意の識別子を持ち，-oを指定するとその情報も復元する
-p port	バックエンドのポート番号を指定
-s	スキーマ定義のみを出力
-t table	テーブルtableのみをダンプする
-u	パスワード認証を有効にする
-v	verbose（冗長）モード．各種メッセージを出力
-z	grant/revokeのアクセス権設定情報を出力

5.1.3 pg_dumpで保存されない情報

pg_dumpで保存することができない情報には以下のものがあります.

- viewとrule（6.4ではOKです）
- large object

これらのものについては，オリジナルのデータを保存しておき，そこから復元するしかありません.

5.1.4 pg_dumpall

pg_dumpが1つのテーブルまたは1つのデータベースをバックアップするツールであるのに対し，pg_dumpallはPostgreSQLのデータベースインスタンス全体をバックアップします．pg_dumpallは実際にはpg_dumpを呼び出すスクリプトです．pg_dumpのオプションは全部使えますが，データベース名を指定する必要はありません.

pg_dumpallを使うことにより,

- PostgreSQLのバージョンアップ
- PostgreSQLを運用しているマシンの機種変更

の際にもデータ移行を行うことができます．pg_dumpallの典型的な使い方は以下のようになります.

```
$ pg_dumpall -o >db.out （バックアップ）
        :
        :
$ psql -e template1 <db.out （復元）
```

残念ながら，pg_dumpやpg_dumpallにはまだ取りきれていないバグが残っているようです．少なくとも，6.3.2や6.4では権限[注2]，継承を使ったテーブルが正しく復元されないことがわかっています．実際にpg_dumpやpg_dumpallを使う場合は，必ず事前にテストを行ってデータベースが正しく復旧されるかどうかを確認してください.

●注2
grant/revokeで設定します.

5-2 ベンチマークテスト Wisconsin Benchmarkを使って

PostgreSQLをインストールしてひと通り使いこなせるようになってくると，次に気になるのはどの程度の性能が出ているかです．PostgreSQLにはWisconsin Benchmarkという性能測定のツールが付属しており，これを使ってある程度の性能の目安を得ることができます．Wisconsin Benchmarkは，史上はじめて作られたRDBMS用のベンチマークテストです．詳細は参考文献4をご覧ください．

現在使われているTPCなどの近代的なベンチマークと違って，測定項目が古い，マルチユーザに対応していない，などの限界がありますが，PostgreSQLどうしを相対的に比較する目安にはなります．

PostgreSQL ML in Japanメーリングリストでは，メーリングリストの会員の方から送っていただいたテスト結果をグラフ化して，WWWで公開しています．486マシンからPentiumのデュアルプロセッサマシンまで各種データが揃っているので，自分のマシンと同クラスのデータを見付けて比較し，一喜一憂するなどして楽しんでください:-)

では，ベンチマークの取り方を説明します．準備作業として以下のことを行ってください．

①PostgreSQLのソースにアクセスできることが必要です．また，postmasterが起動中の場合はそれを停止します．
②環境変数PGDATAにデータベースディレクトリを設定します．bashなら

```
$ export PGDATA=/usr/local/pgsql/data
```

csh/tcshなら

```
% setenv PGDATA /usr/local/pgsql/data
```

とします．

5.2.1　ベンチマークテストの実施

6.3.2の場合

```
$ cd /usr/local/src/postgresql-6.3.2/src/test/bench
$ make bench.out
```

Linuxの場合は，bench.outの最後の2行[注1]の，

```
> 36.38user 9.54system 0:52.27elapsed 87%CPU (0avgtext+0avgdata
0maxresident)k
0inputs+0outputs (0major+0minor)pagefaults 0swaps
```

を削除します．

```
$ make bench.out.perquery
```

●注1
マシンによって数値は異なります．6.4でも同様です．

6.4の場合

```
$ cd /usr/local/src/postgresql-v6.4/src/test/bench
```

create.shの13行目の

```
echo "drop database bench" | postgres -D${1} template1 > /dev/null
```

を

```
# echo "drop database bench" | postgres -D${1} template1 > /dev/null
```

に変え，以下を実行します．

```
$ sh create.sh $PGDATA
$ sh runwisc.sh $PGDATA >& bench.out
```

なお，FreeBSDでは最後にコアを吐きますが，ベンチマークデータそのものは取れています．

Linuxの場合は，bench.outの最後の2行にある，

```
> 36.38user 9.54system 0:52.27elapsed 87%CPU (0avgtext+0avgdata
0maxresident)k
0inputs+0outputs (0major+0minor)pagefaults 0swaps
```

を削除します．

```
$ sh perquery < bench.out >&bench.out.perquery
```

5.2.2　ベンチマークデータの見方

筆者が調査したかぎりでは，文献の記述と，実際にPostgreSQLに用意されているテストプログラム／データの間に食い違いがあります．また，一部明らかに誤っているところもあります．というわけであまり厳密な説明をしても意味がないので，ここではごく大雑把に解説することにします．

テスト結果はbench.out.perquery（リスト5.2.1）というファイルに最終的に出力されます．テスト項目は全部で1～32までの32項目で，それぞれデータベースの性能の違った側面を反映するようになっています[注2]．

●注2
なお，query 0:は意味のないデータなので無視してください．また，21，22，24，25番も実際には何も測定していないダミー項目なので，考慮の対象から外します．

一番左の「query 9:」のような番号が番号です．次の「1.247 real」は実際にかかった時間です（秒単位，以下同様）．したがって，この数字が小さいほど性能がよいことになります．その隣の「1.190 user」と「0.060 sys」は，それぞれユーザ空間に費した時間とカーネル空間内で費した時間を表しています．本来はuser＋sys＝realなのですが，FreeBSDではreal以外はおかしな値になっています．とりあえずrealだけ注目すればよいでしょう．

これらのテスト項目はすべてに意味があるわけではありません．テスト項目の中には，実際にはまったく同じことを測定している項目があります．番号で言うと，

```
1 = 3 = 5
2 = 4 = 6
9 = 12
10 = 13
11 = 14
20 = 23
```

```
27 = 30
28 = 31
```

となっています．また，query 7と8はbackend/frontendの通信性能および端末の表示性能を測定するはずの項目ですが，PostgreSQLでは，バックエンドを直接起動しているため，ここでのデータは意味がありません．結局，意味があるのは1，2，9，10，11，15，16，17，20，26，27，28だけということになります．

次にこれら意味のある項目について，どのようなことを調べているのか説明します．

■1，2

単純なSELECT（projection）を行い，結果を別のテーブルに挿入します．1では，10000レコードの中から1％を選択，2では10％を選択しています．選択率が上がるほどシステムに負荷がかかるので，1と2を比較すると負荷によって性能がどう変わるかの目安になります．CPU性能よりはディスク性能が測定結果を左右します．

