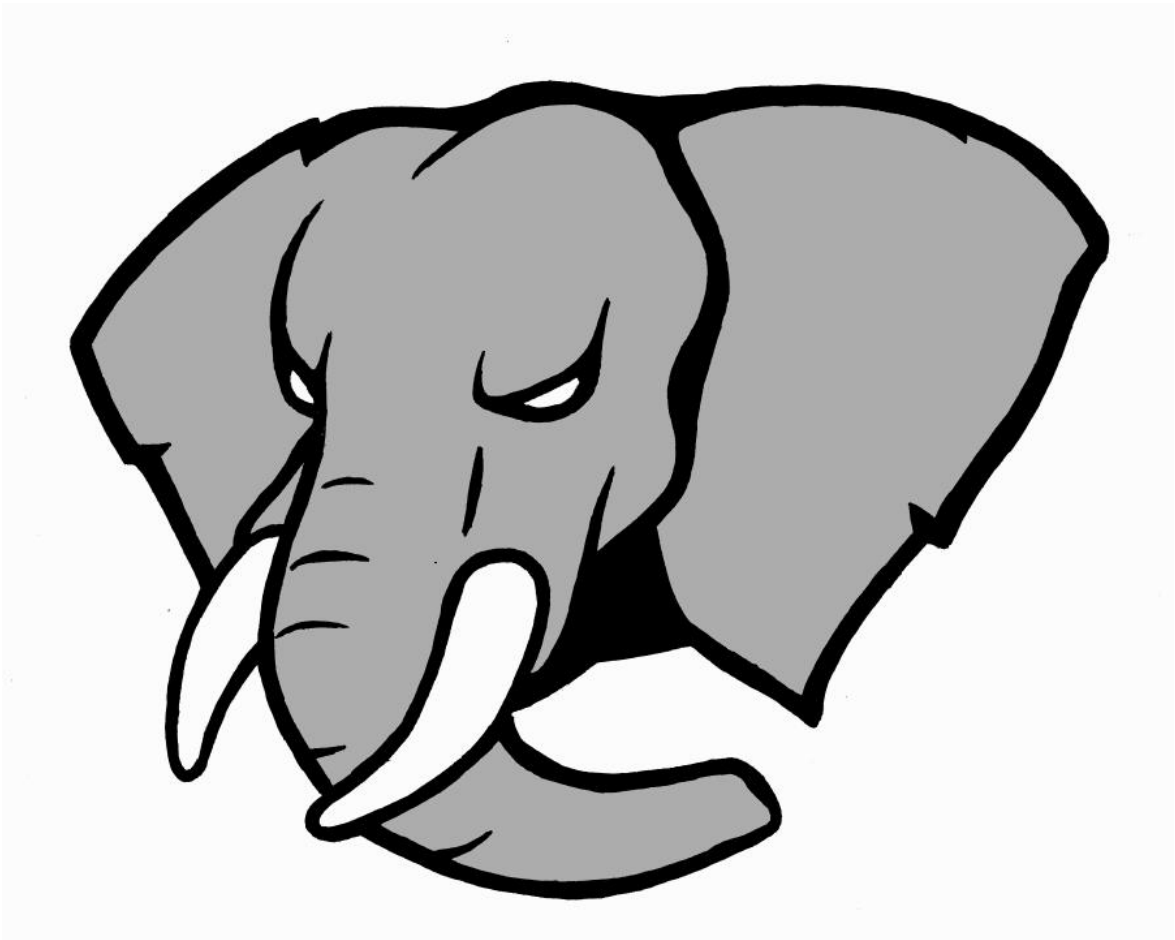

Slony-I
PostgreSQL レプリケーションシステム

概念

Jan Wieck

Afilias USA INC.
Harsham, Pennsylvania, USA

概要

この文書は Slony-I の実装に関する設計目標と技術的概要を記載しています。Slony-I は PostgreSQL ORDBMS に対応する一群の新規レプリケーション手法の中で一番初めに仲間入りしたものです。

目次

1. 設計目標

本章では最終製品に反映される最も重要な設計目標について簡単に概要を示します。

Slony-I 開発の掲げている大きな構想は大規模データベースを相応な限定された数のスレーブシステムに複製するのに必要な機能と能力を備えたマスター・スレーブシステムを構築するものです。PostgreSQL に対する既存のレプリケーションシステムの分析によって初期設計でレプリケーション機能が企画されていないので、現存するシステムとして基本機能のレプリケーションを追加するのは文字通り不可能なことが判りました。

本章で定義される中心的能力は最初のリリースでは全て実装されるとは限りませんでした。とは言っても、これらは、運用されているシステムに与える影響を最小限に抑える形で、後で追加されるべきシステムのメタデータと管理構造の統合の部分である必要があります。

利用可能ないくつかのレプリケーション手法は「1つの型で全てに適合させる」という理論に基づいていますが、データベースのレプリケーションに限ってはその通りになりません。Slony-I は全てのノードが有用であるいう通常の運用を行うデータセンターとバックアップサイトのシステムとして計画されました。運転休止期間が長くなると、故障した構成ノードを削除もしくは無効にしなくてはなりません。同期を取るためにだけ利用される散発的なオフラインノード（旅中のセールスマン）およびマルチマスター、もしくは同期レプリケーションは今のところサポートされませんが、将来 Slony-I の一群になる予定の案件です。

1.1. カスケードされた複数スレーブに対するマスター

Slony-I インストレーションで結合されたシステムの基本構造は1つ以上複数のスレーブノードを持つ1つのマスターで構成されます。全てのスレーブノードがマスターから直接レプリケーションデータを受け取る必要はありません。有効な発信先からデータを受け取るノードはそのデータを他のノードに転送できるように構成することができます。

この能力の背景には明瞭な3つの考え方があります。第1は拡張性です。1つのデータベース、特にクライアントアプリケーションから全てのトランザクション更新を受け取るマスターは、レプリケーション工程でスレーブノードの問い合わせを満足させると言う限られた機能のみ所有します。数多くの読み取り専用スレーブシステムの要求を満足するためにはカスケードできるようになっていなければなりません。

