実践テクニックをご紹介

徒然PostgreSQL散策

第9回　マテリアライズドビュー

日本PostgreSQLユーザ会　石井 達夫
ISHII Tatsuo●ishii@postgresql.jp

マテリアライズドビューとは

データベースに対して大きなテーブルをいくつも結合するような問い合わせを実行すると，かなり時間がかかることがあります．こういうときに有効な手段の1つが，マテリアライズドビュー（materialized view）です．マテリアライズドビューの基本的な考え方は，検索結果の途中経過をテーブルにしまっておき，検索を高速化しようというものです．いわば，検索結果のキャッシュとも言えます（図1）．

商用データベースには大抵マテリアライズドビューを作るためのコマンドが「CREATE MATERIALIZED VIEW」のような形で提供されています．PostgreSQLにはそのようなものはありませんが，既存のコマンドを組み合わせることによって同様の効果を得ることが可能です．

OSDL DBT-3が例題

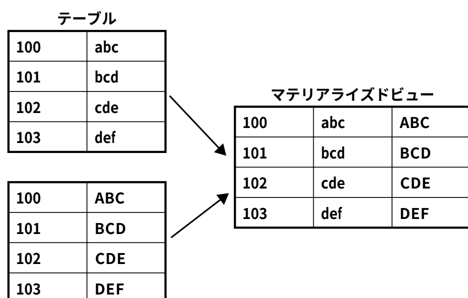
マテリアライズドビューの効果を確かめるには，ある程度の大きさのデータが必要です．適当にデータをでっち上げてもよいのですが，せっかくなのでOSDL DBT-3（http://sourceforge.net/projects/osdldbt/）というソフトウェアを使わせていただくことにします．

OSDL DBT-3（以後「DBT-3」と呼ぶことにします）は，データベースベンチマークの標準化団体であるTransaction Processing Performance Council（TPC）で制定されたTPC-Hというベンチマーク規格を参考にして作られています．TPC-Hはいわゆる「デシジョンサポートシステム」（Decision Support System：DSS）で使われるデータや問い合わせを想定しています．DSSは，複雑大量の情報をコンピュータで処理し，企業経営者やマネージャが経営上の意思決定を迅速に行えるように支援するシステムです．

TPCでは，TPC-Hのほかに，On Line Transaction Processing（OLTP）用のベンチマークであるTPC-Cをはじめ，たくさんのベンチマーク規格を制定しています．OLTPでは中小規模のデータに対して大量の単純な問い合わせを処理するのに対し，DSSでは大規模かつ複雑な問い合わせを処理する点が異なります．したがって，今回の目的にはTPC-Hのほうが適していると言えます．

さて，さきほどDBT-3はTPC-Hを参考にしていると言いましたが，TPC-Hに準拠しているとは言いませんでした．これには理由があります．もともとTPCは商用データベースのベンチマークを行うことを目的にしており，ベンチマークを実施した結果は必ずTPCによって承認されなければTPCベンチマークの結果として発表することはできません．この作業には多くの労力と莫大な費用が必要で，オープンソースコミュニテ

図1◎マテリアライズドビュー



ィが気軽に利用できるようなものではありません．

そういうわけでDBT-3はTPC-Hを参考にしているものの，TPC-Hに準拠しているとは言えないのですが，その分オープンソースコミュニティで利用しやすくなっています．これを利用しない手はありません．

DBT-3の入手

DBT-3の実施とその結果の解析を行うとかなりの情報量になります．今回は誌面の都合もあり，DBT-3によるテストデータの生成機能だけを利用することにします[注1]．

DBT-3のソースコードはhttp://sourceforge.net/projects/osdldbt/から入手できます．本稿執筆時点の最新版はdbt3-v1.4.tar.gzです．これを/tmpにでもダウンロードしてください．

この中にPostgreSQLは自体は含まれていませんから，必要ならば別途PostgreSQLをインストールしてください．本稿ではPostgreSQL 7.4.5がソースからインストール済であるものとします．

