さわって実感

リレーショナルデータベース設計 4

データベース設計（3）

日本PostgreSQLユーザー会 理事長 石井達夫 ISHII Tatsuo
ishii@postgresql.jp

PostgreSQL関連情報

PostgreSQL 7.2 ベータテスト開始

PostgreSQLの最新バージョン7.2の開発も山場を迎え，ベータテストの開始となりました．新バージョンでは，VACUUM処理中にテーブルがロックされるというPostgreSQLの最大の弱点が解消されます．これによって，PostgreSQLを24時間365日稼働させることがようやく本当の意味で可能になりました[注1]．PostgreSQL 7.2は早ければ本稿が世に出るころには正式リリースされているかもしれません．そうなればPostgreSQLユーザにとって嬉しいお年玉ということになりますね:-)

PostgreSQL オフィシャルマニュアル

PostgreSQLユーザにとってもう1つのお年玉は，PostgreSQL付属のマニュアルの日本語訳が書籍化されることです．タイトルは『PostgreSQLオフィシャルマニュアル』（インプレス刊，詳細はhttp://www.ips.co.jp/mer/99168.htm）です．翻訳は日本PostgreSQLユーザー会などが行いました（公開済みのものがベースですが，書籍化にあたり誤訳の訂正，表記が統一されて読みやすくなっています）．オンラインマニュアルもよいのですが，PostgreSQLのように大規模なソフトでは，マニュアルも膨大な量になります．書籍の形でじっくり読んでみたい方も多かったのではないでしょうか．

ユーザー会のドメイン名変更

私のemailアドレスが変更されたことにお気づきでしょうか．日本PostgreSQLユーザー会のドメイン名が“jp.postgresql.org”から“postgresql.jp”に変更されました．これにともなってユーザー会のWebページもhttp://www.postgresql.jpに変更されております．ご注意ください．

物理データベース設計とは

前回で論理データベース設計が完了したので，いよいよデータベース設計最後のフェーズである物理データベース設計に取り掛かります．

実際に作業に取り掛かる前に，物理データベース設計の目的をはっきりさせておきましょう．物理データベース設計では，実際に使用するデータベース製品を前提として，性能，記憶容量，プログラミングのしやすさ，保守のしやすさなどを考慮して論理データベース設計で作成したスキーマ定義を変換します．言い換えれば，そのデータベース製品が最大の性能を発揮できるような設計を行いますし，場合によってはそのデータベース製品しか持っていない機能を積極的に利用

注1）PostgreSQL 7.2については，『Software Design』の2001年11月号の特集「PostgreSQLパワーアップテクニック」に筆者による解説があります．

することもあります[注2].

したがって，物理データベース設計を行う前に，採用するデータベース製品が決まっていることが必要になります．今回はPostgreSQLを前提にして設計を行います．

キー項目の簡略化

図1とリスト1に論理データベース設計が完了した時点でのスキーマ定義を示します．

まず気が付くのは，ほとんどのテーブルで長い結合キーが使われていることです．「記事」テーブルに至っては，{出版社名,雑誌名,発行日,タイトル}というように，4つもの列の結合キーになっています．データベース理論的にはなんら問題はないのですが，実装上以下のような不便さがあります．

記憶容量の無駄

たとえば，出版社名などの文字列が平均して日本語で8文字だとすると，主キー1個あたり64バイトも消費してしまいます．PostgreSQLでは，文字列データ型では各文字列の先頭にバイト数を持つ4バイトのデータが別に必要になるため，実際には80バイト必要です．しかも「タイトル」を除く列はすべて「発行雑誌」テーブルなどにも存在する情報であり，無駄に感じます．

プログラミングの繁雑さ

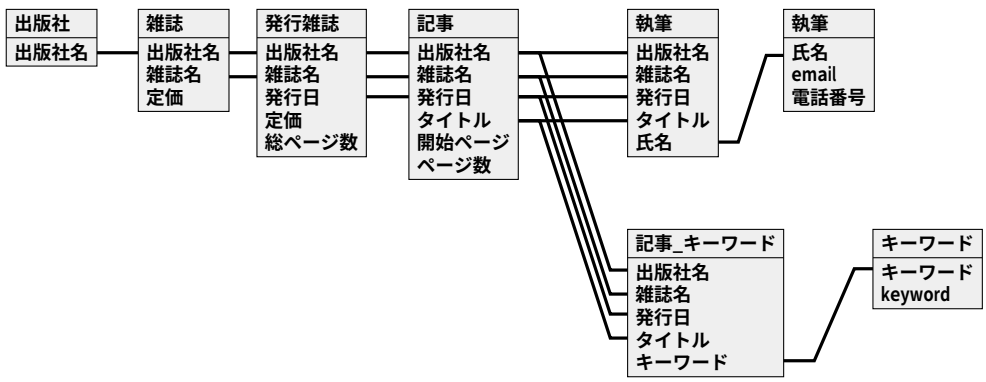
記事テーブルに1件行を追加することを考えてみます．「タイトル」「開始ページ」「ページ数」は記事固有の情報なので，新たに登録する必要があるのはやむを得ないことですが，「出版社名」「雑誌名」「発行日」は「発行雑誌」テーブルからコピーして持ってこなければいけません．これは繁雑ですし，間違いの元になります．