第2の概念は遠隔地の複数スレーブを有効に保ちながらバックアップサイトに対するネットワーク帯域幅に限度を設けることです。

第3の概念はフェイルオーバーに対する筋書きを構成可能にすることです。複数スレーブに対するマスター構成では、マスターが故障した時点で全てのスレーブノードが確実に同じ同期状態にあることは稀でしょう。1つのスレーブがマスターに格上げされることを確実にするには、残っている全てのシステムがデータのステータスについて一致できなければなりません。ひとたびトランザクションがコミットされるとロールバックはできませんので、このステータスは紛れもなく残りの全てのスレーブノードの最新の SYNC 状態です。これと全ての他のノードとの差分は容易かつ迅速に生成され（同一システムでなければ）格上げされる前に最低限新規マスターに適用されなければなりません。

1.2. 稼働中のインストレーションと構成

全てのレプリケーションシステムをクライアントアプリケーションの停止をさせず、稼働中のデータベースシステムにインストールもしくはアンインストールできることが必須です。これにはマスターシステムの初期構成、1つもしくはそれ以上のスレーブの構成、データを複製し、全稼動のマスター・スレーブ状態への捕捉が含まれます。

構成の変更は同時にカスケードされたスレーブノードが、すぐさまそのデータ提供ノードを変更できることも含みます。特に前節で説明したフェイルオーバーの筋書きに対して最上位層にあるスレーブのひとつをマスターに昇格させ、その他の最上位層にあるスレーブを新しく昇格されたマスターからレプリケーションを行うようにリダイレクトさせることで新規マスターの付加を軽減します。

さらに、データベース稼動中のインストレーションと構成の変更はレプリケーションソフトウェアそのものを、現在使用しているバージョンと内部メタデータで互換性を持たない新しいバージョンにアップグレードする機能を唯一保障します。

この条件を満足したとしてもスレーブをアップグレードするためにはスレーブに割り込みを掛けなければなりません。古いバージョンと平行して新しいバージョンをインストールする機能を最低限提供するために、新しいスレーブを作成し既存のスレーブがシステムから削除される前に立ち上げます。

1.3. データベーススキーマの変更

スキーマの変更をレプリケーションすることはよく論議される問題で、この機能の実装に必要な切り口を提供するデータベースシステムは殆どありません。PostgreSQL はスキーマ変更時に呼び出されるトリガーを定義する機能を持っていないため、PostgreSQL システムの中核に膨大な手入れを行なわずにスキーマ変更を複製する平明な手段はありません。

更に、データベーススキーマの変更は稼動中のシステムではいつ発生するかわからない単独で、隔離された DDL 命令文ではありません。そうではなく、複数データベースオブジェクトに変更を掛ける DDL および DML 命令文がグループ化される傾向の性質があって、新規の列をその初期値に更新するといった大規模なデータ操作を行います。

Slony-I レプリケーションシステムは、複製工程の一部として管理された様式で SQL 構文を実行する機能を備える予定になっています。

1.4. 複数のデータベースバージョン

あるデータベースのバージョンから異なるバージョンへアップグレードする過程に対応させるには、異なった PostgreSQL のバージョン間での複製ができなくてはなりません。

マスターのデータベースアップグレードはスレーブへのフェイルオーバーが実行可能でなければなりません。Slony-I のような純粋な非同期マスター・スレーブシステムはトランザクションのロス無しでフェイルオーバーの機能を提供できません。コミットされたトランザクションをまったく損失なくフェイルオーバーするためにはどうしても同期レプリケーションが必要ですが、これは Slony-I ではサポートされません。ですから、マスター交換を目的とする処理管理に強要されるフェイルオーバーには、現在は新しいマスターに格上げされたクライアントが作業を再開する前、スレーブシステムが状況を捕

捉しマスターにならせるようにクライアントアプリケーションの短時間の割り込みが必要です。

1.5. バックアップとポイントインタイムリカバリー

なぜバックアップとリカバリーがレプリケーションシステムの話題となるかを判りきっている必要はありません。Sloly-I の設計次第という理由は PostgreSQL のデータベースシステムがポイントインタイムリカバリーに失敗し、フェイルオーバーを担当するシステム設計がアプリケーション障害によるデータ破壊をかばうことなしでは不完全だからです。

本ドキュメントの後の方で紹介する技術設計では、バックアップ目的に、1つ以上複数のスレーブを使用すること比較的容易にさせます。更にカスケードされたスレーブが存在するかしないかに関係なく、ある遅延をおいて単独のスレーブに複製したデータを適用するに構成することもできます。高度な可用性を求める筋書きでは、通常バックアップをリストアして、ポイントインタイムリカバリーを行う時間的余裕はありません。バックアップメディアは目的に耐えうる十分な速度を持っていません。1時間遅れの余裕で複製データを適用するスレーブは、論理的に過去60分以内のいかなる時点でもマスターに昇格可能です。少なくとも1つの他のノード（遅延なしに複製を行うマスターもしくはいかなる他のノード）は直前1時間のログ情報を持っていて、これを利用できるのであればバックアップノードはある特定の時点までに状況を捕捉するように指示されて、その後マスターに格上げされます。ノードがマスターより迅速に複製できると仮定すれば（どう維持するかは別として）使えます。この場合、内包する遅延よりも短い時間で済みます。

2. 技術の概要

この章では Slony-I の部品と論理的処理を説明します。

2.1 ノード、セットおよび転送

Slony-I レプリケーションシステムはテーブルとシーケンス番号を複製できます。シーケンス番号を複製することは問題が無くは無いので、詳しく 2.3 節で説明します。

テーブルとシーケンスオブジェクトは物理的にセットにグループ化されます。各セットは同じマスターから生じた他のオブジェクトとは独立したオブジェクトのグループを保有していなければなりません。端的には、外部キー制約として意味付けられ、それらのテーブル内のいかなる連番を生成するのに使用される、全てのシーケンスと関係を持つ全てのテーブルは、1つにまとめられて同じセットに属さなければなりません。

