さわって実感

リレーショナルデータベース設計 5

トランザクション設計

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

PostgreSQL 7.2リリース

ついに待望のPostgreSQL 7.2がリリースされました．今回のバージョンでは，スケーラビリティ，可用性など，エンタープライズ向け用途を強く意識した改良が加えられています．詳細は本号の別掲記事（153ページ）をご覧ください．

今回の狙い

前回でデータベースの設計が完了したので，今回からプログラムの設計を行いますが，その前にデータベースの動的な振る舞いを決定する大事な要素であるトランザクションを設計しましょう．

トランザクションとは

トランザクションとは，データベースに対して行う操作の単位です．プログラムの実行単位がプロセスであるのと対比されます．

トランザクションの範囲外で実行されるような操作は，データベースの初期立ち上げなどの特殊な場合を除きあり得ません．また，トランザクションには必ず開始と終了があります．

トランザクションの開始

SQLコマンドBEGINによってトランザクションの開始を宣言します．データベース製品によっては何もしなくても自動的にトランザクションが開始するものもありますが，PostgreSQLではBEGINを使う必要があります．

では，BEGINをせずにいきなりSELECT……などとするとどうなるのでしょう？ この場合，PostgreSQLはそれぞれのSQLコマンドを1つのトランザクションとして実行します．

トランザクションの終了

トランザクションの中では，一連のSQLコマンドを実行できます．SQLコマンドはいくつあっても構いません．それらのSQLコマンドの結果を確定（コミット）させたい場合はCOMMITコマンドでトランザクションを終了させます．

逆に，一連のSQLコマンドの効果を取り消したい場合はROLLBACKコマンドを実行します．こうすると，どんなにたくさんのINSERTやUPDATE，DELETEが実行済であってもすべてのコマンドの効果は打ち消され（ロールバック），データベースの状態はBEGINコマンドを実行する以前の状態に戻ります[注1]．

また，何らかの障害がシステムに発生した場合，自

注1）後述のように，シーケンス関係の一部のコマンドはロールバックされません．また，CREATE DATABASE，DROP DATABASE，TRUNCATEコマンドはトランザクションブロック（BEGINからCOMMITないしROLLBACKまで）の中では実行できないため，必然的にロールバックの対象になりません．

5 トランザクション設計

動的にトランザクションはロールバックされます．

トランザクションの原子性

このことは非常に便利であるだけでなく，データベースの信頼性を保つ上でも重要です．たとえば，銀行口座Aから別の口座Bに1万円のお金を移す処理を考えましょう．次のようなSQL操作になります．

```
-- トランザクションの開始
BEGIN;

-- 口座Aから1万円を引き出す：
UPDATE 銀行口座 SET 金額 = 金額 - 10000 WHERE 口座
= 'A';

-- 口座Bに1万円を振り込む：
UPDATE 銀行口座 SET 金額 = 金額 + 10000 WHERE 口座
= 'B';

-- トランザクションの終了
COMMIT;
```

ここで，1番目のUPDATE文を実行した後，2番目のUPDATE文を実行する前にDBサーバとの接続回線が切れてしまったとします．この場合，PostgreSQLは自動的にトランザクションをロールバックし，1番目のUPDATEがなかったことにしてくれます（そうでなければ口座Aの1万円が空中に消えてしまったことになります）．もしもトランザクションがなければ，1番目のUPDATEを打ち消すような操作，つまり

```
UPDATE 銀行口座 SET 金額 = 金額 + 10000 WHERE 口座
= 'A';
```

をしなければならなかったことでしょう．この例では少数のSQLしか実行していないので操作の取消しを自分で行っても簡単ですが，たくさんのSQLを実行していたり，複雑な条件分岐をしている場合は，以前行った操作をすべて覚えておくのは事実上不可能です．

このように，トランザクションは，たくさんのSQLコマンドが完全に実行されたか，あるいはまったく実行されなかったかのどちらかの状態しか取らない性質があります．このことをトランザクションの原子性と呼び，トランザクションの重要な性質の1つです．

