実践テクニックをご紹介

徒然PostgreSQL散策

第10回 テーブルの構造とディスク容量の見積もり

石井 達夫
ISHII Tatsuo●ishii@postgresql.org

秋のニューヨークとペンシルバニア

この10月にニューヨークに9ヵ月ぶりに仕事で行ってきました．筆者がニューヨークに行くのはこれで3回目ですが，いつも冬のものすごく寒い時期しか知らなかったので，今回はとても快適に過ごせた気がしました．また，今回は列車でペンシルバニアまで足を伸ばし，コアメンバの一人Bruce Momjian氏のお宅にお邪魔させていただきました．閑静な住宅街の中にあるお宅で奥様の手料理をご馳走になるなど，すっかり歓待されました．

ペンシルバニアには，ちょっとした規模のイタリア人街があり，とてもアメリカとは思えない雰囲気で，そこで食べたパスタは最高の味でした．また，露店で野菜や果物が売られ，チーズを扱う専門店があるなど，まさに気分はイタリアでした注1（写真1）．

写真１◎ペンシルバニアのイタリア人街

DB設計とディスク容量の見積もり

DB設計をしていて必ず問題になるのがディスク容量の見積もりです．当面必要になるディスク容量は？　1年ごとにどれだけディスク消費量が増えるのか？　今あるディスクはいつまで持つのか？　こういった疑問に答えるためには，テーブル設計時にできるだけ正確なディスク容量の見積もりができることが必要です．

ディスク容量の見積もりができるためには，PostgreSQLがどのようにデータをディスクに格納しているのかということ，すなわちDBの物理的な構造を知ることが必要です．本稿では，DBの物理構造を学びつつ，最終的にディスク容量の見積もりができることを目標にします．

なお，この本稿では，本誌が世に出るころにはたぶん正式リリースされているであろうPostgreSQL 8.0を前提にしています．

PostgreSQLのDB構造の概略

データベース

PostgreSQLでは，デフォルトのインストール状態で/usr/local/pgsql/data/以下にすべてのファイルが納められており，ここを「データベースクラスタ」と呼びます．テーブルやインデックスの本体はデータベース配下の「base」というディレクトリに格納されています．

注1）と言っても，筆者はイタリアどころか，ヨーロッパのどこにもまだ行ったことはないのですが:-)．

```
$ ls data/base/
1/  17229/  17230/
```

base以下の数字のディレクトリ1つ1つが，1つの「データベース」に対応します．どのデータベースがどのディレクトリに対応しているかはpg_databaseというシステムカタログでわかります（図1）．

ヒープファイルとブロック

あるデータベースに所属するテーブルは，テーブルスペースを指定しない限り，データベースと同じテーブルスペースに作成されます．

テーブルの実体は，「ブロック」という固定長（デフォルトでは8192バイト）の領域が0個以上131072個以下集まったものです（図2）．テーブルを作った直後は対応するファイル（「ヒープ（heap）ファイル」と呼びます）の大きさは0です．データを追加していくにつれてヒープファイルは大きくなります．ブロックが131072個，すなわちヒープファイルのサイズが1Gバイトを超えると，新しいヒープファイルが作られます．こうして，1つの論理的なテーブルは32Tバイト（＝32×1024Gバイト）までの大きさを取ることができます．

ページとタプル

ブロックの中には「ページ」があります．PostgreSQL（postgres）の歴史上ある時点では1ブロック中に複数のページを納める計画もあったようですが，現在では1ブロックの中に入るページ数が1以外になることはないので，実際にはブロック＝ページになっています．ただし，ブロックとページの使い分けが一応あり，ブロックは物理アクセスの単位であり，ブロック内の情報の構造を問題にするときはページという言葉を使います．

▼図1　データベースとディレクトリの対応

```
test=# SELECT oid, datname FROM pg_database ORDER BY oid;
  oid  |  datname
-------+-----------
     1 | template1
 17229 | template0
 17230 | test
(3 rows)
```

ページの中には行が0個以上入ります．PostgreSQLのソースコード上では，行ではなくて「タプル」という用語がよく使われているようなので，以後タプルという用語を使います．

ページの構造を図3に示します．ページの先頭には24バイトの管理領域があり注2，「ページヘッダ」と呼ばれています．ページヘッダの中にはトランザクションログの管理データのほか，このページの中の空き領域を管理する情報が入っています．