代理識別子の利用

キーは行を識別するために重複しない値を持っています，ということは，数字の1，2，3…のように重複しない値であれば「出版社名」「雑誌名」のような列値の組合わせでなくても，キーとして使用できるわけです．このように，列値の組み合わせの代りに使う人工的なキーを**代理識別子**と呼ぶことにします[注3].

PostgreSQLには，SERIAL型という特別なデータ型があります．たとえば，

図1　代理識別子を使わない場合の簡易ER図



注2）いつもそうするわけではありません．他のデータベース製品に移行できる可能性を残すために，あえて標準的なデータベースの機能しか使わないという選択もあり得ます．

注3）この用語は，参考文献1から拝借しました．一般的な用語かどうか筆者は知りませんが，なかなか適切な命名だと思います．

リスト1　論理データベース設計完了時点でのスキーマ定義

```
DROP TABLE 出版社;
CREATE TABLE 出版社 (
    出版社名 TEXT PRIMARY KEY,
    CONSTRAINT 出版社_出版社名_check CHECK(character_length(出版社名) > 0)
);

DROP TABLE 雑誌;
CREATE TABLE 雑誌 (
    出版社名 TEXT NOT NULL,
    雑誌名 TEXT NOT NULL,
    定価 INTEGER NOT NULL,
    PRIMARY KEY (出版社名,雑誌名),
    CONSTRAINT 雑誌_出版社名_check CHECK(character_length(出版社名) > 0),
    CONSTRAINT 雑誌_雑誌名_check CHECK(character_length(雑誌名) > 0),
    CONSTRAINT 雑誌_定価_check CHECK(定価 > 0)
);
ALTER TABLE 雑誌 ADD CONSTRAINT 雑誌_出版社名_外部キー
  FOREIGN KEY(出版社名) REFERENCES 出版社;

DROP TABLE 発行雑誌;
CREATE TABLE 発行雑誌 (
    出版社名 TEXT NOT NULL,
    雑誌名 TEXT NOT NULL,
    発行日 DATE NOT NULL,
    価格 INTEGER NOT NULL,
    総ページ数 INTEGER NOT NULL,
    PRIMARY KEY(出版社名,雑誌名,発行日),
    CONSTRAINT 発行雑誌_出版社名_check CHECK(character_length(出版社名) > 0),
    CONSTRAINT 発行雑誌_雑誌名_check CHECK(character_length(雑誌名) > 0),
    CONSTRAINT 発行雑誌_価格_check CHECK(価格 > 0),
    CONSTRAINT 発行雑誌_総ページ数_check CHECK(総ページ数 > 0)
);
ALTER TABLE 発行雑誌 ADD CONSTRAINT 発行雑誌_雑誌_外部キー
  FOREIGN KEY(出版社名,雑誌名) REFERENCES 雑誌;

DROP TABLE 記事;
CREATE TABLE 記事 (
    出版社名 TEXT NOT NULL,
    雑誌名 TEXT NOT NULL,
    発行日 DATE NOT NULL,
    タイトル TEXT NOT NULL,
    開始ページ INTEGER NOT NULL,
    ページ数 INTEGER NOT NULL,
    PRIMARY KEY(出版社名,雑誌名,発行日,タイトル),
    CONSTRAINT 記事_出版社名_check CHECK(character_length(出版社名) > 0),
    CONSTRAINT 記事_雑誌名_check CHECK(character_length(雑誌名) > 0),
    CONSTRAINT 記事_タイトル_check CHECK(character_length(タイトル) > 0),
    CONSTRAINT 記事_開始ページ_check CHECK(開始ページ > 0),
    CONSTRAINT 記事_ページ数_check CHECK(ページ数 > 0)
);
ALTER TABLE 記事 ADD CONSTRAINT 記事_発行雑誌_外部キー
  FOREIGN KEY(出版社名,雑誌名,発行日) REFERENCES 発行雑誌;

DROP TABLE 執筆;
CREATE TABLE 執筆 (
    出版社名 TEXT NOT NULL,
    雑誌名 TEXT NOT NULL,
    発行日 DATE NOT NULL,
    タイトル TEXT NOT NULL,
    氏名 TEXT NOT NULL,
    PRIMARY KEY(出版社名,雑誌名,発行日,タイトル,氏名),