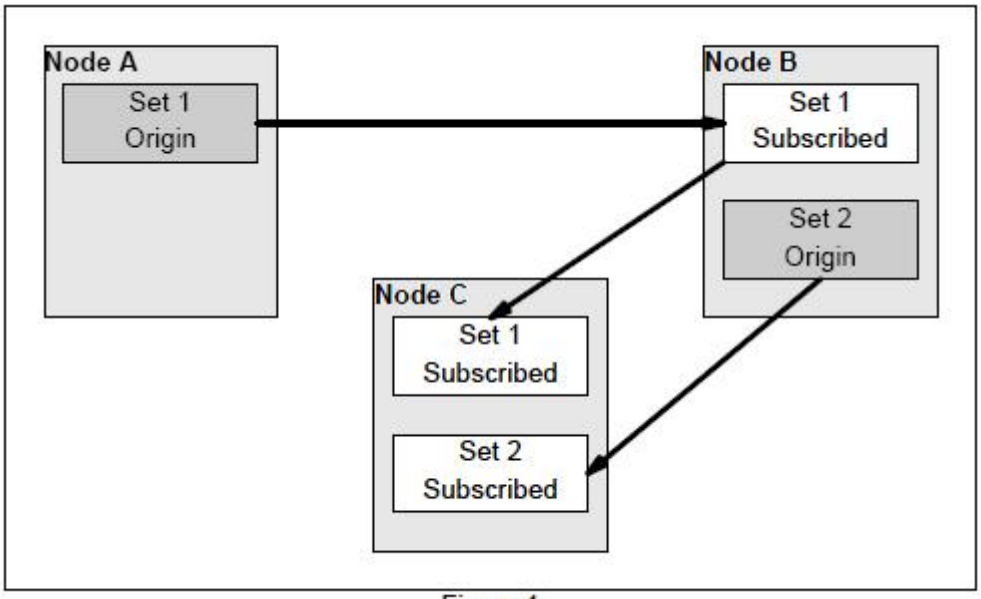

Figure 1

Figure 1はオリジンが異なる2つのデータセットを所有するレプリケーションの構成を図示したものです。ノードCに2つのデータセットを複製する場合、ノードCはセット1のオリジンと連絡を取り合う必要はありません。この筋書きは全てのノードに対し完全な冗長性を有しています。ノードCに障害が発生したとしても、明らかにセット1とセット2のマスターは稼動しているので問題はありません。ノードAに障害が起こるとノードBが2つのセットのマスターに昇格されます。扱いにくい状況はノードBに障害が起こった時です。

ノードBに障害が起きた場合、ノードCがセット2のマスターに昇格されノードAからセット1を複製し続ける必要があります。これを実現するにはノードAはノードCとセット1に対する取り込み分が判っていなければなりません。一般的に、複製ログ情報を保存する全てのノードは、影響を受けるセットの全ての加入者がそのデータを複製済みと認識するまで、ログ情報を保有し続けなければなりません。

この論理を単純化するとすれば、全てのノードを含むネットワーク全体の構成としてセットと加入状況は全てのノードに対して転送され、そこに保存されなければなりません。

2.2. データベース稼動のログ取得

Slony-I は AFTER ROW トリガーを元にした複製（レプリケーション）システムで、これは実際のデータ行の変更を表現する、意義のあるいくつかの SQL 文を再構成するための NEW と OLD 行を分析します。ログの中の行を特定するには、テーブルに UNIQUE（一意）制約が必要です。これにはいかなるデータ型が混在しても構いません。もしも制約が無い場合、Slony-I インストール過程でそのテーブルに int8 の列の追加が必要です。UPDATE 事象で変更されないフィールドは構文に含まれません。現存するいくつかのレプリケーション方式は、複製周期の期間、保存されなければならないログ情報が増加するに関わらず、この技術は、システムに全体にわたってどのアプリケーションテーブ

ルが複製されなければならないかに対する情報を、現在の行から複製時に最新の値を取ってくることで優位性を持っています。

安定性: 履歴情報が失われた場合簡単に解決することが困難な複製されたキーの衝突がある場合があります。最も単純な場合を示すとすれば、以下のような2つの行がそれぞれの値を交換する一意のフィールドです。

```
UPDATE table SET col = 'temp' WHERE col = 'A';
UPDATE table SET col = 'A' WHERE col = 'B';
UPDATE table SET col = 'B' WHERE col = 'temp';
```

'temp'の値に対し余分な手を加えなくしてレプリケーションエンジンがこれらの更新を複製できる順序がありません。

分割: 2.4.1.節で説明してあるように数秒間の作業量で Slony-I はレプリケーション活動全体量をより小さい単位に分割します。これは2つのシリアライザブルトランザクションの可視的境界上で行われます。ですから、複数のマスタートランザクションがあたかも一気に行われれたごとく、スレーブシステムはある矛盾の無い状態から別の状態に移行します。履歴情報が無いと、このことは不可能で、スレーブは最後の同期点から現時点に移行する機会があるだけです。理由の如何を問わずある期間停止させられると、マスター上で行われる全ての仕事の間に及んで、1つの大きなトランザクションで捕捉されなければならず、上に述べた重複キーのリスクも増します。

1.5 に記載された、複製データの遅延したアプリケーションによるポイントインタイム待機機能も、この分割を必要とします。

フェイルオーバー:

スレーブはマスターが故障した時に最も直近のスレーブがある、1つのマスターに対して複数のスレーブでの筋書きでは比較的容易に知らせることができる一方、2つのスレーブ間での行の差分を告げることはほとんど不可能です。ですから、マスターが故障した場合、1つのスレーブがマスターに昇格しますが、その他全てのスレーブは新しくなったマスターと再同期する必要があります。