表1にページヘッダの構造を示します．

LSNとタイムラインIDについては解説が必要だと思うので，説明します．LSNは，PostgreSQLがトランザクションログを使ってデータベースをリカバリするときに使用します．トランザクションログには「更新を行った」とか「新しいレコードを追加した」というような情報が入っているので，これをデータベース

図2◎テーブルの構造

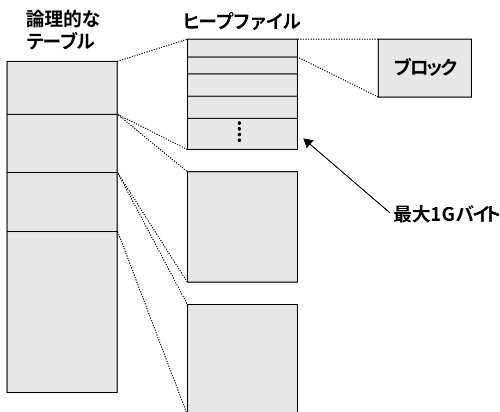
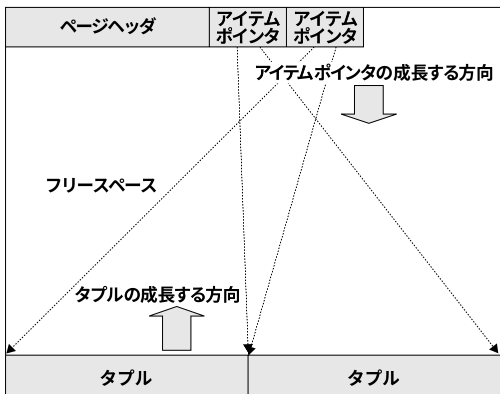


図3◎ページの構造



注2）バージョン7.4では20バイトです．

表1◎ページヘッダの構造

●フィールド名	●バイト長	●用途
pd_lsn	8	LSN(Log Sequence Number)
pd_tli	4	タイムラインID
pd_lower	2	ページ内の空き領域の開始位置
pd_upper	2	ページ内の空き領域の終了位置
pd_special	2	ページ内のスペシャルスペースの終了位置
pd_pagesize_version	2	ページサイズとバージョンに関する情報

に適用することによってデータをリカバリすることができます．このときにLSNを調べることにより，誤って同じログを2回適用してしまうことを防ぐことができます．詳細は本誌Vol.20に掲載された「第6回トランザクションログ」をご覧ください．

「タイムライン（TimeLine）ID」もトランザクションログによるリカバリに関係しています．PostgreSQLは，トラザンクションログを使ってリカバリをかける際に，直近ではなくて任意の時点まで戻ることができ，このときにタイムラインIDが生成されます．タイムラインIDを使うことにより，一旦リカバリをかけた後に再度別の時点を選んでリカバリすることができます．これは，リカバリをかけて戻るべき時点がはっきりしておらず，何度もリカバリをかけてデータ内容を確認しなければならないときなどに有効です．

ページヘッダの後には「アイテムデータ」が続きます．アイテムデータは可変長の領域で，1項目につき4バイトの大きさがあり，ページ内のタプルの位置を管理しています．タプルはページの後方から追加されていきます．タプルが1個追加されるたびにアイテムデータの1項目がページヘッダの後ろに追加されます．つまり，ページの中央の空き領域を，前からはアイテムデータ，後ろからはタプルが攻めていくようなイメージでページの空き領域が埋まっていくわけです．

ページの最後には「スペシャルスペース」がある場合もありますが，通常のテーブルではこのサイズは0であり，使用されていません．

タプルの構造

では個々のタプルの構造を見てみましょう．

タプルの先頭には管理領域である「タプルヘッダ」があります．タプルヘッダは普通のPCでは27バイトの大きさです[注3]．タプルヘッダの構造を表2に示します．

タプルヘッダの後ろには「ヌルビットマップ」があり，NULLデータを含むタプルにおいて，ある列がNULLかどうかを記録します．NULLを含まない行ではヌルビットマップは存在しません．NULLを許す列が1つでも含まれる場合は，列の数分のビットを持つヌルビットマップが存在します．そのサイズは，（列の数＋7）／8バイトになります．もしこのバイト数が4の倍数にならない場合は，4の倍数に切り上げるためにパディングが行われます．