DBT-3のインストール

/tmp/dbt3-v1.4.tar.gzにダウンロードしたDBT-3のソースコードを適当な場所に展開します．ドキュメント（doc/pgsql/dbt3-user-manual-pg.sxw．OpenOfficeのファイルです）によれば，pgsqlというユーザで実行することが推奨されているようですが，手元で試した限りでは，PostgreSQLのスーパユーザであれば他のユーザでも大丈夫でした．ここでは，t-ishiiというユーザでインストールするものとします．また，検証した環境はVine Linux 2.6r4です．

```
$ mkdir ~/dbt3
$ mkdir ~/src
$ cd ~/src
$ tar xfz /tmp/dbt3-v1.4.tar.gz
$ cd dbt3-v1.4
```

scripts/pgsql/set_run_env.sh.inをカスタマイズします．**リスト1**のように，必要なところだけを変更します．

次に，TPCのWebページからデータ生成ツールであるDBGENとQGEN（http://www.tpc.org/tpch/spec/20000511.tar.z）をダウンロードします．/tmpに保存したものとします．

```
$ cd ~/src
$ mkdir tpc
$ cd tpc
$ tar xfz 20000511.tar.z
$ cd ~/src/dbt3-v1.4/datagen
$ cp -f ~/src/DBGEN/appendix/dbgen/* dbgen/
```

PostgreSQL用のパッチをDBGENに当てます．なぜか一部うまくいきませんが，図2のようにして回避します．

データの生成

無事にコンパイルが終わったらデータを生成します．

ここで注意事項を．DBT-3では，データベースクラスタのディレクトリ（今回の例では/usr/local/pgsql/data）を触ります．一応すでにデータベースクラスタがあればそのまま使うようになっているようですが，念のために最初にデータベースクラスタの物理的なバックアップを取っておくことをおすすめします．また，途中でpostmasterの再起動なども入りますので，実運用中のシステムではDBT-3を実行しないほうがよいでしょう（そういう人はいないと思いますが:-)．

データの生成は2段階で行われます．最初にテキストファイルの形でデータファイルを生成します．

```
$ cd ~/src/dbt3-v1.4/datagen/dbgen
```

テキストファイルの生成にはdbgenを使います．引

▼リスト1　set_run_env.sh.inのカスタマイズ

```
export DSS_QUERY=@TOPDIR@/datagen/@DATABASE_TO_USE@-queries
export DSS_PATH=/dbt3_data ➡ /home/t-ishii/dbt3に変更
export DSS_CONFIG=@TOPDIR@/datagen/dbgen
export SID=DBT3
export DBT3_PERL_MODULE=@TOPDIR@/perlmodules
export PATH=/usr/local/pgsql/bin:$PATH
export PGDATA=/dbt3/pgsql ➡ /usr/local/pgsql/dataに変更
export PGUSER=pgsql ➡ t-ishiiに変更
```

注1）DBT-3を実際に最後まで動かしてみたい方は，まずDBT-3の背後にある考え方をきちんと理解したほうがよいでしょう．そのためには，TPC-Hの規格書を読むのが一番です．TPC-Hの規格はTPCのWebページ（http://www.tpc.org/）から入手できます．本稿執筆時点の最新版はバージョン2.1.0で，PDFファイルの形でhttp://www.tpc.org/tpch/spec/tpch2.1.0.pdfから入手できます．

