異種データベースからDB2ファミリーへの自動レプリケーション
― PostgreSQLからDB2へ自動差分コピーの実証と考察 ―

東 忠克*　曽田 俊明**

Automated Data Replication to DB2 from Heterogeneous Data Sources

Tadakatsu Azuma*　Toshiaki Sota**

PostgreSQLは現在日本で最もよく知られているオープンソース・データベースの一つである. 本論文では, DB2®のInformation Integratorの汎用ODBCラッパー下でデータソースとしてLinux上のPostgreSQLに注目し, DB2へのデータの複製を実施するため, その標準機能であるトリガーを開発し, DPropRのアプライ機能を使用してPostgreSQLのデータを自動的にDB2 for z/OS®へレプリケーションするプロトタイピングを実証した記述である. 本論文では今回開発したトリガーなどを中心にその過程で判明したODBC ラッパー接続時の注意点, 他システムへ複製する仕組みを解説し, 今後の異種間レプリケーション機能の実装要件について言及する.

Replication is often used in many distributed systems to provide a higher level of performance. The main purpose of this study is to demonstrate the feasibility of the replication from the PostgreSQL database to the IBM DB2® Family by utilizing the IBM DPropR. In order to accomplish this goal, the triggers have been used to gather the changed data on the source site. Also, the generic ODBC wrapper used by the Information Integrator was applied to access the remote data source. The processes of the replication based on the newly developed trigger are described and also possibilities of other non-IBM DB replication are stated.

Key Words & Phrases：DPropR, replication, DB2 Family, PostgreSQL, Information Integrator

1. はじめに

エンドユーザーが重要なデータを容易, かつ迅速にアクセスできるようになることは大多数の企業にとって大きな目標である. データが1カ所に保管され, そのデータをユーザーがローカル側とリモート側の両方からアクセスしたい場合, 運用時間帯やアクセスの集中度を考慮した場合に定期的に相手のロケーションへデータを複写する技術が重要になっている. この機能をデータベース管理者がFTPによるファイル転送や独自の複雑な仕組みを構築することで実現することは可能ではあるが, 運用要件, 性能要件, 構成変更などに対応するためには, 多くの労力を費やすことを強いられる. そのため. いくつかのデータベースでは, この機能をレプリケーション機能（データベースのテーブルのデータを自動的に他のデータベースのテーブルへ伝播させる機能）として実装されている. 一方, 現在のマルチベンダー環境ではDB2 ファミリー, Oracle, Microsoft® SQL Serverなどの商用データベース, PostgreSQL, MySQLなどのオープンソース・データベースの普及が進み, システム内で多種のデータベースが使用される異種混合環境が増えており, そして, そのデータベース間でのデータ連携が求められることもある. しかし, 異機種データベース間のレプリケーション機能を提供しているデータベースはそれほど多くない.

IBMでは異種データソースの連携に古くから取り組んでおり, 現在では, 異種データソースの共用や分析を可能にするInformation Integrator（以下II）[1][2]を提供している. また, DB2ファミリー間でのレプリケーション機能を提供するDB2 DPropR [3] とIIの機能を組み合わせることにより異機種データベース間のレプリケーション機能を提供している. そして, 既

提出日：2004年6月1日
*ozuma@jp.ibm.com **TSOTA@jp.ibm.com

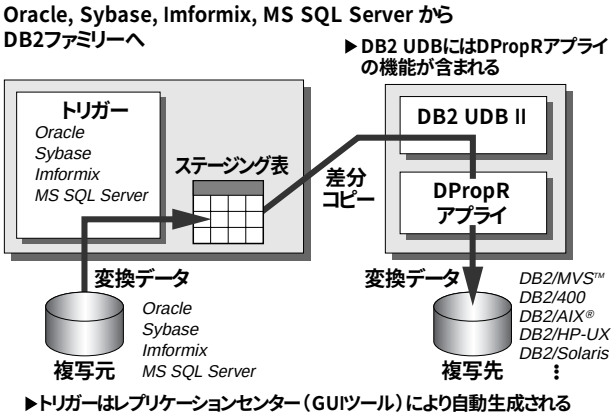

図1. 異機種RDBからのレプリケーション

に多くの商用システムで実装され稼動している. そのレプリケーションで現在サポートするものは**図1**に示す. 現在、製品としてサポートしているのは, 主要な商用データベースだけである. 現在普及が進んでいるオープンソース・データベースへは対応していない.