    CONSTRAINT 執筆_出版社名_check CHECK(character_length(出版社名) > 0),
    CONSTRAINT 執筆_雑誌名_check CHECK(character_length(雑誌名) > 0),
    CONSTRAINT 執筆_タイトル_check CHECK(character_length(タイトル) > 0),
    CONSTRAINT 執筆_氏名_check CHECK(character_length(氏名) > 0)
);
ALTER TABLE 執筆 ADD CONSTRAINT 執筆_記事_外部キー
  FOREIGN KEY(出版社名,雑誌名,発行日,タイトル) REFERENCES 記事;
ALTER TABLE 執筆 ADD CONSTRAINT 執筆_執筆者_外部キー
  FOREIGN KEY(氏名) REFERENCES 執筆者;

DROP TABLE 執筆者;
CREATE TABLE 執筆者 (
    氏名 TEXT PRIMARY KEY,
    email TEXT UNIQUE,
    電話番号 TEXT UNIQUE,
    CONSTRAINT 執筆者_氏名_check CHECK(character_length(氏名) > 0)
);

DROP TABLE 記事_キーワード;
CREATE TABLE 記事_キーワード (
    出版社名 TEXT NOT NULL,
    雑誌名 TEXT NOT NULL,
    発行日 DATE NOT NULL,
    タイトル TEXT NOT NULL,
    キーワード TEXT NOT NULL,
    PRIMARY KEY(出版社名,雑誌名,発行日,タイトル,キーワード),
    CONSTRAINT 記事_キーワード_出版社名_check CHECK(character_length(出版社名) > 0),
    CONSTRAINT 記事_キーワード_雑誌名_check CHECK(character_length(雑誌名) > 0),
    CONSTRAINT 記事_キーワード_タイトル_check CHECK(character_length(タイトル) > 0),
    CONSTRAINT 記事_キーワード_キーワード_check CHECK(character_length(キーワード) > 0)
);
ALTER TABLE 記事_キーワード ADD CONSTRAINT 記事_キーワード_記事_外部キー
    FOREIGN KEY(出版社名,雑誌名,発行日,タイトル) REFERENCES 記事;
ALTER TABLE 記事_キーワード ADD CONSTRAINT 記事_キーワード_キーワード_外部キー
    FOREIGN KEY(キーワード) REFERENCES キーワード;

DROP TABLE キーワード;
CREATE TABLE キーワード (
    キーワード TEXT NOT NULL,
    keyword TEXT NOT NULL,
    PRIMARY KEY(キーワード),
    CONSTRAINT キーワード_キーワード_check CHECK(character_length (キーワード) > 0),
    CONSTRAINT キーワード_keyword_check CHECK(character_length(keyword) > 0)
);
```

```
CREATE TABLE t1(i SERIAL, j INTEGER);
```

というSERIAL型を使ったテーブルがあったときに，

```
INSERT INTO t1(j) VALUES(1);
```

というように，わざとiに対応する値を省略してデータを登録すると，1から始まる決して重複しない値がiに設定されます．たとえば，上記のINSERT文を2度実行すると，

```
 i | j
---+---
 1 | 1
 2 | 1
```

となり，jには重複した1が設定されているのに対し，iは最初は1，次は2というように重複しない値が自動的に設定されます．このようにSERIAL型は代理識別子の要件にぴったりなので，今回はSERIAL型を代理識別子に採用します．

図2にSERIAL型による代理識別子を使った簡易ER図を示します．前回作成した簡易ER図（図1）と比べてみると，かなりすっきりしたのがおわかりかと思います．

代理識別子を使用する場合の注意点

代理識別子を使用すると良いことばかりではありません．たとえばある記事を含む雑誌の出版社名を知りたいとしましょう．図1では，「記事」テーブルの中に出版社名も含まれているので，問い合わせは「記事」テーブルに対してだけ行えば十分です．

しかし，図2の場合には，「記事」テーブルから直接出版社名を得ることはできず，記事⇨発行雑誌⇨雑誌⇨出版社と外部キーの連鎖を辿る必要があります．具体的にSELECT文を示すと，

