PostgreSQL をベースとしたカラムストア機構の実現検討

中村 実† 田原 司睦† 宇治橋 善史† 橋田 拓志† 河場 基行† 原田 リリアン†

†富士通研究所 ICT システム研究所 〒211-8588 川崎市中原区上小田中 4-1-1

E-mail: †{nminoru, tabaru, ujibashi, hashida.takushi, kawaba, harada.lilian}@jp.fujitsu.com

あらまし 近年、トランザクション処理と高速分析処理を兼ね備えた OLXP の必要性が高まっている。我々は OSS の RDBMS である PostgreSQL 9.4 をベースとして、OLXP システムを実現するための設計・開発を行っている。我々のシステムではテーブルに Column Store Index(CSI)と呼ぶ独自インデックスを付与することで列形式の複製テーブルを生成し、スキャン・ソート・集計処理を高速化する。ユーザは行形式テーブル、列形式テーブルの参照を意識する必要がなく、実行時に自動的に選択される。本発表では PostgreSQL ベースの OLXP システム実現に向けた拡張機能について概要を説明する。

キーワード カラムナー，ベクター処理，PostgreSQL，OLXP

1. はじめに

リレーショナルデータベースマネージメントシステム(RDBMS)において、一件あたりのデータ量は比較的小さいが挿入・更新・削除の操作が短期間に大量に発生するオンライントランザクション処理(Online Transaction Processing、OLTP)のワークロードと、すでに蓄積されたデータを広範囲に読み込み複雑な統計処理を行うオンライン解析処理(Online Analytical Processing, OLAP)のワークロードの 2 種類があることが知られている。そして RDBMS の構成方法として、OLTP ワークロードに特化する場合、データストアを行形式で格納すると効率よく処理できることが知られている。一方、OLAP においては蓄積されたデータの中で特定のカラムに注目して分析することが多く、同じカラムを固めて格納する列形式が向いていることが知られている。

しかし OLTP&行形式データストアと OLAP&列形式データストアは相反する性質を持っているため、一つの RDBMS で処理することは困難である。そのため別々のシステムを用意して OLTP システムから OLAP システムへ ETL(Extract Transform Load)ツールなどを用いて一定間隔でデータをロードするという使い方が一般的であった。

しかし近年、ビジネスの世界では OLTP システムの持つデータをリアルタイムに分析できる RDBMS の存在が必要とされるようになってきた。このような OLTP と OLAP が混在したワークロードは OLXP と呼ばれ[4]、OLAP ワークロードを効率的に処理可能なデータベースは OLXP システムと呼ぶことができる。このような OLXP システムは研究レベル・商業レベルの両方で発表されている(SAP HANA[1],Oracle Database 12c[2], HyPer[3])。しかし OLXP システムは、OLTP と OLAP それぞれが分離したシステムと同程度の性能を出すことは困難である。

我々は OSS の RDBMS である PostgreSQL 9.4[5]をベースとした OLXP システムの開発をスタートした[6][7][8]。PostgreSQL を選択した理由は、PostgreSQL が OLTP RDBMS として PowerGres Plus や Symfoware Server V12 などの実績があるためである。我々の OLXP システムは、特に従来と同程度の OLTP 性能を実現した上で、OLAP 性能を向上させることを目標とする。さらに以下のような機能要件を設けて設計を行っている。

1. 既存の行形式テーブルに対するインデックスとして列形式テーブルを実現する。これにより行形式テーブルに対する INSERT/DELETE/UPDATE 操作を捕捉することが可能になる。ただしインデックスとして実装するため、同じデータを行形式と列形式で二重に保持することになる。
2. 列形式テーブルへの反映を非同期に行い、OLTP 処理の負荷が高い時は反映を遅延させることで OLTP 性能の低下を防ぐ。
3. 行形式テーブルに対してアクセスするか列形式テーブルにアクセスするかをコスト評価によって自動的に切り替える。ユーザーはアクセスするテーブルを意識する必要がない。この実現のために列形式テーブルに対するアクセスが PostgreSQL と完全に互換な MVCC を実現する必要がある。
4. 列形式データをベクター処理することでプランツリーの実行に対するオーバーヘッドを削

減し高速化する。
5. マルチコアを利用しクエリーを並列処理する。
6. 列形式テーブルの圧縮することで、使用ディスク量を減らす。また圧縮によりディスクから読み込み時の I/O 量を減らすことで高速化する。

