題名：
「**PostgreSQL** における負荷分散のための相互レプリケーション実装に関する報告」

著者氏名：三谷　篤

所属：株式会社 **SRA** 西日本　広島支社
所属団体：日本 **PostgreSQL** ユーザー会(**JPUG**)　理事/広島地区支部・支部長

E-Mail：**mitani@sraw.co.jp**

要旨：
負荷分散を目的とした **PostgreSQL** の双方向レプリケーションの実装方式を検討し実装しました．２台の **DB** を使ったベンチマークでは，同時接続数が **16** 未満では **10%～17%**のオーバーヘッドがみられましたが, 同時接続数が **16** 以上ではオーバーヘッドも殆ど見られず, 特に参照系のクエリー（**SELECT**）では高い処理能力が計測され，負荷分散に有用であると思われました

　今後の課題としてデータベースサーバを増設する際，同期のとれた初期データの投入方法や，レプリケーションを実行しているレプリケータの多重化がありますが，双方向レプリケーション機能を用いることで，高負荷なシステムでも **PostgreSQL** を利用することは可能であると考えます.

1.　はじめに

PC が各家庭に普及したことに加え，近年急激に進んだ携帯端末の普及やインターネット・インフラ整備のおかげで，インターネットの利用者は増加し，また利用者層も多様化しています．利用者層の多様化に伴いインターネットサービスへのニーズも多様化する傾向にあります．これに対応するため，インターネットサービス提供側でも CRM のように顧客毎にカスタマイズした多様なサービスを提供できるシステムが求められています．

インターネットを用いたサービスの１つである，Web アプリケーションでもこれらの多様化のニーズを受け，静的なコンテンツを羅列するだけではなく，データベースを用いて動的なコンテンツを自動生成する，いわゆる Web-DB システムが広く使われるようになってきました．

Web サービスとデータベースを連携することで，利用者に多様なサービスを提供したいというニーズは満たされるようになりましたが，一方で利用者数の急激な増加によって Web サーバやデータベースへの過負荷が問題となる事例も増えています．Web サービス単体で考えると，Web サーバの増設し，ロードバランサーとよばれる装置を導入することで負荷分散を図ることができます．しかしデータベースはデータの整合性，一貫性をとる必要があるため，単純にデータベースサーバを増設して，アクセスを振り分けるという方法では負荷を分散することができません．

今回これらの問題を解決するために，非常に高負荷となることの多い Web アプリケーションにおいて，データベースの負荷分散を実施することを目的に，負荷分散方式を検討しました．データベースにはオープンソースのデータベースである PostgreSQL を選定しました．PostgreSQL は商用データベースに勝るとも劣らない高機能を備えている上フリーですので，Web-DB システムに広く利用されています．しかし，PostgreSQL は分散機能を持っていないため，システムが高負荷になった場合，致命的な問題となることがあります．負荷分散を目的とした機能追加が PostgreSQL に実装できれば，これまで利用を諦めていた高負荷なシステムでも利用可能になり，その可能性が広がると思われます．

2.　PostgreSQL の負荷分散を行うための実装に関する検討

2.1.　分散方式に関する検討

まずどのような方法で負荷分散を行うか、分散方式について検討しました。

データベースの負荷分散を行うためには、処理（負荷）を受け持つデータベースサーバを複数台に分散する必要があります。分散の仕方には大きく分けて以下の 2 種類が考えられます。

１つは、複数のデータベースサーバが個々の記憶装置内に個々のデータを格納してデータベースエンジンが複数のデータベースサーバにまたがって処理を行う分散方式です（以下、Non Data Shared 方式とします）。

もう１つは複数のデータベースサーバが１つの記憶装置を共有して処理を行う分散方式です（以下、Data Shared 方式とします）。

2.1.1. Non Data Shared 方式の特徴

Non Data Shared 方式では、個々のデータベースサーバは個々のデータを保有してよいため、システム内にデータベースサーバを追加することが簡単です。また、各データベースサーバ間で重複するデータを持つ必要がないため、全体として **1** 台のデータベースサーバでは扱うことのできないような大規模なデータを扱うことのできます。

しかし、**1** 台でもデータベースサーバが壊れた場合、全件検索はできなくなるなど、データベース全体に影響を及ぼしてしまいます。また、分散しているデータベースサーバにはバランス良くデータを配置しないと、特定のデータベースサーバに負荷が集中することになってしまいます。データベース稼動中は、データは動的に変化し増減します。また、個々のデータのアクセス頻度はデータ格納時には推定することが困難なため、データをバランス良く配置することは実際には非常に困難です。