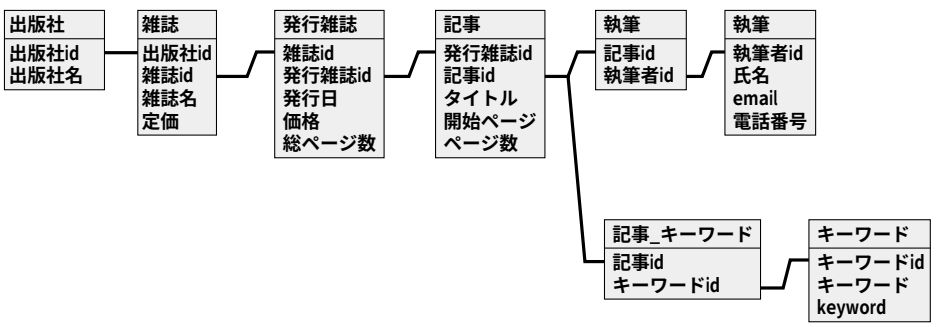
```
SELECT 出版社名 FROM 出版社,雑誌,発行雑誌,記事 WHERE
    出版社.出版社ID = 雑誌.出版社ID AND
    雑誌.雑誌ID = 発行雑誌.雑誌ID AND
    発行雑誌.発行雑誌ID = 記事.発行雑誌ID AND
    記事.タイトル = 'リレーショナルデータベース設計';
```

のような複雑なSQL文になってしまいます．性能的に不利になる傾向もあります．もっとも，このような問い合わせは最悪のケースであり，こうした問い合わせがまれにしか必要にならないのであれば，さほど，問題にならないと言えるでしょう．システムに対する要件を整理し，よく出現する問い合わせが低速にならないかどうかよく検討する必要があります．

インデックスとは

インデックスとは，問い合わせ処理を高速化するための付加的なデータ構造です[注4]．インデックスがなく

図2 SERIAL型による代理識別子を使った場合の簡易ER図



注4）インデックスで高速化されるのはSELECT文だけではありません．UPDATE文やDELETE文でも，WHERE句を使用するものはインデックスによって高速化されます．

4 データベース設計（3）

てもデータベースの機能が損なわれたり，誤った結果を返すことはありません．もっぱらインデックスは性能にのみ関係しますが，実際問題として，インデックスなしにはデータベースは実用にならないと言っても過言ではありません．したがって，インデックスの設計はデータベースの物理設計においても重要なポイントになります．

インデックスの種類

インデックスにはいくつかの種類があり，目的によって使い分けます．

● Btree

最も一般的なインデックスです．WHERE a = b のような等号を含む問い合わせはもちろん，a < b のような大小比較も高速化します．

● Hash

Btreeとは異なり，基本的に等号を含む問い合わせだけを高速化することができます．

● Rtree

PostgreSQLには実装されていますが，ほかのデータベースではほとんど見ることのできない特殊なインデックスです．矩形などの2次元以上のデータの比較を高速に行うことができます．地理情報システムなどに利用されています．

◇ ◇ ◇

インデックスの種類はSQL標準で規定されていませんので，データベース製品によって使用できるインデックスの種類は異なります．ここでは，多くのデータベース製品でサポートされ，PostgreSQLでも利用できるBtreeインデックスについて説明します．以後，単に「インデックス」と言う場合，Btreeインデックスのことを指すものとします．

Btreeインデックスの動作原理

これからご紹介するのは，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，本稿では単にBtreeと言った場合，B+treeを指すものとします．

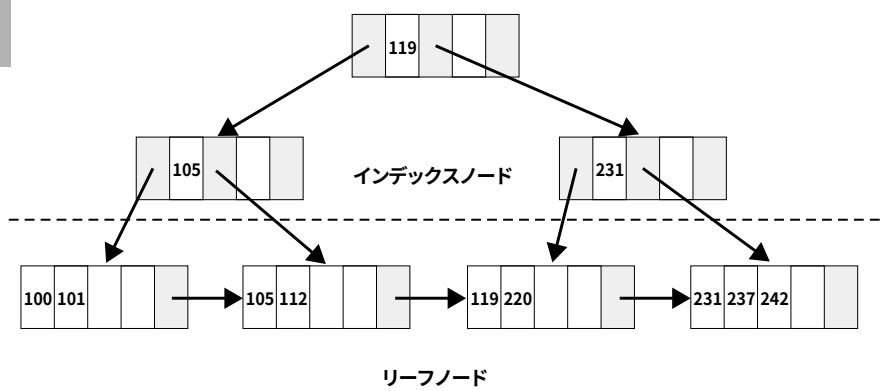
今，整数型の列

```
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3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

図3 Btreeの例



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．このポインタをたどると，昇順にソートされた列の値を得ることができます．また，この図には書いてありませんが，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，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，1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．

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付けます．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たどって右から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を見付けます．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付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の実装もその位です）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，目的のデータを見付けるまでのアクセス回数はせいぜい2～3回程度です[注5]．

Btreeの構造は図3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．

インデックスの作成

実際にインデックスを作成するためには，CREATE INDEXというSQL文を使います．データベース製品によって構文は多少異なりますが，PostgreSQLの場合，基本的な構文は以下のようになります．