▼図2　パッチを当ててからコンパイル

```
$ patch -b -p0 < osdl_dbgen.patch
patching file dbgen/Makefile
The next patch would create the file dbgen/Makefile.in,
which already exists!  Assume -R? [n] n
Apply anyway? [n] n
Skipping patch.
1 out of 1 hunk ignored -- saving rejects to file dbgen/Makefile.in.rej
patching file dbgen/bm_utils.c
patching file dbgen/driver.c
patching file dbgen/print.c
patching file dbgen/tpcd.h

$ autoconf
$ ./configure
checking for gawk... gawk
checking for gcc... gcc
checking for C compiler default output file name... a.out
〜途中省略〜
config.status: creating scripts/sapdb/restore_db.sh
config.status: creating scripts/sapdb/set_run_env.sh
config.status: creating scripts/sapdb/update_statistics.sh
$ make
cd datagen/dbgen; make
make[1]: 入ります ディレクトリ `/home/t-ishii/src/dbt3-v1.4/datagen/dbgen'
cc -g -O -DDBNAME=\"dss\" -Dpgsql  -DTPCH -DSTDLIB_HAS_GETOPT -DEOL_HANDLING -D_FILE_OFFSET_BITS=64
-c -o build.o build.c
cc -g -O -DDBNAME=\"dss\" -Dpgsql  -DTPCH -DSTDLIB_HAS_GETOPT -DEOL_HANDLING -D_FILE_OFFSET_BITS=64
-c -o driver.o driver.c
〜途中省略〜
gcc -c -shared -fpic -g -Wall -I/home/t-ishii/src/dbt3-v1.4/dbdriver/utils/include -g -O2 -
D_FILE_OFFSET_BITS=64 -Dpgsql -o get_statement.so get_statement.c
gcc -I/home/t-ishii/src/dbt3-v1.4/dbdriver/utils/include -Dpgsql -g -Wall -I/home/t-ishii/src/dbt3-
v1.4/dbdriver/utils/include -g -O2 -D_FILE_OFFSET_BITS=64 -o parse_query main.so get_statement.so
gcc -g -Wall -I/home/t-ishii/src/dbt3-v1.4/dbdriver/utils/include -g -O2 -D_FILE_OFFSET_BITS=64 -o
gtime get_time.c
make[1]: 出ます ディレクトリ `/home/t-ishii/src/dbt3-v1.4/dbdriver/utils'
```

数は「-s スケールファクタ」です．DBT-3では，データベースの規模をスケールファクタで指定します．公式なスケールファクタは1以上で，一番小さな1のときでもテキストファイルにして1Gバイト，最終的なデータベースクラスタのサイズは4Gバイト近くにもなります．

それではちょっと大変なので，今回はスケールファクタ0.1で実施します注2（図3）．これでもデータベースクラスタは600Mバイト弱，テキストファイルは100Mバイトほどになります．

▼図3　テキストファイルの生成

```
$ ./dbgen -s 0.1
TPC-H Population Generator (Version 1.3.0)
Copyright Transaction Processing Performance
Council 1994 - 2000
creating links in /tmp to file ./supplier.tbl
creating links in /tmp to file ./customer.tbl
creating links in /tmp to file ./orders.tbl
creating links in /tmp to file ./lineitem.tbl
creating links in /tmp to file ./part.tbl
creating links in /tmp to file ./partsupp.tbl
creating links in /tmp to file ./nation.tbl
creating links in /tmp to file ./region.tbl
```

すると図3のように，8個のファイルができます．一番大きいのはlineitem.tblで，71Mバイトあります．

いよいよテーブルの生成です．

```
$ cd ~/src/dbt3-v1.4/scripts/pgsql
$ source set_run_env.sh
```

続いてset_db_env.shを変更します．43行目辺りの

```
initdb -D $PGDATA > /dev/null 2>&1
```

を，

```
initdb --no-locale --encoding EUC_JP -D
$PGDATA > /dev/null 2>&1    (実際は1行)
```

に変更します．また，なぜかシェルスクリプトの実行権限が落ちていたので，訂正します．

注2）スケールファクタ1未満ではDBT-3によるベンチマークは正しく動作しません．あくまで本稿のための仮の設定であると考えてください．

```
$ chmod 755 *.sh
```

最後に，実際にテーブルを生成します．

```
$ ./build_db.sh
+ db_param=
+ shift 1
+ data_file=0
～途中省略～
+ echo 'updating optimizer statistics done'
updating optimizer statistics done
+ date
2004年  9月 19日 日曜日 14:51:53 JST
```