■9，10，11

複数のテーブルのJOINを行います．9に比べ，11はJOINの条件が複雑です．また，9と11ではJOINの対象となるカラムはあらかじめソートされているので，処理の負荷は軽いはずです．それに対し10は，ソートされていないテーブルとの比較になっています．CPU性能で測定結果が左右される傾向が強いようです．

ちなみに，前述のWWWで公開しているベンチマークデータは，この9番目の項目で優劣を比較しています．

■15，16，17

9，10，11と同じようなテストですが，JOIN対象のカラムがソートされていないので，より負荷が高くなっています．CPU性能だけではなく，ディスク性能も影響します．

●リスト 5.2.1　bench.out.perqueryの例

```
query  0:   0.001 real   0.000 user   0.000 sys
query  1:   0.150 real   0.150 user   0.010 sys
query  2:   1.575 real   0.590 user   0.610 sys
query  3:   0.114 real   0.080 user   0.030 sys
query  4:   1.525 real   0.570 user   0.550 sys
query  5:   0.109 real   0.100 user   0.010 sys
query  6:   0.395 real   0.330 user   0.060 sys
query  7:   0.019 real   0.010 user   0.000 sys
query  8:   0.523 real   0.380 user   0.150 sys
query  9:   1.247 real   1.190 user   0.060 sys
query 10:   2.977 real   2.670 user   0.290 sys
query 11:   1.739 real   1.550 user   0.170 sys
query 12:   1.920 real   1.210 user   0.240 sys
query 13:   2.848 real   2.500 user   0.330 sys
query 14:   1.750 real   1.550 user   0.170 sys
query 15:   3.261 real   1.790 user   1.290 sys
query 16:   3.049 real   2.570 user   0.360 sys
query 17:   3.582 real   2.140 user   1.350 sys
query 18:   2.126 real   1.700 user   0.380 sys
query 19:   0.327 real   0.230 user   0.090 sys
query 20:   0.001 real   0.000 user   0.000 sys
query 21:   0.001 real   0.000 user   0.000 sys
query 22:   0.001 real   0.000 user   0.000 sys
query 23:   0.001 real   0.000 user   0.000 sys
query 24:   0.001 real   0.010 user   0.000 sys
query 25:   0.001 real   0.000 user   0.000 sys
query 26:   0.001 real   0.010 user   0.000 sys
query 27:   0.011 real   0.010 user   0.000 sys
query 28:   0.032 real   0.020 user   0.010 sys
query 29:   0.001 real   0.000 user   0.000 sys
query 30:   0.008 real   0.000 user   0.010 sys
query 31:   0.028 real   0.030 user   0.010 sys
query 32:   0.031 real   0.020 user   0.020 sys
```

■18，19

18は10000レコード，19は1000レコードをそれぞれSELECT INTOにてコピーします．ほとんどディスク性能で結果が決まります．

■20

MIN( )を実行します．

■26

INSERTを実行します．

■27

DELETEを実行します．

■28

UPDATEを実行します．

まとめ

テスト内容は厳密に言うといろいろ問題もありますので，あくまで目安として考えてください．

比較的，実用状態の性能を反映していると思われるのは，9，10，11と15，16，17です．9，10，11はCPU性能が支配的です．それに対し15，16，17はディスクをも含めた，より全体的な性能がより反映していると言えそうです．

5-3 PostgreSQLのパフォーマンス

PostgreSQLに限らず，RDBMSは使い方によってかなり性能が違ってきます．テーブルの設計や問い合わせの書き方における一般的な注意点は，参考文献5などが参考になると思います．

本節ではPostrgeSQL特有の注意事項について述べます．

5.3.1 メモリ

PostgreSQLが使用するメモリには共有メモリとソートバッファの2つがあり，これらの大きさはpostmaster起動時のオプションで指定できます．

共有メモリは，複数のバックエンドプロセスで共有されるメモリ領域で，データベースをアクセスする際のキャッシュとして使われます．postmasterの-Bオプションで使用量を変更できます．たとえば

```
postmaster -B 1024 -S -i
```

-Bの後の数字は共有メモリバッファの数です．バッファ1個あたりの大きさは8192バイトで，デフォルトでは64個のバッファが割り当てられます．この例の場合，1024×8192＝8Mバイトの共有メモリが使われることになります．-Bで指定する数値が大きいほどキャッシュが大きくなり，ディスクアクセスが減るので性能が向上しますが，システムによって使用可能な共有メモリの大きさには制限がありますので，その範囲内で使用してください．

また6.3.2では，-Bに大きな値を指定したときに不具合が出ることが報告されています[注1]．

ソートバッファは，プロセス内のヒープ領域に取られ，複数のバックエンドで共有されないメモリです．ソート処理はORDER BYを指定した場合だけでなく，テーブルの結合の際にも行われるので，ソートバッファを大きくすることで多くの場合性能の

●注1
6.4ではこの問題は解決しているようです．

向上が期待できます．たとえば

```
postmaster -o '-S 1024' -S -i
```

-oの後に''で括った部分はバックエンドへのオプションになります．postmaster自身の-Sオプションと混同しないようにしてください．数値は，ソートバッファの大きさを1024バイト単位で指定します．この例の場合，1024×1024＝1Mバイトのバッファ領域が指定されたわけです[注2]．

ソートバッファは共有メモリではありませんので，かなり大きなサイズが指定できます．もっともその分プロセスサイズが大きくなってしまうので，ほどほどにする必要はありますが．

●注2
デフォルトでは512Kバイトのバッファが使用されます．

5.3.2 fsync( )

UNIXにはfsync( )というシステムコールがあり，OSが管理するバッファとディスクの内容を同期させる働きをします．PostgreSQLは，トランザクションのコミット時にfsync( )を呼び出し，コミットによって確定したデータがディスクに書き込まれることを保証します．ただ，fsync( )はかなりのオーバヘッドになるため，ユーザの判断でfsync( )の呼び出しをやめることができます．ただし，システムが異常終了してしまった場合には，せっかくコミットしたバッファ上のデータがディスクに書き込まれないため，データが失われる可能性があります．したがって，fsync( )をやめるかどうかは，リスクと性能の兼ね合いで慎重に判断する必要があります．たとえば，検索中心の使い方でほとんどデータの更新がない場合にはデータが失われる可能性が少ないので，コミットのたびにfsync( )をしなくてもよいかもしれません．

自動fsync( )をやめるには，postmasterの起動オプションで，

```
postmaster -o '-F'
```

とします．なおメモリ上のバッファとディスクの同期は，通常，OSが定期的に行いますので，PostgeSQLがfsync( )を呼ばないからといって，まったくバッファのデータがディスクに書かれなくなるわけではありません．