```
CREATE INDEX インデックス名 ON テーブル名(列名[..列名]);
```

つまり，インデックスは列または列の組み合わせに対して設定します．例を示します．「発行雑誌」テーブルの「発行日」列にインデックスを作成するには，以下のSQL文を入力します．

```
CREATE INDEX 発行雑誌_発行日インデックス ON 発行雑誌(発行日); (実際は1行)
```

インデックスが設定された列に対するWHERE句の条件を含むSQL文は，インデックスによって問い合わせが高速化されます．それ以外の列は高速化の対象に

注5）Btreeインデックスのデータ量に関する詳しい見積もりは，参考文献[1]などを参照してください．

はなりません．ではすべての列にインデックスを設定すれば良いかというと，残念ながらそうではありません．

インデックスを作成すると，問い合わせ処理は高速化されますが，その分余計にディスクスペースを消費しますし，データの更新時にはインデックスを更新する分だけシステムに負担がかかります．したがって，むやみにインデックスを作成すべきではありません．

また，重複が多いデータに対してインデックスを作るのも得策ではありません．たとえば性別のように，2種類の値しかない列に対してインデックスを作っても実際には使用されないので，更新処理が遅くなるだけ無駄です．

自動的に作成されるインデックス

ほとんどのデータベース製品でもそうですが，PostgreSQLの場合も特定の場合には自動的にインデックスが作成されます．

- 主キー
- ユニークキー

したがって，これらの列に対してあえてインデックスを作成する必要はありません．

マルチカラムインデックス

複数の列に対してまとめて作成するインデックスを**マルチカラムインデックス**（multi column index）と呼びます．たとえば，i，jという列があり，以下のような問い合わせが頻繁にあるとします．

```
SELECT * FROM t1 WHERE i = 10 AND j = 100;
```

このような場合

```
CREATE INDEX t1index ON t1(i,j);
```

とすると，問い合わせにインデックスが使われます．ただし，同じi,jに対する問い合わせでも，

```
SELECT * FROM t1 WHERE i = 10 OR j = 100;
```

のようなOR条件による問い合わせに対しては，j = 100の部分に対してインデックスは使われません[注6]．

また，このインデックスは，

```
SELECT * FROM t1 WHERE i = 10;
```

のような問い合わせにも使われます．しかし，

```
SELECT * FROM t1 WHERE j = 100;
```

には使われません．これは，（i，j，k…）のマルチカラムインデックスは，1番左の列であるiを含み，かつANDで結合された問い合わせに対してインデックスに対してのみ使用できるからです．少々わかりにくいかもしれないので，まとめてみましょう．

- インデックスが使われる問い合わせの例

```
i = 0
i = 0 AND j = 1
i = 0 AND k = 2
i = 0 AND j = 1 AND k = 2
```

- インデックスが使われない問い合わせの例

```
j = 1
k = 2
i = 0 OR j = 1
j = 1 AND k = 2
```

このように，マルチカラムインデックスでは，そのものずばりの列の組み合わせだけではなく，他の場合にもインデックスを有効利用できる場合があります．実際に使われる問い合わせで必要になる最小限のインデックスを作るように工夫しましょう[注7]．

部分インデックス

PostgreSQL 7.2から**部分インデックス**というものが使えるようになります．

たとえば，商品の販売日，販売数などを格納したデータベースを考えます．商品の売れ行きは片寄って

注6）別にjに対してもインデックスを作ればもちろんインデックスが使われるようになります．
注7）マルチカラムインデックスを有効に使ってくれるかどうかは，そのデータベース製品の「賢さ」にもよります．ここでの結果は，あくまでPostgreSQLによるものです．

おり，売れている商品の数はごくわずかだとします．そこでもっぱら売れている商品に関心が集まり，検索も売れている商品に対してもっぱら行うとします．この場合，普通のインデックスを作ると，販売数が少ない商品がインデックスのデータの大半を占めるわけですが，それらは実際には検索対象にならないので無駄になります．そこで，実際に検索の対象となるわずかの商品に対してだけインデックスを作ることを考えます．これが部分インデックスです．

部分インデックスの使い方は以下のようになります．

