実践テクニックをご紹介

徒然PostgreSQL散策

第11回 最終回

テーブルの構造とディスク容量の見積もり（2）

石井達夫 ISHII Tatsuo
ishii@postgresql.org

はじめに

寒い日が続きますが，読者の皆さんはお元気にお過ごしでしょうか．

2005年は待望のPostgreSQL 8.0がリリースされるという幸先のよいスタートでした．PostgreSQL 8.0の機能に引かれながらも，安定するまで採用を待つべきかどうか，嬉しい悩みを抱えている方もいらっしゃるのではないでしょうか．

さて，前回に引き続き，今回もディスク容量の見積もり方法の解説を行います．今回は積み残しになっていたインデックスやトランザクションログ容量計算の方法を解説し，データベースの容量見積もりを完成させます．前回同様，解説の前提にしているのはPostgreSQL 8.0です．

インデックスとBtree

インデックスもヒープと同じように固定ブロックに分かれて管理されています．1ブロックの大きさがデフォルトで8192バイトであることもまったく同じです．ここでは，PostgreSQLでもっともよく使われているBtreeを前提に解説します．

Btreeとは

Btreeはディスク上のインデックスの実装方法の1つで，数々の優れた特徴を持っているため，データベースなどに広く利用されています．PostgreSQLで実装されているのは正確にはB+treeと呼ばれるBtreeの発展形です．しかし，実際にはB+treeのほうが広く使われており，以後単にBtreeと言った場合，B+treeを指すものとします．

Btreeの動作原理[注1]

今，整数型の列，

```
INTEGER i
```

を含むテーブルがあったとします．もしiに対してインデックスを作成すると，WHERE i = 100のような問い合わせをBtreeを使って高速に実行できます．この列に，100，101，105，112，119，220，231，237，242の9個の値を登録してあったとして，Btreeインデックスを作成したときの例を図1に示します．

点線から上は**インデックスノード**，下は**リーフノード**と呼ばれます．

まずリーフノードですが，各々の箱は固定の大きさで，列の値を格納する白い箱の部分と，ポインタを格納する灰色の部分に分かれています．この例では，各リーフノードについて白い箱は4つあり，最大4個の値を格納することができます．灰色の箱に入っているポインタは次のリーフノードを指しています．このポインタを辿ると，昇順にソートされた列の値を得ることができます．また，この図には描いてありませんが，実際には各列の値に対応した行を知るためのポインタも格納されています．

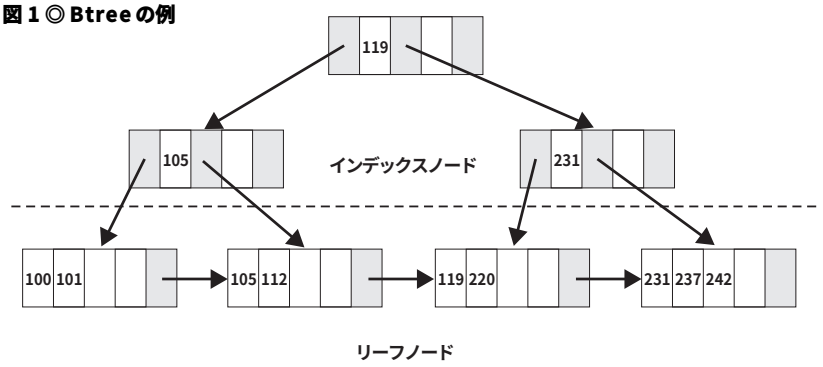
一方インデックスノードは，リーフノードへ到達するための道筋を示す役割を果たします．この例では，

注1）本節は拙著『PHPxPostgreSQLで作る最強Webシステム』（技術評論社）から引用しています．

連載postgre11 06.2.27 10:33 AM ページ221

各リーフノードあたり最大2個までのポインタを格納することができます．白い箱には列の値が入っており，その値よりも小さな値を含むインデックスノードまたはリーフノードへのポインタは値の左側にあり，逆にその値と同じか，あるいはその値よりも大きな値を含むインデックスノードまたはリーフノードへのポインタは値の右側にあります．したがって，一番上のノード（特に**ルートノード**と呼ぶことがあります）から探したい値の大小でポインタを辿れば，目的のリーフノードに素早くたどり着くことができるわけです．