5.3.3 vacuum

RDBMSで検索を速くするテクニックとして，インデックスを定義する方法があります．検索対象のカラムにインデックスが設定されていればテーブル本体を見に行かずに済むので，場合によっては検索が劇的に速くなることがあります．

このように「インデックスがあるから，テーブル本体を見るかわりにそちらをアクセスしよう」などという判断はオプティマイザというサブシステムが行います．ただし，オプティマイザが適切な判断を下すためには，そのための適切な情報が提供される必要があります．

PostgreSQLでは，この情報を最新のものにするためにvacuumというSQLコマンドを使います．せっかくインデックスを定義しても，vacuumを実行しておかなければインデックスを見てくれない場合もあります．また，データを大量に更新した場合にもvacuumを再実行したほうがよいでしょう．

vacuumはpsqlから実行します[注3]．

```
psql -c 'vacuum テーブル名' データベース名
```

テーブル名を省略するとデータベース中の全テーブルがvacuumの対象になります．巨大なテーブルがあったりすると非常に時間がかかるので，できればテーブル名を指定したほうがよいでしょう．

またvacuumには，不要なレコードを削除する働きもあります．PostgreSQLでは，updateやdeleteの対象となったレコードは直接更新されず，新しいレコードにその内容が移されたり（updateの場合），削除マークだけ付けてレコード自体は削除されることなく残ります（deleteの場合）．したがって，データベースの更新が頻繁に行われると，不要レコードが溜まってきます．vacuumはこれらのゴミになったレコードを削除します．不要レコードが削除されればテーブルの大きさが小さくなるので，ディスクが節約できるだけでなく，性能が向上する場合もあります．

vacuumの問題点としては，実行時間がかかること，また，vacuum実行中はそのテーブルにアクセスできなくなってしまうことが挙げられます．したがって，深夜などあまりアクセスのない時間帯を狙ってvacuumをかけるほうがよいでしょう．あるいは，1週間に一度，1ヵ月に一度などの割合でデータベースのメンテナンスの時間を取り，バックアップとvacuumを同時にやってしまうことも考えられます．

逆に，非常に更新が頻繁で，しかもデータベースの運用を止められないような用途の場合は，PostgreSQLの採用自体をあきらめなければならないケースもあるかもしれません．

●注3
もちろんプログラミングインターフェースを使ってプログラムの中から行ってもかまいません．

5.3.4 WWWとPostgreSQL

CGIやPHPを使ってWWWサーバとPostgreSQLを連係するような使い方の場合，同じホストでWWWサーバとPostgreSQLを運用すると，WWWサーバプロセスとPostgreSQLのバックエンドプロセスがCPUやメモリなどの各種資源を奪い合うことになり，アクセスが多くなった場合に極端にレスポンスが劣化することがあります．とくに，PostgreSQLはファイルテーブル[注4]を多量に消費する傾向があり，デフォルトでファイルテーブルの少ないFreeBSDの場合，ほとんどシステムが使用不能になることさえあります．

したがって，WWWサーバにアクセスが集中することが予想される場合は，できればWWWサーバとPostgreSQLバックエンドを別々のマシンで動かすことをお勧めします．

●注4
システム内で開かれているファイルに関する情報を保管するためのカーネル内のメモリ領域のこと．

5.3.5 PostgreSQLのバージョンによる性能の違い

PostgreSQLは，今のところバージョンが上がるたびに性能が向上しています．たとえば前述のWisconsin Benchmarkで見ると，6.3.2と6.4を比較すると，まったく同じハードウェアでも6.4の方がかなり性能がよくなっています．また，6.3.2ではインデックスが使われなかったOR検索[注5]でも，6.4ではインデックスが使われるようになりました．したがって，OR検索を使っているアプリケーションでは，6.3.2から6.4に変えただけで劇的に性能が向上することになります．

というわけで，PostgreSQLはなるべく新しいバージョンを使いましょう．

●注5
“select * from foo where i = 1 or i = 2”あるいは“select * fromfoo where i in (1,2)”のような検索のこと．

5.3.6 環境変数PGCLIENTENCODING

PostgreSQL 6.4では，使用するデータベースの文字コード（エンコーディング）が固定ではなく，データベースの生成時に決定されるようになりました．また，PostgreSQLではバックエンドとフロントエンドの文字コードは別々に設定することができます．このため，フロントエンドがバックエンドに接続する際，以下のようなやりとりがあります．

①環境変数PGCLIENTENCODINGが設定されていない場合，データベースの使用している文字コードをフロントエンドでも使うものとする．そのため，フロントエンドはバックエンドにデータベースの文字コードを問い合わせる．また，その結果を環境変数PGCLIENTENCODINGにも設定する．
②フロントエンドは，環境変数PGCLIENTENCODINGをフロントエンドの使用する文字コードとしてバックエンドに伝える．
③フロントエンドの文字コードとバックエンドの文字コードが異なる場合，バックエンドは必要に応じてコード変換を行う．

ご覧のように，環境変数PGCLIENTENCODINGが設定されていない場合，バックエンドにデータベースの文字コードを問い合わせるという，よけいなステップ①が必要になり，頻繁にデータベースへの接続／切断を繰り返す場合に無視できないオーバーヘッドになります．

そこで，事前にデータベースのエンコーディングがわかっている場合には，環境変数PGCLIENTENCODINGを設定しておくことをお勧めします．たとえば，データベースの文字コードが日本語EUCで，フロントエンド側も同じ日本語EUCを使用する場合，bashであれば

```
$ export PGCLIENTENCODING=EUC_JIS
```

cshおよびtcshであれば

```
% setenv PGCLIENTENCODING EUC_JIS
```

とします．

5.3.7 更新トランザクションの競合

PostgreSQLでは，排他制御の単位はテーブル単位であり，データベースの整合性を保つためにあるトランザクションがテーブルを更新中は，そのテーブルをアクセスするすべてのトランザクションが待たされます．たとえば，Webページへのアクセス数を管理する以下のようなテーブルがあったとします．

```
create table www_counter (
       url text,       -- Web page の URL
       int cnt         -- アクセス数カウンタ
```

)

あるURL http://www.foo.co.jp/のカウンタを更新するトランザクションAと，URL http://www.bar.co.jp/のカウンタを更新するトランザクションBは同時に実行できず，どちらかが更新を終了するまで実行を待たされます[注6]．

このようなケースでは，異なるURLに対応するレコードはお互いに無関係なので，同時に更新処理ができてもよさそうなものですが，残念ながらPostgreSQLではそうなっていません[注7]．