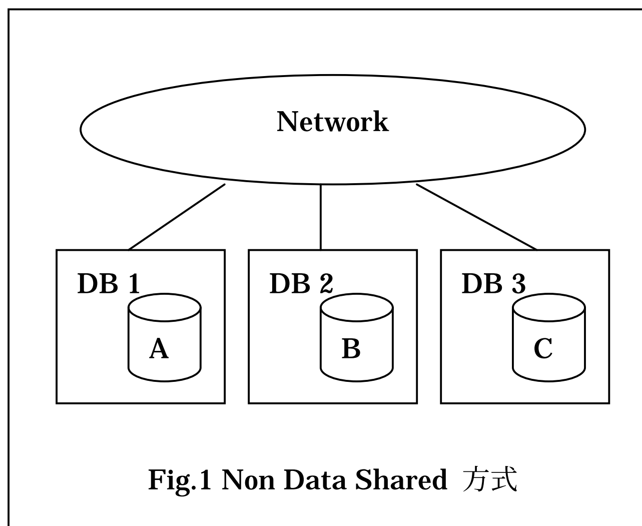


Fig.1 Non Data Shared 方式

2.1.2. Data Shared 方式の特徴

Data Shared 方式では、個々のデータベースサーバは同じデータを共有していますので、**1** 台のデータベースサーバが壊れても、それ以外のデータベースサーバでサービスを継続して提供することが可能です。また、データベースサーバのマシンスペックが同じであれば、どのデータベースサーバにアクセスしてもレスポンス速度は同じであり、データによってアクセスするデータベースサーバを選択する必要がなく、難しい調整なしにバランスのとれた並列処理が可能です。

しかし、システムにデータベースサーバを追加する場合、サーバ追加の通知や初期データの同期をとる必要があり（実装方式に依存しますが）、一旦システムを停止するなどの手順が必要です。また、個々のデータベースサーバは同じデータを扱いますので、全体として扱うことの出来るデータ量は大きくなりません。

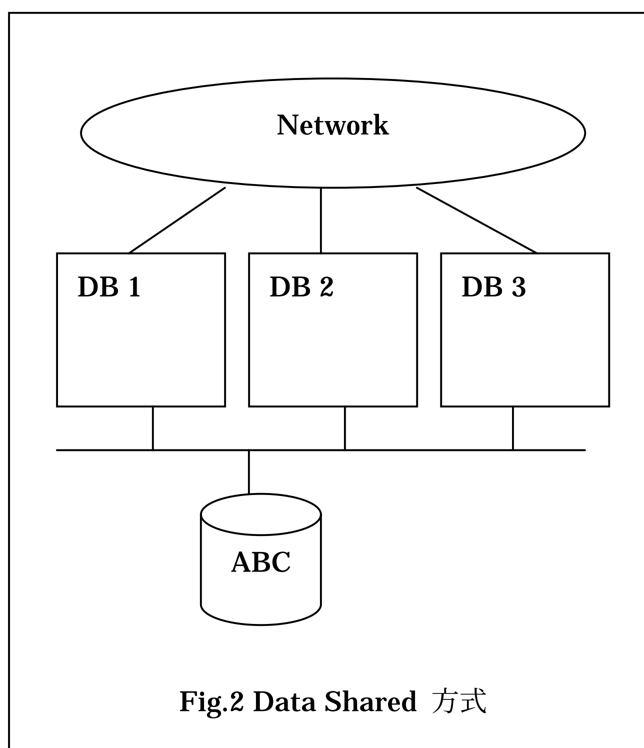


Fig.2 Data Shared 方式

2.1.3. 分散方式の選択

これらのことから、**Non Data Shared** 方式は大規模なデータを処理する場合に向いており、**Data Shared** 方式は大量のアクセスを処理する場合に向いていると考えられます。

今回の負荷分散の目的は、大量のアクセスによるデータベースの負荷を分散・軽減することですので、**Data Shared** 方式を選択することが目的に沿っていると考えます。

3. PostgreSQL における実装方式の検討

次に **PostgreSQL** における、**Data Shared** 方式の実装方法を検討しました。

Data Shared 方式ではデータを各データベースサーバ間で共有しますので、データを格納している装置を共有する必要があります。データを格納する装置には大きく分けて主記憶装置（以下、メモリとします）と補助記憶装置（以下、ハードディスクとします）の **2** 種類があります。また共有方法も物理的に共有する方法と仮想的に共有する方法の **2** 種類が考えられます。したがって実装方法はこれらを組み合わせて、**(1)**ハードディスクを物理的に共有する方法、**(2)**ハードディスクとメモリを物理的に共有する方法、**(3)**ハードディスクを仮想的に共有する方法、**(4)**ハードディスクとメモリを仮想的に共有する方法の **4** 種類の方法が考えられます。

3.1.1. ハードディスクを物理的に共有する方法

物理的な共有方法の場合、**CPU** とハードディスクをバスで共有することになるため、物理的に分散することができません。また、複数の **CPU** で記憶装置の制御が可能か否かは、バスコントロールを行うチップセットや **CPU** の仕様に依存します。物理的な共有方法は、いわゆる **2CPU** や **16CPU** サーバと呼ばれるマシン構成を指すことになります。**PostgreSQL** は既にマルチタスクで並列処理ができるように実装されており、この共有方法ではアプリケーションプログラム側で考慮できることは特にないと考えます。