図1◎Btreeの例



例を示します．

今，220を探したいとします．まずルートノードを調べます．ルートノードに119が登録されていますが，220はこれよりも大きいので，119の右側のポインタを辿ってルートノードの右下のインデックスノードを見つけます．このノードには231が登録されており，目的の220よりも大きいので，231の左側のポインタを辿って右から2番目のリーフノードに到達します．この中に目的の220が存在します．

一方，220よりも大きい値を探すときはどうなるでしょう？ この場合もまず220を見つけます．すると，そのリーフノードに220よりも大きな値が含まれているので，それがまず検索結果になります．さらに，リーフノードにはそれよりも大きな値を含むリーフノードへのポインタがあるので，それを辿ることによって，231，237，242を含むリーフノードを素早く探し出すことができます．

なお，この例では一方向のポインタのみが存在していますが，逆方向のポインタを設けることにより，目的の値よりも小さな値を素早く見つけ出すようにすることも可能です（実際，PostgreSQLはそのような実装になっています）．

Btreeのメリット

Btreeは，目的の値を見つけ出すまでに最小限のアクセスで済むことが特長です．上の例では3回のアクセスで目的のデータを見つけることができました．データ量が多くなっても，インデックスノード中にある程度の数のポインタを収めることができればアクセス回数はさほど増えません．たとえば，ノードの大きさが8000バイト程度（PostgreSQLの実装もそのくらいです）で，この例のように整数型のデータを格納している場合，100万件以上のデータに対してBtreeインデックスを作成しても，目的のデータを見つけるまでのアクセス回数はせいぜい2～3回程度です[注2]．

Btreeの構造は図1からわかるように，バランスの良い左右対称の形をしています．これは偶然ではありません．このようなバランスの取れた形のときにアクセス数が最小になることがわかっているからです．実はBtreeはデータを挿入していってもバランスが自動的に取られるようになっています．これもBtreeの特長です．

PostgreSQLにおけるBtreeの実装

Btreeの動作原理がわかりましたので，PostgreSQLにおけるBtreeの実装を調べてみましょう．

PostgreSQLが採用したアルゴリズムは，おもに1981年にP. Lehman and S. Yaoが発表した論文「Efficient Locking for Concurrent Operations on B-Trees」に基づいており，一意でないキーを許すなど

注2）Btreeインデックスのデータ量に関する詳しい見積もりは，『トランザクション処理（上）（下）』（日経BP社）などを参照してください．

連載postgre11 06.2.27 10:33 AM ページ222

の拡張が加えられています[注3].

ページの構造

前回テーブルにおけるページの構造を説明しましたが，インデックスでもページの構造は基本的に同じです．ただし，ページの最後にあるスペシャルスペースがテーブルでは使われていなかったのに対し，インデックスでは使われています．その構造を表1に示します（PostgreSQL 7.4も同じ構造です）.

btpo_prevとbtpo_nextは「同じレベル」のページをつなぐリンクで，インデックスをスキャンするときに使用されます.

ページの中から完全にデータが削除されると，btpo_flagsフラグにそのことが示されると共に，xactには次のトランザクションIDか，非常に古いことを表すトランザクションID（VACUUM FULLのとき）がセットされます．Btreeインデックスでは，データが削除されたページが物理的に削除されることはありませんが，次にデータを格納する必要があるときに再利用されます．空いたページを物理的に削除するには，REINDEXコマンドなどを使ってインデックスを再構築する必要があります.

btpo_flagsはページの種類を識別するためのフラグです．全部で5種類あります．以下，順に説明します.

● メタページ

PostgreSQLでは，インデックスファイルの最初の1ページがメタページという特別なページになっています．メタページの構造を表2に示します.

メタページは必ず存在しますし，インデックスからすべてのデータが削除されてもメタページが削除されることはありません.

● ルートページ

ルートノードを格納するページで，必ず存在します．ルートページもインデックスからすべてのデータが削除されてもメタページが削除されることはありません.

● リーフページ

リーフノードを格納するページです.

● インターナルページ

ルートページでもリーフページでもないページ，すなわち「木」の大部分を占めるページです.

● 削除ページ

削除扱いになっているページです.

● 空白ページ

データを含んでいない空のページです.