原子性の確認

トランザクションの原子性について，実際に確かめてみます．

まず，テスト環境を作りましょう．以下の例では，データベースとして“webdb”，ユーザ名も“webdb”とします．以下のように登録しておいてください（postgresユーザで実行します）．

```
$ createuser webdb
Shall the new user be allowed to create
databases? (y/n) y
Shall the new user be allowed to create more
new users? (y/n) n
CREATE USER

$ createdb webdb
CREATE DATABASE
```

次にテーブルを作ります．これは前回作成したテーブル定義を使用します[注2]（リスト1）．これを“create.sql”というファイル名でセーブしておいてください．そして，以下のようにpsqlコマンドで登録します．

```
$ psql -U webdb -e -f create.sql
```

ここで“-U webdb”は，webdbユーザでデータベースにログインすることを指示します[注3]．このコマンドを実行すると，図1のようなログが吐き出されます．

続いて，とりあえず技術評論社のデータをpublishersテーブルに登録してみましょう．

```
webdb=> INSERT INTO publishers(pb_name)
VALUES('技術評論社');
```

注2）http://www.gihyo.co.jp/wdpress/から入手できます．
注3）webdbユーザにパスワードを設定している場合は，パスワードの入力が求められます．

リスト1　今回使用するテーブル定義

```
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;
```

5 トランザクション設計

図1　テーブルの作成ログ（抜粋）

```
DROP SEQUENCE publishers_publisherID_SEQ;
psql:create.sql:1: ERROR:  sequence "publishers_publisherid_seq" does not exist
DROP TABLE publishers;
psql:create.sql:2: ERROR:  table "publishers" does not exist
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)
);
psql:create.sql:8: NOTICE:  CREATE TABLE will create implicit sequence 'publishers_publisherid_seq'
for SERIAL column 'publishers.publisherid'
psql:create.sql:8: NOTICE:  CREATE TABLE/PRIMARY KEY will create implicit index 'publishers_pkey' for
table 'publishers'
psql:create.sql:8: NOTICE:  CREATE TABLE/UNIQUE will create implicit index 'publishers_pb_name_key'
for table 'publishers'
CREATE
DROP SEQUENCE magazines_magazineID_SEQ;
psql:create.sql:10: ERROR:  sequence "magazines_magazineid_seq" does not exist
DROP TABLE magazines;
psql:create.sql:11: ERROR:  table "magazines" does not exist
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)
);
(以下，省略)
```

```
INSERT 4454788 1
webdb=> SELECT * FROM publishers;
 publisherid |  pb_name
-------------+------------
           1 |  技術評論社
(1 row)
```

上のようにpb_nameのデータだけを登録し，publisheridはデータを指定していませんが，“1”が登録されています．これはpublisheridのデータ型がSERIALになっているからです．

ではさっそくトランザクションの効果を試してみましょう．図2のようになります．ご覧のように，ROLLBACKを使って間違った更新の取消しができています．

ただし，いくつか注意する点があります．

●SERIAL型と連続値の取り扱い

publisheridが1（技術評論社）から3（PostgreSQL出版）に飛んでいます．これは，“INSERT INTO publishers(pb_name) VALUES('技術評論者');”によって一度2というpublisheridが割り当てられたものの，その後ROLLBACKされたためにその値が捨てられたからです．PostgreSQLのSERIAL型（およびその実装に使われているSEQUENCE）では，一度割り当てられた値はたとえROLLBACKされても元には戻りません．したがって，このように必ずしも連続した値を取らないことがあり得ます．今回はpublisheridとして，とにかく一意の値が得られれば十分なのでSERIAL型を使っていますが，どうしても連続した値が必要な場合は別の実装を考える必要があります．

●自動アボートとエラー処理

“INSERT INTO publishers(pb_name) VALUES('技術評論社');”がエラーになったために，トランザクションが自動的にアボートしています．こうなると以後ROLLBACKを入力するまではどんなSQLコマンドも受け付けなくなります．しかも，SQLコマンドを投入しても“NOTICE: current transaction is aborted, queries ignored...”のようなメッセージが返ってくる