3.1.2. ハードディスクとメモリを物理的に共有する方法

この共有方法も上記と同じで、アプリケーションプログラム側で考慮できることは特にないと考えます。

3.1.3. ハードディスクを仮想的に共有する方法

ハードディスクの仮想的な共有方法には、**NFS** のようにリモートのハードディスクをネットワーク経由で仮想的にローカルのハードディスクであるように扱う共有方法や、ミラーリング処理によって物理的に異なるハードディスクでデータだけを共有する方法が考えられます。

PostgreSQL の場合、データベースのデータはデータベースサーバ起動時に指定したデータパスのディレクトリ以下にファイルとして保存されますので、データパス以下のデータファイルを仮想的に共有することは可能です。

しかし、**PostgreSQL** ではデータの検索や追加、変更、削除等の処理は直接データファイルを使って行われるのではなく、全てメモリ上で行われます。したがってデータファイルだけを共有してもデータベースサーバで保有しているデータを全て共有したことにはなりません。バージョン **7.1** で導入された **WAL(Write Ahead Log)**により、データの変更はメモリでの操作が終了した後、データファイルに書き込まれる前に **XLOG** というファイルにデータの変更部分を保存するようになっています。**XLOG** ファイルを共有することでデータ

の変更自体は共有することが出来るようになりましたが、**XLOG** だけを共有してもメモリへ反映しないとデータの整合性をとることはできません。

3.1.4. ハードディスクとメモリを仮想的に共有する方法

上記の理由から、**PostgreSQL** においてデータを共有するためにはハードディスクとメモリの両方を仮想的に共有する必要があることが分かります。

PostgreSQL はマルチタスクで動くように設計されているため、タスク間で同じメモリを利用できる共有メモリ上に全てのデータは展開されています。そのため、ネットワークを経由して仮想的にメモリを共有することは可能です。

しかし、これには負荷分散の面で問題があります。最初にデータをメモリに展開したデータベースサーバのメモリを全てのデータベースサーバで共有してしまうため、最初にデータをメモリに展開したサーバに全システム分の負荷がかかってしまいます。

3.1.5. 実装方式の選択

ハードディスクとメモリを仮想的に共有し、かつ、負荷が **1** 箇所に偏らないようにするにはどのような方法としてどのような方法を検討しました。

負荷が均一に分散できるためには、分散しているデータベースサーバの **CPU** がそれぞれメモリを保有し、そこに「同じデータ」が展開されていることが必要です。この「同じデータ」とはデータの内容が「同じ」という意味で、格納されているメモリ番地まで同じである必要はありません。これはハードディスクに対しても同じことが言えます。そうであれば、ハードディスクやメモリなどの物理的な記憶装置の仮想共有ではなく、データだけを仮想共有できればよいということになります。これを行う方法としてデータベースサーバ間で相互に動的なデータ複製（以下、リプリケーションとします）が考えられます。以上のことから、**PostgreSQL** にリプリケーションを導入することによってデータを仮想的に共有し、負荷を分散させることが可能であると考えます。

3.2. 実装するレプリケーションの選択

レプリケーションを行うには大きく分けて **2** 種類の方法があります。１つはデータの変更が行われる前に、受信したクエリーを各データベースサーバに配信する方法です。もう１つはデータ変更が終了した後、データ変更の差分を各データベースサーバに通知し、各データベースサーバの記憶装置のデータに更新をかける方法です。

前者の方法は **SQL** や **PostgreSQL** のコマンド仕様によって標準化されたクエリーをレプリケーションの対象として扱いますので、**PostgreSQL** のバージョンやユーザー定義のテーブル構成などにそれほど影響を受けません。しかし、後者の方法ではデータ変更の差分の持ち方やメモリへの展開方法など、**PostgreSQL** の内部仕様に依存するため、**PostgreSQL** のバージョンやユーザー定義のテーブル構成などの影響を受けると予想されます。したがって、レプリケーションは、受信したクエリーを各データベースサーバに配信する方法の方が汎用的なレプリケータ実装に適していると考えます。

負荷分散を目的とした場合，クエリーを受信し，処理を行うデータベースサーバは複数に分散する必要があります．そのため，負荷分散を目的としたレプリケーションでは，各々のデータベースサーバが受信したクエリーをお互いにレプリケートしあう，双方向レプリケーション方式にする必要があると考えました．

4.　双方向レプリケーション実装に関する検討

4.1.　双方向レプリケーションにおける問題点と解決方法

　双方向レプリケーションにおいて知られている問題として，同一レコードに個々のサーバでデータ更新が行われた場合，データ整合性が崩れるというものがあります.