我々はこれらの目標を満たす OLXP システムを PostgreSQL 本体から分離したエクステンションとして実現するように設計した。これを Column Store Index(CSI)と名付けた。

なお現在の設計では、更新を含まない参照のみのクエリーの中でテーブルスキャン、ソート、集約(Aggregation)を高速化の対象とする。テーブル結合(Join)は今後の課題としている。

トランザクション分離レベルは READ COMMITTED と REPEATABLE READ のみカラムストアを用いた高速実行を試み、SERIALIZABLE ではオリジナルの PostgreSQL で動作する。

2. PostgreSQL

PostgreSQL 9.4 は DBMS デーモンプロセス(postmaster)を中心とした複数のプロセスで構成される。クライアントがソケット経由で postmaster へ接続すると、postmaster は自身を fork してバックエンドプロセスを作成する。以降はクライアントとバックエンドプロセス間でセッションを確立し、バックエンドプロセスがクライアントからのクエリーを処理する。PostgreSQL はマルチスレッドに対応しておらず、各セッションのクエリーは一部の処理を除いてシングルプロセスで逐次処理される。

PostgreSQL のデータベースはファイルシステム上に記録される。1 つのテーブルは 1GB 単位で分割されたファイル群として構成される。PostgreSQL はこのファイル群を Relation を呼ぶ。テーブル以外にインデックスやビューも relation として記録される。例えば CREATE INDEX によってインデックスを作成すると、Index Relation が 1 つ新規に作成されることになる。

PostgreSQL は共有メモリ上に共有バッファを持ち、relation を 8KB を単位のブロックでロードして、やはり 8KB 単位のページにマップする。共有バッファは PostgreSQL のプロセス間で共有されている。Relation 上で連続したページが共有バッファ上で連続したページに割り当てられる保証はない。

3. Column Store Index の設計

3.1. 利用方法

我々は CSI を PostgreSQL のインデックスの一つとして実現する。ユーザーが CSI を使う場合は、以下のように CREATE INDEX 文を実行する。

```
CREATE INDEX indexname ON tablename USING csi (a, b, c);
```

PostgreSQL の CREATE INDEX 文が持つ USING 句は nbtree や hash などインデックス内部のアルゴリズムを指定するが、これと並列に"csi"キーワードを追加して CSI をカスタムインデックスとして追加する。ON 句でカラムストアを設けるテーブル名を指定し、その後に ON 句で指定したテーブルの中のカラム名を a, b, c のように列挙する。ここで指定するカラム名のリストはオリジナルの PostgreSQL ではインデックスのソートキーとなるが、CSI では列形式データへ変換するカラムを指定する効果のみでカラムの並び順にも意味はない。

既成のテーブルに対して CREATE INDEX を実行すると、テーブルにすでに格納されているデータを一括でカラムストアに格納する。

いったん CSI インデックスを作成した後に、INSERT/UPDATE 文を実行すると、PostgreSQL はインデックス毎に設定されたコールバック(aminsert)を呼び出す。このコールバックを CSI インデックスでも捕捉する。一方、PostgreSQL はテーブル中のタプルが DELETE された場合、その削除契機をインデックスには通知しない。CSI はこの機能の不足を補うために、タプル削除の契機でもコールバック(amdelete)が呼び出されるように PostgreSQL 本体側に変更を加えている。

3.2. ストレージ構成

カラムストアは追記的なデータ挿入に適しているが、OLTP のように DELETE/UPDATE 文がランダムな位置に発生するワークロードでは効率が低下する。そのため過去列形式 RDBMS では、列形式ストアの前段として Write-Optimized-Storage(WOS)[9]あるいは Delta Store などと呼ばれる行形式格納のキャッシュを設け、短期的な更新は行形式で貯めるのが一般的である。そして WOS へ一定量のデータが貯まった後に非同期なタイミングで列形式に変換しストレージへ反映する。この時、WOS に対して列形式ストアを Read-Optimized-Storage(ROS)と呼ぶ。

CSI でも WOS と ROS の二つのストアを設ける。また CSI が WOS から ROS へ反映することを WOS→ROS 変換と呼ぶことにする。CSI の WOS と ROS を構成のイメージは Figure 1 のようになる。