性能: 1つもしくは非常に数の少ないローテイトログテーブルにログ取得情報を保存することの意味は、複製エンジンが1つの複製ステップに対する実際のデータを、ほんの短い問い合わせで1つのテーブルのみから select するだけで抽出できます。これとは対照的に、複製時にアプリケーションテーブルから現在値をフェッチするシステムでは**複製されたテーブル毎**に同じ数の問い合わせを発行する必要があり、そしてこれらの問い合わせはアプリケーションデータテーブルと共にログテーブル(複数も)と結合することもあります。このシステムの性能は複製されたテーブルの数に反比例することは明らかです。全ての差分が適用されたある時点で(既に指摘したように分割はできません)、PostgreSQL データベースシステムに、これらの問い合わせで返される行数を処理するメモリー内ハッシュ結合問い合わせ計画よりもより最適化を欠かせ、複製システムがマスターの作業量を大幅に削減させること無しに、

状態の捕捉に到達することができません。

通常の状態でログは1つのログテーブルに集められ、そこで定期的に削除され、そして vacuum されます（2.4.2.を参照）。十分な空き領域を持った、程ほどに大きいテーブルの方が、区画の終わりに拡張された空テーブルよりも、INSERT 操作でよりよい性能が得られます。これは、PostgreSQL の空き領域の取り扱いは、複数のバックエンドが、同時に異なるブロックに新規タプルを追加することを認めるからです。同時に、区画の終わりにテーブルを拡張すると、これらのブロックはキャッシュの中には決して見つからず、ファイル容量の増加による OS 内のファイルシステムメタデータの変更が必要になるため、既存のブロックを再使用することより高価です。他のテーブルへのログ切替機構は、テーブルに排他ロックを掛け、効果的にはクライアントアプリケーションを停止することになる VACUUM FULL を実行しないので、ログテーブルを縮小可能にするように、ログテーブルが一度適切な大きさを超えた場合に備えて、提供されています。

それぞれのログ行には現在のトランザクション識別子、局所ノード識別子、適用されるテーブル識別子、ログ行為シーケンス番号、およびスレーブシステム上で同じ変更を行う SQL 命令文を再構築するのに必要な情報が入っています。行為シーケンスは AFTER ROW トリガーに割り当てられているため、その昇順は自動的に、テータベーステーブルに起こる同時更新の順序と競合しない順番になります。全く同じ順番で更新がされる必要は無く、それらが可視となる、言い換えればそれらのトランザクションのコミットがなされる順序では無いことは確かです。とは言っても、可視となる順序付けを持った、論理的に昇順のトランザクショングループ内で、この順序で実行される命令文は正確に同じ結果をもたらします。この順序は合意可能順序（agreeable order）と呼ばれます。

2.3. シーケンスの複製

PostgreSQL におけるシーケンス番号の生成器は同時性に関して高度に最適化されています。と言うことは、重複した識別番号を生成しないことを保障しているだけです。ロールバックを行わないので、従ってギャップを生成します。他の問題として、シーケンス番号に基づいたトリガーを定義することができません。

PostgreSQL のシーケンス番号は64ビット整数であることから、全ての有効な範囲を複数セグメントに分割し、マスターに結局は格上げされる可能性のある全てのノードに、それぞれ一意の範囲を割り当てることは、まったく問題ありません。この様にして、複製工程でシーケンスを単に無視することができます。欠点は、バックアップ/リストア工程では無視されず、シーケンスを再調整せずに正しくないバックアップをリストアしてしまう危険性が高いことです。

他の可能性として、ユーザ定義関数を使い、複製されたテーブルが所有している行でシーケンスを効率的に交換し、そして同時性を無効にし、シーケンスがクライアントアプリケーション全体に渡り主たるボトルネックになることがあります。

他に見られる手法では、シーケンスを複製せず、スレーブがマスターに昇格する時に調整します。このことには、シーケンスが生成した値を所有する全てのテーブルに、少なくとも1回完全なテーブル走査必要で、フェイルオーバー工程で大幅な遅延が発生することを意味します。

Slony-I の手法は異なります。シーケンス番号を生成する標準関数、*nextval()* および *setval()*

は、他の名前とオブジェクト識別子を持った新規の pg_proc カタログエントリを作成することにより排除されます。シーケンス複製の場合、関数はログテーブルに複製動作を挿入します。ログテーブルに何らの更新もされず、クリーンアップ工程で過去のログエントリのみが削除されるのであれば、シーケンスを割り当てないように現在のトランザクションをブロックします。中止されたトランザクションは、割り当てられたシーケンスを失うという事実は、いずれにしても次の割り当てでスキップされるので、無視することができます。

スレーブは複製の過程でシーケンス番号を後ろ向きに調整しないように注意しなければなりません。とい言うのは、シーケンス記録行為の合意可能順序を保証する副作用、アプリケーションテーブルの行ロックはシーケンスのために存在するのではありません。シーケンス番号の割り当ては論理的に同時に起こり、そしてそれは、BEFORE ROW トリガーが起動を掛け、取って代わるための nextval() 関数内で、同時性の理由から直列化が望まなれい競争状態（元の netval() を呼び出しログレコードを挿入する間）以前であっても起こります。

2.4. ノードデーモン

Slony-I で複製システムに参加する全てのデータベースはノードです。データベースは異なるサーバに存在する必要も、異なるpostmasterでサービスが提供される必要もありません。2つの異なるデータベースは2つの異なるノードです。

複製システムのそれぞれのデータベースで、Slon と呼ばれるノードデーモンが動き出します。このデーモンは複製エンジンそのもので、マスターとスレーブの機能を持った複合プログラムです。ノードの役割はデータベースレベルではなく、セットレベルで定義されるだけなため、Slony-I でマスターとスレーブを差別化するのは、とにかく適正でありません。Slon には以下の義務があります。

2.4.1. ログデータの分割

ログデータを倫理的に昇順のトランザクションのグループに分割することは、想像する以上に簡単です。Slony-I デーモンは局所ノードのログ行為シーケンス番号が変更になったかを、構成可能なタイムアウト時間で検査しますので、変更が認められると SYNC 事象を生成します。システムが生成した全ての事象は直列化可能（serializable）なトランザクション内で生成され、1つのオブジェクトにロックを掛けます。この様にして、それら事象のシーケンスが生成されるコミットされたと全く同じ順序であることが保障されます。