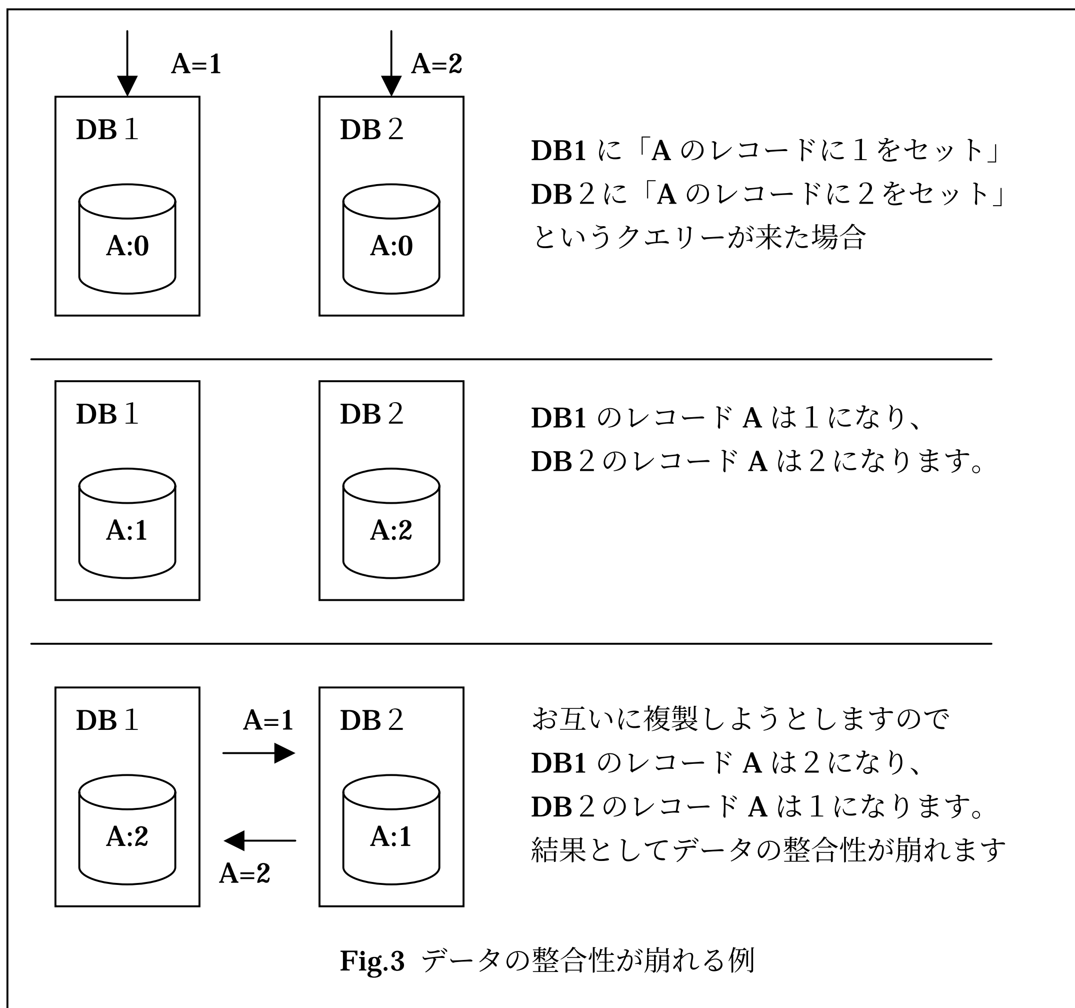


Fig.3 データの整合性が崩れる例

これを解決する方法として，レプリケート対象のクエリーをキューに入れ，シーケンシャルにクエリーを配信する方法を考えました.