これで，DBT3という名前のデータベースにテストデータが作成されました．確認してみましょう（図4）．この中で一番大きなテーブルはlineitemで，60万行以上あります．

使用する問い合わせ

DBT-3では多数の問い合わせが実行されますが，今回使用するのはリスト2のものです．この問い合わせは，筆者の手元のノートPCでは実行に1秒以上かかります．これをマテリアライズドビューを使って改善しましょう．

このままではマテリアライズドビューにならない

この問い合わせは，そのままではマテリアライズドビューにする意味がありません．そこで以下のように変形します．

① WHERE 句に問い合わせによって可変になる条件を含んでいるので，それを排除．具体的には o_orderdate >= date '1997-10-01' and o_orderdate < date '1997-10-01' + interval '3 month' の部分．もちろん，この条件が常に適用されるのであればそのまま残しておいてもかまいません．

② GROUP BYを使っているのでそれを排除．GROUP BYは問い合わせの結果行をサマライズするものであり，サマライズしたものをマテリアライズドビューにしたのではいつも同じ結果しか得られません．もちろん，常にサマライズしたものしか参照しないのであれば十分マテリアライズドビューにする意味はありますが，ここではマテリアライズドビューをなるべく柔軟に使いため，このように考えました．

③ 同様の理由により，ORDER BYを使っているのでそれを排除．

以上を適用して得られたSELECT文はリスト3のようになります．

マテリアライズドビューの定義

リスト4のようにマテリアライズドビューmvを定義します．マテリアライズドビューmvを使ってオリジナルと同じ結果を得るための問い合わせはリスト5の

▼図4　テストデータの確認

```
$ psql DBT3

DBT3=# \dt
            List of relations
 Schema |       Name       | Type  |  Owner
--------+------------------+-------+---------
 public | customer         | table | t-ishii
 public | lineitem         | table | t-ishii
 public | nation           | table | t-ishii
 public | orders           | table | t-ishii
 public | part             | table | t-ishii
 public | partsupp         | table | t-ishii
 public | region           | table | t-ishii
 public | supplier         | table | t-ishii
 public | time_statistics  | table | t-ishii
(9 rows)
```

▼リスト2　使用する問い合わせ

```
SELECT o_orderpriority, count(*) AS order_count FROM orders
WHERE o_orderdate >= date '1997-10-01' AND o_orderdate < date '1997-10-01' + interval '3 month'
AND EXISTS
(SELECT * FROM lineitem WHERE l_orderkey = o_orderkey AND l_commitdate < l_receiptdate)
GROUP BY o_orderpriority ORDER BY o_orderpriority;
```

▼リスト3　変更した問い合わせ

```
SELECT * FROM orders WHERE EXISTS
(SELECT * FROM lineitem WHERE l_orderkey = o_orderkey AND l_commitdate < l_receiptdate);
```

▼リスト4　マテリアライズドビューの定義（1）

```
CREATE TABLE mv AS SELECT * FROM orders WHERE EXISTS
(SELECT * FROM lineitem WHERE l_orderkey = o_orderkey AND l_commitdate < l_receiptdate);
```

▼リスト5　マテリアライズドビューを使った問い合わせ

```
SELECT o_orderpriority, count(*) AS order_count FROM mv
WHERE o_orderdate >= date '1997-10-01' AND o_orderdate < (date '1997-10-01' + interval '3 month')
GROUP BY o_orderpriority ORDER BY o_orderpriority;
```

▼リスト6　マテリアライズドビューの作り直し

```
TRUNCATE mv;
INSERT INTO mv SELECT * FROM orders WHERE EXISTS
(SELECT * FROM lineitem WHERE l_orderkey = o_orderkey AND l_commitdate < l_receiptdate);
```

ようになります．これを実行してみると，0.5秒ほどで結果が返り，だいぶ改善されたことがわかります．

マテリアライズドビューの自動更新

さて，マテリアライズドビューを使って検索の高速化ができることはわかりましたが，マテリアライズドビューは一種のキャッシュですから，元のテーブルが更新されたときに何らかの方法でマテリアライズドビューも更新されるようにしなければなりません．