この問題に対処するためには，以下のような方法が考えられます．

●注6
これは更新の場合だけの話で，単にwww_counterのレコードを読むだけのトランザクションは並行して実行されます．

●注7
将来的には，このような同時更新の機能を盛り込む予定はあるそうです．

■更新トランザクションは極力短時間で終了する

トランザクションの中でユーザからの入力待ちをするなどは論外です．また，とくにレスポンスが要求されない場合は，更新対象のカラムに対して不必要なインデックスを作成しないようにしましょう．インデックスを設定した場合，検索は高速化されますが，更新は遅くなるからです．

■ひとつのテーブルに更新が集中しないようにする

www_counterの例のように，各レコードの独立性が高く，また集約関数（sum，avgなど）を使う必要がない場合は，思い切ってテーブルを分割することも考えられます．たとえば，URLの各文字を数値として合計し，得られた数値の下位1桁の値が0～9までに応じて10個のテーブルに分割します[注8]．これだけでアクセスの集中が1/10に軽減されます．

●注8
これは「ハッシュ」という考え方に基づいた方法ですが，ほかにも方法は考えられます．要は，アクセスができるだけ均等に分散するようにテーブルを分割できればよいわけです．

5-4 PostgreSQLの問題点

どんなソフトウェアも完全というものはありません．6.4にバージョンアップしたPostgreSQLでは，6.3.2の多くのバグが解消されていますが，残念ながら取り切れなかったバグや制限事項があります．また逆に，6.4で新しく入ってしまったバグも見受けられます．ここでは，PostgreSQL 6.4の問題点を説明します[注1]．なお，開発側が認識している問題点に関しては，

```
/usr/local/src/postgresql-v6.4/doc/TODO
```

にも記述されています．ただし，このドキュメントにはすでに対応済みにもかかわらず，問題点として上げられているものもあるので注意してください．

●注1
本稿執筆時点ではまだですが，おそらくこの本が世に出るころには6.4のバグ修正版である6.4.1がリリースされていることと思います．6.4.1ではここに述べたバグのうち，かなりの部分が修正されるはずです．

5.4.1　ファイルテーブルエントリの不足

5.3.4でも触れましたが，同時接続ユーザが増えた場合，ファイルテーブルを使い果たして事実上システムが使用不可能になってしまいます．対策としては，

①カーネルの設定を変更し，使用可能なファイルテーブルエントリを増やす．
②PostgreSQLをリコンパイルし，同時使用可能ユーザを減らす．

という方法があります．①の方法はシステムによって異なるので，ここでは具体的には触れません．

②ですが，PostgreSQLの同時使用可能ユーザは，デフォルトでは32（6.3.2）または64（6.4）に設定されています．これを減らすわけです．変更方法ですが，src/include/storage/sinvaladt.hに

```
#define MaxBackendId 32                    /* maximum number
of backends            */
```

という行がありますので，32または64という数字を適当に減らします．
もしpostmasterが動いているのなら停止してから，6.3.2であれば

```
$ cd /usr/local/src/postgresql-6.3.2/src
```

6.4であれば

```
$ cd /usr/local/src/postgresql-v6.4/src
```

とし，

```
$ make clean
$ make
$ make install
```

でPostgreSQLを再インストールしてからpostmasterを再起動すればOKです．

5.4.2 オプティマイザがメモリを使い果たす

非常に複雑な問い合わせ，たとえば

```
select * from foo where a = 1 and b = 1 and ...（以後20個程続く） ... and z = 1
```

のような問い合わせを実行すると，異常に実行に時間がかかったり，途中でメモリを使い果たしてバックエンドが停止してしまいます．これはGEQOという，問い合わせを最適化するモジュールが最適な解を出すための計算量が非常に多くなってしまうからです．このような状況はGEQOの調整パラメータを適当に設定することにより回避できる場合があります．

まず，/usr/local/pgsql/data/pg_geqo.sampleをpg_geqoという名前でコピーします．

```
$ cd /usr/local/pgsql/data
$ cp pg_geqo.sample pg_geqo
```

次にこのファイルの後ろのほうにある

```
#Generations            200
```

の行頭の#を消し，200という数字を適当に小さく（たとえば20くらい）します．

```
Generations          20
```

postmasterの再起動は必要ありません．以後，接続し直したセッションからこの処置が有効になっているはずです．

5.4.3 正しくバイナリデータが取得できない

binary cursorを使うと正しくバイナリデータが取得できない場合があります．そのようなときは第2章で述べたように，CD-ROM付属の修正パッチを適用してください．

5.4.4 large objectをcreateしようとするとエラーに

これは6.3.2では発生しない現象です．また，同じ6.4でもFreeBSDでのみ起きる確認で，筆者の手元のLinuxPPCでは再現しません．ですが，ソースを見ると明らかにおかしいので，これも第2章で述べたように，CD-ROM付属の修正パッチを適用してください．

5.4.5 pgdump_allのバグ

マルチバイト拡張が有効なとき[注2]，pgdump_allが正しいcreate database文を出力しません．第2章で述べたCD-ROM付属の修正パッチを適用してください．

●注2
configure --with-mb=を指定したときです．

5.4.6 FreeBSDとTcl

FreeBSDで，ja-tcl-7.6/ja-tk-4.2を併用してTclを有効にしようとすると，configureでエラーになります．6.4のみの問題です．第4章で述べた方法で回避してください．

5.4.7　libpgtcl.soがロードできない

libpgtcl.soをロードしようとすると，cryptが見つからないというエラーになります．これも第4章で述べた方法で回避してください．

5.4.8　対応方法がわかっていないバグ

現象が確認されているものの，まだ修正方法がわかっていないバグもあります．ここではその現象のみを列挙しておきます．

①`select * from pg_shadow where (usesysid is null and oid is null);`はOKだが，`select * from pg_shadow where not (usesysid is null and oid is null);`はバックエンドが落ちてしまう．

②pg_dumpで，grant/revokeで設定される権限および継承を使ったcreatetableの情報がうまくセーブされない．

③“xinv”で始まるテーブルを作ることができるが，large objectを作った際に自動的に作られるテーブル名と混同してしまう可能性がある[注3]．

④2／3次元の配列の扱いがうまくいかない場合がある．

⑤`select a[1] from test;`は駄目で，`select test.a[1] from test;`とする必要がある．

⑥char( )とvarchar( )の配列が作れない[注4]．

⑦`UPDATE table SET table.value = 3;`は駄目で，`UPDATE table SET value = 3;`とする必要がある．

⑧副問い合わせを使ったviewが定義できない．

⑨トランザクションがアボートした際にメモリが正しく解放されない．

●注3
これについては“xinv”で始まるテーブル名を使わないようにするしかありません．

●注4
こうしている理由はよくわかりませんが，ソースを見ると明示的に禁止しています．対策としては，かわりにtext型で配列を定義してください．

5-5 PostgreSQL 6.4で追加された機能

PostgreSQL 6.4は1998年11月にリリースされました．ここでは，新たに追加されたものの中からとくに興味深い機能を取り上げて紹介します．

5.5.1 PL/pgSQL

PL/pgSQLは，SQL言語を使用するユーザ定義関数です．第3章で紹介したSQL関数と似ていますが，機能が大幅に強化されています．

- 制御構造が記述できる
- 組み込み関数やユーザ定義関数を呼び出せる
- trusted functionなのでfunctional indexにも使用できる

PL/pgSQLで書いた関数とその実行例を図5.5.1に示します．

●図5.5.1 PL/pgSQLで書いた関数

```
create function odd_even(int) returns text as '
        declare
                num alias for $1;
        begin
                if num % 2 = 0 then
                        return ''偶数です'';
                else
                        return ''奇数です'';
                end if;
        end;
' language 'plpgsql';
CREATE

select odd_even(2);
odd_even
--------
偶数です
(1 row)
```