本論文では, 現在製品でサポートしている異機種データソース以外のデータソースにおいても, ODBCインターフェースを利用しデータ連携を行うIIの機能（ODBCラッパー）とデータベースの標準機能であるトリガー（データベースのテーブルへのイベントで、事前に定義されたルーチンの起動を行う機能）の機能を用いることでDPropRを利用して自動レプリケーションが可能であることを, PostgreSQL［4］を使用したプロトタイプを構築することで実証した. DPropRでは, 異種データベースがレプリケーションソースとなる場合には, トリガーを利用するが, このトリガー利用に関しては筆者が1995年に開発部門と協業した内容が製品化された経緯を持つ［5］.

まず, DPropR, IIによって提供される異機種間レプリケーションの仕組みについて述べ, 今回行ったPosgreSQLからDB2へのレプリケーションのプロトタイピングの内容について述べる.

このプロトタイピングによる実証により, 今回実装を行ったPostgreSQLのみでなくODBCインターフェースとトリガー機能をサポートするすべてのデータソースからのレプリケーションが可能になることが言える. DPropRで対応していないデータソースのようにFTPなどを利用した独自のアプリケーションを構築する必要はなくデータベースの標準的な機能であるトリガーの定義を実装することでレプリケーションが可能になる. またターゲットをニックネームに設定することでPostgreSQLからOracleへといったDB2ファミリー以外のデータソース間でのレプリケーションもIBMソリューションの仕組みで提供できる. 今後, 製品でサポートされていないオープンソース・データベースとDB2の連携処理をソリューション要件とする新規案件に対し, 本論文で示す事例や考察が役に立てば幸いである.

2. DB2 DPropRによるレプリケーション

DB2では, レプリケーション機能をDPropRで提供し, レプリケーションの構成の定義を行うことで実現できる. DPropRでは実行時に二つのコンポーネントが個別に稼動し変更をコピーする. 各コンポーネントの関係を**図2**に示す. 変更収集プログラム（キャプチャー）は, DB2のログを非同期に読み取り, ソース表がある場所のステージング表（変更データを扱うCD表とトランザクションデータを扱うUOW表）に書き出す. 変更適用プログラム（アプライ）は, 書き出された変更データを読み取りターゲット表に書き出す.

異機種データソース間でのレプリケーションでは, IIの機能を利用する. レプリケーションでの役割を平易に言うとIIは離島に架けられた“ 橋 ”であり, アプライはその島から荷物を運び出す“ トラック ”と言える.

3. 異種間接続と異種間レプリケーション

マルチベンダー環境においては, データ連携を行う必要のあるシステムが同種のデータベース管理システム（DBMS）であるとは限らない. しかし, DB2 DPropRではDB2ファミリーのみでなく, 異機種DBMS間（Oracle, Sybase, Informix®, MS SQL Serverのみサポート）で, 複写元で変更されたデータのみを複写先へ自動的にレプリケーションを可能にする. 合併や統合を進める企業内でデータの断片化や情報の非共有などが生まれ1994年のDPropR出荷以来多くの企業にレプリケーションの採用がされてきている.

ソース表がDB2ファミリーの場合はキャプチャーがログを読み込むことができるが, 異機種DBMSには現行ではキャプチャーは存在しない. そのため, 異機種DBMSのレプリケーションソース側にはトリガーを定義し, ソース表への変更データをステージング表に書き出す仕組みで変更データの収集を行っている. また, データ反映時に関してはステージング表の読み取りが必要となるが, ステージング表は異機種DBMSのレプリケーション元のソース側のデータベースに存

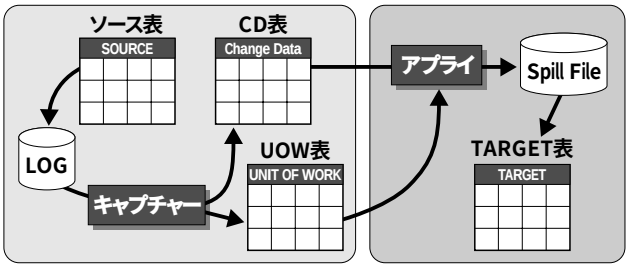

図2. キャプチャーとアプライの内部動作

表1. 主要DBMSの専用ラッパー

データソース	ラッパー
Oracle	NET8
Sybase	DBLIB, CTLIB
MS SQL Server	DJXMSSQL3, MSSQLODBC3
Informix	INFORMIX
DB2 ファミリー	DRDA®

在するのでIIの連合機能を使用しアクセスする仕組みを構成する.