マテリアライズドビューを一から作り直す

もっとも簡単なのは，定期的にマテリアライズドビューを一から作り直すことです．元のテーブル（上の例ではordersとlineitem）がたまにしか更新されず，厳密に新しいデータを表示する必要がない場合には実用的な方法です．

単純にテーブルを削除して作り直してもよいのですが，システムカタログにゴミが溜りがちになるので，**リスト6**のようにTRUNCATEを利用するのも一手です．このようなSQL文を適当な頻度，たとえば毎晩実行するようにすればよいでしょう．

▼リスト7　マテリアライズドビューの定義（2）

```
CREATE TABLE mv (
    o_orderkey numeric(10,0),
    o_custkey numeric(10,0),
    o_orderstatus character(1),
    o_totalprice numeric(12,2),
    o_orderdate date,
    o_orderpriority character(15),
    o_clerk character(15),
    o_shippriority numeric(10,0),
    o_comment character varying(79)
);
```

トリガを使って最新データを維持

最新のデータがすぐにマテリアライズドビューに反映されるようにするためには，リスト6のスクリプトを頻繁に実行する必要があります．しかし，大きなマテリアライズドビューではそのために多くの時間がかかるだけでなく，システムに大きな負荷がかかるので，DBのレスポンスが低下しがちになります．

そこで効果的なのが，元のテーブルの更新に直接関係するマテリアライズドビューの変更部分だけを作り直す方法です．そのためには元のテーブルにトリガを仕掛け，その中でマテリアライズドビューの更新を行うようにします．

●ステップ1：主キーを設定

トリガを使って更新を行うためには，マテリアライズドビューに主キーが必要です．主キーがないと，更新すべき行が一意に定まらないからです．

今回使用するマテリアライズドビューmvの定義は**リスト7**のようになっています．リスト7の中で，o_orderkeyは元のordersテーブルの主キーから直接生成されているため必ず一意になり，mvにおいても主キーに適しています．**図5**のようにo_orderkeyに主キーを設定することにします．

●ステップ2：ordersテーブルへのトリガの設定

今回マテリアライズドビューmvに関係しているテーブルはordersとlineitemです．ordersテーブルとlineitemテーブルは1：nの親子関係にあり，ordersの主キーo_orderkeyをlineitemのl_orderkeyが外部

キーとして参照しています.

つまり，ordersテーブルの行は単独で存在できますが，対応するordersテーブルの行を持たないlineitemテーブルの行は存在しません．このことを念頭に置いてトリガを設定します.

まず，ordersテーブルのトリガを検討します.

① ordersテーブルへのINSERT時

新規にordersにINSERTされた行に対応するlineitemの行はまだ存在していないはずなので，トリガの設定は必要ありません.

② UPDATE時

ordersで更新のかかった行内容を同じo_orderkeyを持つmvの行にそのままコピー更新します.

③ DELETE時

ordersで削除された行と同じo_orderkeyを持つmvの行を削除します.

リスト8にordersテーブルへのトリガの設定を示します.

●ステップ3：lineitemテーブルへのトリガの設定

次にlineitemテーブルです．mvの定義が，リスト4の形であったことを思い出しましょう．すなわち，lineitemにINSERT/UPDATE/DELETEされた行がこのWHERE句の条件を満たすかどうかで処理内容が変わってきます.

① INSERT時

l_commitdate < l_receiptdateを満たしていなければ無処理で終了．満たしている場合はl_orderkeyと一致するo_orderkeyを持つmvの行があるかどうかをチェックします．もしまだなかった場合には，ordersテーブルから該当行をmvにコピーして終了します.

② UPDATE時

l_orderkeyが等しく，かつl_commitdate < l_receiptdateを満たしている行がlineitemにあるかどうかチェックします．もしなければ該当行をmvから削除します．もしあればmvに該当行があるかどうか調べ，もしなければordersテーブルから該当行をmvにコピーします.