その後には4バイトのOID（オブジェクトID）があります．「WITHOUT OID」を指定して作成されたテーブルでは，OIDは存在しません．

表2◎タプルヘッダの構造

●フィールド名	●バイト長	●用途
t_xmin	4	行を挿入したトランザクションID
t_cmin	4	行を挿入したコマンドID
t_xmax	4	行を削除したトランザクションID
t_cmax/t_xvac	4	行を削除したコマンドIDまたはVACUUMによって移動された行のバージョン
t_ctid	6	この行あるいは新しい行のTID（タプルID）
t_natts	2	列の数
t_infomask	2	フラグビット
t_hoff	1	行データへのオフセット

注3）バージョン7.3/7.4では23バイトです．

テーブルサイズの計算の簡単な例

それでは以上の知識をもとに，もっとも簡単な例でテーブルサイズを計算してみましょう（**リスト1**）．本稿では，ブロックのサイズはデフォルトの8192バイトであるものとします．

前述のように，ページの先頭には24バイトのページヘッダがあるので，実際に使えるのは8192－24＝8168バイトです．

一方タプル1個あたりの大きさは，

タプルヘッダ（27バイト）
＋1バイトのパディング
＋OID（4バイト）
＋i列の大きさ（4バイト）
＋j列の大きさ（4バイト）

から，40バイトになります[注4]．

以上から，1ページに格納できるタプル数は，

8168／（40＋4）＝185.63

を切り下げて185個になります．したがってt1に4096行のデータを挿入したとすると，

4096／185＝22.14

から，23ページ＝23ブロックのテーブル領域が必要であることがわかります．これはディスク容量で言うと，23×8192＝188416バイト＝184Kバイトになります．

逆にディスク容量が決まっている場合はどうなるでしょう？ たとえば，1Mバイトのヒープファイルに何行格納できるか計算してみましょう．

まず1Mバイト＝1024×1024＝1048576バイトですから，ブロック数で言うと1048576／8192＝128になります．各ブロック＝ページには185行入りますから，トータルでは128×185＝23680行格納できることがわかりました．

以上，INTEGER（4バイト整数型）でのテーブル容量の計算方法を示しました．

▼リスト1　サンプルテーブル

```
CREATE TABLE t1(
    i INTEGER NOT NULL,
    j INTEGER NOT NULL
);
```

固定長データ型の場合の計算方法

データ型が固定長の場合は，同じようにして比較的簡単に計算できます．INTEGERの場合と違うのはデータ型によってバイト数が異なることだけです．**表3**に標準的なプラットフォームでの各データ型の記憶容量を示します．

可変長データ型の場合の計算方法

文字列型，NUMERIC，配列の場合は3つの理由に

表3◎データ型の記憶容量

●型	●記憶容量（バイト）
SMALLINT（INT2）	2
INTEGER（INT4）	4
BIGINT（INT8）	8
SERIAL（SERIAL4）	4
BIGSERIAL（SERIAL8）	4
REAL	4
DOUBLE PRECISION	8
FLOAT	8
FLOAT（1-24）	4
FLOAT（25-53）	8
TIMESTAMP	8
TIMESTAMP WITHOUT TIME ZONE	8
TIMESTAMP WITH TIME ZONE	8
INTERVAL	12
DATE	4
TIME	8
TIMEP WITHOUT TIME ZONE	8
TIME WITH TIME ZONE	12
BOOLEAN	1
XID	4
OID	4

注4）ここで1バイトのパディングが入っているのは，OID（OIDがない場合は列の先頭）が4バイトバウンダリになることが要求されるからです．このバウンダリ要求はアーキテクチャによって異なりますが，普通のPCでは4です．

より容量計算がかなり難しくなります．

① ユーザが入力するデータ長によって容量が変わる
② データによっては自動圧縮が行われる
③ 文字コードによっては，文字数とバイト数の関係が文字種によって変わる

①③については平均データ長を決めて計算すれば対応できますが，②はプログラムの挙動がわからないと計算できません．

TEXT型の場合

まず，TEXT型について調べてみましょう．ご存じの方も多いと思いますが，TEXT型はPostgreSQL固有のデータ型で，無限長（実際には圧縮後で1Gバイト）までの文字列を格納できます．文字数の制限がないのでPostgreSQLのユーザには広く使われています．