IIの連合機能とは, 別のデータベースに存在する表をあたかも同じデータベース上に存在する表として取り扱うことができる機能である. 別のデータベースに存在する表が異機種データベースのものであってもDB2の表と同じようにアクセスが可能である. このような他のデータベースへの透過的なアクセスはラッパーと呼ばれるデータベース・オブジェクトを介して行われる. ラッパーはデータソースへの接続, 操作を行うモジュールであり, データソースへアクセスするための情報を持っている. そして, それぞれのデータソースごとにラッパーが提供されアクセスを可能にしている. 現在, 専用のラッパーが提供されているデータソースを表にまとめたのが**表1**である. Oracle, MS SQL Serverといった有名な商用のデータソースには専用ラッパーが提供されている.

現在, 製品でサポートされている異機種DBMS間のレプリケーションはそのデータソースに最適化された専用のラッパーを用いて行われる. 専用のラッパーが提供されていないデータソースであってもデータベース・アクセスの標準的なインターフェースであるODBCをサポートしていれば, ODBCラッパーを使用することにより連合機能を使用することができる.

ただし, ODBCラッパーは, 多くのデータソースを対象とできるようにするために, 専用ラッパーのようにそのデータソース独自の最適化が行われていない. そのため, ODBCラッパーは専用ラッパーと比べいくつかの制約事項と考慮点が必要となる.

今回, プロトタイピングを行った, PostgreSQLに関しても専用のラッパーが提供されていないため, ODBCラッパーを使用するが, このプロトタイピングを行う上でも制限事項や考慮点が存在した. それらについてはラッパー設定やトリガーの実装により回避することが可能であった. 詳しくは4.2節で述べる.

4. レプリケーションのプロトタイピング

データソースにPostgreSQLを使用したプロトタイピングを行い, ODBCラッパーを使用したレプリケーションのフィージビリティを評価する. 内部的にトリガーを使用してキャプチャーの内部的振る舞いをエミュレートし, レプリケーションを実現するプロトタイピングを実施し評価した.

連合サーバーは, 連合機能を提供するサーバーであり, 前に述べたようにDPropRで異機種間のレプリケーションを行う際に必要となり, 今回のプロトタイピングでも**図3**のようにODBCラッパーを使用してPostgreSQLにDB2のインターフェースでアクセスできるように構築を行う.

4.1 連合サーバーの構成

最初にPostgreSQLに連合サーバーを介してアクセスを行うためにODBCラッパーを使用した連合サーバーの構成を行う. ODBCのデータソースを設定し, 連合サーバー上のODBCラッパーに従属するデータベース・オブジェクトを作成することになる. そのオブジェクトの階層関係は**図4**のようになっている. それぞれのオブジェクトはPostgreSQL（データソース）に透過的にアクセスを行うためにPostgreSQL上の必要な情報をそれぞれ持っている.

以下, 構成の手順を示しながら各設定について説明を行う.

ODBCデータソースの設定

(1) PostgreSQLのODBCドライバーを導入

PostgreSQLのODBCドライバーには, PostgreSQLクライアントとしての機能も含まれており, 連合サーバー側で導入が必要なPostgreSQL関連のモジュールはODBCドライバーのみである.

(2) ODBC DSN（Data Source Name）の設定

DSNの設定には, サーバー名, ポート番号, データベース名が設定されており, DSNで接続するデータベースを特定できる.