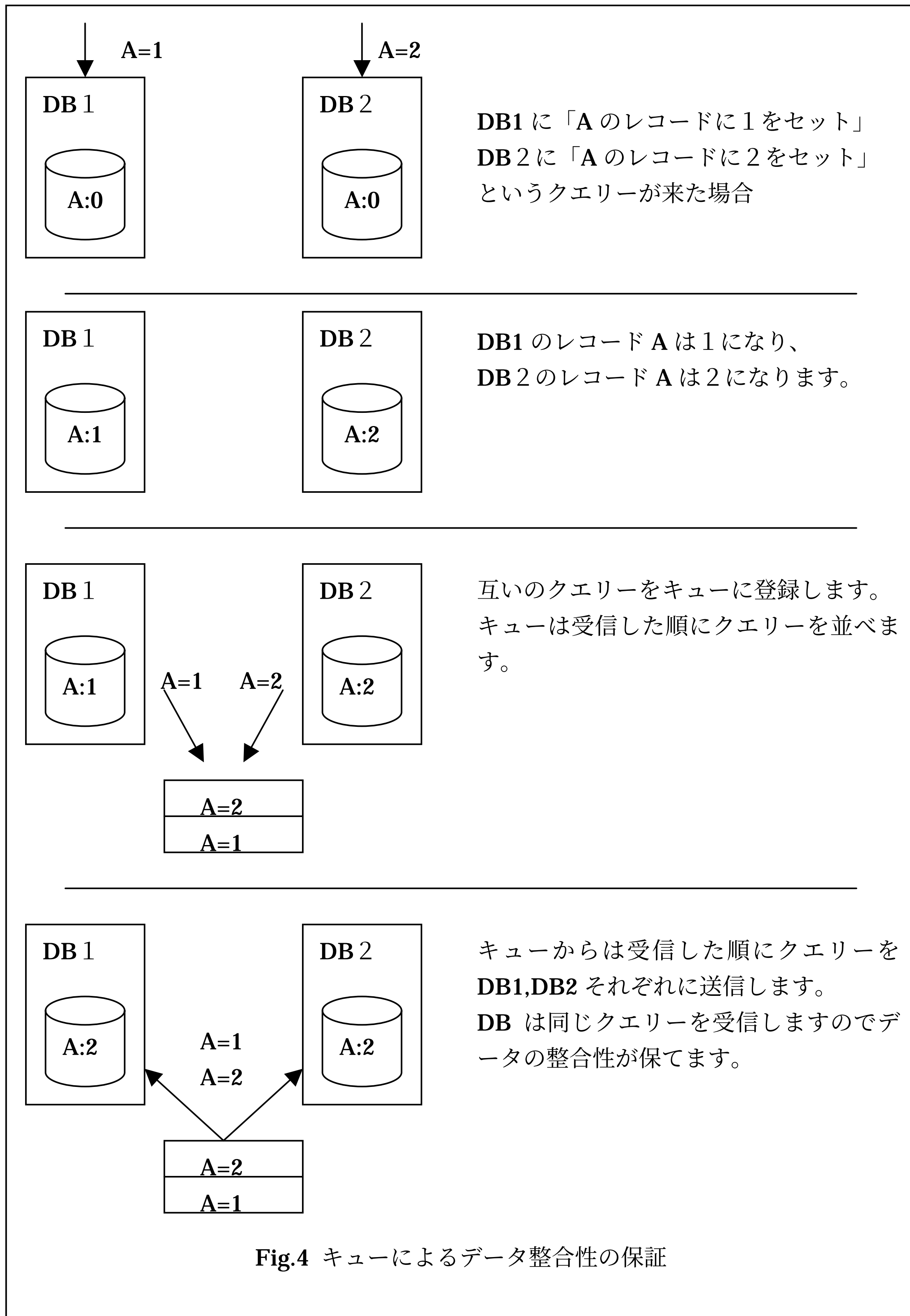


Fig.4 キューによるデータ整合性の保証

4.2.　双方向レプリケーションにおけるトランザクション処理

トランザクション処理を双方向レプリケーションで行う場合，一連のトランザクション処理はそれぞれ独立して処理される必要があります．上記のクエリーをキューに入れる方法だけでは，キューの中で各データベースサーバから発せられたトランザクション処理が入り交ざってしまいます．

これを解決する方法として，トランザクション・クエリーはトランザクション毎に独立したキューに入れる方法を考えました．各々のトランザクション・クエリーを専用のキューに入れるためには，一連のトランザクション・クエリーを識別する必要があります．これには，トランザクション・クエリーに，クエリーを受信したデータベースサーバのプロセスの **ID** を付与し，データベースサーバのホスト名とプロセス **ID** でトランザクション・クエリーを識別する方法を考えました．