```
CREATE INDEX aindex on t1(amount) WHERE amount
> 1000;（実際は1行）
```

これで販売数が10000以上の商品にのみインデックスが作成されます．部分インデックスはデータの無駄が少なく効率が良いだけでなく，インデックスのサイズが減少することによって検索効率も高まります．

インデックスが使われているかどうかの確認

本当にインデックスが使われているかどうか自信がないときは，EXPLAINコマンドで確認しましょう．EXPLAINの使い方は簡単で，単にEXPLAINの後にSELECT文を続けて書くだけです．EXPLAINは実際に問い合わせを実行はせず，代わりに問い合わせの実行方法を表示します．

例を示します．

```
test=# EXPLAIN SELECT * FROM t1 WHERE i = 1
AND j = 1;（実際は1行）
NOTICE:  QUERY PLAN:

Index Scan using t1index on t1  (cost=0.00
..14.63 rows=10 width=12)（実際は1行）

EXPLAIN
```

“Index Scan”とあれば，インデックスが使われています．逆に，

```
test=# EXPLAIN SELECT * FROM t1 WHERE j = 1
AND k = 1;（実際は1行）
NOTICE:  QUERY PLAN:

Seq Scan on t1  (cost=0.00..2089.00 rows=10
width=12)（実際は1行）

EXPLAIN
```

のように，“Seq Scan”と表示された場合はインデックスは使われていないことになります．

なお，EXPLAINコマンドはPostgreSQL固有のものですが，たいていのデータベース製品にも似たような機能が実装されているはずです．

インデックスが使われない？

インデックスを作ったし，問い合わせもインデックスを利用しても良いように見えるのに，なぜかインデックスが使われないことがあります．これはインデックスを使わずにテーブルを直接見たほうが速い場合があるからです．例をあげましょう．

●重複が多い

先ほども述べたように，性別のように非常に重複が多いデータではインデックスは使われません．

●データ件数が少ない

データ件数が小さい場合は，テーブル全体を1度にメモリにロードできるため，わざわざインデックスを見る必要がありません．この場合もインデックスは必要ありません．

●出力データ件数が多い

検索の結果出力されるデータ件数が多いと予想される場合は，インデックスが使われません．たとえば，WHERE句がないSELECTではテーブルの全データが抽出されますが，この場合はテーブル本体を直接見たほうが効率が良くなります．

●文字列検索

文字列の検索では，完全一致あるいは前方一致検索以外はインデックスは使われません．前方一致検索

4 データベース設計（3）

とは，たとえば『“あ”で始まる文字列」のような検索条件です．具体的には，

```
WHERE t LIKE 'あ%'
WHERE t ~ '^あ'
```

のような検索条件です．最初のほうはLIKE検索で，どのデータベースシステムでも使えます．後のほうは**正規表現検索**というPostgreSQL固有の機能です．

●VACUUMをかけていない

データベースがインデックスを使うべきかどうか判断するためには，そのための統計情報が必要です．この情報にはデータ件数，データの重複率などが含まれます．VACUUMによってこの統計情報を更新しないと正確な判断が下せないので，インデックスが使われないことがあります．なお，VACUUMはPostgreSQL固有のコマンドです．

実際にインデックスを追加する

それでは今までの議論に基づき，インデックスを追加してみましょう．ただし，主キーやUNIQUE制約が課せられた列にはすでにインデックスが存在するため，まずそれを確認します．

表1で，“◉”印は単独の列に付けられたインデックス，列名1→列名2は（列名1，列名2）のマルチカラムインデックスを表します．

ご欄のように，すでに多くの列にインデックスが設定されていることがわかります．後は実際にアプリケーションで頻繁に必要となるような列（列の組み合わせ）にインデックスを付けていきます．連載の順序の関係でまだアプリケーションの設計ができていないので，本当はこの段階ではどの列に対する検索が多いかはわからないのですが，とりあえず記事のタイトルと執筆者名の検索が多そうだということにして，インデックスを付けてみましょう．

まず記事タイトルですが，すでに（タイトル，発行雑誌ID）の組に対するインデックスが存在しているので，タイトルに対するインデックスは必要ありません．

次に執筆者名ですが，これはまだインデックスが付いていないので早速追加します．