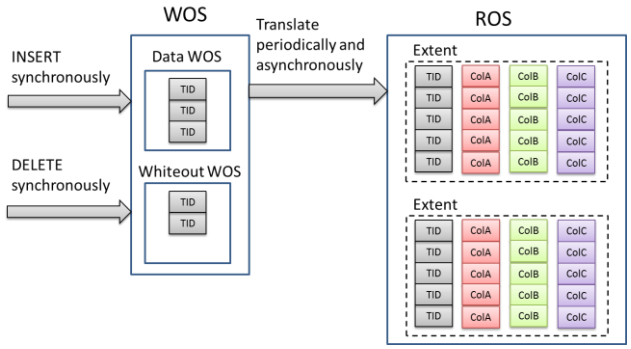

Figure 1 WOS と ROS の構成

WOS と ROS は PostgreSQL の relation としてディスクに記録する。ROS は列ごとに別個の relation を作成して、別々にディスクに書き出す。Relation を利用しているため WOS と ROS は PostgreSQL の他のテーブルと同様に Write-Ahead-Logging(WAL)ログを生成することができ、システムダウン時には自動的にリカバリーされる。

CSI はカスタムインデックスとして機能するが、ROS に格納するデータはソートをせずに追記順に書き込んでいる。ROS に対するインデックスやプロジェクションを追加するのは今後の課題とする。

3.3. Write-Optimized-Store(WOS)

CSI は本来のテーブルとカラムストアを二重に保持するため、WOS はテーブル内の ROS へ反映していない箇所を記録するだけの装置となる。OLTP 性能を低下させないために、WOS はできるだけ処理のオーバーヘッドを小さくする必要がある。

PostgreSQL はテーブル内の行(タプル)の位置を記録する Tuple-ID(TID)と呼ぶ情報を持つ。TID は 32 ビットのページ番号と 16 ビットのページ内アイテム番号が対になった 48 ビットの識別子である。WOS はこの TID だけを記録し、タプル内のカラムデータは保持しない設計とした。これにより INSERT/DELETE に対するオーバーヘッドが最小限になる。

CSI の WOS はタプルの挿入を記録する Data WOS とタプルの削除を記録する Whiteout WOS の二種類を設ける。インデックスへ挿入(aminsert)が行われた場合は、Data WOS 内に挿入した行の TID を追記する。行の削除が行われた場合(amdelete)は、Whiteout WOS 内に削除した行の TID を追記する。WOS→ROS 変換時には、Data WOS と Whiteout WOS の両方に存在する TID は削除し、Data WOS または Whiteout WOS にのみ存在する TID を ROS へ移動する。

3.4. Read-Optimized-Store(ROS)

CSI のカラムストアである ROS は、最大 262,144 行を 1 つにまとめたエクステント(extent)単位で管理する。これは一定量の行をまとめて処理することで ROS の管理オーバーヘッドを減らし、同時に読み出しを効率化するためである。WOS→ROS 変換は、Data WOS が 1 エクステントを構成するために必要な数だけ貯まった時に変換を開始する。WOS には TID のみが記録されているので、ROS へ格納するカラムデータはインデックス元となるテーブルからフェッチする。

エクステントのデータは圧縮が可能な場合には、列単位で圧縮を行う。圧縮方式は固定サイズの列にはランレングス圧縮を、可変サイズの列には辞書圧縮を行う。

また WOS→ROS 変換では、Whiteout WOS にのみ存在する TID に対応する行を ROS から削除する必要がある。しかしエクステントの一部行だけが削除された場合に既存のエクステントを再作成するのはコストが高いので、削除行を示す delete vector を導入する。

3.5. MVCC

CSI は OLXP をサポートするため、WOS→ROS 変換中も含めて、オリジナルの PostgreSQL に準拠した MVCC を実施する[7]。CSI が MVCC を守るためには以下の 2 つの条件を守る必要がある。

1. INSERT したトランザクションがロールバックした場合その行は INSERT されなかったことになる。DELETE したトランザクションがロールバックした場合、その行は DELETE しなかったことになる。
2. WOS→ROS 変換中にカラムストアへの参照をした場合でも、挿入された行が Data WOS と ROS で二重に出現することは許されない。また削除された行の情報が Whiteout WOS から消え delete vector に出現せず削除されてないように見えることも許されない。

PostgreSQL はテーブル内にある各行が生成トランザクション ID(Xmin)と削除トランザクション ID(Xmax)の情報を持っており、テーブルスキャンの際には現在のトランザクション情報と照合して「可視な(visible)」な行だけを読む。これが MVCC 制御である。

