バッファキャッシュ効率化の PostgreSQL への適用

板垣 貴裕　寺本 純司　土川 卓哉　芳西 崇

日本電信電話株式会社 NTT サイバースペース研究所

〒239-0847 神奈川県横須賀市光の丘 1-1

E-mail: {itagaki.takahiro, teramoto.junji, tsuchikawa.takuya, honishi.takashi}@lab.ntt.co.jp

あらまし 近年、サーバが搭載可能なメモリは大容量化しており、RDBMS の性能に占めるバッファキャッシュの影響が大きくなっている。本稿では、バッファキャッシュの効率化に関する手法について検討し、オープンソース DBMS である PostgreSQL に対して適用を行った場合の性能評価の結果について報告する。キャッシュの利用効率を向上させ、I/O 回数の削減するためには、データベースのホットスポットを割り当てられたキャッシュに見合うサイズに調整することが有効であった。さらに、その最適化された状態を維持するために、恒常的なガベージコレクションについての検討を行った。これらを適用した PostgreSQL について、TPC-C (DBT-2) 300WH の負荷での性能評価を行ったところ、適用前の状態に比べ、約 2 倍のスループット向上を確認できた。

キーワード 性能評価, RDBMS, PostgreSQL

1. はじめに

近年、データベースサーバが搭載可能なメモリは大容量化している。そのため、ディスク I/O が性能的なボトルネックとなるシステムであっても、メモリによるバッファキャッシュによって I/O 負荷を軽減することで、安価に高性能が得られるようになっている。

このバッファキャッシュの効果をより高めるためには、データベース全体の中でアクセス頻度の高いデータ部分、すなわち、ホットスポット領域の管理が重要になる。要求されたデータがキャッシュされている場合には、メモリへのアクセスだけで処理は完了し、実際の I/O は発生しない。さらに、複数の書き出し要求をメモリ上で一つにまとめられることで、I/O 回数の削減が期待できる。

本稿では、バッファキャッシュの効果を高める手法を、オープンソースの DBMS である PostgreSQL へ適用する。近年のシステム開発では、構築コストの削減などを目指し、OSS の利用が拡大しており、より重要で、信頼性が求められる分野にも進出しつつある。そのような分野の中には、OLTP (online transaction processing) 用途があり、複数の端末から要求される膨大な数のトランザクションを短時間で完了することを要求される。そこでは、膨大なデータアクセスを効率よく処理する必要がある。発行される大量のディスク I/O をバッファキャッシュによって負荷を軽減し、PostgreSQL の性能を改善する手法について提案する。そして、OLTP の評価モデルである TPC-C での有効性を示す。

2. バッファキャッシュの効率化の方針

一般的な OLTP では、大容量のデータ領域に対してアクセス要求がある。そのため、データを参照するためにディスクへの I/O が必要となり、データベースの性能は、この I/O 性能に大きな影響を受ける。

バッファ管理の方針としては、大きく分けて、要求回数の多い領域をより優先してキャッシュすることと、キャッシュすべきデータそのものをコンパクト化し、数多くのデータをキャッシュすることの 2 つに大別できる。また、ホットスポットのサイズを最適に保つために、ガベージ領域の増加への対策が必要となる。

そこで、バッファキャッシュの効率の改善として、PostgreSQL への適合性を考慮し、下記の 5 つを検討した。

1. 不必要な領域のキャッシング抑制
2. キャッシュ可能なデータ数の増加
3. キャッシュすべきデータ領域の絞込み
4. ガベージ領域の増加抑制
5. ガベージ領域の早期回収

これら 5 つの着目点について、4 つの具体的な改善を行った。3 章では、不要な領域であるログのキャッシングを抑制する手法について提案する。4 章では、データベース全体の容量を削減し、キャッシュ可能なデータを増加させる、データの高密度化について提案する。5 章では、キャッシュすべきデータ領域をまとめ、さらに処理によって生じるガベージ領域の増加を抑える垂直パーティショニングについて提案する。6 章では、発生したガベージ領域を回収し、メモリの圧迫を軽減するための継続的なガベージコレクションについて検討する。7, 8 章で実験をし、結果を考察する。

3. ログキャッシングの抑制

メモリによるデータのキャッシングは、データベースと OS の両者によって行われる。この章では、データベースにとって不必要な領域のキャッシングを避けるために行った、OS キャッシュを経由しないログの書き出しについて説明する。