PL/pgSQLのインストール

PL/pgSQLを使うには，create language文によってPL/pgSQL言語をデータベースに登録する必要があります．読者の皆さんのサイトで，もし今後標準的にPL/pgSQLを利用するなら，template1データベースにPL/pgSQLを登録すれば，以後作成されたデータベースではcreate language文を発行することなく，すぐにPL/pgSQLを使うことができます．

psqlなどを使って，リスト5.5.1のSQL文を実行してください．

PL/pgSQLの文法

PL/pgSQLの文法の詳細は，PostgreSQL付属ドキュメントProcedural Languagesの“PL/pgSQL”をご覧ください．ここでは概要を説明します．

■文字とコメント

PL/pgSQLでは，大文字／小文字が区別されません．

コメントはSQL同様，--で始めるスタイル（double dash comment）と，C言語同様の/*…*/（block comment）が使えます．

定数はおおむねそのまま書けますが，文字列だけは'ではなく，''で囲んで記述します．

▶例： ''abc''

■プログラムの構造

PL/pgSQLで書かれたプログラムは，以下のような構造を取ります．

```
[<<label>>]
[DECLARE
    declarations]
```

●リスト5.5.1　PL/pgSQLの登録

```
create function plpgsql_call_handler() returns opaque
        as '/usr/local/pgsql/lib/plpgsql.so'
        language 'C';

create trusted procedural language 'plpgsql'
        handler plpgsql_call_handler
        lancompiler 'PL/pgSQL';
```

```
BEGIN
    statements
END;
```

declarationsは変数の宣言です．statementsが実行文です．statementsは普通の文だけでなく，入れ子上にBEGIN...END;（ブロック）を書くことができます．

関数からの戻りは

```
RETURN 値;
```

または

```
RETURN 変数名;
```

です．

■宣言

宣言（declarations）には，変数や参照するカラムやタプルの宣言を書きます．宣言なしに変数を使うことはできません．例外は関数の引数で，$1 $2 $3...などの変数名が自動的に割り当てられます．

宣言には複数の書式があります．

● 変数名 [CONSTANT] 型 [NOT NULL] [DEFAULT := 値]

CONSTANTを指定すると，実行時に値を変更できなくなります．NOT NULLを指定すると，NULLをその変数に代入したときにエラーになります．なお，変数の初期値はNULLなので，NOT NULLを指定した場合には，必ずDEFAULTで初期値をNULL以外にしなければなりません．型はシステム組み込みのデータ型か，以下のものが使えます．

● 変数名 %TYPE…宣言済みの変数と同じ型
● 変数名　テーブル名 . カラム名 %TYPE…指定テーブル . カラムと同じ型

▶ 例：

```
i int;
t text;
j cont int not null default := 1;
ii i%TYPE;
jj words.word%TYPE;
```

● name ALIAS FOR $n;

関数の引数$nをnameという名前で参照できるようにします．プログラムを見やすくするのに使うほか，引数がcomposit type（タプル）の場合には，必ずaliasを使います．

▶ 例：

```
name alias for $1;
```

● name class%ROWTYPE;

変数nameをclassで指定されるテーブルまたはviewと同じ構造を持つものとして宣言します．テーブルのカラムは.を用いてアクセスします．

▶ 例：

```
i int;
foovar foo%ROWTYPE;

i := foovar.int_column;
```

● name RECORD;

特定の型を持たない変数を宣言します．SELECTを実行した結果を保持する場合にバッファとして用います．

▶ 例：

```
buff RECORD;

select into buff * from foo;
```

なお，検索結果が2件以上ある場合，2件目以降は無視されます．

■式，expression，関数呼び出し

代入文が:=となる以外は普通のSQL文と同じです．各種演算子も普通に使えます．関数の呼び出しは，代入文の右辺に書くことで実行されます[注1]．

▶ 例：

```
t text;

t := ''この PostgreSQL のバージョンは以下です：''||version();[注2]
```

●注1
内部的にはSELECT文が実行されます．

●注2
||は文字列連結演算子です．

■特殊変数FOUNDと例外

FOUNDという名前の特殊変数があり，SELECTの実行結果，該当データが存在すればtrueになります．

▶例：

```
SELECT * INTO myrec FROM EMP WHERE empname = myname;
IF NOT FOUND THEN
    RAISE EXCEPTION ''employee % not found'', myname;
END IF;
```

ちなみにRAISEは例外処理で，内部的にはelog( )を呼び出しています．EXCEPTIONを指定するとトランザクションをアボートします．EXCEPTIONのほか，DEBUG（デバッグメッセージの出力），NOTICE（通常メッセージの出力）が選択でき，DEBUG，NOTICEの場合には関数の実行は継続し，トランザクションもアボートされません．その次の文字列はメッセージのフォーマット指定です．%の部分に後続の変数の内容が代入されます．

なお，変数の部分に直接文字列を書くような，

```
RAISE EXCEPTION ''employee % not found'', ''John'';
```

は許されていないようです．

■条件判断

条件判断の構文は以下です．

```
IF expression THEN
    statements
[ELSE
    statements]
END IF;
```

■ループ

```
[<<label>>]
LOOP
    statements
END LOOP;
```

この形のループは，何もしなければ無限ループになってしまうので，明示的にEXITを発行してループを脱出します．EXITの引数にラベルを指定すれば入れ子になったループを脱出できます．なおEXITは

```
EXIT [ label ] [ WHEN expression ];
```

の形でexpressionに脱出条件を書くことができます．

```
[<<label>>]
WHILE expression LOOP
    statements
END LOOP;
```

expressionがtrueの間ループを実行します．

```
[<<label>>]
FOR name IN [ REVERSE ] expression1 .. expression2 LOOP
    statements
END LOOP;
```

nameをループ変数にして，開始条件のexpression1から終了条件のexpression2までループを回ります．nameは整数型で，ループの間だけ生存する変数として扱われます．ループを1回まわるごとにnameは+1されます．

```
[<<label>>]
FOR record | row IN select_clause LOOP
    statements
END LOOP;
```

select_clauseで実行した検索結果を，1行ずつRECORD型またはROWTYPE型で宣言した変数に入れながらループを回ります．

▶例：

```
w words%ROWTYPE;

for w in select * from words loop
      raise notice ''word: %'',w.word;
end loop;
```

PL/pgSQLでできないこと

PL/pgSQLでは以下のことができません.