表1◎スペシャルテーブルの構造

●フィールド名	●バイト長	●用途
btpo_prev	4	左の兄弟ページのアドレス
btpo_next	4	右の兄弟ページのアドレス
level（xact）	4	木のレベルまたは次のトランザクションID
btpo_flags	2	ページの種類を表すフラグ
パディング	2	
（計）	16バイト	

表2◎メタページの構造

●フィールド名	●バイト長	●用途
btm_magic	4	Btreeであることを示すマジックNo（BTREE_MAGIC固定）
btm_version	4	Btree実装のバージョン番号（今のところ2）
btm_root	4	ルートページへのポインタ
btm_level	4	ルートページの木のレベル
btm_fastroot	4	「fast」ルートページへのポインタ
btm_fastlevel	4	「fast」ルートページへの木のレベル

インデックスページ数の見積もり

インデックスにおける容量見積もりの難しさは，これらのページがどのくらいの割合で混在するかを正確に予測できない点にあります．そこで実データではどうなるのか，調べてみることにしました.

おなじみの簡易ベンチマークツールpgbench[注4]使い，

```
pgbench -i -s 10 test
```

で100万件分のデータを作ります．すると，図1のよ

注3）詳細は，ソースコードに付属するsrc/backend/access/nbtree/README をご覧ください.
注4）pgbenchはPostgreSQLのソースコードに付属しています．インストール方法はcontrib/pgbench/README.pgbench_jisを参照してください.

連載postgre11 06.2.27 10:33 AM ページ223

うにaccounts_pkeyというINTEGER型のBTreeインデックスが生成されますので，この中身を調べてみましょう．リスト1にあるような簡単なプログラムを作り，ページの中身を印字させます．

このプログラムの引数はインデックスファイルのパス名です．該当するインデックスファイルのパス名は，図2のようにシステムカタログを検索することによって調べることができます．ここで「relfilenode」がファイル名を示しており，7536346がaccounts_pkeyのファイル名だとわかります．次にDB名に対応するディレクトリ名を調べます（図3）．

これで，目的のファイルが/usr/local/pgsql/data/base/17230/7536346であることがわかったので，先ほどのプログラムの引数に与えてみます（図4）．1ページに付き1行で情報が表示されます．

この結果を集計したのが表3です．ご覧のように，

▼リスト1　Btreeインデックスのページの種類を印字

```
#include <stdlib.h>
#include <stdio.h>

#define BTP_LEAF      (1 << 0) /* leaf page, i.e. not internal page */
#define BTP_ROOT      (1 << 1) /* root page (has no parent) */
#define BTP_DELETED   (1 << 2) /* page has been deleted from tree */
#define BTP_META      (1 << 3) /* meta-page */
#define BTP_HALF_DEAD (1 << 4) /* empty, but still in tree */

main(int argc, char **argv)
{
  FILE *fd;

  typedef struct {
    char dummy[8192-16];
    int btpo_prev;
    int btpo_next;
    int level;
    short btpo_flags;
    short pad;
  } index_page;

  index_page index_buf;

  fd = fopen(argv[1], "r");
  if (!fd)
  {
      fprintf(stderr, "cannot open %s\n", argv[1]);
      exit(1);
  }

  while (fread((char *)&index_buf, 8192, 1, fd) != 0)
  {
      char *kind;

      if (index_buf.btpo_flags & BTP_LEAF)
          kind = "leaf";
      else if (index_buf.btpo_flags & BTP_ROOT)
          kind = "root";
      else if (index_buf.btpo_flags & BTP_DELETED)
          kind = "deleted";
      else if (index_buf.btpo_flags & BTP_META)
      kind = "meta";
      else if (index_buf.btpo_flags & BTP_HALF_DEAD)
      kind = "empty";
      else
      kind = "internal";

      printf("flag: %04x kind: %s level: %d\n", index_buf.btpo_flags,
        kind, index_buf.level);
  }

  fclose(fd);
}
```

▼図1　作成したインデックス

```
       Table "public.accounts"
  Column  |     Type      | Modifiers
----------+---------------+-----------
 aid      | integer       | not null
 bid      | integer       |
 abalance | integer       |
 filler   | character(84) |
Indexes:
  "accounts_pkey" primary key, btree (aid)
```