CSI は 1 を守るために、Data WOS と Whiteout WOS にはテーブルと同様に Xmin/Xmax の情報を記録する(Figure 2)。その上で WOS→ROS 変換時には、Data WOS 内の TID はトランザクションの可視性をチェックし、TID を挿入したトランザクションがコミット済かつ他

の全てのトランザクションから見て可視状態な TID だけを ROS に移動させる。同様に Whiteout WOS 内の TID も全てのトランザクションから見て可視状態となり削除されたと判定できる場合にのみ ROS の delete vector へ反映する。このため ROS はロールバックが起きても巻き戻す必要のないデータだけを ROS へ移動することができる。

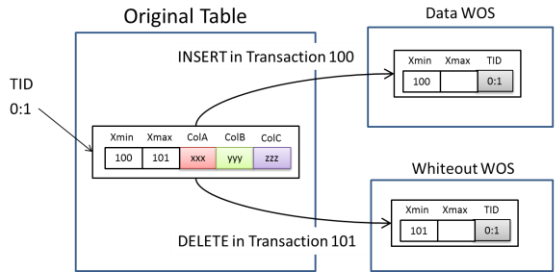

Figure 2 WOS のトランザクション情報

次に 2.を守るため、クエリーの開始時点の WOS と ROS のスナップショットが読めるようにする。WOS は相対的に容量が小さいため、クエリー開始時点に WOS 内の可視な TID の行データをテーブルから読み込み、メモリ上で一時的に列形式データに変換する。我々はこのデータ構造を Local ROS と名付けた。Local ROS の作成と WOS→ROS の変換は排他的に行っており、プラン実行中は Local ROS のみを参照することでクエリー開始時点の WOS のスナップショットを見ることができる。

一方、ROS は巨大なためクエリー実行前にスナップショットを保存することも、クエリー実行中に WOS→ROS 変換を停止させることも実用的ではない。そこでエクステント単位にトランザクション情報を記録することで、二段目の MVCC を導入する。我々はエクステント生成時のトランザクション ID を Xgen、エクステント無効化時のトランザクション ID を Xdel と名付けた。その上で WOS→ROS 変換自体を独立したトランザクションで行い、トランザクション ID をエクステントの Xgen に記録する。また delete vector に記録される削除された行の数が閾値を越えたエクステントは生きている行だけを取り出してエクステントを再作成するが、処理後に不要になった古いエクステントの Xdel にはトランザクション ID を記録する。CSI は ROS を更新した最後のトランザクション ID を記録しおり、クエリー開始時にそのトランザクション ID を読み込みクエリー実行中は「Xgen ≦クエリー開始時点で最後に ROS を更新したトランザクション ID ＜ Xdel」が成立するエクステントのみをフェッチすることで、クエリー開始時の ROS のスナップショットを読むことが可能になる。

3.6. プランの書き換え

CSI はあるクエリーがカラムストアを用いてアクセスした方が高速であるとコスト評価した場合に限り、プランを書き換える。プランの書き換えには PostgreSQL 9.5 から導入される Custom Scan API をバックポートして使用している[10]。

プランの書き換えの対象とするのは、SeqScan ノード、Agg ノード、Sort ノードとする。これらを Custom Scan API を派生させた CSI Scan ノード、CSI Agg ノード、CSI Sort ノードに置換する。

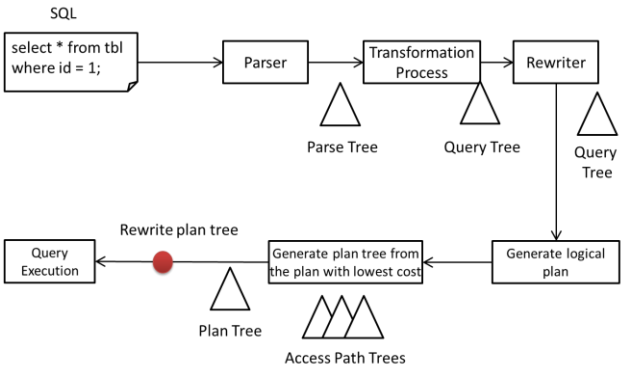

Figure 3 プランの書き換え位置

PostgreSQL は planner のフェーズで複数のアクセスパスを提案し、最もコストが小さいものを採用する。この際、インデックスごとにアクセスパスが生成されるので、CSI インデックスに対してもアクセスパスを提出する機会がある。ただし CSI は pg_hint_plan などのプランを書き換える他のエクステンションと併用することを想定している。アクセスパスを使ったプランの書き換えは他のエクステンションと干渉する危険性がある。そこで CSI は常に最悪値となるコストを提出してアクセスパスとしての採用を回避し、PostgreSQL 本来の planner にプランを確定させる。そのプランツリー中に SeqScan ノード、Agg ノード、Sort ノードがあれば、再度コスト評価を行い CSI ノードへの書き換えを実施する(Figure 3)。