■トランザクションの扱い

PL/pgSQL関数の中でbegin，commit，abortなどはできません.

■テーブルを返す関数

たとえば，fooというテーブルがあり，そのタプルを返すような関数

```
create function bar() returns foo as ...
```

は作れません．また，SQL関数では可能だったタプルの集合を返すような関数をsetofを使って作ることもできません.

なお，前述のように引数にタプルを取るような関数を作ることは可能です.

■配列を返す関数

配列の一部や全部を返す関数は作れません.

例題

前述のドキュメントに例題があります．その他，src/pl/plpgsql/test/やsrc/test/regress/sql/plpgsql.sqlをご覧になるとよいでしょう.

5.5.2 view/rule システム

これは正確には新しい機能ではありませんが，view/ruleシステムが大幅に書き換えられ，従来「使えない」と言われていたPostgreSQLのview/ruleがかなり実用的なものになりました.

PostgreSQLでは，viewもruleシステムという機能によって実現されています[注3]．そこで，本節ではまとめてお話しすることにします.

viewは仮想的なテーブルで，たとえば以下のように定義します.

```
create view rtest_v1 as select * from rtest_t1;
```

●注3
PostgreSQLではviewはSQL92の定義のサブセットになっています．とくに重要な違いはviewは更新できないことです．しかし後で述べるように，PostgreSQLではruleを使って，見かけ上更新可能なviewを作ることができます.

なお，rtest_t1は，

```
create table rtest_t1 (a int4, b int4);
```

により，すでに存在しているものとします．

これにより，実際には存在しないrtest_v1というテーブルを検索するSQL文

```
select * from rtest_v1;
```

が実行できるようになります[注4]．

このとき，PostgreSQLでは内部的に

```
create rule "_RETrtest_v1" as on select to rtest_t1 do instead
        select * from rtest_v1;
```

というruleが定義されます．ご覧のように，create rule文では，

```
select * from rtest_v1
```

というアクションがあったときに，代わりに

```
select to rtest_t1
```

を実行するという定義が作成されます．

ところで，このままではrtest_v1はINSERTやUPDATE，DELETEなどの更新処理ができません．なぜなら，create viewではINSERTやUPDATE，DELETEをしたときに，代わりに何をするのか定義されていないからです．逆に言うと，rtest_v1に対する更新処理を定義することにより，見かけ上更新可能なviewを作ることができます．リスト5.5.2で，rtest_v1に対してINSERT/UPDATE/DELETEの動作が定義されたことになります．ここで，currentとnewは予約語で，それぞれ現在の値および新しくセットされる値を表しています．

●注4
作成されたviewの定義はpg_viewsというシステムカタログをSELECTすることにより，確認することができます．実はpg_views自身もviewです．ちなみに，6.4ではpg_user，pg_rules，pg_views，pg_tables，pg_indexesというviewがあらかじめ定義されています．

●リスト5.5.2　rtest_v1に対する更新処理の定義

```
create rule rtest_v1_ins as on insert to rtest_v1 do instead
        insert into rtest_t1 values (new.a, new.b);

create rule rtest_v1_upd as on update to rtest_v1 do instead
        update rtest_t1 set a = new.a, b = new.b
        where a = current.a;

create rule rtest_v1_del as on delete to rtest_v1 do instead
        delete from rtest_t1 where a = current.a;
```

ところで，UPDATEとDELETEの定義にはどちらも`where a = current.a`が使われていますが，これはなかなか微妙な問題を含んでいます．

今，rtest_v1には

```
a| b
-+--
1|11
2|12
2|13
```

のようなデータが入っているとします．

```
delete from rtest_v1 where a = 1;
```

は素直にa=1,b=11の行が削除されます．では，

```
delete from rtest_v1 where b = 12;
```

ではどうでしょう．deleteのrule定義では，`where a = current.a`となっていますが，この場合，まずb=12のタプルを検索します．すると，a=2,b=12の行が見付かります．そこでこのタプルを`current`とします．すると`current.a = 2`ですから，`where a = current.a`は`where a = 2`に読み変えられますので，

```
delete from rtest_t1 where a = 2;
```

が実行され，結局a=2,b=12とa=2,b=13のタプルが削除の対象になります．

ところで，商用データベースにはtriggerという機能が備わっているものがあり，ruleと同じような目的で使われます[注5]．triggerも何らかのアクションがあったときの動作を定義する点では同じですが，検索を行ったときに，何らかのデータ更新を行うような動作を定義できず[注6]，ruleに比べると一般性に欠けるとされています（参考文献1参照）

また，PostgreSQLのドキュメントによれば，オプティマイザはruleから多くの情報を得ることができるため，オプティマイズできる可能性が高い点もruleのメリットであるとしています．

このように優れた点の多いruleですが，6.4がリリースされるまではほとんど使えない状態でした．これはruleというものが非常に強力な半面，セマンティクスが難しいこともひとつの原因だと思われます．先に「微妙な問題」の例を挙げましたが，このような問題ひとつひとつについて適切な仕様を定め，実装を行うのは大変なことです．

●注5
PostgreSQLはruleだけでなく，triggerも持っています．

●注6
たとえば何か重要なテーブルがあり，それを参照（SELECT）したときにログテーブルに記録を残すような処理が該当します．

またruleシステムがパーサやプランナ，オプティマイザと深く結び付いていることがいっそう難しさを助長しています[注7]．

しかし，それらの困難を乗り越え，6.4でよやくruleシステムが使えるようになったのは本当に嬉しいことです．ユーザ定義関数やデータ型と並んでruleシステムはPostgreSQLの最大の武器と言えると思います．今まではあまり注目されることのなかったruleシステムですが，6.4のリリースをきっかけにして広く活用されるようになることを期待します．

●注7
このあたりの苦労話は，ドキュメントの“The PostgreSQL Rule System”からも伺えます．

5-6 PostgreSQLの開発体制と今後の予定

本節では，PostgreSQLの開発体制と今後について触れます．

5.6.1　PostgreSQLの開発体制

PostgreSQLの開発は完全にボランティアベースで，特定の企業や組織の干渉はまったく受けていません．開発用のマスターソースはcvsというツールにより，Marc G. Fournier氏の提供するサーバ上で管理されていますが，このサーバもMarc G. Fournier氏が自費で提供しているものです．