▼図2　パス名の調査

```
$ psql test
test=# \x
Expanded display is on.
otest=#  SELECT  *  FROM  pg_class  WHERE
relname = 'accounts_pkey';
-[ RECORD 1 ]--+--------------
relname        | accounts_pkey
relnamespace   | 2200
reltype        | 0
relowner       | 1
relam          | 403
relfilenode    | 7536346
reltablespace  | 0
relpages       | 2193
reltuples      | 1e+06
reltoastrelid  | 0
reltoastidxid  | 0
relhasindex    | f
relisshared    | f
relkind        | i
relnatts       | 1
relchecks      | 0
reltriggers    | 0
relukeys       | 0
relfkeys       | 0
relrefs        | 0
relhasoids     | f
relhaspkey     | f
relhasrules    | f
relhassubclass | f
relacl         |
```

▼図3　ディレクトリ名の調査

```
test=#  SELECT  oid  FROM  pg_database  WHERE
datname = 'test';
  oid
-------
 17230
(1 row)
```

連載postgre11 06.2.27 10:33 AM ページ224

▼図4 リスト1の実行

```
$ ./a.out /usr/local/src/pgsql/current/data/
base/17230/7536346
flag: 0008 kind: meta level: 0
flag: 0001 kind: leaf level: 0
flag: 0001 kind: leaf level: 0
flag: 0000 kind: internal level: 1
flag: 0001 kind: leaf level: 0
flag: 0001 kind: leaf level: 0
flag: 0001 kind: leaf level: 0
flag: 0001 kind: leaf level: 0
flag: 0001 kind: leaf level: 0
flag: 0001 kind: leaf level: 0
～以下略～
```

このケースでは99％以上がリーフページであり，リーフページの数を見積もることができればインデックスの容量を見積もることができると言えそうです．もちろん，値の分布や，データ型，データ長によって割合は異なってきます．

念のため，char(10)の場合のインデックスの状況を同じように調べてみます（図5）．その結果，表4のようになりました．char(10)のデータサイズは14バイトで，INTEGERの4バイトから10バイト増えた結果，全体のページ数とインターナルページが増えていますが，それにしても依然として99％以上の高い割合でリーフページが大半を占めており，インデックスの容量見積もりのためにはリーフページだけを見積もれば充分であることがわかります．

リーフページ数の見積もり

リーフページに格納される個々のデータ（インデックスタプル）は表5のような構造になります．

リーフページの中には，ページヘッダとスペシャルデータが固定長で必ず含まれており，それ以外がインデックスタプルで使用できる領域です．個々のタプルデータの位置を表すアイテムポインタがありますので，結局1ページの中に格納できるインデックスタプル数は，INTEGERの場合次のようにして求められます．

▼図5 char(10)の場合

```
test=# CREATE TABLE t1(t CHAR(10));
CREATE TABLE
test=# INSERT INTO t1 SELECT aid FROM accounts;
INSERT 0 1000000
test=# CREATE INDEX t1index ON t1(t);
CREATE INDEX
```

表3◎インデックスのページの割合

●ページの種類	●数	●全体に対する比率
メタページ	1	0.03％
ルートページ	1	0.03％
インターナルページ	7	0.3％
リーフページ	2184	99.6％
（計）	2193	100％

1ページに格納できるインデックスタプル数＝
（ページサイズ（8192）－ページヘッダ（24）
－スペシャルデータ（16））
／（インデックスタプル（12）
＋アイテムポインタ（4））
＝509.5

これを切り下げて509となります．したがって，100万個のインデックスタプルを格納するためのリーフページ数は，

1000000 / 509 = 1964.63

これを切り上げて1965ページとなりました．しかし表3を見ると，実際には2184ページのリーフページがありました．この違いはどこから来るのでしょう？これは，Btreeではページの利用率が100％にはならないという特性から来ています．つまり，Btreeでは必ず使用されない領域が出てしまうのです．

表4◎char(10)の場合

●ページの種類	●数	●全体に対する比率
メタページ	1	0.03％
ルートページ	1	0.03％
インターナルページ	19	0.49％
リーフページ	3832	99.5％
（計）	3853	100％

表5◎インデックスタプルの構造

●フィールド名	●バイト長	●用途
t_tid	6	対応するテーブル内行のTID
t_info	2	タプル長とフラグ
実データ	4	INTEGERの場合
（計）	12バイト	