3.7. プラン実行

書き換えたプランは ROS と WOS を組み合わせたデータセットを参照して実行する必要がある。そのためにプラン実行に先立って Local ROS を作成する。Local ROS は Data WOS に登録された TID の行データをテーブルから参照し、ROS と同様の列形式データに変換する。また Whiteout WOS 内の TID も読み込み、ソートしてメモリ上に削除リストとして記録する。ROS から

読み込んだデータは、delete vector とソート済み削除リストをマージして削除された行を決定する(Figure 4)。

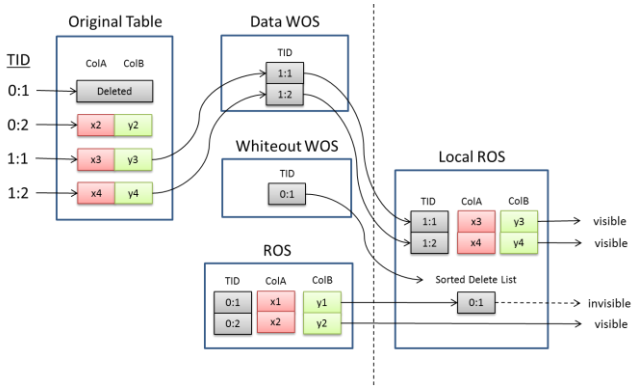

Figure 4 プラン実行のカラムストアスキャンのイメージ

Local ROS を作成することで CSI の実行エンジンは列形式のみに対応すればよい。しかし Local ROS をクエリー毎に作成し、毎回破棄しているのはオーバーヘッドがかかる。ただしこのオーバーヘッドは、WOS→ROS 変換が定期的に行われ WOS は一定量に抑えられるので、実際の運用では無視できると考えられる。

3.8. ベクター処理

PostgreSQL の各プランは選択リストや WHERE 句を処理するために expression tree を持つ。プラン実行時にはこの expression tree を走査しながら処理することで演算を行う。例えば(A* 3) + B という句に対して PostgreSQL は Figure 5 のような expression tree を作成する。

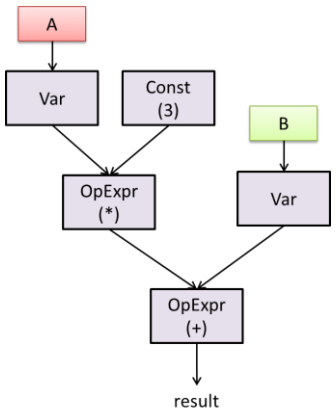

Figure 5 PostgreSQL の expression tree

しかしこのような expression tree の走査は条件分岐ミスを大量に発生させ、性能低下の要因となっている。そこで CSI ではカラムストアから 128 行単位でデータを読み込み、これをベクター(vector)とし、expression tree の各ノードは 1 行単位ではなくベクター単位で処理を実行する(Figure 6)。

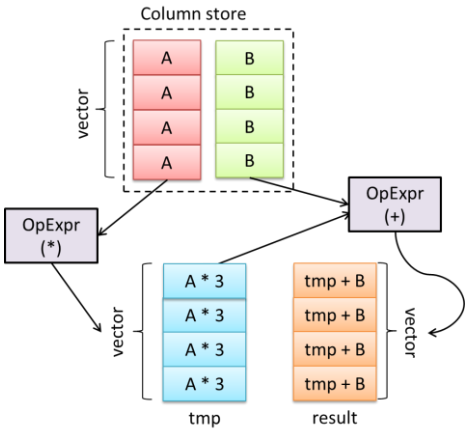

Figure 6 ベクター処理のイメージ

3.9. 並列実行

CSI は可能な場合にはクエリーの並列実行を実現する。これには PostgreSQL 9.4 から導入された Dynamic Background Worker を利用して行う。