事象は電文コードとそれの運ぶ情報、トランザクション全体の直列化可能スナップショット情報、これらの中からのものを含みます。どんな2つの昇順 SYNC 事象間でコミットされた全てのトランザクションは、従って以下の様に定義されます。

```
SELECT xid FROM logtable
   WHERE (xid > SYNC1_maxxid OR
         (xid >= SYNC1_minxid AND xid IN (SYNC1_xip)))
   AND   (xid < SYNC2_minxid OR
         (xid <= SYNC2_maxxid AND xid NOT IN (SYNC2_xip)));
```

2.4.5. 節で述べた動作による実際の問い合わせはさらにもっと複雑です。しかし、結局のところ一般原理はこの様に単純で、局所ノードのデーモンは局所ログ行為シーケンスを検査するだけで、シーケンスが変更されれば行を挿入し、通知を生成します。

2.4.2. 電文交換

ノードの追加、セットの購読もしくは購読取り止め、セットのテーブル追加、などなどの全ての構成の変更は、事象としてシステムを通じ通信されます。事象は事象情報をテーブルに挿入することによって生成され、同じと所の全ての監視ノード（listener）に通知されます。SYNC 電文は同じ機構で通信されます。

Slony-I システム構成は、どの事象かを問い合わせる他の全てのノードの情報を所有します。

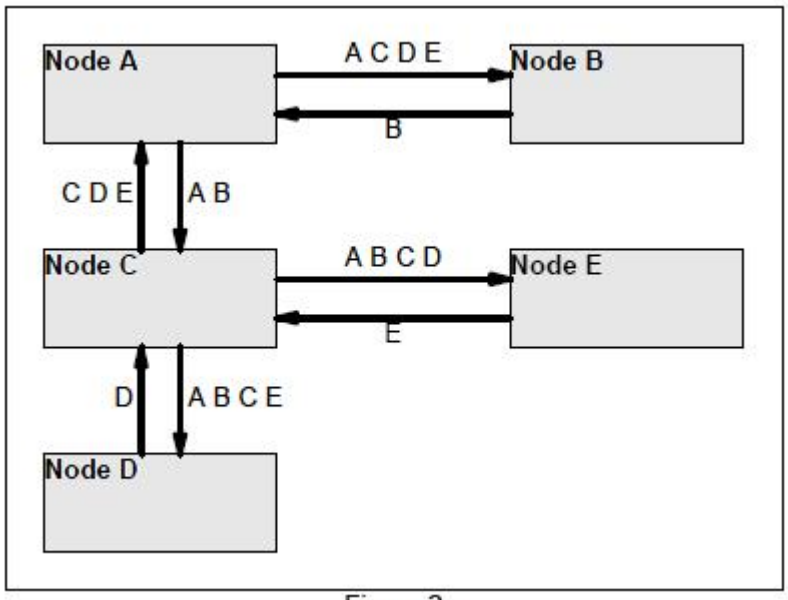

Figure 2

Figure 2 は5ノード構成の事象フローを表しており、そこでは以下のノードの組み合わせ間に存在する直接の接続のみが存在します。

```
NodeA <-> NodeB
NodeA <-> NodeC
NodeC <-> NodeD
NodeC <-> NodeE
```

全てのデーモンはそのノードへの、そこから（Figure 2 で示されている様に、事象オリジンである必要はありません）事象を受け取る遠隔データベース接続を確立します。デーモンは事象生成についてお互いに通知し合うため、PostgreSQL の LISTEN/NOTIFY 機構を使用します。

新規事象を受け取る時、同じトランザクション内でデーモンはその事象を処理します。この様にして、事象は転送されるようになり、転送が保障され、連鎖の中で次の受け取りノードに事象が到達した時点で、全ての必要なデータは転送ノードに保存され、使用可能になります。

ノード D もしくはノード E で生成された事象がノード B で見られる以前、ちょっとの間移動する事実は問題ありません。事象がそこから発生した同じノードで発生する如何なるセットで購読されるのであれば、如何なるノードに対しても SYNC 電文を含んだ事象のみが重要です。

双方とも転送を行うことができるノード B とノード C で、現在購読されているノード A で発生するデータセットを仮定します。このデータセットは現在ノード D で購読されなくてはなりません。実際の購読事象は、データセットのオリジンであるノードＡで生成されなければならず、セットの全ての購読ノードに SYNC 事象のフロー内で移動しなければなりません。さもないと、ノード B とノード C はどの論理ポイントインタイムにおいて、ノードＤがセットを購読済みかを知ることができず、Ｄに転送する可能性のある複製データを保持する必要があるのか、知る由がありません。ノードＣの事象待ち行列を観察することで、事象をノード D が受け取るとき、この購読事象に先駆ける SYNC 事象までに、C が全ての複製差分を処理したことが保障され、現在 C は、D が将来の SYNC 事象の結果として引き続いて起こる全ての差分を必要とする可能性が判ります。

同様に、ノードBがフロー内での同じ論理ポイントインタイムで電文を購読し、知ったとすれば、この時点から、たとえ現在のデータのスナップショットを提供できる前、もしくはDへの購読電文、そしてDが購読ノードとしてBと交信できるように構成される以前においても、差分情報は、ノードCがいつ故障しても良い様に保存されなければなりません。

余談として、ノードＡでセットが発生するFigure 2の構成では、プロトタイプ開発中に著者が使用したそのものの設定です。全ての構成をインストールすることが可能で、アプリケーションによりノード A が定期的にオンラインで、書き込みアクセスできれば稼動します。