図1で，新たにインデックスに値235と244を追加する必要が出てきたとします．これらのデータは一番右のリーフノードに入ることになります．ところが，ここでリーフノードが最大4個までのインデックスタプルしか格納できないとすると，244は入りきれません．このとき，このリーフノードは2つに分割され，231，235，237の入ったリーフノードと242，244の入ったリーフノードになります．そしてインデックスノードにもポインタが追加されます．

この例ではリーフノードが分割されただけですが，もし値が追加されていけばインデックスノード中のポインタエリアが満杯になることもあります．その場合はインデックスノードも分割されます．

このように，Btreeでは値の追加があるとページが分割，追加されていくことがあります．その結果，どのページもきっちり満杯，というような状況はまず起きません．では実際にどのくらいの割合でページが利用されるのでしょうか．これもなかなか計算しにくいのです．そこで実際に試してみると，表3のケースでは使用率は，

1965 / 2184 = 0.8997 = 90％

でした．表4のケースでは，

1ページに格納できるインデックスタプル数＝
（ページサイズ（8192）－ページヘッダ（24）
－スペシャルデータ（16））
／（インデックスタプル（22）
　＋アイテムポインタ（4））
＝313.53 → 313

100000/313 = 3194.88 → 3195ページ
3195/3832 = 0.8337 = 83％

でした．試しにchar(40)のデータで同様の実験をしてみると，使用率は89%になりました．どうやら，PostgreSQLのBtreeでは領域使用率は80～90%くらいのようです．もちろんこの値は削除も更新もない非常に理想的な状態のものですから，実運用に入ってからは実際にどの程度の領域使用率になるかチェックしたほうがよいでしょう．

トランザクションログの容量見積もり

ディスク容量の見積もりの最後に，トランザクションログの容量見積もりを行います．この節を理解するためには，できれば本連載の第6回「トランザクションログ」（本誌Vol.20）を一読していただくとよいと思います．

トランザクションログの書出しタイミング

トランザクションログはデータベースに対する永続的な変更操作があったときに書き出されます．したがって，副作用のない純粋なSELECT操作や一時テーブルの操作はログの対象になりません．

リソースマネージャとログレコードの種類

ログレコードはリソースマネージャと呼ばれるモジュールが管理し，ログレコードは少なくともリソースマネージャの種類分だけ存在します（表6）．

表6の「詳細処理」がわかれば，ログの大きさは見積もれます．難しいのはSQL文と「処理詳細」の対応です．たとえば，Btreeインデックスがあるテーブルに行を追加すると，HEAP_INSERT，BTREE_INSERT_LEAFだけではなく，ページの分割が起これば BTREE_SPLIT_L(R)，さらにBTREE_INSERT_UPPERやBTREE_SPLIT_L(R)_ROOTなどが起こり得ます．したがって，ログレコードに関しては正確な容量見積もりはかなり難しいと言わざるを得ません．

簡単な例

ここでは簡単な例として，INSERT処理を考えます．簡単のために，挿入されるのはINTEGER型のデータが1つだけであるとします．まず，HEAP_INSERTにより，表7（227ページ）のようなログレコードが作成されます．

一方，このタプルはテーブル上では，タプルヘッダ（27バイト）＋パディング（1バイト）＋OID（4バイト）＋データ本体（4バイト）＝36バイトですから，ログレコードは，テーブル本体よりも大きくなるということになります．