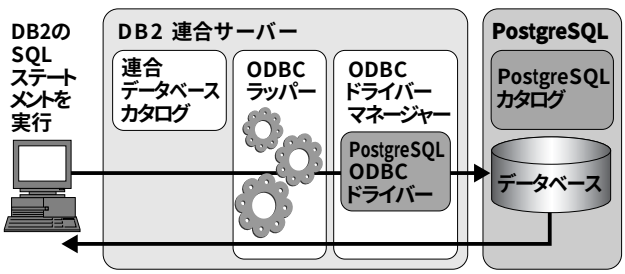

図3. ODBCラッパー接続概要

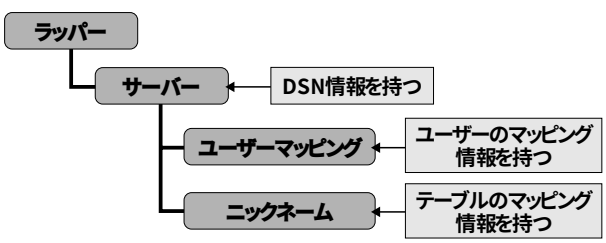

図4. 連合サーバー構成オブジェクト

ODBCのデータソースとしてPostgreSQLのデータベースを登録した後に連合サーバーを構成するデータベース・オブジェクトの作成を行う. これには, ラッパー, サーバー, ユーザーマッピング, ニックネームといったオブジェクトがあり, 以下の順にオブジェクトを作成していく.

(1) ラッパーの作成…ODBCラッパーを作成.
(2) サーバーの作成…DSNを指定する.
(3) ユーザーマッピングの作成…PostgreSQLのユーザー情報を設定する.
(4) ニックネームの作成…PostgreSQL上の対象表を指定する.

以上の設定を行い, 連合サーバーを構築する.

4.2 ODBCラッパー制約事項と考慮点

このプロトタイピングで, 問題となった制約事項, 考慮点について述べる.

(1) 対象データを指定したUpdate, Delete文

ODBCラッパーを使用した場合,Update, Delete文にWhere句を指定した時, Update, Delete文が発行できないという制限がある. これは後に述べる. プッシュダウンの設定を行うことで対応可能となる.

(2) ニックネームに対してLock Table文の発行

ODBCラッパーの場合, Lock Table文がニックネームに対して発行できない. この制約は, ODBCのデータソースとして, ExcelなどLock Table文をサポートしないデータソースが存在するためである. 一方, PostgreSQLは, Lock Table文をサポートしており, Lock Table文でTableにLockを取得することが可能である. レプリケーションを行うロジックの中で未コミットデータがある際にアプライが実行された場合でのデータ漏れを防ぐためにステージング表にLock Table文を含める必要がある. 今回この制約のためPostgreSQLのトリガー内でLock Table文を発行するデザインとすることで対応した.

(3) プッシュダウンを行えるように設定の変更が必要

連合サーバーでのSQLの処理の方法として二つの方法がある. 一つは, 連合サーバーに発行されたSQLの処理をすべてデータソースに行わせる方法. これをプッシュダウンと呼ぶ. もう一つは, SQLの処理の一部を連合サーバー側で行う方法. この方法は, データソースの能力を連合サーバーが判断してデータソースで行えない処理を連合サーバーで補完するものである. 多くの場合, 前者の処理の方のパフォーマンスが良い. ODBC以外のラッパーの場合デフォルトでプッシュダウンが行えるように設定してある. しかし, ODBCラッパーの場合, プッシュダウンが行えないように設定してある. ODBCラッパーの場合, データソースに成り得るものがRDBMS以外も存在し, それらがデータソースの場合, 連合サーバーが要求する処理を満たせない場合を考慮し, あえてNにあらかじめ設定されている. PostgreSQLは能力を持ったRDBMSであるのでプッシュダウンが行えるように設定の変更を行うべきである. この変更をしないと, データソースにある表の全行を連合サーバー側に転送してしまう. 設定変更前後でアクセスパスを取得したものが, **図5**である. プッシュダウンを行わない場合, PostgreSQLに対してWhere句を取って実行されるため, 表の全行を取得してしまう. この設定のままでデータソースに大量のデータがある時は著しいパフォーマンスの低下が発生してしまうので設定の変更が必要である.

PUSHDOWN N (デフォルト) の場合

```
SQL Statement:
  DECLARE C1 CURSOR
  FOR
     select *
     from t1
     where c1 = 1

Estimated Cost = 126.430458
Estimated Cardinality = 1.000000

Ship Distributed Subquery #1
|  #Columns = 2
Residual Predicate(s)
|  #Predicates = 1
Return Data to Application
|  #Columns = 2

Distributed Substatement #1:
Server: POSTGRESQL  (ODBC 3.0)
SQL Statement:

     SELECT A0."c1", A0."c2"
     FROM "t1" A0

Nicknames Referenced:
   E503230.T1  ID = 32768
      Base = t1

#Output Columns = 2
```