3.1. WAL とは

トランザクションがコミットされた場合、たとえ異常終了した場合でも、データベースはその結果を失ってはならない。結果を保証しながら性能を高めるため PostgreSQL は、WAL(Write Ahead Log)という機構を持っている。これは、データの書き出しに先立って、変更の手順をディスクに書き出すことにより、実際に変更のあったデータ本体を書き出す必要性を減らす工夫である。突然の電源断などで異常終了した場合にも、WAL 中の操作履歴から、コミット済みの変更を復元できる。

データベースが正常に稼動している間、WAL は書き出されるのみであり、書かれた WAL が再び読み込まれることはない。読み込みが発生するのは、異常終了後の復元作業中のみである。データベースが正常に稼動している間は、WAL は二度とアクセスされないため、キャッシングは不要である。

3.2. Direct I/O によるキャッシング抑制

データベースの外部に位置する OS には、WAL の書き出しと通常データの書き出しの、どちらが要求されているかを判断することは難しい。そのため、本来不要である WAL 領域をキャッシュしてしまう動作が確認できた。

OS にログをキャッシュさせないためには、OS のページキャッシュを経由しない書き出しが必要となる。これを、Linux, FreeBSD, Windows などの OS が持つ Direct I/O と呼ばれる機能によって実現した。

Direct I/O 自体は、WAL ファイルの open() 呼び出し時に、O_DIRECT フラグを指定することで利用できる。ただし、Direct I/O の制限として、書き出すバッファの位置、長さ、書き出し先のファイル内でのオフセットは、OS ごとに制限がある。典型的には、512byte～4KB 程度の 2 の累乗であることが多い。この制限を満たすため、WAL 用のバッファを PostgreSQL のページサイズ(4,8,16,32KB/標準 8KB)境界に整列するように確保する変更を加えた。

4. データの高密度化

PostgreSQL は、可変長データを扱いやすい構造になっている。しかし、そのために必要な構造が、OLTP のような単純な構成では、オーバースペックとなってしまい、性能上のデメリットになっている。

この章では、値とインデックスの表現をコンパクトなものに変更し、データベースファイルサイズを縮小する試みについて説明する。

4.1. 固定長文字列

PostgreSQL の固定長文字列は、可変長文字列に長さの制約を追加したものになっている。このため、固定長であっても、文字列の長さを保持する領域を確保してしまう。

そこで、冗長な長さ情報を持たない固定長文字列を作成し、PostgreSQL の拡張モジュールとして追加した。

ただし、本稿の固定長文字列実装では、SQL 標準に準拠していない。これは、SQL 標準では、テーブル定義の際の“CHAR(長さ)”という書式により、固定長文字列型の最大長さを指定する。一方、PostgreSQL では、ユーザ定義型に対して追加パラメータを渡すことがサポートされていない。このため、本稿では、必要な長さごとに、スクリプトを用いて固定長文字列を定義するソースコードを自動生成するアプローチを取った。

SQL 標準に準拠するためには、ユーザ定義型に対して追加パラメータを渡す機能が必要であるが、これについては今後の課題とする。

4.2. 可変長文字列

可変長文字列は、文字列一つ一つで長さが異なるため、長さ情報を文字列型内部に保持する必要がある。PostgreSQL は、この長さ情報のために、常に 4byte の領域を確保していた。

一般的な OLTP では短い長さの文字列を多用する。このため、長さを表すための領域のオーバーヘッドを無視できない。例えば、TPC-C の場合は、20 文字程度の可変長文字列を使用するが、この場合には、長さフィールドのオーバーヘッドが文字列型全体の 20%に及ぶ。

これを改善するため、より効率的な表現で可変長の文字列を保持できるデータ型を提案する。データの長さによって長さを保持するフィールド長そのものを可変とすることで、格納効率を高めることを試みた。表 1 のように、ヘッダ先頭の数ビットで後続のデータの長さに応じた保持形式を表す。ヘッダの大きさは、127 byte までは 1 byte、16KB までは 2 byte となる。多用される短い可変長データの範囲において、より無駄のない格納が可能になる。利点として、長さ 0 から 4GB まで、シームレスに扱えることと、上位の数ビットを見て長さ判定処理をディスパッチできることが挙げられる。

表 1：可変長型ヘッダ表現

格納状態	ヘッダ表現	ヘッダ [byte]	データ [bit]
127 byte	0*******	1	7
16 KB	10****** ********	2	14
4 GB	110----- [4byte]	5	32

4.3. インデックスの隙間除去

PostgreSQL のインデックスのページ構造を図 2(左)に示す。インデックスタプルは、常にその環境の最大配置境界(典型的には 8byte)に整列されていた。