連載postgre11 06.2.27 10:33 AM ページ226

表6◎リソースマネージャの処理

●リソースマネージャの名称	●詳細処理名	●説明
XLOG（OIDなどの管理）	NEXTOID	OIDカウンタの管理
	CHECKPOINT_SHUTDOWN	終了処理中のチェックポイント
	CHECKPOINT_ONLINE	チェックポイント
Transaction（トランザクション管理）	XACT_COMMIT	コミット処理
	XACT_ABORT	アボート処理
Storage（ストレジマネージャ管理）	XLOG_SMGR_CREATE	ファイル作成処理
	XLOG_SMGR_TRUNCATE	TRUNCATE処理
CLOG（トランザクションコミットステータス管理）	CLOG_ZEROPAGE	pg_clogへのページ追加
Database（データベース管理）	XLOG_DBASE_CREATE	CREATE DATABASE処理
	XLOG_DBASE_DROP	DROP DATABASE処理
Tablespace（テーブルスペース管理）	XLOG_TBLSPC_CREATE	CREATE TABLESPACE処理
	XLOG_TBLSPC_DROP	DROP TABLESPACE処理
Heap（ヒープ管理）	HEAP_INSERT	INSERT処理
	HEAP_DELETE	DELETE処理
	HEAP_UPDATE	UPDATE処理
	HEAP_MOVE	VACUUMによるタプルの移動
	HEAP_CLEAN	タプルの消去
Btree（Btree管理）	BTREE_INSERT_LEAF	リーフページの追加
	BTREE_INSERT_UPPER	上位ページの追加
	BTREE_INSERT_META	メタデータの追加
	BTREE_SPLIT_L	ページの分割（左）
	BTREE_SPLIT_R	ページの分割（右）
	BTREE_SPLIT_L_ROOT	rootページの分割（左）
	BTREE_SPLIT_R_ROOT	rootページの分割（右）
	BTREE_DELETE	削除
	BTREE_DELETE_PAGE	ページの削除
	BTREE_DELETE_PAGE_META	メタページの削除
	BTREE_NEWROOT	rootページの追加
	BTREE_NEWMETA	メタページの追加
Sequence（シーケンス管理）	XLOG_SEQ_LOG	シーケンス更新処理

実際にINTEGER型の列のみを持つテーブルを作り，以下のような方法でおおよそどのくらいのトランザクションログが生成されるのか調べてみました．

PostgreSQL 8.0では，PITR（Point In Time Recovery）といって，トランザクションログを保存する機能があります．したがって，テーブルにデータを追加したときに保存されたログ（アーカイブログ）が実際にどのくらいあるか調べることで，ログの生成量がわかるというわけです．

PITRはデフォルトでは有効になっていません．有効にするためには，以下の手順を実行します．

① アーカイブログの保存先ディレクトリを作ります[注5]

```
$ mkdir /usr/local/pgsql/archivedir
```

注5）実際の運用では，アーカイブログの保存先はデータベースクラスタとは物理的に別のディスクドライブにすることをお勧めします．

連載postgre11 06.2.27 10:33 AM ページ227

表7◎作成されたログレコード

●フィールド名	●バイト長	●用途
xl_crc	8	CRCチェック
xl_prev	8	前のログレコード
xl_xid	4	トランザクションID
xl_len	4	リソースマネージャデータ長
xl_info	1	処理詳細
xl_rmid	1	リソースマネージャID
パディング	2	
target	18	タプルの識別（テーブルスペース，データベース，テーブル，TID）
t_natts	2	列数
t_infomask	2	タプルフラグ
t_hoff	1	オフセット
t_data	4	データ本体
（計）	56（55を4バイトバウンダリに切り上げ）	

② postgres.confを設定します（リスト2）

③ postmasterを再起動します

④ バックアップを取得します

psqlから以下を実行します．

```
SELECT pg_start_backup('mybackup');
```

tarコマンドでバックアップを取得します．

```
$ cd /usr/local/pgsql/data
$ tar cvfz mybackup.tar.gz data
```

psqlから次を実行します．

```
SELECT pg_stop_backup();
```

以上でPITRが有効になりました．そして，以下のようなテーブルを作り，

```
CREATE TABLE t1(i INTEGER) WITHOUT OIDS;
```

2097152件の行を挿入したところ，アーカイブログは113Mバイトになりました（図6）．

先ほどのアーカイブログ容量の計算式では，56バイト*2097152＝112Mバイトですから，かなり実際に近い数値になりました．

●UPDATEの場合

UPDATEの場合は，ログされるデータは1件あたり表8のようになります．

●DELETEの場合

DELETEの場合は，ログされるデータは1件あたり表9のようになります．

インデックスの追加処理

Btreeインデックスのリーフノードが追加された場合（BTREE_INSERT_LEAF）のログのレコードは表10のようになります．データは整数型（4バイト）とします．

ページが分割されると（BTREE_SPLIT_LまたはBTREE_SPLIT_R），さらに表11のようなログレコードが生成されます．これにさらに，左右のページに格納されているタプルの全データが追加されます．