図2　トランザクションの効果を検証

```
webdb=> BEGIN;                                                    ← トランザクションの開始
BEGIN
webdb=> INSERT INTO publishers(pb_name) VALUES('技術評論者');      ← 間違ったデータを登録してしまった！
INSERT 4454789 1
webdb=> ROLLBACK;                                                 ← でも取り消しできるから大丈夫
ROLLBACK
webdb=> BEGIN;                                                    ← 別のトランザクションの開始
BEGIN
webdb=> INSERT INTO publishers(pb_name) VALUES('PostgreSQL出版');  ← 別のデータを登録
INSERT 4454790 1
webdb=> COMMIT;
COMMIT
webdb=> BEGIN;
BEGIN
webdb=> DELETE FROM publishers WHERE pb_name = 'PostgreSQL出版';   ← 間違ったデータを消してしまった！
DELETE 1
webdb=> SELECT * FROM publishers;                                 ← 確かに消えている
 publisherid |  pb_name
-------------+------------
           1 | 技術評論社
(1 row)

webdb=> ROLLBACK;                                                 ← でも取り消しできるから大丈夫
ROLLBACK
webdb=> SELECT * FROM publishers;                                 ← 復活していることを確認
 publisherid |    pb_name
-------------+----------------
           1 | 技術評論社
           3 | PostgreSQL出版
(2 rows)

webdb=> BEGIN;
BEGIN
webdb=> INSERT INTO publishers(pb_name) VALUES('技術評論社');      ← 同じ出版社を重複して登録したのでエラーとなり，自動的にトランザクションがアボート
ERROR:  Cannot insert a duplicate key into unique index publishers_pb_name_key
webdb=> SELECT * FROM publishers;                                 ← 一度トランザクションがアボートすると，ROLLBACKを入力するまではコマンドを受け付けない
NOTICE:  current transaction is aborted, queries ignored until end of transaction block
*ABORT STATE*
webdb=> ROLLBACK;
ROLLBACK
```

だけで，SQLの実行結果はエラーになっていません．このために，実行したつもりのSQLが効いていない…というわかりにくいトラブルに巻き込まれることがあります．したがって，INSERTを実行した段階でエラーになったことをきちっと認識してエラー処理をすることが重要です．

トランザクションによる同時実行制御

ここまでは，単一のトランザクションの性質を見てきました．しかしトランザクションの機能はこれだけにとどまりません．トランザクションのもう1つの重要な性質が**同時実行制御**（Concurrency Control）です．これは，複数のトランザクションが同時に実行された場合でもデータベースのデータがおかしなことにならないように制御する機能です．

では，複数のトランザクションを実行するときに考慮すべき点を見ていきましょう．

基本的な排他制御

PostgreSQLでは，テーブルは1個のファイルとして実装されています[注4]．また，テーブルの中には0個以上の行があります．

INSERT時には，このファイルの最後尾に行を構成するデータを追加していきます．もしも複数のトランザクションが何も排他制御せずにデータをファイルに追加していったとしたら，それぞれのデータが混じり合った行が追加されてしまうかもしれません．このようなことが決して起きないように，データベースシステムは排他制御を実施します．

注4）実際には大きなテーブル（1Gバイトを超えるようなもの）は複数の物理的なファイルに分割されるような実装になっていますが，ここでは論理的に1個のファイルであるとみなしています．

5 トランザクション設計

ダーティーリードの防止

ダーティーリード（dirty read）とは，あるトランザクションが更新中のデータが他のトランザクションから読めてしまう現象です．たとえば，先の例では「技術評論者」という誤った出版社名を登録してしまいました．もしこれが他のトランザクションから見えてしまうといろいろと不都合が起きるでしょう．

たとえば，データベースから出版社一覧表を作成してWebで公開することを考えると，「技術評論者」という誤ったデータが検索エンジンに登録されてしまうかもしれません．実際には，この誤ったINSERTはROLLBACKされていますから，最終的にはデータベースには存在しません．ですから，これは矛盾です．