最大配置境界に整列する理由は、CPU の制限のためである。一般的な CPU で数値を扱う場合、メモリ上の適切な境界に配置されなければならない。例えば、32bit 整数は 4byte 境界、64bit 実数は 8byte 境界などである。配置制限を確実に満たすため、PostgreSQL は、常にタプルをその環境の最大配置境界に並べていた。

しかし、本来は、インデックスのキーによっては、より制限の緩い境界に配置することができる。例えば、キーとして 32bit 整数が使われた場合には、ヘッダの配置境界 2byte と、キーの境界 4byte の、より制限の厳しい境界、すなわち 4byte 境界に配置すればよい。

配置制限を緩和することで、図 2(右)のような構造を取れ、インデックスタプル間の隙間を無くすことができる。これは、特にキー長が短い場合に効果が大きくなり、インデックスサイズを最大 20%削減できる。

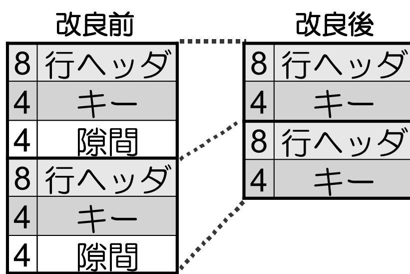


図 2：インデックスタプル配置

5. 垂直パーティショニング

パーティショニングとは、テーブルを物理的な複数のテーブル断片（パーティション）に格納する方法である。パーティショニング方式のひとつである、垂直パーティショニングを用いてデータ配置を最適化することで性能向上を目指した。

まず、パーティショニングの概論に触れた後、PostgreSQL に適用した場合の 2 つの効果について述べる。そして、垂直パーティショニングの実装方法について説明する。

5.1. パーティショニングの利点

パーティショニングでは、データが物理的に格納される領域は分割されているが、論理的にはひとつのテーブルとして扱える。そのため、アプリケーションは分割のされ方を意識する必要は無く、透過的に参照できる。さらに、特定のパーティションにのみ含まれるデータを参照する場合には、データベースが自動的に検索範囲を絞り込むことで、効率良く処理される場合もある。

一方、明示的に、特定のパーティションのみを操作することもできる。分割した際の条件の範囲に対して、一括削除やメンテナンス作業を行えることが利点である。

パーティショニングには、大きく分けて、表を行単位で分割する水平パーティショニングと、列単位で分割する垂直パーティショニングがあるが、本稿では垂直パーティショングを扱う。

5.2. 効果 1：キャッシングの効率化

利用頻度の異なる複数の列が存在する場合、アクセス頻度の高い列を主表に格納し、残りを副表に分離することで、主表をオンメモリに留めることができ、主表のみにアクセスする場合の I/O を削減できる。

分割した主表と副表の両方共がメモリ上に無い場合は、通常 1 度で済むディスク読み込みが 2 度に増えるため、性能は劣化する。しかし、主表がメモリに乗り切ってしまう場合には、副表の読み込みしか発生しないため、このデメリットは生じなくなる。垂直パーティショニングのキャッシュ率向上効果を活かすためには、列毎のアクセス頻度を考慮する必要がある。

5.3. 効果 2：更新量の削減

PostgreSQL が追記型アーキテクチャであることに関連し、垂直パーティショニングによって更新時のデータサイズの増加を減少させることができる。

PostgreSQL では、ある行が削除された場合には、行そのものは実際には削除されず、「削除された」という印だけがつけられる。また、更新は、削除と挿入の組み合わせとして実現されている。このしくみにより、MVCC (Multi-Version Concurrency Control: 多版型同時実行制御)を実現し、更新と参照が競合しない。これにより、マルチユーザ環境における並列性能を向上させている[1] 。

更新または削除により生じた古いバージョンの行は、しばらくすると完全に不要になり、以降アクセスされなくなる。このガベージの増加が、性能低下の原因となる。

垂直パーティショニングによって行を分割するこ

とで、ひとつの行内の一部のフィールドのみが更新される場合に、変更の無いフィールドを複製せずに済む場合がある。特に、更新頻度が低く、サイズの大きなフィールドを副表に分離した場合に有効に動作する。（図 3）

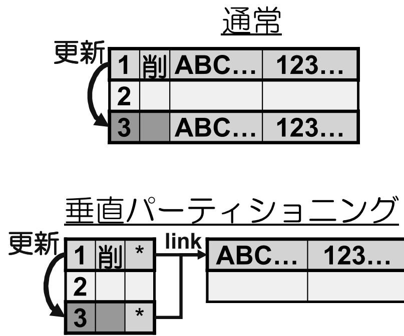


図 3： ガベージ領域増加の抑制

5.4. 実装

テーブルを列で切り離し、主表と副表に分離するため、主表と副表の行の対応を取る必要がある。そのための方式として、以下の 3 つについて検討を行った。

1. 共通 PK(Primary Key)型:
   主表と副表に共通の PK を持たせる。
2. 追加 PK 型:
   副表に PK を追加し、主表から参照する。
3. 直接参照型:
   主表が副表の物理位置を持つ。