並列実行は、カラムストアをエクステント単位に分割し、それぞれの並列ワーカーが別々のエクステントを処理する。並列ワーカーが行う処理のアルゴリズムは、元のプランで選択されたアルゴリズムと同じものを利用する。例えばハッシュ・アグリゲーションが選択された場合は、並列ワーカーはハッシュ・アグリゲーションを行い、バックエンドプロセスは結果を統合して上位プランノードに返却する。

並列実行には並列ワーカー間のデータ共有が必要となる。PostgreSQL 9.4 は Dynamic Background Worker と伴に Dynamic Shared Memory(DSM)という共有メモリ機構が導入された。しかし DSM は同一のメモリ実体を共有する機構であっても、プロセス間ではマッピングする位置が異なり、ポインタを含んだデータをそのまま転送することができない。

そこで我々は DSM に替わる機構として、Shared Memory Context(SMC)機構を開発した[8]。SMC はバックエンドプロセスと並列処理を行う Dynamic Background Worker 間で共有される共有メモリだが、関連するプロセス間で同一のメモリアドレスにマッピングする。

また SMC は PostgreSQL の Memory Context 機構のインターフェイスと合致させている。このため SMC は既存の PostgreSQL のルーチンをそのまま呼び出し、ルーチン内で新規に割り付けられるメモリを共有メモリに割り当てることが可能になっている。このため

PostgreSQL の提供するプランツリーやハッシュテーブルなどをそのまま共有することができる(Figure 7)。

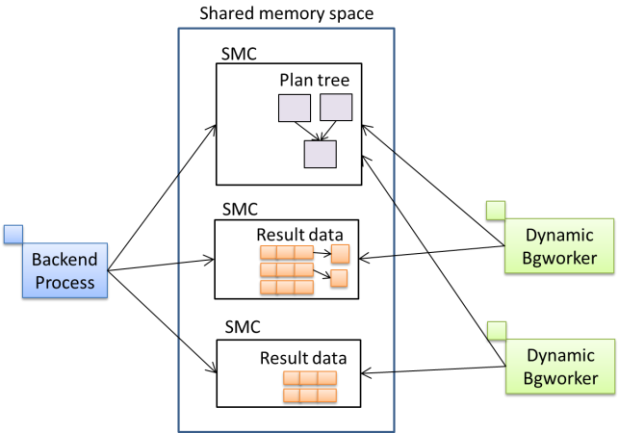

Figure 7 SMC の利用イメージ

4. まとめ

PostgreSQL をベースに OLTP 処理を低下させずに OLAP 処理を高速化する OLXP システムを開発するため、列形式格納のインデックスを実現する CSI の設計・実装を進めている。本稿では CSI の基本な設計を述べた。

今後は実装を進め OLAP の性能向上率、OLTP と OLAP 混在時の性能の評価を実施する。またテーブル結合についても設計・実装を行う。

参 考 文 献

[1] V. Sikka, F. Färber, W. Lehner, S. K. Cha, T. Peh, and C. Bornhövd, "Efficient Transaction Processing in SAP HANA Database: The End of a Column Store Myth", Proc. SIGMOD, 2012.

[2] Oracle Database In-Memory, Oracle, http://www.oracle.com/technetwork/database/in-memory/overview/twp-oracle-database-in-memory-2245633.html

[3] Florian Funke, Alfons Kemper, and Thomas Neumann, "Compacting Transactional Data in Hybrid OLTP&OLAP Database", Proc. VLDB, 2012.

[4] Hasso Plattner,"The Impact of Columnar In-Memory Databases on Enterprise Systems", Proc. VLDB, 2014.

[5] PostgreSQL, http://www.postgresql.org/

[6] 田原 司睦他,カラムナデータをサポートするための PostgreSQL 拡張,第 77 回日本情報処理学会全国大会

[7] 橋田 拓志他,PostgreSQL ベースのカラムナ機構へのトランザクション実現検討,第 77 回日本情報処理学会全国大会

[8] 宇治橋 善史他,PostgreSQL ベースの並列処理向けの共有メモリ機構の設計,第 77 回日本情報処理学会全国大会

[9] Mike Stonebraker, et al., "C-store: a column-oriented DBMS", Proc. VLDB, 2005.

[10] 海外 浩平氏のカスタムプランのパッチ, http://www.postgresql.org/message-id/9A28C8860F777E439AA12E8AEA7694F8F8F6BA@BPXM15GP.gisp.nec.co.jp