PUSHDOWN Yの場合

```
SQL Statement:
  DECLARE C1 CURSOR
  FOR
     select *
     from t1
     where c1 = 1

Estimated Cost = 50.046124
Estimated Cardinality = 1.000000

Ship Distributed Subquery #1
|  #Columns = 2
Return Data to Application
|  #Columns = 2


Distributed Substatement #1:
Server: POSTGRESQL  (ODBC 3.0)
SQL Statement:

     SELECT A0."c1", A0."c2"
     FROM "t1" A0
     WHERE (A0."c1" = 1)

Nicknames Referenced:
   E503230.T1  ID = 32768
      Base = t1

#Output Columns = 2
```

図5. プッシュダウン設定によるアクセスパス比較

4.3 レプリケーション機能要件と仕組み

トリガーはステージング表に変更されたデータを挿入する際, 昇順の番号を各レコードに割り振る必要がある. これは、通常キャプチャーがDB2のログを読み取る場合にログ内に記録されるログのシーケンスナンバー(LSN: Log Sequence Number)を使用するが, キャプチャーが稼動しない環境ではトリガー内でその番号を生成する必要があるためである. トリガー内で番号生成のため, PostgreSQLに実装されているシーケンスオブジェクトを使用した. 生成された番号の値はnextval, currvalを使用して次の番号の生成, 現行の番号を参照することが可能である. (**図6**の①, **図7**) アプライはステージング表にあるCOMMITSEQ値をキャプチャーによって更新されたSYNCHPOINT値と比較し, それ以下のデータのみフェッチする仕組みを採用している. これをフェッチ前にアプライのBefore-

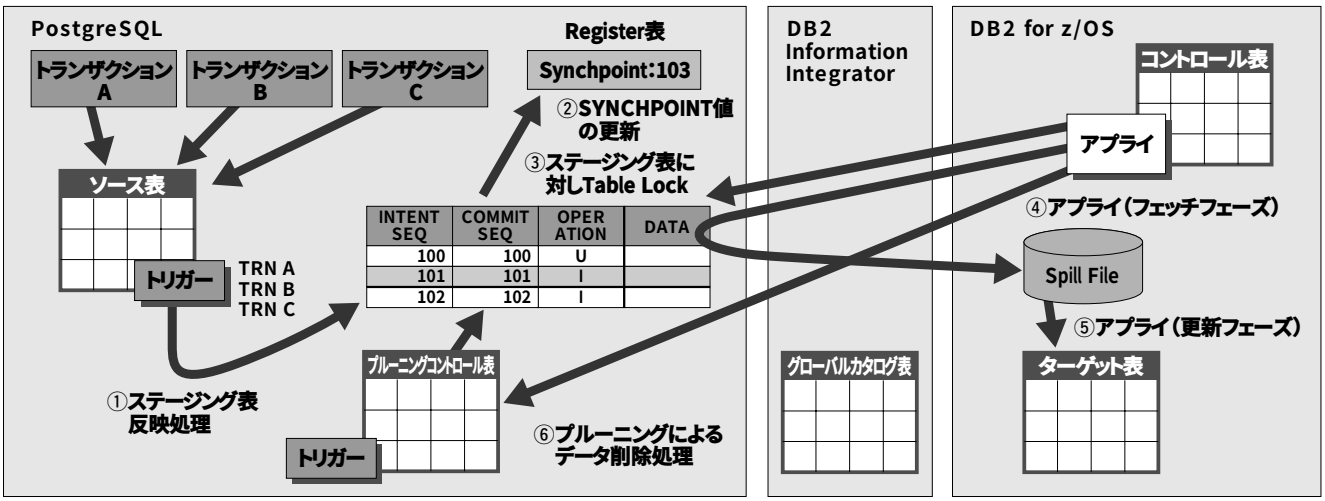

図6. トリガーとアプライの内部動作

SQLを使用しソース側のREG_SYNCH表の更新を実施しそのトリガー内でREGISTER表のSYNCHPOINTの更新を実施するようにし, 同じ仕組みを可能にした. (**図6**の②)

ステージング表に書き出された変更データをアプライが読み取る前に未コミットのトランザクションが存在した場合, データのレプリケーション漏れが発生しないように非常に短い時間であるがLock Table文を発行する必要がある.