1 は扱いやすいが、PK の重複が発生するため、PK のサイズが大きな場合にオーバーヘッドが生じる。

2 は、キー長の短い新たな PK を副表に定義できるため、1 にあった冗長性がなくなり、インデックスが縮小される。しかし、必ず主表経由で副表にアクセスしなければならない。

3 は、機能的には 2 と同等だが、副表のためのインデックスが不要になり、これら 3 つの中では最もオーバーヘッドが小さい。ただし、副表を更新する場合には、物理位置が変化するため、同時に主表も変更する必要がある。

どの方式を選択すべきかについては、ワークロードに依存する。本稿の実験では、副表の更新頻度を考慮し、更新がある場合には 2 を、無い場合には 3 を適用した。

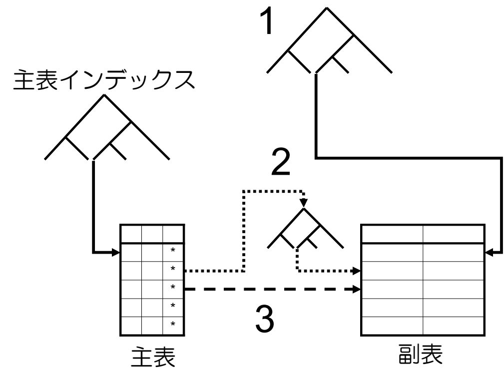


図 4： 垂直パーティショニングの 3 方式

6. 常時稼動型メンテナンス

前述のように、PostgreSQL は追記型アーキテクチャである。追記のために発生したガベージ領域をより効率よく再利用するため Background Vacuum を作成した。

この章では、まず、追記によって生じたガベージを再利用するための Vacuum コマンドについて説明する。次に、ある行が不要だと判断されるまでの処理の流れについて整理する。そして、ガベージ処理を行うモジュールとして着目した Background Writer プロセスの説明を行う。そして、Background Writer へガベージ処理機能を追加した手順について説明する。

6.1. Vacuum コマンド

PostgreSQL には、ガベージを回収し、再利用するための Vacuum コマンドがある。不要になった行は再利用可能であると認識され、後の挿入または更新の際に使われる。定期的な Vacuum は、データベースの膨張を防止し、性能を維持する上で重要なメンテナンスである。

Vacuum は、ガベージが発生した直後ではなく、深夜などの負荷の低い時間帯にまとめて実行するように設計されている。これは、Vacuum のリソース消費による、処理本体への悪影響を避けるためである。

しかし、ガベージの発生から回収まで遅れがあるため、常時いくらかのガベージが残ることは避けられない。データベースを常には最良の状態に保つことはできず、特に高負荷の場合には、少量のガベージでも影響は無視できない。

また、ディスクに比べ、CPU の処理能力は大幅に向上している。従来は、オンライン中のメンテナンス処理は、処理本体の妨げになる恐れがあったが、現在のハードウェアにおいては、ストレージへの付加に比較して、CPU には余力がある。そのため、オンラインでメンテナンスを行うことによるガベージ削減効果は、

それに必要な CPU 負荷の増加を上回る利点があると考えられる。

6.2. ガベージが発生する条件

ある行が削除または更新された場合には、元の行の古いバージョンという扱いになる。すぐに不要となるわけではなく、しばらくの間は、他のトランザクションの読み取り一貫性のために必要になる。本当に不要になるのは、変更を行ったトランザクションがコミットし、そのコミットよりも世代の古いトランザクションがすべて終了した時点である。

OLTP のような軽量トランザクションが連続する場合には、古い行がガベージになるまでの時間は、Vacuum コマンドが実行される一般的な間隔よりも遥かに短い。ガベージの定常的な量を少なく抑えるためには頻繁なガベージ回収が必要であるが、Vacuum コマンドはテーブル単位でしか処理を行えない。追加の処理コストを抑えながらガベージ回収までの時間を早めるためには、Vacuum コマンドよりも軽量のガベージ処理機能が必要であった。

6.3. Background Writer とは

Background Writer (以下、bgwriter)は、バッファ管理機構を構成するモジュールの一つである。ガベージ処理との直接的な関連は無いが、Vacuum すべきページの対象と、bgwriter の処理の対象がほぼ一致するという特徴がある。