2.4.3. 事象の確認

事象型の大多数は構成変更です。例外は SYNC と SUBSCRIBE 事象で、2.4.5.節と 2.4.6.節でより詳細に取り上げられます。

構成変更事象は、事象データ行の中の局所構成情報を変更するための全ての必要となる情報を伝えます。処理は高々、Slony-I 管理テーブルのうちの１つの行を格納、もしくは削除することです。

同じトランザクション内で局所ノードデーモンは、事象オリジンと一致する局所テーブルに、事象シーケンス番号と局所ノード識別子の確認行を挿入し、事象を処理します。

事象搬送機構に話を戻すと、デーモンはそこで、接続されている全ての遠隔ノードの確認テーブルに同じ確認行を挿入し、そのテーブルに通知（NOTIFY）します。遠隔ノードデーモンはそのテーブルを監視（LISTEN）し、新しい確認行があれば拾い上げて、ネットワークを通じて伝播させます。この様にして、クラスタ内の全てのノードは、局所ノードが事象を首尾よく処理したことがわかります。

2.4.4. クリーニングアップ

ここまでで、多くの事象、確認、そして（うまく行けば）より多くのトランザクションログデータを生成しました。これら全ては、ちょっと後になって放棄されることは言うまでもありません。定期的にノードデーモンは事象、確認、およびログテーブルをクリーンアップします。これは2つの段階で行われます。

1. 確認データが圧縮されます。全てのノードは昇順でオリジン毎の事象を処理することから、＜オリジン、レシーバ＞毎の最も高い事象シーケンス番号を所有した行だけが必要になります。
2. 古くなった事象とログデータを削除します。2.4.5.節で見たごとく、オリジン毎に、最後の事象を保存しておかなければなりません。そして、クラスタ内の全てのノードによっていまだ確認されていない、オリジン毎の最小の事象シーケンスを所有した SYNC 事象を選択し、その結果セットにループを掛けます。SYNC が見つかるたびに、そのオリジンから、SYNC 内のスナップショット情報に基づいて可視である、全ての古い事象と全てのログデータを削除します。

巨大容量化したログデータが一度集積された場合、ログ交換機構がノードを基準として提供されます。ディスク区画を再回収するたった1つの他の方法は full vacuum であり、これにはテーブルに排他ロックを獲得し、そして効率的にクライアントアプリケーションを停止させなければならため、これは必要な機構です。交換モードに入った後、ログテーブルに挿入するトリガーと関数はテーブル変更を使用し開始します。ノードが交換モードにある間、ログデータは論理的に2つのログテーブル間の和集合です。古くなったログテーブルが空であるとクリーンアップ工程が検出すると、ログ交換モードを終了し、変換モードでシステムが見た可能性のある全てのトランザクションが終了するまで待って、古いログテーブルを空にします。

2.4.5. データの複製

遠隔 SYNC を受け取ると、ノードは、事象が生成されたノード上上で生成された如何なるセットに実際に購読されたかを検査します。購読されていなければ、他と同じように単に事象を確認するだけで、それで終わりです。全ての他のノードは、この SYNC 事象以前のログ情報を決して問い合わせないので、（少なくともこのノードに対し）ログデータを持っておく必要がありません。

そのオリジンから、1つもしくはそれ以上のセットが購読されると、実際の複製作業は以下のような工程を取ります。

1. ノードは、SYNC オリジンから購読された如何なるセットに対する転送情報を提供する、全ての遠隔ノードへの接続を検査します。

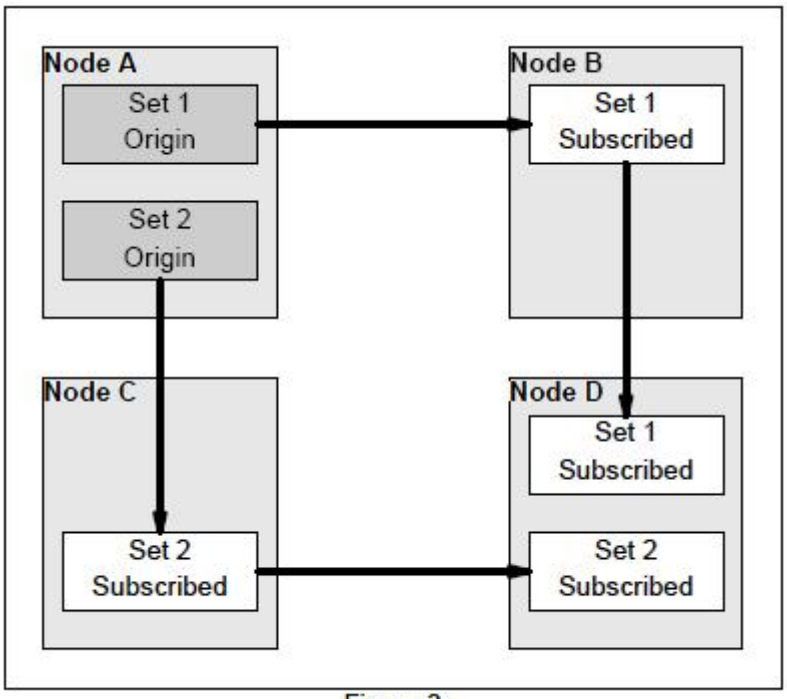

Figure 3

Figure 3 はノード B がセット1のみを複製するように構成された筋書きを示しています。同様に、ノード C はセット2のみを複製するように構成されています。報告目的のため、ノード D は両方のセットへ購読されますが、主ノード A の作業量をできるだけ最小とするため、セット1はノード B から、そしてセット2をノード C から複製します。

この分散データ経路に拘わらず、ノード A で生成された SYNC 事象は双方のセットに対応するように定められており、最終の SYNC 事象から蓄積された両方のセットに対するログデータは、1つのトランザクションで、ノード D に適用されなければなりません。そして、ノード D は、もし両方のノードが既に SYNC 事象を完了済みの場合にのみ、先に進んで、複製作業を開始します。