TEXT型の構造は，文字列の長さを格納する4バイトのヘッダとその後に続く実際の文字データになっています．したがって，圧縮を考えなければ「4＋文字列バイト長」がTEXT型のバイト長です．

文字種とバイト長

日本語の場合，主に使用できる文字コードはEUC-JPかUNICODE（UTF-8）です．表4に文字種と1文字あたりのバイト長を示します．

ここで「漢字」と言っているのは，いわゆる全角文字のことを意味します．ですから「あ」や「α」などもこの分類に入ります．使いたい文字がどの分類になるかよくわからない場合は，octet_lengthを使ってバイト長を確認できます（図4）．

表4◎文字種とバイト長

●文字種	●文字コード	●1文字あたりのバイト長
ASCII	EUC-JP	1
1バイトカタカナ	EUC-JP	2
JIS X 0208漢字	EUC-JP	2
JIS X 0212漢字	EUC-JP	3
ASCII	UTF-8	1
1バイトカタカナ	UTF-8	3
JIS X 0208漢字	UTF-8	3
JIS X 0212漢字	UTF-8	3

TOASTと圧縮を考慮した場合

先ほど説明したように，タプルはページの中に格納されています．ページの大きさは8192バイトですから，このままでは8192バイトを超えるような大きなデータは格納できません．そこでPostgreSQLでは，TOASTテーブルという別のテーブルにはみ出た部分を格納することによって大きなデータを格納できるようにしています．

TOASTテーブルの構造

TOASTテーブルにはデータがBYTEA型に変換された後「チャンク（chunk）」と呼ばれる単位に分割されて格納されます．1つのチャンクの大きさは最大でも通常1994バイトを超えない大きさです．TOASTテーブルは表5のような構造を持っています．ここでchunk_idは，ある行のある列データを識別するための固有のIDです．chunk_seqは複数のチャンクに分割したデータに付けられた一連の番号です．chunk_dataがチャンク本体です．TOASTテーブルにはインデックスもあり，chunk_idとchunk_seqの複合ユニークインデックスが作られます．

● TOASTテーブルの中身を調べる

実際にデータを格納して確認してみましょう．文字コードはEUC-JPを使用するものとします．格納するテーブルはリスト2のものを使用します．

テストデータとしては，執筆中の本稿のテキスト原

▼図4　バイト長の確認

```
test=# \encoding
EUC_JP
test=# SELECT octet_length('α');
 octet_length
--------------
            2
(1 row)
```

表5◎TOASTテーブルの構造

●列名	●データ型
chunk_id	OID
chunk_seq	INTEGER
chunk_data	BYTEA

稿を使いました．データの登録には，

```
INSERT INTO t2 VALUES('.....');
```

のようなSQL文を使います．テキスト中に「'」（単一引用符，シングルクォート）がある場合は，バックスラッシュでエスケープが必要です．もちろん本稿にはご覧のように「'」が出てくるので，sedを使い，

```
sed "s/'/\\\\'/g"
```

としてエスケープしました（もちろんsedではなくて他のツールを使っても結構です）．

登録データのバイト長は図5でわかります．

テーブルに可変長の項目が含まれていると自動的にTOASTテーブルとインデックスが作成されます．TOASTテーブルは「pg_toast_テーブルのrelfilenode[注5]」，インデックスは「pg_toast_テーブルのrelfilenode_index」という命名規則があるので，以下に解説するようにして名前と大きさ（ブロック数）が検索できます．

検索前に一度VACUUMを実行して，ブロック数に関する情報を更新しておくのがポイントです（図6）．pg_toast_17288がTOASTテーブルで，pg_toast_17288_indexがインデックスです．それぞれ，1ブロック，2ブロックの大きさがあります．

それではいよいよTOASTテーブルの中身を調べてみましょう．TOASTテーブルはpg_toastというTOAST専用のスキーマに所属しており，またpg_toastはスキーマサーチパスに含まれていないので，検索するときはスキーマ名をテーブル名の前に付けておきます[注6]（図7）．一連のチャンクには17294というIDが付けられ，0から2までのシーケンス番号を持つ3つのチャンクに分割されています．0と1のチャンクは最大長である1994バイトで，残りのデータがチャンク2に格納されています．ですから，TOASTデータの全長は5593バイトということになります．