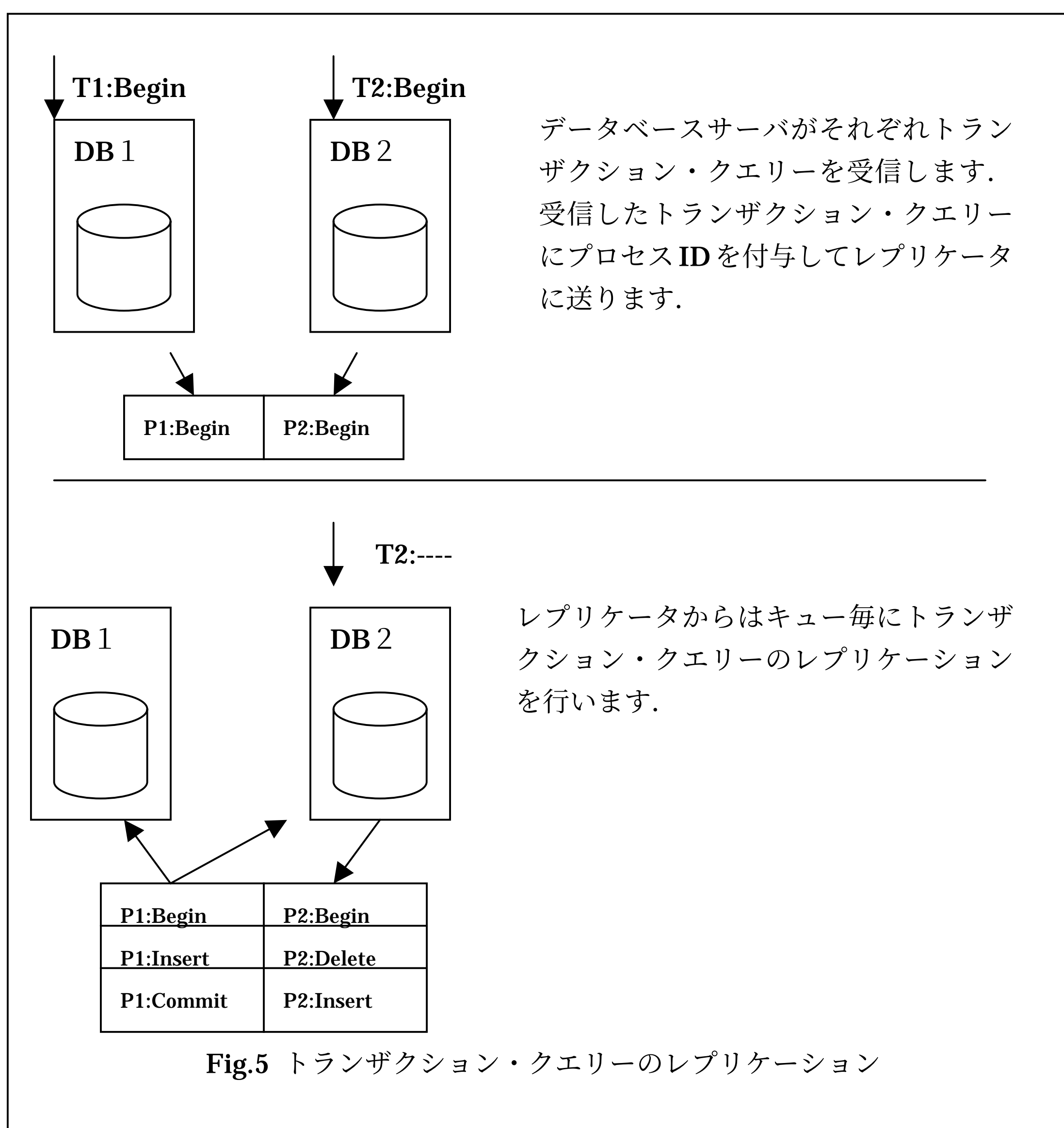


Fig.5 トランザクション・クエリーのレプリケーション

なお，**PostgreSQL** には２相コミットが実装されていませんので，完全な分散トランザク

ションを実装することはできませんが，レプリケーション実行時に各データベースサーバからの戻り値を判定して，**COMMIT** 以外のクエリーが失敗した時には **ROLLBACK** を行い，**COMMIT** の失敗時には，失敗したデータベースサーバを「データベース不良」としてレプリケーション対象から外す方法を考えました.

5.　レプリケーション実装によるシステム構成

システムは，レプリケーション機能を追加したデータベースサーバと，クエリーをキューに並べ，各データベースサーバに配信するレプリケーションサーバ（以下，レプリケータとします）から構成されます.