```
CREATE INDEX 執筆者_氏名インデックス ON 執筆者(氏名);
```

ここでは，インデックスの名前は「テーブル名＋列名」というルールにしています．これは別にこうしな

▼表 1-1　「出版社」テーブル

出版社ID	◉
出版社名	◉

▼表 1-2　「雑誌」テーブル

出版社ID	
雑誌ID	◉
雑誌名	
定価	

▼表 1-3　「発行雑誌」テーブル

雑誌ID	
発行雑誌ID	◉
発行日	
価格	
総ページ数	

▼表 1-4　「記事」テーブル

発行雑誌ID	
記事ID	◉
タイトル	
開始ページ	
ページ数	

▼表 1-5　「執筆者」テーブル

執筆者ID	◉
氏名	
email	
電話番号	

▼表 1-6　「執筆」テーブル

記事ID	
執筆者ID	

▼表 1-7　「キーワード」テーブル

キーワードID	◉
キーワード	
keyword	

▼表 1-8　「記事_キーワード」テーブル

記事ID	
キーワードID	

ければならない，というわけではなく，単にデータベース内でインデックスの名前が重複しないように，またどの列に対するインデックスかわかりやすいようにこのようにしているだけです．きちんとドキュメントで管理できるのなら，単なる連番でインデックスの名前を付けても一向に構いません．

これ以外のインデックスについては，アプリケーションの設計ができてから追加しても遅くありません．一般に，インデックスを付けすぎるよりは，足りないほうがましです．インデックスが足りない場合は，検索速度が落ちるのですぐにわかりますし，インデックスは後からでも容易に追加できるからです．逆にインデックスが多すぎる場合，無駄なインデックスがあることはわかりづらく，知らない間に更新処理に負担がかかっていることがあります．

テーブル名，列名などの日本語を英語に

さて，ここまではわかりやすさのためにテーブル名，列名などに日本語を使ってきたわけですが，最後にそれらを英語に直します．プログラミングの際には日本語を使用することは稀です．また，プログラミング言語によっては日本語の扱いに問題があるものもあります．そうした制約を取り除くために，あえて英語に直すわけです．

いきなり日本語に変更するよりは，まず「対訳表」を作るのが良いでしょう（表2）．なお，実際に英訳するときは，テーブル名は複数形にするのが一般的のようです．

PostgreSQLは内部的に名前を生成します．たとえば，

```
iss_magID SERIAL
```

という列があると，“iss_mags_iss_magid_seq”というシーケンスオブジェクトを内部的に作ります．ところがこの名前は31文字以内という制限があり，31文字を超えると適当に31文字以下に省略してしまいます．このため人間にはどのような名前が付くのか検討が付かなかったり，ときには名前が衝突してしまうこともあります．そこで，“issue_magazines”のような長い名前はあらかじめ適当に省略しているわけです．苦しいところもありますが，ご容赦ください．

また，magIDの“ID”のような大文字は，PostgreSQLの内部では小文字に変換して扱われます．つまり，大文字／小文字の区別はできないので，その点もご注意ください[注8]．

では，完成したスキーマ定義をリスト2に，また例によってER図もどきを図4に示します．

まとめ

今回は，データベースの物理設計を行いました．その場合のテクニックとして，代理識別子，インデック

▼表2　対訳表

日本語	英語
出版社	publisher
出版社名	pb_name
雑誌	magazine
雑誌名	mag_name
定価	list_price
発行雑誌	iss_mag
発行日	issue_date
価格	price
総ページ数	total_pages
記事	article
タイトル	title
開始ページ	start_page
ページ数	num_pages
執筆者	writer
氏名	wrt_name
email	email
電話番号	ph_number
執筆	writing
キーワード	keyword
keyword	english_keyword
外部キー	fkey
インデックス	idx

注8）テーブル名や列名などの識別子において大文字・小文字の区別を行わないのはSQLの仕様です．

リスト2　物理設計の完成したスキーマ定義

```
DROP SEQUENCE publishers_publisherID_SEQ;
DROP TABLE publishers;
CREATE TABLE publishers (
    publisherID SERIAL NOT NULL,
    PRIMARY KEY(publisherID),
    pb_name TEXT NOT NULL UNIQUE,
    CONSTRAINT publishers_pb_name_check CHECK(character_length(pb_name) > 0)
);

DROP SEQUENCE magazines_magazineID_SEQ;
DROP TABLE magazines;
CREATE TABLE magazines (
    publisherID INTEGER NOT NULL,
    magazineID SERIAL NOT NULL,
    mag_name TEXT NOT NULL,
    list_price INTEGER NOT NULL,
    PRIMARY KEY(magazineID),
    UNIQUE(mag_name,publisherID),
    CONSTRAINT magazines_mag_name_check CHECK(character_length(mag_name) > 0),
    CONSTRAINT magazines_list_price_check CHECK(list_price > 0)
);
ALTER TABLE magazines ADD CONSTRAINT magazines_pb_name_fkey
      FOREIGN KEY(publisherID) REFERENCES publishers;