●ヒープファイル本体

では，ヒープファイル本体のほうにはどの程度の大きさのデータが格納されているのでしょうか？ contribに登録されているpgstattupleを使えば確認できます．

まずpgstattupleをインストールしましょう．ここでは，インストールに使った8.0ベータ4のソースツリーがカレントディレクトリにあるものとします．

▼リスト2　サンプルテーブル（2）

```
CREATE TABLE t2(
    t TEXT NOT NULL
);
```

▼図5　バイト長の確認

```
test2=# SELECT length(t) FROM t2;
 length
--------
   6022
(1 row)
```

▼図6　VACUUMの実行

```
test=# VACUUM;
VACUUM
test=# SELECT c2.relname, c2.relpages FROM pg_class c,pg_class c2 WHERE (c2.relname = ('pg_toast_' ||
c.relfilenode) OR c2.relname = ('pg_toast_' || c.relfilenode || '_index')) AND c.relname = 't2';
       relname         | relpages
-----------------------+----------
 pg_toast_17288_index |        2
 pg_toast_17288       |        1
(2 rows)
```

▼図7　TOASTテーブルの中身を検索

```
test=# SELECT chunk_id, chunk_seq, length(chunk_data) FROM pg_toast.pg_toast_17288;
 chunk_id | chunk_seq | length
----------+-----------+--------
    17294 |         0 |   1994
    17294 |         1 |   1994
    17294 |         2 |   1605
(3 rows)
```

注5）relfilenodeはテーブルに割り当てられた固有のIDで，システムカタログのpg_classで管理されます．

注6）chunk_dataの中身はBYTEAで，しかも圧縮がかかった後のバイナリ値を表現したものですから，そのまま見ても「\346%\000\000\000\012\305\314\301...」のようになってしまうので，ここでは長さだけを表示させています．

```
$ cd postgresql-8.0.0beta4/contrib/pgstattuple
$ make
$ make install
$ psql -f pgstattuple.sql test
```

これでインストールは完了です．さっそくpgstattupleを使ってヒープファイル本体のタプルの長さを調べてみましょう（図8）．この中で，tuple_lenがテーブルの中のタプルの長さの合計です．ここではタプルが1個だけなので，52というのがそのタプルの長さです．tuple_lenにはタプルヘッダも含まれており，その長さは，

タプルヘッダ（27バイト）
　＋1バイトのパディング
　＋OID（4バイト）
　＝32

バイトです．ですから，データ本体の大きさは52－32＝20バイトということになります．20バイトの中身は表6のようになっています．ご覧のように，すべて管理データであり，ユーザデータ本体はTOASTテーブルに格納されていることがわかります．

ここまで調べて，ようやくTEXT型でTOASTと圧縮がある場合のデータ容量を計算できるようになりました．

●データ容量の計算

圧縮がある場合は圧縮後の正確なデータ長を正確に見積もるのが難しいのですが，ここではとりあえずテストデータで圧縮率8％（TOASTデータ長（5593）／元データ長（6022）＝0.92から計算）として，リスト2の定義を持つテーブルに10Kバイトのテキストを4096個格納した場合の容量見積もりを行ってみましょう．

- テーブル本体に格納されるタプル長は一律52バイト．したがって1ページには8168／（52＋4）＝145.85から145タプルが格納できる．4096タプルならば4096／145＝28.24だから，29ブロック必要になる．

- 10Kバイトのデータを圧縮すると（10×1024×0.92＝9420.8）から9421バイトになる．これはチャンクに分割すると9421／1994＝4.72から5つのチャンクになる．したがって4096個ならば4096×5＝20480チャンクが必要で，1つのブロックには4チャンク格納できるので，20480／4＝5120ブロックが必要．

- 以上から，テーブル本体とTOASTテーブルで合計29＋5120＝5149ブロック必要．これは，5149×8192／1024／1024＝40.22から約41Mバイトになる．