トランザクション処理中に新たな読み込みが発生すると、ページを使い切っていた場合には、使用頻度が低いページを破棄することで、必要な空きページを確保する。このとき、置き換えの対象になったページが更新されていた場合には、破棄の前に、書き出しを行う必要がある。bgwriter は、まもなく置き換え対象になる更新済み(dirty)のページを予め書き出しておく。ページの置き換えの際に書き出しが不要になることで、本体処理のメモリ確保がスムースに行われるようになる。

6.4. 提案手法 :Background Vacuum

Vacuum の実行間隔をできるかぎり短くするため、bgwriter にガベージ回収機能を追加する手法を実装した。

Vacuum の実行間隔を短くする際に注意すべきは、Vacuum による新たな I/O の発生を避けることである。そのためには、オンメモリのページのみを処理すべきである。また、計算負荷を抑えるため、全バッファのスキャンは避けたい。さらに、処理本体との競合を避けるため、別の処理がアクセスしていないページを処理すべきである。

上記を満足するページは、bgwriter が処理の対象になるページと一致する。この方式は、PostgreSQL 開発メーリングリストにて議論されたが[2]、実現には至っていなかった。本稿では、これを具体化した。

ページ単位の Vacuum を行うにあたって、ロックに注意する必要がある。PostgreSQL には、あるページのヘッダを対象とする共有／排他ロックと、そのページの参照プロセスを表す pin 数という概念がある。排他ロックを取得した上で、pin 数が 1 である状態を、完全排他ロック(Super Exclusive Lock)と呼ぶ。ガベージを処理するために、そのページ上のタプルを移動するためには、この完全排他ロックを取得していなければならない。Background Vacuum においても、このロックのルールは守らなければならないが、トランザクション本体の処理と bgwriter としての処理の邪魔にならないようにするため、ガベージ処理のためのロック取得は、楽観制御を行った。完全排他ロックを取得できる可能性がある場合にのみロックを試み、完全排他ロックが取得できている場合にのみ、ガベージ処理を行う。ただし、前述のように、bgwriter は、他から参照されていないページを処理の対象にすることが多いため、完全排他ロックを取得できる確率は非常に高い。

ページのガベージ処理に成功した場合は、Free Space Map への登録を試みる。Free Space Map とは、空き領域のあるページを記録しておくシステム領域である。

最後に、bgwriter の書き出しページ数の調整機能が加わる。従来、bgwriter の書き出しが間に合わなかった場合には、トランザクションを処理中の各 backend プロセスが自ら書き出しを行っていた。本稿の Background Vacuum においては、トランザクション処理本体への悪影響を避けるため、backend プロセスの書き出し時にはガベージ処理を行わない。Vacuum の効果を高めるためには、ページの書き出しを可能な限り bgwriter が行う必要がある。そのために、単位時間当たりに書き出すページ数を自動調整する機能を加えた。

6.5. Background Vacuum の処理の流れ

1. bgwriter が書き出す対象バッファの参照カウントがゼロの場合、排他ロックの取得を試みる。参照カウントがゼロでないか、ロックが取得できなければ、共有ロックを取得できるまで待機し、通常の書き出しを行う。
2. 排他ロックを取得できた場合、もう一度参照回数を確認する。完全排他ロック状態で無かった場合は、この書き出しでの Vacuum を諦める。排他ロックを共有ロックにダウングレードし、通常の書き出しを行う。

3. 完全排他ロック状態であった場合は、ページ単位での sweep を行う。現行のすべてのトランザクションから不可視のタプルが見つかった場合には、その領域を切り詰める。
4. 再利用可能か否かによらず、このタイミングで排他ロックを共有ロックにダウングレードする。
5. 再利用可能な領域が見つかった場合には、そのページを Free Space Map に登録する。Free Space Map は予めページ ID でソートされているため、バイナリサーチで検索し、見つかれば更新。見つからなければ、その前後の一定の範囲内から最も空き容量の少ないページを探し、置き換える。
6. 書き出しを行う。
7. 共有ロックを解放する。

7. 実験

本稿で提案した手法について、TPC-C(DBT2) 300WH の負荷にて、性能計測を行った。

「ログキャッシングの抑制」「常時稼動型メンテナンス」は、PostgreSQL へパッチを適用した。また、「データの高密度化」は、PostgreSQL へのパッチの他、DBT2 のテーブル定義の書き換えを伴った。

一方、「垂直パーティショニング」は、トレードオフのある手法である。そのため、適用すべき範囲はアプリケーションに依存する。本実験において適用した箇所については後述する。