③ DELETE時

mvからorderkeyが等しい行を検索します．もしある場合には，他にl_orderkeyが等しく，かつl_commitdate < l_receiptdateを満たしている行がlineitemにあるかどうかチェックし，なければ該当行をmvから削除します.

リスト9にlineitemテーブルへのトリガの設定を示します.

●ロックを計画的にかける

これで一応トリガを使ってマテリアライズドビューを管理できるようになりましたが，もう1つ考えることが残っています．それは，orders，lineitemに同時更新が発生するケースです．この場合，lineitem，mvのおのおのに適切にロックをかけないとデータの一貫性がなくなる可能性があります．かといって，各テーブルに勝手にロックをかけるとデッドロックを引き起こします.

これを防ぐには，常に決まった順序でロックをかけるようにします．今回の例で言えば，orders，lineitemの順に常にロックをかけるようにするのがよいでしょう．具体的には以下のようになります.

```
BEGIN;
LOCK TABLE orders IN SHARE ROW EXCLUSIVE MODE;
:
ordersテーブルの更新
:
LOCK TABLE lineitem IN SHARE ROW EXCLUSIVE MODE;
:
```

▼図5　主キーの設定

```
DBT3=# ALTER TABLE mv ADD PRIMARY KEY(o_orderkey);
NOTICE:  ALTER TABLE / ADD PRIMARY KEY will create implicit index "mv_pkey" for table "mv"
ALTER TABLE
```

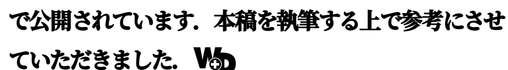
▼リスト8　orders テーブルのトリガ

```
--
-- INSERT処理
--
CREATE OR REPLACE FUNCTION orders_insert() RETURNS TRIGGER
SECURITY DEFINER LANGUAGE PLPGSQL AS '
BEGIN
  INSERT INTO mv(o_orderkey, o_custkey, o_orderstatus, o_totalprice,
    o_orderdate, o_orderpriority, o_clerk, o_shippriority, o_comment)
  VALUES(NEW.o_orderkey, NEW.o_custkey, NEW.o_orderstatus, NEW.o_totalprice,
    NEW.o_orderdate, NEW.o_orderpriority, NEW.o_clerk, NEW.o_shippriority,
    NEW.o_comment);
  RETURN NULL;
END;
';

DROP TRIGGER orders_insert ON orders;
CREATE TRIGGER orders_insert AFTER INSERT ON orders
FOR EACH ROW EXECUTE PROCEDURE orders_insert();

--
-- UPDATE処理
--
CREATE OR REPLACE FUNCTION orders_update() RETURNS TRIGGER
SECURITY DEFINER LANGUAGE PLPGSQL AS '
BEGIN
  UPDATE mv SET
    o_orderkey = NEW.o_orderkey,
    o_custkey = NEW.o_custkey,
    o_orderstatus = NEW.o_orderstatus,
    o_totalprice = NEW.o_totalprice,
    o_orderdate = NEW.o_orderdate,
    o_orderpriority = NEW.o_orderpriority,
    o_clerk = NEW.o_clerk,
    o_shippriority = NEW.o_shippriority,
    o_comment = NEW.o_comment
  WHERE o_orderkey = OLD.o_orderkey;
  RETURN NULL;
END;
';

DROP TRIGGER orders_update ON orders;
CREATE TRIGGER orders_update AFTER INSERT ON orders
FOR EACH ROW EXECUTE PROCEDURE orders_update();

--
-- DELETE処理
--
CREATE OR REPLACE FUNCTION orders_delete() RETURNS TRIGGER
SECURITY DEFINER LANGUAGE PLPGSQL AS '
BEGIN
  DELETE FROM mv WHERE o_orderkey = OLD.o_orderkey;
  RETURN NULL;
END;
';

DROP TRIGGER orders_delete ON orders;
CREATE TRIGGER orders_delete AFTER DELETE ON orders
FOR EACH ROW EXECUTE PROCEDURE orders_delete();
```