元データは4096×10×1024／1024／1024＝40Mバイトですから，ほぼ同じくらいのディスクスペースが必要であることがわかります．

TOASTと圧縮がない場合

PostgreSQLでは，圧縮をしてデータを格納するかどうかはデータの内容によることになります．すなわち，同じ文字が連続しているような場合には圧縮効率

▼図8　ヒープファイル本体のタプルの長さ

```
test=# \x
Expanded display is on.
test=# select * from pgstattuple('t2');
-[ RECORD 1 ]------+------
table_len          | 8192
tuple_count        | 1
tuple_len          | 52
tuple_percent      | 0.63
dead_tuple_count   | 0
dead_tuple_len     | 0
dead_tuple_percent | 0
free_space         | 8112
free_percent       | 99.02
```

表6◎データ本体の中身

●フィールド名	●データ長	●意味
va_header	4	フラグ＋データ長※
va_rawsize	4	圧縮前のサイズ
va_extsize	4	TOASTに格納されているサイズ
va_valueid	4	chunk_id
va_toastrelid	4	TOASTテーブルのID

※va_headerの頭1ビットがオンならばTOASTテーブルを使用していることを表し，次のビットがオンならば値が圧縮されていることを表します．したがって，今回のテストデータでは両方のビットがオンになっているはずです．残りの30ビットは全データ長です．30ビットを使って表現できる最大の数字は1Gバイトです．このため，PostgreSQLでは，データの最大長は1Gバイトに制限されています．

がよくなるので，比較的小さいデータでも圧縮が行われます．したがってあらかじめデータ圧縮が行われるかどうか判断するのは難しいのですが，ソースコード中のコメントによれば，1Kバイト以下のデータの場合，圧縮が行われる確率は低いとのことです[注8]．

TOASTも圧縮も行われない場合は，タプル長は，

52＋4＋データ長

となります．たとえば512バイトの文字列データを4096個格納すると，8168／（52＋4＋512）＝14.38ですから，1ブロックには14タプル入ります．したがって，4096タプルでは4096／14＝292.57から293ブロック必要になり，これは約2.3Mバイトです．元データは512×4096／1024／1024＝2Mバイトですから，PostgreSQLに格納する場合はちょっとだけデータが余計に必要になっています．

そのほかの可変長データ型

TEXT以外の可変長データでは，CHAR，VARCHAR，BYTEAがTEXTと同様の方法で容量計算ができます．

NUMERIC型の場合

NUMERIC型も可変長データ型ですが，データ本体のバイト数の計算式は以下のようになります．

8＋精度（スケール）／4×2

ですからリスト3のようなテーブルにおいて，タプルの長さは，28（タプルヘッダ）＋4（OID）＋8＋128（スケール）／4×2＝104バイトとなります．

配列の場合

PostgreSQLでは任意のデータ型[注9]を配列にすることができます．データ量としては，基本的に今まで説明したデータ型を配列要素数格納したときのデータ量と考えてよいのですが，配列としての管理領域が加算されることだけが異なります．

管理領域は全部で16＋4×次元数＋4×次元数バイトで，内容は表6になります．

例を示します．

```
CREATE TABLE t5(
    i INTEGER[]
);
INSERT INTO t5 VALUES(ARRAY[1,2,3,4,5,6,7,8,9,10]);
```

この場合，1次元配列なので，タプルの長さは28（タプルヘッダ）＋4（OID）＋24（配列管理領域）＋4（INTEGERのデータサイズ）×10＝96バイトになります．

以下のような2次元配列を登録した場合は，

```
INSERT INTO t5
VALUES(ARRAY[[1,2,3,4,5,6,7,8,9,10],
[1,2,3,4,5,6,7,8,9,10]]);
```

28（タプルヘッダ）＋4（OID）＋32（配列管理領域）＋4（INTEGERのデータサイズ）×20＝144バイトになります．

次回はインデックスの容量計算

ここまで書いたところで誌面が尽きました．データベースにはテーブルだけでなく，インデックスもあります．また，トランザクションログもあります．次回はそれらについて解説し，ディスク容量見積もりを完成させたいと思います．

▼リスト3　サンプルテーブル（3）

```
CREATE TABLE t4(
    n NUMERIC(128) NOT NULL
);
```

表6◎管理領域の内容

●フィールド名	●バイト長	●用途
size	4	この配列データのバイト長
ndim	4	次元数
flag	4	未使用
elemtype	4	データ型OID
dim	4×次元数	各次元のサイズ
dim_lower	4×次元数	次元の下限値

注8）src/backend/utils/adt/pg_lzcompress.c
注9）ユーザ定義データ型とドメインを除きます．