PostgreSQLは，COMMITによって確定したデータだけを他のトランザクションに見せることによってこの問題を解決しています．

ノンリピータブルリード，ファントムの防止

しかし，逆にいうと他のトランザクションがコミットした結果は見えてしまうわけです．その結果，以下のようなことが起こり得ます．

- 他のトランザクションがあるデータを更新してコミットした結果，別のトランザクションではそのデータの値が，以前読み込んだときと異なります[注5]．これを**ノンリピータブルリード**（nonrepeatable read）と呼びます．

- 他のトランザクションがあるデータを追加してコミットした結果，別のトランザクションで以前読み込んだときにはなかったデータが読み込めるようになります．これを**ファントム**（phantoms）と呼びます．

トランザクションの分離レベル

SQL標準では，ダーティーリード，ノンリピータブルリード，ファントムの発生具合によってトランザクション管理のレベルを4つに分けて定義しています．これを**トランザクションの分離レベル**（Transaction Isolation Level）と呼びます（**表1**）．

PostgreSQLでは，このうちリードコミッティドとシリアライザブルをサポートしており，多くの商用データベース同様，デフォルトのトランザクション分離レベルはリードコミッティドです．設定によってシリアライザブルも利用できます．

シリアライザブルの利用

PostgreSQLでは，デフォルトのトランザクション分離レベルからより高度なシリアライザブルに移行するためには，3つの方法があります．

① “ SET TRANSACTION ISOLATION LEVEL SERIALIZABLE; ” を “ BEGIN; ” の直後に実行します．この方法では，COMMITまたはROLLBACKによってこのトランザクションが終了するまではシリアライザブルになります．

② “ SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL SERIALIZABLE; ” を実行します．この方法では，このデータベースセッション[注6]が終了するまでシリアライザブルになります．

▼表1　トランザクションの分離レベルと排他制御のスコープ

分離レベル	ダーティーリード	ノンリピータブルリード	ファントム
リードアンコミッティド（read uncommitted）	起きる	起きる	起きる
リードコミッティド（read committed）	起きない	起きる	起きる
リピータブルリード（repeatable read）	起きない	起きない	起きる
シリアライザブル（serializable）	起きない	起きない	起きない

注5）ここで「以前」と言っているのは，そのトランザクションがBEGINコマンドによって開始された以降，現時点までの間のどこかを指しています．トランザクション開始以前ではありません．以下同様．

注6）あるユーザがデータベースにログインしてからログアウトするまでの間のことを指します．

③postgresql.conf（通常/usr/local/pgsql/data/にある設定ファイル）中に，“default_transaction _isolation = 'serializable' ”と書きます．この方法では，すべてのトランザクションのデフォルトのトランザクション分離レベルがシリアライザブルになります[注7]．

シリアライザブルのメリットとデメリット

シリアライザブルは，高度なトランザクションの分離性を提供します．あるトランザクションがデータを更新中であっても，シリアライザブルで実行中のトランザクションではそのトランザクションが開始した時点のデータだけが参照できます．時間はかかるが参照のみを行う集計処理のようなトランザクションは，シリアライザブルで実行するのが最適です[注8]．

しかし，更新処理を伴う場合は考慮が必要です．シリアライザブルでは，同じ行に対して複数のトランザクションが同時に更新を行うとトランザクションがアボートしてしまうのです．

例を挙げます．以下のように，アクセスカウントに使用するテーブルt1があったとします．このテーブルには初期値として0が1行だけ入っています．

```
CREATE TABLE t1(i INTEGER);
INSERT INTO t1 VALUES(0);
```

ここでカウンタをカウントアップする2つのトランザクションをシリアライザブルで実行します．2つの端末ウィンドウを開き，それぞれpsqlを上げて実験します．2つのpsqlでのSQLコマンドの実行順序に関わる問題ですので，以下，1番目のpsqlのセッションをT1，2番目をT2として表し，前後関係がわかりやすくなるようにします（図3）．

ここでは，同じ行に対して更新処理を行ったため，後からUPDATEを実行したT2は結局エラーとなってしまいました．もしT2が最初iが0だったことを重視してiを1にすると，T1の更新が消えてしまったことになります．逆に，T1の更新を尊重すると，iの値を2にしなければなりませんが，これはT2の開始時にiが0であったことと矛盾します．ジレンマを解消するためには，T2がなかったことにするしかなく，T2がアボートしたわけです．