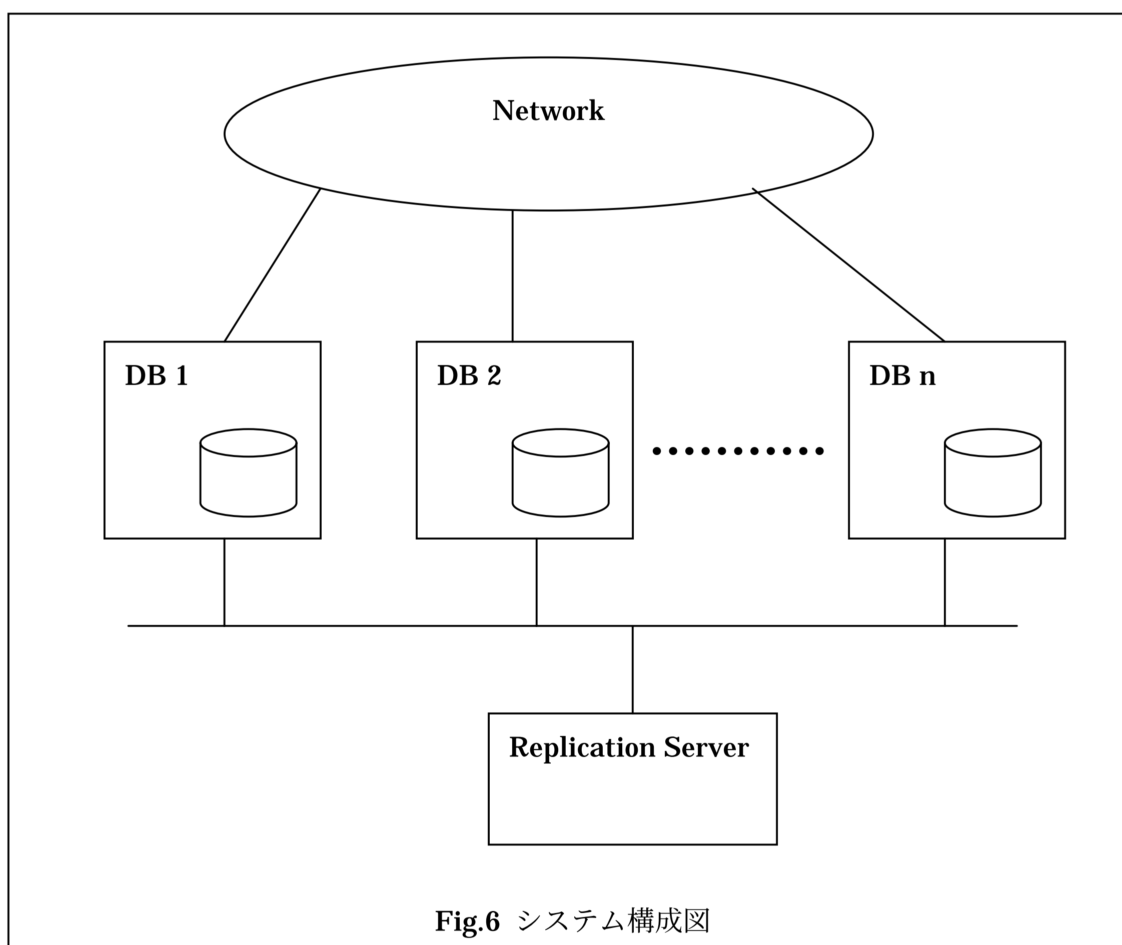


Fig.6 システム構成図

5.1.　データベースサーバにおけるレプリケーション処理

データベースサーバではクエリーを受信後，クエリーの種別を判定します．**INSERT**，**UPDATE**，**DELETE** などであればレプリケーション対象クエリーと判定し，レプリケータにクエリーを送信します．また，**CREATE** や **DROP** などのコマンド・クエリーもレプリケーション対象クエリーと判定してレプリケータに送信します.

SELECT のようにデータ変更を伴わないようなクエリーであっても，データ変更を伴うストアド・プロシジャを呼び出す場合がありますが，クエリーの種別判定ではデータ変更を伴うクエリーか否かの判定はできません．そのため，レプリケーション対象クエリーで

なくても，クエリーが処理された結果，データ変更が実施されて **XLOG** に書き込みが発生した場合，そのクエリーをレプリケータに送信します．

また，クエリー種別判定でレプリケーション対象クエリーと判定しても，クエリーがレプリケータから送信されたものであった場合，レプリケータへの再送をせず，クエリーの処理だけを行います．

5.2. レプリケータにおけるレプリケーション処理

レプリケータは受信した順にクエリーを，送信元以外のレプリケート対象のデータベースサーバに配信します．このとき，**UPDATE** クエリーは送信元のデータベースサーバにも配信します．

6. 双方向レプリケーションの実装

6.1. 環境

6.1.1. ハードウェア環境

実装に用いたマシンの主なスペック及び **OS** は以下の通りです．

Table 1 ハードウェア環境

	OS	CPU	メモリ
データベースサーバ 1	Linux(RedHat 7.2)	Pentium3 800MHz	1GB
データベースサーバ 2	Linux(RedHat 7.2)	Pentium3 800MHz	1GB
レプリケータ	Linux(RedHat 7.2)	Pentium3 600MHz	256MB

6.1.2. ソフトウェア環境

データベースサーバは **PostgreSQL7.2** をベースに開発しました．また，レプリケータは **PostgreSQL7.2** のライブラリを使用して開発しました．開発は **Linux** 上で行い，**PostgreSQL** のソースコードに合わせて **C** 言語で作成しました．

6.1.3. ネットワーク環境

データベースサーバとレプリケータは **100BASE-TX** の **LAN** で結びました．

6.1.4. ベンチマーク環境

性能評価に用いたベンチマークには **PostgreSQL** に標準で同梱されている **pgbench** を使いました．ベンチマーク計測用のレコード数は **10** 万件で，同時接続ユーザ数を１から **128** まで変化させ，**1** 秒間の処理トランザクション数を計測しました．計測は同時接続数毎にそれぞれ **10** 回行い，平均値と標準偏差を算出しました．

ベンチマークは **2** 種類の方法で行いました．

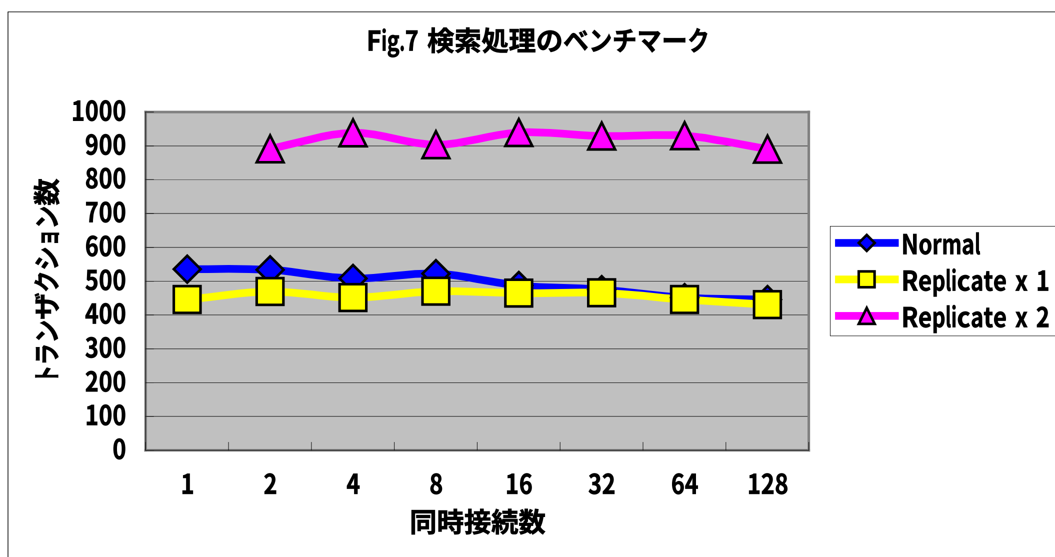
１つは検索処理だけのベンチマーク，もう１つはデータ追加・更新・削除・検索を複合させた総合的な処理のベンチマークです．