2. 今、ノードデーモンが行うことは、ログ行為シーケンス順の SYNC 事象オリジンから如何なるセットをも提供する全ての遠隔ノードの使用中のログテーブルの和集合の論理的な選択です。選択されたデータは、特定のノードにより提供された全てのセットに含まれるテーブルと、最後と実際の事象との間に存在するテーブルに限定されます。Figure 3 の例では、ノード D は以下のようにノード B に問い合わせを行います。

```
SELECT * FROM log
      WHERE log_origin = id_of_node A
      AND   log_tableid IN (list_of_tables_in_set_1)
      AND   (log_xid > last_maxxid OR
            (log_xid >= lasit_minxid
            AND log_xid IN (last_xip)))
      AND   (log_xid < SYNC_minxid OR
            (log_xid <= SYNC_maxid
            AND log_xid NOT IN (SYNC_xip)))
      ORDER BY log_origin, log_actionseq;
```

まあ、少なくとも理論を開始することができます。実際には、購読の過程はセット毎に、これらの OR 条件でリストされるため、問い合わせを受けるノードのログ交換の間で、両方のログテーブルの和集合に対し、全てのことを行います。幸いなことに、PostgreSQL は、これが未だオリジンに沿ったインデックス走査で、並び替えを必要としないログテーブルの actionseq であることを認識する、十分に円熟した問い合わせおプティマイザを持っています。

3. これら全ての遠隔結果セットは、ここで複製ノードと併合され、局所データベースに適用されます。これらは正しく並び替えられて到達するので、1行先を見るだけで、ノードは直ぐにそれらを兵具で着ます。どんなテーブル上に定義されたトリガーであっても、SYNC 処理の全期間に渡って無効になります。テーブルにトリガーが定義されている場合、それは同時にセットオリジンの同じテーブルにも定義されているはずです。トリガーによって行われる全ての動作は、複製されたテーブルに影響を及ぼす動作である限り、同様に複製されます。ですから、スレーブで再度トリガーを実行する必要は無く、トリガーをどう掛けるかによっては、マスターとスレーブ間での矛盾を導くこともあります。
4. この全ての問題を引き起こす SYNC 事象は通常保存され、局所トランザクションがコミットされ、そして、確認が全てのほかの事象のために送り出されます。

2.4.6. セットの購読

セットを購読することは、セットのオリジンで発生されなければならない操作です。これは、それらのオリジンで実際に使用されているセットの購読を Slony-I が許可するためで、アップリケーションは同時性をもって、セットデータを変更します。より大きいデータセットの場合、そのデータのスナップショットを生成するのに多少の時間を要し、その間では、該当するセットの複製提供ノードになる可能性のあるノードは、将来ログデータを求めるかもしれない新奇購読ノードが存在するか否かを知らなければなりません。セットオリジンで購読（SUBSCRIBE）事象を生成することは、そのセットのオリジンからやって来る同一の2つの SYNC 事象間で、この事象を全てのノードが受け取ることを保障します。ですから、同じ時点で全てがログデータの保存を開始します。

購読（SUBSCRIBE）事象は、そのセットのログデータを提供するノードから直接受け取らなけれ

ばなりません。これは、ログデータ提供ノードは同時に、新規購読ノードが初期スナップショットを複写するノードである理由からです。

購読（SUBSCRIBE）事象が正式なノードから受け取られると、どのように購読するかの正確な手順は、新規の購読ノードが首位階層のスレーブでか、もしくは、そのセットに関連して、転送するスレーブとそこからカスケードする新規ノードであるため、ログデータ提供ノードがセットオリジンかどうかに依存します。

1. セット内にある全てのテーブルに対し、スレーブはテーブル構成を問い合わせ、それを局所に保存します。同時に、全てのそれらのテーブルに複製トリガーを生成します。
2. セット内のテーブル上の全てのトリガーは、データの複写工の短縮正、正しくない順序のデータを複写することに起因して起こりうる外部キーの衝突を回避、もしくは通達依存の理由により、無効にされます。
3. それぞれのテーブルに対し、双方で PostgreSQL の COPY コマンドを使用し、データストリームを転送します。
4. トリガーがリストアされます。

5a. もしもデータを複写するノードが他の（カスケードしている）スレーブであれば、現在のトランザション内でセットオリジンから来た、最後の可視であるSYNC事象をそのままの状態で、全体のセットとして単に複写するだけです。セットの複写を開始したあと何が起ころうとも、そのセットはこのトランザクションで不可視です。ですから、局所セットの SYNC 状態は、そのようにされた結果として記憶されます。

5b. もし初期複写を受け取ったノードがセットオリジンであれば、問題はセットデータが1つの SYNC 時点から他の時点に「急に移行しない」ことです。この場合、現在処理している購読（SUBSCRIBE）事象と、その最後の SYNC の後に既に見た全ての行為シーケンスの以前に、最後の SYNC 事象を使用することが必要です。既にこれらの行為が適用されたデータ行を既に複写しているので、次の SYNC 事象を跡で処理するときに、それらを明示的に取り除かなければなりません。 これは、新規セットをそのオリジンから直接購読した後に作成された最初の SYNC 事象にだけ適用されます。

6. いつもの様に、購読（SUBSCRIBE）事象はトランザクションがコミットされ、事象の処理が確認された局所ノードに格納されます。

2.4.7. 格納とアーカイブ（保管場所）

カスケードができるようになるため、2.4.5. で併合され適用されたログデータは、同時に局所ログテータテーブルに格納されなければなりません。これは、ログデータが結果として生じた SYNC 事象の挿入と同じトランザクション内で起こるので、このデータを受け取る全てのカスケードしているスレーブは、SYNC 事象が提供ノードにより受け渡されると仮定すれば、SYNC 事象を受け取るのとまったく同じ時に、それが見えるようになります。ログデータには、たまたま、このノードで発生したセットに対する局所ログデータと一緒にクリーンアップされます。この過程は 2.4.4.で既に扱われています。