前に述べた通り, Lock Table文自身はPostgreSQLの標準SQLとしてサポートされるのであるが, ODBCラッパーの制約でDB2から発行が許されない. そのためトリガー内で, Lock Table文を発行するデザインにした. (**図6**の③)

アプライはステージング表にある変更データをREGISTER表のSYNCHPOINT値と比較し, その値以下のデータをSPILLファイルと呼ばれるワークファイルにカーソルをオープンしフェッチ処理をして書き出す. (**図6**の④)

アプライはソースサーバーから接続をターゲットサーバーに切り替え, SPILLファイルを読み取りターゲット表に変更反映する. (**図6**の⑤)

最後にアプライは処理したSYNCHPOINT値をソース側にあるPC表に書き出す. これは本来レプリケーション済みの不要なデータをキャプチャーが削除するために行うが, UPDATEトリガーにて不要なデータの削除(プルーニング)をできるようにデザインした. (**図6**の⑥)

4.4 レプリケーション機能要件の実装方法

実際にPostgreSQL内に作成したトリガーを紹介する. まず, **図7**ソース表の変更収集のトリガーを示す. このトリガーはソース表のINSERT, DELETE, UPDATE時に起動されステージング表にシーケンス

```
create sequence azseq;
create table ccd (
ibmsnap_commitseq char(10) null,
ibmsnap_intentseq    char(10) not null,
ibmsnap_operation    char(1) not null,
ibmsnap_logmarker timestamp(0)  not null,
c1 int,c2 int,c3 int,c4 int);

create or replace function insert_ccd ()
returns trigger as '
begin
  if TG_OP = ''INSERT'' then
     insert into ccd values
           (lpad(nextval(' 'azseq''),10,''0''),
            lpad(currval(' 'azseq''),10,''0''),
            ''I'', current_timestamp, new.c1,new.c2,new.c3, new.c4);
  end if;
  if TG_OP = ''UPDATE'' then
     insert into ccd values
           (lpad(nextval(' 'azseq''),10,''0''),
            lpad(currval(' 'azseq''),10,''0''),
            ''U'', current_timestamp, new.c1, new.c2,new.c3, new.c4);
  end if;
  if TG_OP = ''DELETE'' then
     insert into ccd values
           (lpad(nextval(' 'azseq''),10,''0''),
            lpad(currval(' 'azseq''),10,''0''),
            ''D'', current_timestamp, old.c1, old.c2, old.c3, old.c4);
  end if;
return new;
end;
' LANGUAGE 'plpgsql';

create trigger iccd
    after insert or update or delete
    on source for each row
    execute procedure insert_ccd();
```

図7. シーケンスとソース表トリガー定義

```
create or replace function signal_fun ()
returns trigger as '
begin
  if TG_OP = "INSERT" then
      update ibmsnap_pruncntl set

synchpoint=lpad(nextval("azseq"),10,"0"),
          synchtime=current_timestamp(0)
        where
            map_id=new.signal_input_in
        and  synchpoint is null;
  end if;
return new;
end;
' LANGUAGE 'plpgsql';

create trigger signal_trig after insert
    on ibmsnap_signal for each row
    execute procedure signal_fun();
```

図8. シグナル表トリガー定義

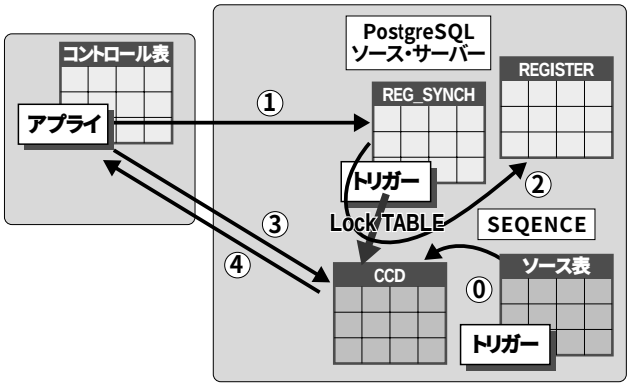

図9. REG_SYNCHトリガー定義の動き

```
create or replace function reg_fun () returns trigger as '
begin
  if TG_OP = ''INSERT'' then
  lock table ccd in exclusive mode;
  delete from ibmsnap_reg_synch;
  update ibmsnap_register set
         synchpoint=lpad(nextval(''azseq''),10,''0''),
         synchtime=current_timestamp ;
  end if;
return new;
end;
' LANGUAGE 'plpgsql';

create trigger reg_trig after  insert
on ibmsnap_reg_synch for each row
execute procedure reg_fun();
```