インデックスにデータが追加される際には，これ以外にも上位ページの追加（BTREE_INSERT_UPPER）

▼図6 アーカイブログ

```
$ ls -lh archivedir/
合計 113M
-rw-------    1 t-ishii  t-ishii       16M Jan 16 22:52 000000010000000000000005F
-rw-------    1 t-ishii  t-ishii       244 Jan 16 22:48 000000010000000000000005F.00B4F74C.backup
-rw-------    1 t-ishii  t-ishii       16M Jan 16 22:55 000000010000000000000060
-rw-------    1 t-ishii  t-ishii       16M Jan 16 22:56 000000010000000000000061
-rw-------    1 t-ishii  t-ishii       16M Jan 16 22:56 000000010000000000000062
-rw-------    1 t-ishii  t-ishii       16M Jan 16 22:56 000000010000000000000063
-rw-------    1 t-ishii  t-ishii       16M Jan 16 22:56 000000010000000000000064
-rw-------    1 t-ishii  t-ishii       16M Jan 16 22:56 000000010000000000000065
（このとき，テーブルサイズは72.17Mバイトでした）
```

▼リスト2 postgres.confの設定

```
archive_command = 'cp %p /usr/local/src/pgsql/current/archivedir/%f'  # command to use to archive a logfile segment
```

連載postgre11 06.2.27 10:33 AM ページ228

表8◎UPDATEでログされるデータ

●フィールド名	●バイト長	●用途
xl_crc	8	CRCチェック
xl_prev	8	前のログレコード
xl_xid	4	トランザクションID
xl_len	4	リソースマネージャデータ長
xl_info	1	処理詳細
xl_rmid	1	リソースマネージャID
パディング	2	
target	18	タプルの識別（テーブルスペース，データベース，テーブル，TID）
newtid	6	新しく挿入されたタプルの識別
t_natts	2	列数
t_infomask	2	タプルフラグ
t_hoff	1	オフセット
t_data	4	データ本体
（計）	64（61バイトを4バイトバウンダリに切り上げ）	

表9◎DELETEでログされるデータ

●フィールド名	●バイト長	●用途
xl_crc	8	CRCチェック
xl_prev	8	前のログレコード
xl_xid	4	トランザクションID
xl_len	4	リソースマネージャデータ長
xl_info	1	処理詳細
xl_rmid	1	リソースマネージャID
パディング	2	
target	18	タプルの識別（テーブルスペース，データベース，テーブル，TID）
（計）	48（46バイトを4バイトバウンダリに切り上げ）	

などが発生しますが，量的に多いのは以上の2つです．

最後に

2回に渡ってテーブルの構造とディスク容量の見積もり方法を解説しました．テーブルやインデックスに関してはかなり正確に見積もることができました．残念ながらトランザクションログ容量に関してはそれほど正確に見積もることはできませんでしたが，概ねテーブルやインデックスの更新データよりやや多いくらいの容量が必要そうです．

表10◎リーフノードの追加時

●フィールド名	●バイト長	●用途
xl_crc	8	CRCチェック
xl_prev	8	前のログレコード
xl_xid	4	トランザクションID
xl_len	4	リソースマネージャデータ長
xl_info	1	処理詳細
xl_rmid	1	リソースマネージャID
パディング	2	
target	18	インデックスタプルの識別（テーブルスペース，データベース，インデックス，TID）
t_tid	6	テーブルTID
t_info	1	フラグ
t_data	4	データ本体
（計）	60（57バイトを4バイトバウンダリに切り上げ）	

表11◎ページ分割時

●フィールド名	●バイト長	●用途
xl_crc	8	CRCチェック
xl_prev	8	前のログレコード
xl_xid	4	トランザクションID
xl_len	4	リソースマネージャデータ長
xl_info	1	処理詳細
xl_rmid	1	リソースマネージャID
パディング	2	
target	18	インデックスタプルの識別（テーブルスペース，データベース，インデックス，TID）
otherblk アドレス	4	ページ分割に関与したブロックの
leftblk	4	左（前）のブロックのアドレス
rightblk	4	右（次）のブロックのアドレス
level	4	木のレベル
leftlen	2	左（前）のタプルデータの長さ
len	4	右（次）のタプルデータの長さ
（計）	68	

容量の見積もりはなかなかややこしい計算が必要ですが，データベースシステムを設計する上では避けて通れません．今回の記事をもとに，自動的に計算を行うプログラムを書いてみるのもおもしろいでしょう．