DROP SEQUENCE iss_mags_iss_magID_SEQ;
DROP TABLE iss_mags;
CREATE TABLE iss_mags (
    magazineID INTEGER NOT NULL,
    iss_magID SERIAL NOT NULL,
    issue_date DATE NOT NULL,
    price INTEGER NOT NULL,
    total_pages INTEGER NOT NULL,
    PRIMARY KEY(iss_magID),
    UNIQUE(magazineID,issue_date),
    CONSTRAINT iss_mags_price_check CHECK(price > 0),
    CONSTRAINT iss_mags_total_pages_check CHECK(total_pages > 0)
);
ALTER TABLE iss_mags ADD CONSTRAINT iss_mags_magazine_fkey
      FOREIGN KEY(magazineID) REFERENCES magazines;

DROP SEQUENCE arctiles_arctileID_SEQ;
DROP TABLE arctiles;
CREATE TABLE arctiles (
    iss_magID INTEGER NOT NULL,
    arctileID SERIAL NOT NULL,
    title TEXT NOT NULL,
    start_page INTEGER NOT NULL,
    num_pages INTEGER NOT NULL,
    PRIMARY KEY(arctileID),
    UNIQUE(title,iss_magID),
    CONSTRAINT arctiles_title_check CHECK(character_length(title) > 0),
    CONSTRAINT arctiles_start_page_check CHECK(start_page > 0),
    CONSTRAINT arctiles_num_pages_check CHECK(num_pages > 0)
);
ALTER TABLE arctiles ADD CONSTRAINT arctiles_iss_mags_fkey
      FOREIGN KEY(iss_magID) REFERENCES iss_mags;

DROP SEQUENCE writers_writerID_SEQ;
DROP TABLE writers;
CREATE TABLE writers (
    writerID SERIAL NOT NULL,
    wrt_name TEXT NOT NULL,
    email TEXT UNIQUE,
    ph_number TEXT UNIQUE,
    PRIMARY KEY(writerID),
    CONSTRAINT writers_wrt_name_check CHECK(character_length(wrt_name) > 0)
);
CREATE INDEX writers_wrt_name_idx ON writers(wrt_name);

DROP TABLE writings;
CREATE TABLE writings (
    arctileID INTEGER NOT NULL,
    writerID INTEGER NOT NULL,
    PRIMARY KEY(arctileID,writerID)
);
ALTER TABLE writings ADD CONSTRAINT writings_arctile_fkey
      FOREIGN KEY(arctileID) REFERENCES arctiles;
ALTER TABLE writings ADD CONSTRAINT writings_writer_fkey
      FOREIGN KEY(writerID) REFERENCES writers;

DROP SEQUENCE keywords_keywordID_SEQ;
DROP TABLE keywords;
CREATE TABLE keywords (
    keywordID SERIAL NOT NULL,
    keyword TEXT NOT NULL,
    english_keyword TEXT NOT NULL,
    PRIMARY KEY(keywordID),
    UNIQUE(keyword,english_keyword),
    CONSTRAINT keywords_keyword_check CHECK(character_length (keyword) > 0),
    CONSTRAINT keywords_english_keyword_check CHECK(character_length(english_keyword) > 0)
);

DROP TABLE arctiles_keywords;
CREATE TABLE arctiles_keywords (
    arctileID INTEGER NOT NULL,
    keywordID INTEGER NOT NULL,
    PRIMARY KEY(arctileID,keywordID)
);
ALTER TABLE arctiles_keywords ADD CONSTRAINT arctiles_keywords_arctile_fkey
      FOREIGN KEY(arctileID) REFERENCES arctiles;
ALTER TABLE arctiles_keywords ADD CONSTRAINT arctiles_keywords_keyword_fkey
      FOREIGN KEY(keywordID) REFERENCES keywords;
```

ス，識別子の英語化を解説しました．特にインデックスはデータベースのパフォーマンスに関わる部分なので，少し詳しく解説を行いました．お陰でまたもや予定より早く誌面が尽きてしまいましたが，次回はいよいよPHPを使って実際にアプリケーションを作成しながら「トランザクション設計」について学ぶ予定です．

参考文献

[1]『トランザクション処理（下）』／Jim Gray，Andreas Reuter／日経BP／2001

図4　物理設計の完成したスキーマ定義のER図もどき