7.1. TPC-C / DBT2

TPC-C(Transaction Processing Performance Council Benchmark C)は、卸売りのトランザクション処理をシミュレートした OLTP ベンチマークである。トランザクションの 92%が更新系であり、軽量であるが、大量の処理を同時平行的に行う必要がある。

本稿では、TPC-C のオープンソースの実装の一つである DBT-2 ワークロードツール[3]に変更を加え、実験を行った。また、相対比較のために MySQL の計測も行った。

7.2. 測定環境

測定環境を表 1 に示す。表中にあるデータベースごとのパラメータ値は、PostgreSQL8.0, 8.1, MySQL5.0 それぞれについて、主要なチューニングパラメータを変更しながら予備測定を行い、最もスループットの高かった組み合わせである。次章では、表に示したパラメータを用いた場合の結果について示す。

表 1: 測定環境

<table>
<tr><td rowspan="3">サーバ</td><td>CPU</td><td colspan="2">4×Opteron 848 (2.2GHz)</td></tr>
<tr><td>メモリ</td><td colspan="2">8GB (DDR-400)</td></tr>
<tr><td>I/Oアダプタ</td><td colspan="2">LSI20320 (SCSI 320MB/s)</td></tr>
<tr><td rowspan="2">ストレージ</td><td>キャッシュ</td><td colspan="2">1GB (write-back)</td></tr>
<tr><td>ディスク</td><td colspan="2">(8+8)×S-ATA (RAID 1+0)</td></tr>
<tr><td rowspan="2">OS</td><td>カーネル</td><td colspan="2">RHEL AS V4U1 (2.6.9-11)</td></tr>
<tr><td>file system</td><td>XFS</td><td>ext3</td></tr>
<tr><td rowspan="5">DB</td><td>種類</td><td>PostgreSQL</td><td>MySQL (InnoDB)</td></tr>
<tr><td>バージョン</td><td>8.0.4 / 8.1.0</td><td>5.0.15</td></tr>
<tr><td>割当メモリ</td><td>512MB</td><td>256MB (fdatasync)</td></tr>
<tr><td>checkpoint</td><td>13分 間隔</td><td>256MB 間隔</td></tr>
<tr><td>接続数</td><td>30</td><td>60</td></tr>
<tr><td rowspan="2">DBT-2</td><td>負荷</td><td colspan="2">300WH</td></tr>
<tr><td>計測時間</td><td colspan="2">準備+計測 = 30分+30分</td></tr>
</table>

(RHEL : Red hat Enterprise Linux AS Version 4 Update 1)

7.3. 垂直パーティショニングの適用判断

垂直パーティショニングは、customer, stock の 2 つのテーブルに対して使用した。

以下の期待値とは、論理的な 1 行を更新した場合に増えるガベージの量である。垂直パーティショニングを使わない場合は、常に元表のサイズになる。

表 2: 垂直パーティショニングの効果

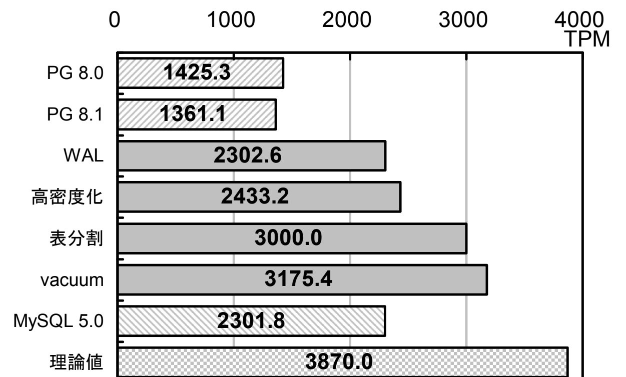
テーブル	元表 [byte/行]	主表 [byte/行]	副表 [byte/行]	期待値 [byte]
customer	600	300	350	335
stock	320	50	300	50

8. 結果と考察

8.1. スループット改善結果

PostgreSQL8.1 に対して、最終的には 233%のスループットを達成した。グラフ 1 は、上から順に、それぞれの改善の累積効果である。

提案方式すべてでスループットの改善が見られたが、特に WAL と表分割の効果が顕著であった。



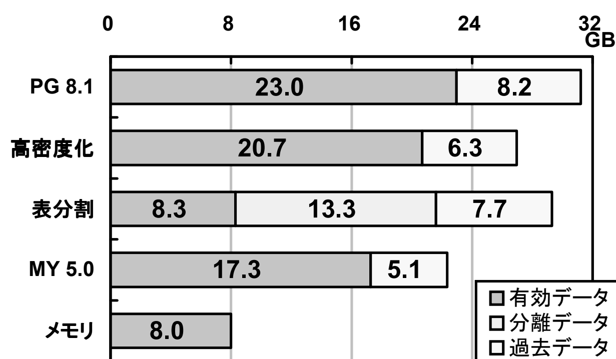
グラフ 1: スループット（累積効果）

8.2. データベースサイズ削減結果

高密度化により、データベース際の削減効果が見られた (グラフ 2)。型の高密度化では、new_order 以外のすべてのテーブルで、データ量の削減効果があった。インデックスの高密度化では、効果があったインデックスのみを以下に示す。それぞれ 4byte のキーを持つインデックスである。