図10. REG_SYNCHトリガー定義

```
create or replace function pc_fun () returns trigger as '
declare min_synch char(10);
begin
  select min(synchpoint) into min_synch from ibmsnap_pruncntl
  where apply_qual=new.apply_qual;
  if TG_OP = ''UPDATE'' then
  delete from ccd where ibmsnap_commitseq < min_synch ;
  end if;
return new;
end;
' LANGUAGE 'plpgsql';

create trigger pc_trig after update
on ibmsnap_pruncntl for each row
execute procedure pc_fun();
```

図11. プルーニングトリガー定義

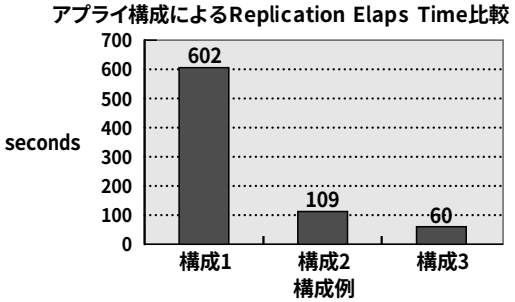

図12. アプライ構成による処理時間比較
構成1：プッシュ構成
構成2：プル構成（フェッチ／インサート形式）
構成3：プル構成（ロード適用）

番号を生成しながら変更されたデータを書き込み，キャプチャーの動きをエミュレートしている．

アプライは全件コピーをしてデータ同期を実施する．この時，キャプチャーに差分収集開始のシグナルを発信する．このシグナルを受け，それ以降の差分を書き出すようにREGISTER表のSYNCHPOINT値を更新するトリガーが**図8**である．

アプライ自身がキャプチャーなしの場合には差分反映前に前述したようにREGISTER表のSYNCHPOINT値を更新する必要がある．REG_SYNCH表にトリガーを作成し，未コミットトランザクションのデータの複製漏れを防止するLock Table文を含める．これをアプライの発行するSQL-Before機能にUPDATE文を含め，トリガーで実装した．各制御表とトリガーとの関係を表したものが**図9**である．また，REG_SYNCH表に実装したトリガーが**図10**である．

ステージング表に書き出され適用済みのデータを削除する機能もPRUNCNTL表にUPDATEトリガーを作成することで実現する．実装したトリガーが**図11**である．このトリガーはソース表を参照している複数のアプライが存在することを考慮しすべてのアプライの進行速度中いちばん低いSYNCHPOINTまでを削除をするデザインした．

4.5 レプリケーション（アプライ）の実行

これまでの設定により，PostgreSQLへ行われた変更が収集可能になる．その変更差分データの反映を行うためにアプライを実行するがアプライの実行場所はソースサーバーとターゲットサーバーにデータベースの接続が行える場所であればどこであっても構わない．今回ターゲットのデータベースとしてDB2 for z/OS®を利用している．ソースサーバー側（PostgreSQL）とターゲットサーバー側（DB2 for z/OS）とアプライの実行場所を変えPostgreSQLからDB2 for z/OSへのレプリケーションを行い，性能測定を行った．ソース表はINTが4カラム50000件の表を用意し，アプライの配置を変更しPush/Pull/Pull with LOADの三つの場合で実行時間を比較した結果が**図12**である．（Push構成：アプライがソースサーバー側で実行される構成，Pull構成：アプライがターゲットサーバーで実行される構成）

構成例1はアプライ for Windowsを使用してホストへPushしたケース，構成例2はアプライ for z/OSを使用してニックネームが存在するDB2 UDBへPull形態でフェッチしターゲットへInsertを実施したケース，構成例3では同じくアプライ for z/OSを使用してニックネームが存在するDB2 UDBへPull形態かつCrossloaderと呼ばれるカーサーからのLoadを実施したケースで