さいごに

今回は検索効率を高めるためにマテリアライズドビューを導入する方法を説明しました．毎回マテリアライズドビューを作り直すのであれば簡単ですが，トリガを使って元データの更新を効率よく管理するのは結構大変です．マテリアライズドビューは一種の非正規化であり，データの整合性維持を自前で行うことになるからです．今回は関係するテーブルが2つだけの簡単な例でしたが，たくさんのテーブルが絡むケースでは，よく考えないとデータの不整合が起きたり，デッドロックが発生します．ただ，マテリアライズドビューをうまく使うと非常に検索効率が上がるので，ここぞというところで採用すると効果的であるのはたしかです．どうしてもデータベースの性能が出ないときの最後の手段にでも使っていただければ，と思います:-)

なお，マテリアライズドビューの考え方を整理したものが，


http://jonathangardner.net/PostgreSQL/materialized_views/matviews.html

で公開されています．本稿を執筆する上で参考にさせていただきました．

▼リスト9　lineitemテーブルのトリガ

```
--
-- INSERT処理
--
CREATE OR REPLACE FUNCTION lineitem_insert() RETURNS TRIGGER
SECURITY DEFINER LANGUAGE PLPGSQL AS '
DECLARE
  cnt INTEGER;
BEGIN
  IF NEW.l_commitdate < NEW.l_receiptdate THEN
    SELECT INTO cnt count(*) FROM mv WHERE NEW.l_orderkey = o_orderkey;
    IF cnt = 0 THEN
      INSERT INTO mv SELECT * FROM orders WHERE o_orderkey = NEW.l_orderkey;
    END IF;
  END IF;
  RETURN NULL;
END;
';

DROP TRIGGER lineitem_insert ON lineitem;
CREATE TRIGGER lineitem_insert AFTER INSERT ON lineitem
FOR EACH ROW EXECUTE PROCEDURE lineitem_insert();

--
-- UPDATE処理
--
CREATE OR REPLACE FUNCTION lineitem_update() RETURNS TRIGGER
SECURITY DEFINER LANGUAGE PLPGSQL AS '
DECLARE
  cnt INTEGER;
BEGIN
  SELECT INTO cnt count(*) FROM lineitem WHERE NEW.l_orderkey = l_orderkey AND
    l_commitdate < l_receiptdate;
  IF cnt = 0 THEN
    DELETE FROM mv WHERE o_orderkey = NEW.l_orderkey;
  ELSE
    SELECT INTO cnt count(*) FROM mv WHERE NEW.l_orderkey = o_orderkey;
    IF cnt = 0 THEN
      INSERT INTO mv SELECT * FROM orders WHERE o_orderkey = NEW.l_orderkey;
    END IF;
  END IF;
  RETURN NULL;
END;
';

DROP TRIGGER lineitem_update ON lineitem;
CREATE TRIGGER lineitem_update AFTER UPDATE ON lineitem
FOR EACH ROW EXECUTE PROCEDURE lineitem_update();

--
-- DELETE処理
--
CREATE OR REPLACE FUNCTION lineitem_delete() RETURNS TRIGGER
SECURITY DEFINER LANGUAGE PLPGSQL AS '
DECLARE
  cnt INTEGER;
BEGIN
  SELECT INTO cnt count(*) FROM mv WHERE OLD.l_orderkey = o_orderkey;
  IF cnt <> 0 THEN
    SELECT INTO cnt count(*) FROM lineitem WHERE OLD.l_orderkey = l_orderkey AND
      l_commitdate < l_receiptdate;
    IF cnt = 0 THEN
      DELETE FROM mv WHERE o_orderkey = OLD.l_orderkey;
    END IF;
  END IF;
  RETURN NULL;
END;
';

DROP TRIGGER lineitem_delete ON lineitem;
CREATE TRIGGER lineitem_delete AFTER DELETE ON lineitem
FOR EACH ROW EXECUTE PROCEDURE lineitem_delete();
```