垂直パーティショニング（表分割）では、全体のサイズは増加しているものの、アクセス頻度の高い領域と低い領域を切り離すことができており、これがスループットの向上に結びついたと考えられる。

また、MySQL のデータベースサイズは、非常にコンパクトである。この理由は、MySQL は Primary Key を基準にするクラスタインデックスが用いられているからだと考えられる。一方、PostgreSQL では、Primary Key と通常のインデックスの区別はなく、順序を持たないテーブルに対する外部インデックスとして実装されている。そのため、Primary Key の B-tree インデックスのリーフに相当するデータの重複が MySQL には無く、これがサイズの差の理由と考えられる。



グラフ 2： データベースサイズ（累積効果）

表 3： 高密度化(型)の効果

(new_order 以外でサイズの削減が見られた)

テーブル	適用前 [byte/行]	適用後 [byte/行]	行数 [行/WH]	差 [/300WH]
warehouse	120	104	1	5KB
district	128	104	10	72KB
customer	720	680	30,000	360MB
history	56	48	21,000	50MB
new_order	8	8	9,000	0
orders	32	24	30,000	72MB
order_line	64	56	300,000	720MB
item	96	88	100,000	240MB
stock	352	312	100,000	1.2GB

表 4： 高密度化(インデックス)の効果

(キー長が 4byte 以下のインデックスに適用)

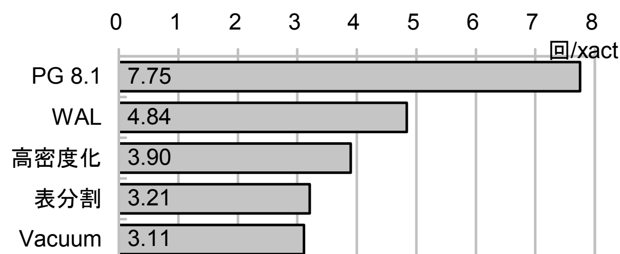
インデックス	行数 [行/WH]	差 [/300WH]
customer primary key	30,000	36MB
orders secondary index	30,000	36MB
stock primary key	100,000	120MB

8.3. I/O 削減効果

Read に関しては、特に WAL への変更で大きな減少が見られる。OS のキャッシュが WAL によって消費されていたと考えられる。それ以降の改造も、Read の減少に結びついている。

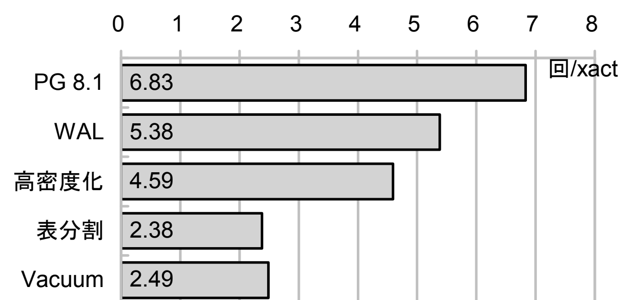
一方、Write は、特に表分割の効果が大きい。垂直パーティショニングによって、アクセス頻度が高い部分がメモリに留まり続けることができ、ページ入れ替えによる書き出しが減少した効果と、更新量の削減効果の両者によって Write が減少したと考えられる。

また、vacuum でも性能向上効果が見られた。これは、ガベージ領域の再利用までの期間が短縮され、ガベージを入出力するために I/O 帯域が消費されることが減少したためと考えられる。



グラフ 3: Read 回数（累積効果）

(WAL Direct I/O による削減効果が高い)



グラフ 4: Write 回数（累積効果）

(垂直パーティショニングによる削減効果が高い)