ある. Push構成(構成例1)はPull構成(構成例3)と比較すると最大10倍の遅延が観測され, Pull構成の通常フェッチ/Insert形式の構成例2と構成例3でカーサーからのLOADを使用した場合45%の改善が見られた. ホスト上のDB2PMアカウティングトレースを取得し各構成例を解析した. すべての構成例を比較すると処理時間はDB2ホストとメッセージに比例して遅延する点が判明した. つまり処理時間改善にはメッセージを減少させることがキーになる. 読み取り専用のカーサーであればDB2が採用する32KB単位でDRDAのブロッキングが使用され, Pull形式がPushよりメッセージ総数少なく, 処理時間が少なくて済むのである. Push形式ではターゲットのDB2からみるとリモートインサートになるため, 行単位で送受信が発生するために極端にブロッキングをする場合に比べ劣化しているのである. つまり32KBのブロックの中に何行の結果行が含まれるのか(ブロッキング)が重要になる. 1ブロック=1行では効果はでない. パフォーマンスを考慮したレプリケーションを実施するには, アプライの実行場所も考慮する必要がある.

5. おわりに

今回, 汎(はん)用のODBCラッパーを使用して異種のレプリケーションを実現した. 理論上PostgreSQLだけでなく未知のデータソースへODBCで接続し, その異種DBの標準機能トリガーを使用しレプリケーションを構築することが可能になることを意味する. このような仕組みでレプリケーションを構築するにはトリガーを異機種のデータベース内に構築できることが前提条件になる. 今回紹介したトリガーと同等の処理が構築できれば異種DBから差分レプリケーションが定義を追加するだけで構築できるため, データベース管理者がFTPや独自の複雑な仕組みを構築する作業が不要になる. またターゲットをニックネームにするとAny to Anyの構成もレプリケーションを可能にする. たとえばPostgreSQLからOracleへなど, ソースとターゲット表両方が異種DBであってもIBMソリューションの仕組のみで提供できることの報告を行った. [6]

一方, トリガー定義やレプリケーション定義は手動で行う必要がある. この作業はソース表が多い場合やひとつの表のカラム数が多い場合, 煩雑な作業をレプリケーション定義者へ課すことになる. 今後はトリガーの自動生成やレプリケーション定義の自動化のエリアの改善を検討したい. マイナーなデータソースではなく, オープンソース・データベースとして認知度が高く世界中で使用されているPostgreSQL, MySQLなどの対応を優先して検討を進めたい.

参考文献

[1] Paolo Bruni, et al., *Data Federation with IBM DB2 Information Integrator* V8.1, IBM Corporation, ISBN0738499161, 2003.10.23

[2] *IBM Systems Journal* -Vol.41, No.4, 2002- Information Integration, http://www.research.ibm.com/journal/sj41-4.html, 2003.10.25

[3] Tadakatsu Azuma et al., *The DB2 Replication Certification Guide*, PRENTICE HALL PTR, ISBN0130824240, 1999

[4] John C. Worsley, Joshua D. Drake, *Practical PostgreSQL*, O'Reilly, ISBN1565928466, 2002

[5] Tadakatsu Azuma et al., *Data Where You Need It, The DPropR Way! DataPropagator Relational Solutions Guide*, IBM Corporation, ISBN 0738408174, 1995.6.22

[6] 東忠克 他, 異種データベースとDB2ファミリーの自動レプリケーション, データベースとWeb情報システムに関するシンポジウム論文集, 情報処理学会, 2003年11月

日本アイ・ビー・エム システム・エンジニアリング株式会社
ICP-ITスペシャリスト
東 忠克 Tadakatsu Azuma

[プロフィール]
日本アイ・ビー・エム1986年入社. 1992年に日本で初めてDB2-SQL/DSのDRDA構成のテストを実施後1995年に日本アイ・ビー・エム システムズ・エンジニアリングへ出向. マルチプラットフォーム, 分散データベース上で稼動するDataPropagatorやInformation Integratorを中心にそれらの普及のため, 多くのお客様に技術支援を実施してきた. これらの経験を生かし現在海外DB2 UDB開発部門へ次期バージョンなどの開発支援も実施している.

日本アイ・ビー・エム システムズ・エンジニアリング株式会社
ITスペシャリスト
曽田 俊明 Toshiaki Sota

[プロフィール]
2002年, 日本アイ・ビー・エム入社. 同年, 日本アイ・ビー・エム システムズ・エンジニアリング出向. DB2 UDBの技術サポートを担当し, 主に他社データベースとの連携に関するお客様のプロジェクトの技術支援, 機能検証, 研修を実施.