開発はインターネットを利用した分散開発体制が取られており，世界中に開発者がいます．ソースに付属のdoc/TODOを読むとわかるように，非常に多くの人たちが開発に参加していますが，古くからPostgreSQLの開発に関わっており，重要な貢献をしているという意味では以下の方たちが筆頭に上げられるでしょう．

■ Marc G. Fournier氏

先に述べたように，開発用のサーバマシンを提供しているほか，WWWサーバやFTPサーバ用のマシンも提供しています．PostgreSQLがスケジュール通りに開発されるように[注1]まとめ役をされています．

●注1
なかなか予定通りにはいかないようですが…．

■ Bruce Momjian氏

PostgreSQL全般について深い知識を持ち，多くの重要な貢献をされています．そのパワフルさは，いったいいつ本業をしているのかと思うほどです:-)

■ Thomas Lockhart氏

日付関連のデータ型，パーサ周りのエキスパート．SQL92への準拠は氏の貢献によるところが多いようです．また，ドキュメントのまとめ役でもあります．

■Vadim B. Mikheev氏

エキュゼキュータ，トランザクション管理など，データベースエンジンの中枢部分について重要な貢献をされていますが，とくにsubselect（副問い合わせ）の実装は氏の力によるものです．今後は，PostgreSQLのトランザクション管理の大幅な改良を計画されているそうなので，楽しみです．

◆ ◆ ◆ ◆ ◆

開発者どうしの連絡はメーリングリストによって行われます．開発の方針決定もこのメーリングリスト上の議論を通じて行われます[注2]．PostgreSQLの開発に参加したい人は誰でもこのメーリングリストを購読して質問したり，意見を言うことができます．詳しくは，http://www.postgresql.org/をご覧ください．

開発者の中でもアクティブな人たちには，このマスターソースへの書き込み権限が与えられています[注3]．それ以外の人でも，PostgreSQLに何らかの有用な改造を加えたい場合は，パッチとしてメーリングリストを通じて発表することができます．このパッチはコミッターが承認すれば，オリジナルソースに反映されます．私自身もこの方法でPostgreSQLの開発に参加しています．

このように，比較的オープンな方法で運営されていますが，現在のところこの開発方法はうまく機能しているようです．

●注2
おそらく開発者どうし，直接顔を合わせたことはないものと思われます．

●注3
コミッターと呼ばれます．

5.6.2 今後のPostgreSQL

今後のPostgreSQLの開発ですが，そのときどきに開発者自身が最もやりたいことを実装していく，というの基本なので，メーカ製のソフトのようにきちんとしたロードマップがあるわけではありません．というわけで，あくまで筆者の主観的な見方ですが，ここ1～2年は以下のような項目がターゲットになるのはないかと思っています．

Windows NTへの対応

従来WindowsではODBCドライバを使ってUNIXのバックエンドに接続する形でしかPostgreSQLが利用できなかったのですが，6.4ではlibpqがWin32に移植されたため，多くのプログラムインターフェースがWindowsでも使えるようになりました．すでに，libpq＋Tcl/Tkの動作が確認されており，Perlなども動くのではないかと思います．

そこで次のステップはバックエンドもWindowsで動かそうということになるわけで

すが，実は6.4の開発段階ですでにWindows NTへの移植の完成度はかなりのところまで来ており，regression testもいちおう通るまでになっています．ただ，共有メモリ関係などでまだ不安なところがあり，実用的に使うには改善の余地があるそうです．

LLL（Low Level Locking）の実装

5.3.7で述べたように，今のPostgreSQLでは排他制御の単位がテーブルであるため，どうしても更新処理の性能が上がりません．根本的な解決としては，タプルやページ[注4]単位の排他制御＝LLLの実装が望まれます．また，SQL92のトランザクション管理の構文[注5]が実装されれば，よりいっそう肌理細かくトランザクションの性質をコントロールできるため，より効果的です．これらについてはPostgreSQLの中枢部分の大改造が必要になりますが，すでにVadim B. Mikheev氏によってかなり実装が進んでおり，次期バージョンの6.5で搭載される予定です．

●注4
データベースの物理的な記憶単位です．

●注5
SET TRANSACTION…の構文．SET TRANSACTION READ ONLY，SET TRANSACTION ISOLATION LEVELなどがあります．詳しくは参考文献をご覧ください．

タプルサイズの制限の緩和

PostgreSQLでは，タプル（レコード）の大きさは8192バイトを超えられません．この制限を超えるようなデータは現状ではlarge objectに格納するしかないわけですが，large objectでは使い勝手や性能に制約があるため，やはり大きいデータも普通のデータ型として扱えることが望まれます．

スケーラビリティ

同時ユーザ数が増えたときの問題点，とくにWWWとの連係における問題点についてはすでに述べた通りですが，この点に関する議論はむしろ本家のメーリングリストよりも日本の方が盛んです．バックエンドをマルチスレッド対応にする，テーブルをUNIXのファイルにマッピングするのではなく，データベース全体を1個のファイルにマッピングするなどの対応もありますが，筆者の考えでは，商用データベースと同様の手法，すなわち「TPモニタ」をフロントエンドとバックエンドの間に入れることが必要ではないかと思います．ちなみにTPモニタは，フロントエンドからの複数の接続要求をまとめ，フロントエンドへの接続数を減らして負荷を軽減する働きをします[注6]．今後の検討課題と言えそうです．

●注6
TPモニタの役割は他にもありますが．

5-7 インターネット上の関連情報

本節では，インターネット上のPostgreSQLに関する情報ポインタを紹介します．

5.7.1 PostgreSQL関連

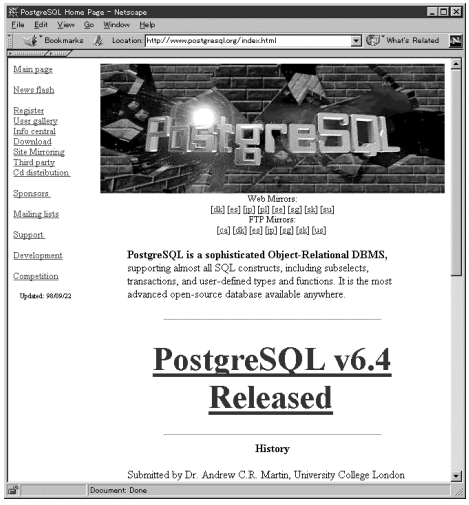
■http://www.postgresql.org/

PostgreSQL開発チームのWWWサイト（図5.7.1）．PostgreSQLに関するオフィシャルな情報はここから得ます．また，PostgreSQLに関する各種メーリングリストを購読することができます[注1]．現在，表5.7.1のリストがあります．

購読方法はメーリングリストによって異なりますので，Webページを参照してください．また，Webページからこれらのメーリングリストのアーカイブを検索することができます．