6.2.　ベンチマーク結果

双方向レプリケーションのベンチマークは以下の通りです.

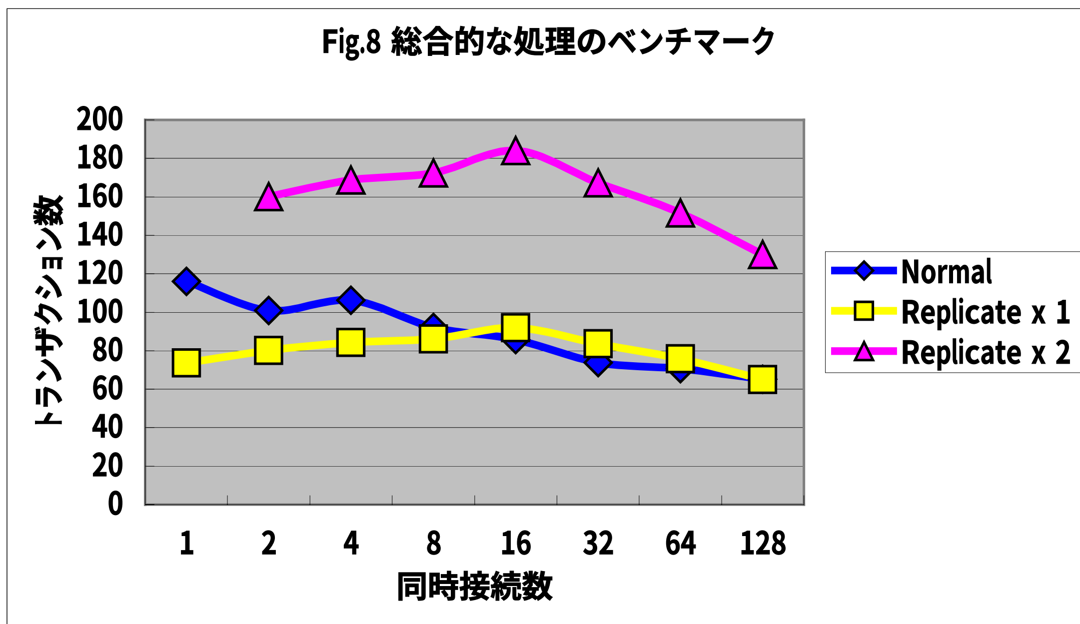
6.2.1.　検索処理のベンチマーク

検索処理のレプリケーションはノーマルの **PostgreSQL** サーバと比較して多少オーバーヘッドはありますが，データベースサーバを **2** 台に増設した場合，ほぼ **2** 倍のトランザクションを処理しました.



6.2.2.　総合的な処理のベンチマーク

検索処理だけでなく，追加，更新，削除クエリーをトランザクションで括った，総合的な処理のベンチマークを計測しました．同時接続数が少ない場合に多少オーバーヘッドは発生しますが，同時接続数が **8** 以上では殆どオーバーヘッドはみられませんでした．データベースサーバを **2** 台に増設した場合，同時接続数が **8** 以上で通常の **PostgreSQL1** 台と比較して，ほぼ **2** 倍のトランザクション数を処理しました.



7.　考察

双方向レプリケーションを利用することで，**PostgreSQL** に負荷分散の機能を付与することができました．これにより，これまで諦めるしかなかった高負荷なシステムにおいて，**PostgreSQL** を利用することが可能になると思います．

今後の課題として，**2** 相コミットを備えた分散トランザクション機能の実装，動的なデータベースサーバの増設およびリカバリー機能の実装，および，レプリケータと連動して高可用性を実現するロードバランサー機能の実装などが考えられます．

2 相コミットは **PostgreSQL** に無い機能ですので，新しく機能追加を行う必要があります．これについては，日本 **PostgreSQL** ユーザー会の分散トランザクション分科会において，実現方式や実装方式の検討が行われており，今年度内に成果報告が行われる予定です．また，高可用性を目的としたロードバランサーとリカバリー機能についても，同じく分散トランザクション分科会において，実現方式の検討が行われ，実装を担当することになりました．

今後これらの課題を実装し，ベータ版のバージョンアップを行いたいと思っています．