このようなトランザクションのアボートを回避するには，明示的なロックを使用します．PostgreSQLにはさまざまな強さを持つロックコマンドがあります．今回のような場合は，参照は許すが更新は同時に1トランザクションだけに限定するロックコマンドを使用します（図4）．

“LOCK t1 IN SHARE ROW EXCLUSIVE MODE;”の部分がロックコマンドです．ただしこのロックをかけると，テーブル全体がロックされてしまいます．PostgreSQLは行ロックをサポートしていますから，本当は処理対象の行だけを明示的にロックしたいのですが，ロックコマンドではそれはできません[注9]．

図3　シリアライザブル処理のアボートする例

```
T1: webdb=> BEGIN;
T1: BEGIN
T2: webdb=> BEGIN;
T2: BEGIN
T1: webdb=> SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
T1: SET VARIABLE
T2: webdb=> SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
T2: SET VARIABLE
T1: webdb=> UPDATE t1 SET i = i + 1;
T1: UPDATE 1
T2: webdb=> UPDATE t1 SET i = i + 1;  ← T1のコミット待ちのため，実行中断
T1: webdb=> COMMIT;
T1: COMMIT
T2: ERROR:  Can't serialize access due to concurrent update  ← T1がコミットしたので実行再開．しかしエラーとなる
```

注7）この方法はPostgreSQL 7.2以降で使用できます．
注8）多くのデータベース製品では，更新中のトランザクションがあると，同じオブジェクトを参照に行ったトランザクションは待たされるのが普通です．PostgreSQLでは，MVCC（Multi-Version Concurrency Control）という機能を使い，更新トランザクションと参照トランザクションの並列実行を実現しています．
注9）SELECT …… FOR UPDATEという特別なSELECT文を使用すれば明示的な行ロックができます．

図4　テーブルロックによるシリアライザブル処理

```
T1: webdb=> BEGIN;
T1: BEGIN
T2: webdb=> BEGIN;
T2: BEGIN
T1: webdb=> SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
T1: SET VARIABLE
T2: webdb=> SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
T2: SET VARIABLE
T1: webdb=> LOCK t1 IN SHARE ROW EXCLUSIVE MODE;
LOCK TABLE
T2: webdb=> LOCK t1 IN SHARE ROW EXCLUSIVE MODE;  ← ロック待ちのため，実行中断
T1: LOCK TABLE
T1: webdb=> UPDATE t1 SET i = i + 1;
T1: UPDATE 1
T1: webdb=> COMMIT;
T1: COMMIT
T2: webdb=> UPDATE t1 SET i = i + 1;  ← 実行再開，UPDATE実行
T2: webdb=> COMMIT;  ← 今度は成功
T2: COMMIT
```

図5　リードコミッティドの使用例

```
T1: webdb=> BEGIN;
T1: BEGIN
T2: webdb=> BEGIN;
T2: BEGIN
T1: webdb=> UPDATE t1 SET i = i + 1;
T1: UPDATE 1
T2: webdb=> UPDATE t1 SET i = i + 1;
T1: webdb=> COMMIT;                    ← T1のコミット待ち
T1: COMMIT
T2: UPDATE 1
T2: webdb=> SELECT * FROM t1;
T2:  i
T2: ---
T2:  2
T2: (1 row)
T2: webdb=> COMMIT;
T2: COMMIT
```

▼表2　テーブル名とその役割

テーブル名	役割
publishers	出版社
magazines	雑誌
iss_mags	発行雑誌
articles	記事
writings	執筆
writers	執筆者
articles_keywords	記事-キーワード関係
keywords	キーワード

シリアライザブルとリードコミッティドの使い分け

発想を変えて，本当にリードコミッティドでは駄目なのか考えてみましょう．図5をご覧ください．T2のUPDATEがT1のコミットを待ってブロックするのはシリアライザブルと同じですが，T1がコミットした途端にUDPATEが実行再開しています．