格納と転送を通じてのカスケードに加え、Slony-I は同時に、バックアップとポイントインタイムリ

カバリ機構を提供することができます。局所ノードのデーモンはそのノードの現在の SYNC ステータスが何であるかを正確に把握しており、pg_dump を開始させるに十分な余裕もたせる、次の SYNC ステータス複製の遅延をさせる能力があり、直列化可能なトランザクションのスナップショットの作成を保障します。結果のダンプは、最後の SYNC 事象が局所でコミットされた時点でのデータベースの正確な姿です。もしそれが、全ての次に引き起こる SYNC 事象に適用される同じ問い合わせを含んだファイルを書き出すのであれば、それらのファイルは一緒になって、マスターで生成される SYNC 事象と同じ粒度でリストアすることができるバックアップを作成します。

2.4.8. 提供ノードの変更とフェイルオーバー

セットを購読する全てのノードが、対応する SYNC 事象を確認する以前に、そのように構成されたノード上にログデータを格納することはすぐさまの提供ノード変更とフェイルオーバーの基礎となっています。

ログデータ提供ノードの変更は、データを格納しないマスターもしくはスレーブの他のノードから、2.4.5. のログデータを選択するための任意のポイントインタイム（もちろん、ある事象でトリガーが掛けられ、通信された時点、その他）に開始を意味する以外の、何者でもありません。

フェイルオーバーは、他のノードと同期を取り、セットのオリジンを変更し、最終的にひねりを入れて提供ノードを変更する論理的シーケンス以上のものではありません。

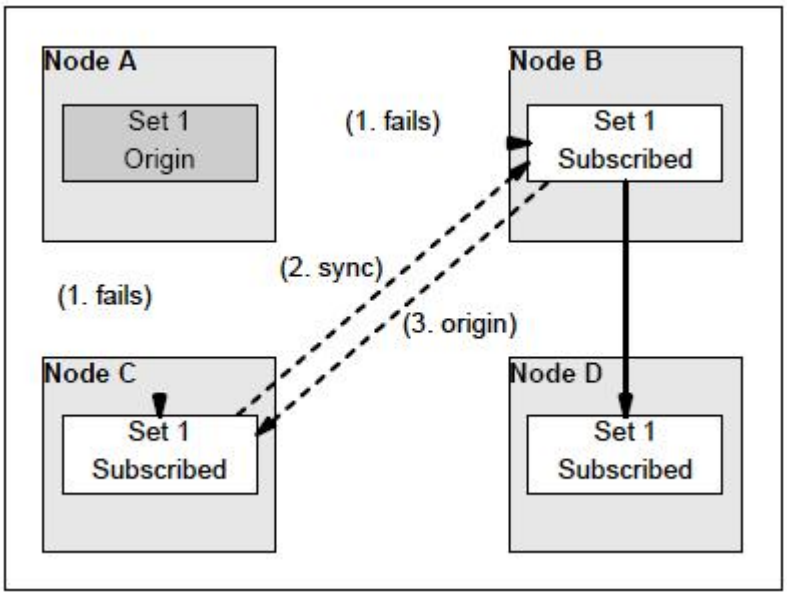

Figure 4

1. Figure 4 でノードAが故障します。それはデータセット1の現在のオリジンです。計画は、ノードBをマスターに昇格させ、ノードCに新規マスターからノ複製を継続させることです。
2. その時点でノードCが複製に関してノードBより、より先行していることがあり得えますので、ノードBは最初に全ての事象（および SYNC 事象に対する関連したログの差分）に、それ自身、未だ

事象を取得していないか尋ねます。この行為において、ノードAに対する複製以外、実際の違いはありません。

3. ノードBがノードCよりも、複製において等しいか、もしくはより先行しているのが確かな時点で、セットを引き取ります（そしてオリジンになります）。ノードCが現在行わなければならない提供ノード変更におけるひねりは、現在まで、ノードCはノードAからの、ノードBが知っている全てのSYNC 事象が複製されたことを保障しません。この様にして、ノードBからのオリジン（ORIGIN）事象はその時点のノードBによって知られた、クラスタ全般に知られた最後のノードA事象でなければならない、最後のノードAの事象を含むことになります。ノードC上でそのオリジン（ORIGIN）事象を処理するひねりは、オリジン（ORIGIN）で列挙された1つが、そしてノードAから事象の全てがノードCで複製されるまで確認されません。その時点でもちろん、ノードCはノードBもしくはDを提供ノードとして使用し、複製を継続するかどうか自由です。

非常に単純なので、この時点で全てのフェイルオーバー過程は比較的簡単に見えます。最初からこの方向に向けられた Slony-I 全体の設計は、実際それ程、驚くに値しません。とは言っても、この単純性には良い値段が付きます。値段とは、もしも（スレーブ）ノードが利用できなくなり、クラスタ内の全てのノードが蓄積された事象情報と可能性のあるログデータのクリーニングアップを停止することです。ですから、もしも長い期間にわたってあるノードが利用できなくなると、構成を変更し、システムに他の手法で再始動させることを知らせることが重要です。これを成就するには、論理的にノードを一時停止させる（無効にする）か、もしくは完全に構成から削除することです。

無効にさせられたノードに対し、まったく最初から再加入をしないで、残りのクラスタに追いつく可能性がまだ残っています。2.4.7. で作成されたポイントインタイムリカバリ差分ファイルを、かなり以前のログテーブルから削除されてしまった情報を供給するために使用することができます。そのノードが再始動のやり直しを終了した時点で、クラスタ内で誰かが再度、再起動された新規ログ情報を保持するため、再起動されたノードは再び差分ログファイルをやり直し、時として、さらにその再始動の前に出現する、最後に知られた SYNC 事象に対応する1つが現れるのを待ちます。それは現在オンラインの背後にあります。

3. 謝辞

いくつかの Slony-I の中心となる原理は、PostgreSQL プロジェクトに寄与されてきた他の複製解決策（ソリューション）から頂きました。すなわち、PostgresSQL で貢献されている、直列化可能隔離レベルに互換性のある、トランザクション境界におけるログ情報一連の流れの分割、ログテーブルを交換させるアイデア、そして、eRServer にどう似通って実装するかです。