●注1
当然ですが，メッセージはすべて英語です．

●図5.7.1
PostgreSQLのオフィシャルサイト



●表 5.7.1　本家で開催している各種 ML

pgsql-admin	PostgreSQLのインストールや管理に関する議論
pgsql-users	PostgreSQLの利用者のためのリスト
pgsql-hackers	PostgreSQLの開発者のためのリスト．インストールや利用方法に関する質問はできない
pgsql-interfaces	主としてPostgreSQLのプログラミングインターフェースに関する議論
pgsql-novice	PostgreSQLの初心者のためのリスト
pgsql-patches	PostgreSQLのバグ修正や改良パッチを投稿／議論
pgsql-sql	SQL文に関する議論

■ http://www.sra.co.jp/people/t-ishii/PostgreSQL/

筆者が管理しているPostgreSQL日本語メーリングリストのサポートを主な目的にしたページです（図5.7.2）．メーリングリストのアーカイブや，その検索[注2]，インストール方法，バグ情報，ベンチマークデータなどのPostgreSQLの各種関連情報を紹介しています．

PostgreSQL日本語メーリングリストを購読するには，メールアドレス`pgsql-jp-request@sra.co.jp`宛てに

```
subscribe
```

1行だけ書いてメールをお送りください．サブジェクトは必要ありません．

無事に受け付けられれば，welcomeメールが届きますので，注意事項を熟読の上，議論に参加ください．

●注 2
実際には検索は別サイトで行われており，箕畑氏作成の検索システムにリンクさせていただいているだけです．

●図 5.7.2
PostgreSQLの日本語サイト：筆者が管理している

■http://www.interwiz.koganei.tokyo.jp/software/PsqlODBC/

片岡氏が開発されているPostgreSQL ODBC Driver日本語版ページです．ODBC（Open Database Connectivity）はRDBMSに接続するためのAPI（APplication Interface）です．データベースベンダの提供するODBCドライバを組み込むことにより，アプリケーションプログラムはデータベース製品の違いをあまり意識することなくデータベースにアクセスできるようになります．PostgreSQLにもODBCドライバがあり，これを使うことによりWindows上のExcelやAccessからPostgreSQLのデータベースが使えるようになります．片岡氏はPostgreSQL用のODBCドライバで日本語を使えるようにしただけでなく，オリジナルのバグ修正などの改良もされています．

なお，同ページで配布されているPostgreSQL用の日本語版ODBCドライバはCD-ROMにも収録してあります（packages/ODBC/）．

■http://www.rccm.co.jp/~juk/

Linuxの新しいディストリビューションPlamo Linuxで積極的にPostgreSQLのパッケージ化をされている桑村氏のページです．また，“Kerberos”という認証システムをPostgreSQLで利用する場合のインストール方法などの解説があります．

■http://pg.cni.co.jp/

PostgreSQLの和訳ドキュメントが充実している前田氏のページです．オンラインマニュアルを検索することもできます．

■http://www.remus.dti.ne.jp/~sim/

PostgreSQLをはじめ，幅広い情報満載の堀田氏によるページです[注3]．

●注3
CD-ROM収録のPostgreSQL 6.3.2日本語オンラインマニュアルは前田氏と堀田氏のご提供によるものです．

■http://www.sra.co.jp/people/t-ishii/sd/index.html

技術評論社発行の『Software Design』に掲載された筆者の記事で，主にPostgreSQLやPHPに関して書いています．

5.7.2 PostgreSQL関連ソフトウェア

■http://www.php.net/

PostgreSQLをサポートするサーバサイドスクリプト言語PHPのページです（図5.7.3）.

■http://www.cityfujisawa.ne.jp/%7Elouis/apps/phpfi/

PHPに関する日本語ページです（図5.7.4）．PHPの日本語ドキュメントなどもあり，大変充実しています．また，PHPに関する日本語メーリングリストも運用されています．購読方法はWebページをご覧ください.

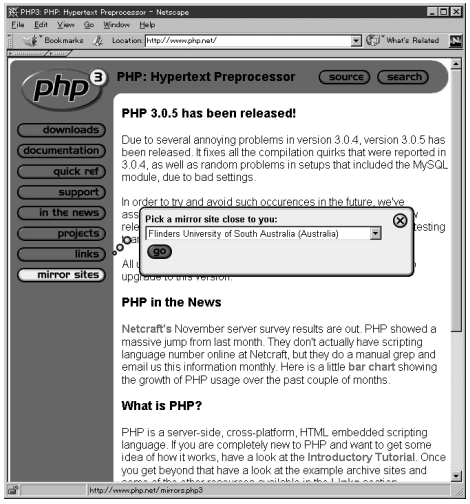
■http://www.apache.org/

世界で最も広く使われているWebサーバApacheのページ．PHPとの併用をお勧めします.

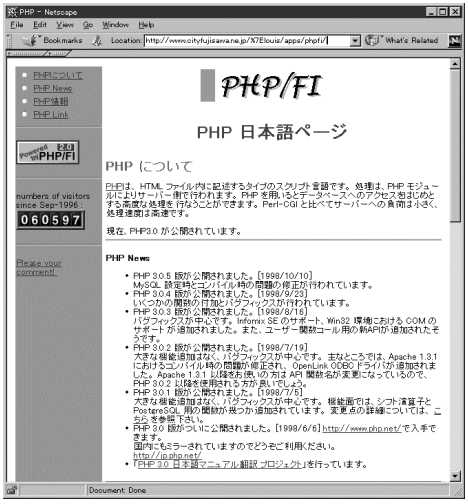
■http://www.netlab.co.jp/ruby/jp/

まつもと ゆきひろ氏が作成したオブジェクト指向スクリプト言語rubyのページです．PostgreSQLが提供されています.

●図 5.7.3　PHP本家のサイト



●図 5.7.4　PHPの日本語サポートのサイト



■**http://www.javasoft.com/**

SunによるJavaのページ．Solaris用などのJDKはここから入手できます．

■**http://www.blackdown.org/java-linux.html**

Intel版などのJDKはここから入手できます．

■**http://business.tyler.wm.edu/mklinux/**

MkLiux，LinuxPPC用のJDKはここから入手できます．

■**http://www.freebsd.org/java/**

FreeBSD用のJDKはここから入手できます．

■**http://developer.java.sun.com/developer/**

JDC（Java Developers Connection）のページ（図5.7.5）．JDK 1.1.xでSwingを使うためには，このページでJDCの会員になってSwingを別途入手する必要があります．

■**http://www.kusastro.kyoto-u.ac.jp/~baba/**

全文検索に関する情報が豊富な馬場さんのページ．本書で使っているkakasiのパッチなどもここから入手できます．

●図5.7.5
JDCのサイト