実はT2は，このときT1が変更した値を再度読み込み，その値を元に更新処理を行います．したがって，SELECTしてみると，期待通り2という値が得られています．単純な処理ではリードコミッティドでも問題なく使用できることがおわかりいただけると思います．

雑誌記事データベースのトランザクション設計

トランザクションに関する解説はこの位にして，雑誌記事データベースの具体的なトランザクション設計にとりかかりましょう．トランザクションに関する細かな注意点があれば，設計の中で触れていくことにします．図6をご覧ください．これは前回掲載したER図もどきです[注10]．

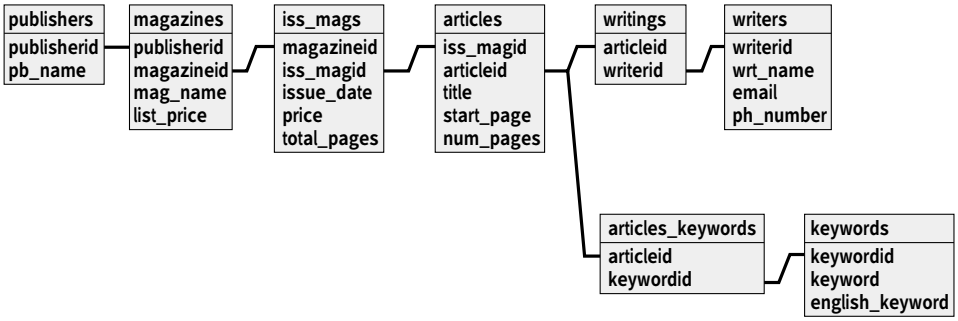
テーブルは全部で8個あり，テーブル名とその役割は表2のようになっています．

トランザクションの概要

今回のシステムの主な用途は，記事などの検索処理にあると考えられます．検索処理だけであれば，あまりトランザクション設計で難しいことを考える必要はありません．一方，データの登録・変更・削除に関しては，複数のテーブルが密接に関連しているため，よ

注10）一部英語のスペルミスがあったため，若干変更されています．

図6　雑誌記事データベースのER図もどき



く考える必要があります．

なお，前提条件として，主キーの更新は許さないことにします．

①被参照テーブルのみの性質を持つテーブル（publishers，writers，keywords）は独立して登録を行ってよい．

②参照テーブル（magazines，iss_mags，artickes，writings，article_keywords）は，被参照テーブルに外部キーが存在することを確認してから登録を行う．このとき，外部キーの存在の確認と登録の間に被参照テーブルの該当行が削除変更されてしまわないように，被参照テーブルにロックをかける．

③被参照テーブルの行削除を行うと外部キーのCASCADE指定により参照テーブルの該当行が自動的に削除されるので，アプリケーションで削除処理を行う必要はない．

④writings，article_keywordsは被参照テーブルを2つ持っている．これらのテーブルにデータを登録するときには②によって2つの被参照テーブルにロックをかける必要があるが，このときにデッドロックが発生しないように，ロックの順番を一定に決めておく．ここでは，articles➡writers，articles➡keywordsの順にロックをかけるものとする．

Webアプリケーションにおける留意点

一般に，データの削除・変更を行う場合は確認のためにデータを表示します．それから削除・変更処理を行うわけですが，Webアプリケーションでは，データを表示し終わったところでデータベースのセッションが終了し，その結果トランザクションも終了します．続く削除・変更処理では，新たなトランザクションを実行することになります．そのため，データの表示から削除・変更までの間に他のトランザクションによってデータが変更されてしまうことがあります．

そこで，削除・変更処理を行う前に，データが変更されていないかどうか確認します．そのためにはいちいちデータの照合を行ってもよいのですが，PostgreSQLでは，各行はその物理的な位置を表す“ctid”という特別な列を持っています．そこで，ctidが変化していないかどうかによってデータの変化を検出するものとします．

次号の予定

ここまで書いたところで誌面が尽きてしまいました．今までに作成したテーブル設計とトランザクション設計に基づき，次回はいよいよ実際にPHPでアプリケーションを作成することにします．