8.4. 性能低下抑制効果

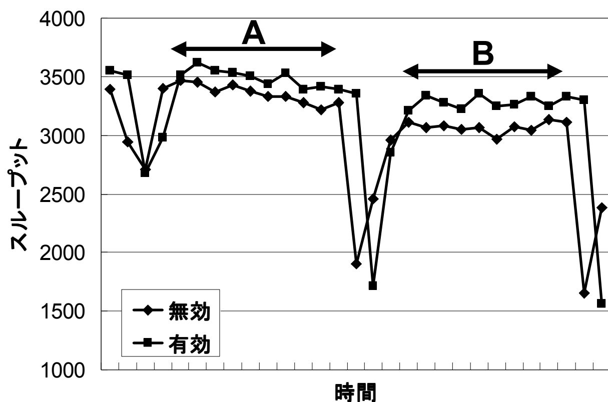
Background Vacuum の導入によって、は、3000 から 3175 TPM へ、5%のスループット向上が見られた

（グラフ 5)。

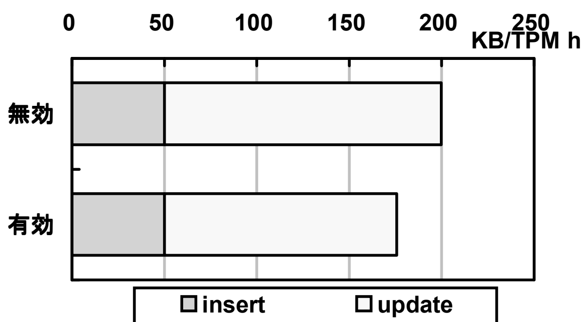
特に計測後半での性能向上が大きいことから、ガベージの早期回収の効果が表れていると考えられる。性能の傾向を区間 A と区間 B で比較すると、TPM 劣化傾向は、-8.46% から -5.93% へ改善している。劣化が 30%程度軽減されることが確認できた。

また、ファイルの増加サイズのうち、削減することが不可能な INSERT を除いた、UPDATE による増加量は、16%の削減が見られた（グラフ 6)。

本稿での Background Vacuum は、テーブル領域のみを対象としており、インデックスに対してはガベージ回収を行わない。そのため、性能劣化が完全には解消できなかったものと考えられる。



グラフ 5: Background Vacuum による
スループットの変化

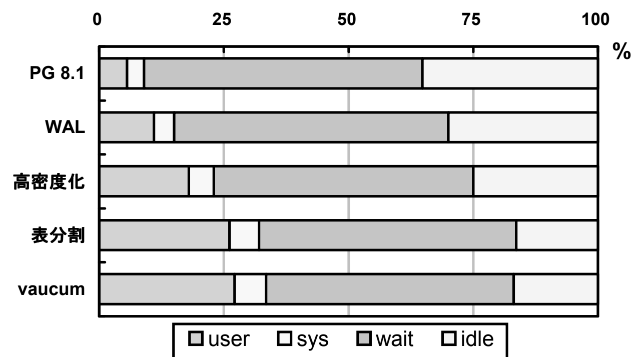


グラフ 6: Background Vacuum による増加抑制効果

8.5. CPU 使用率

CPU 使用率は、スループットの上昇と共に増加しているが、本測定環境では CPU に余裕があったので、大きな問題には至っていない。

高密度化による CPU 使用率の上昇は、スループットに対して相対的に大きい。この理由としては、本稿で追加した可変長型の処理時間と共に、既存型との型変換のオーバーヘッドが考えられる。型変換のオーバーヘッドを避けるためには、既存型の完全な置き換えが必要である。これについては、今後の課題とする。



グラフ 7: CPU 使用率

9. おわりに

9.1. まとめ

PostgreSQL の高速化について検討し、メモリキャッシュ効率に関する 4 つの改善「ログキャッシングの抑制」「データの高密度化」「垂直パーティショニング」「常時稼動型メンテナンス」を行った。それらの改善により、PostgreSQL の性能を手法適用前に比べ、233%に向上できた。

9.2. 課題

9.2.1. 高密度型の自動適用

本稿では拡張モジュールとして実現されているが、PostgreSQL 本体に組み込み、テーブルの定義に応じて、最適な型が自動選択されることが望ましい。

9.2.2. 更新量削減の一般化と自動化

行の中の幾つかのフィールドのみを更新する場合には行全体の複製を避けられれば、垂直パーティショニングを行わないまま、更新量の削減の自動化が可能になる。更新時に差分のみを記録する追記方式に一般化することを検討したい。

9.2.3. 性能劣化改善

テーブル領域だけでなく、インデックス領域の Background Vacuum の検討が必要である。Vacuum の早期化、自動化については、性能面だけではなく、運用面からも改善が望まれている。

文　献

[1] JPUG, “同時実行制御,” PostgreSQL 8.1 文書, http://www.postgresql.jp/document/current/html/mvcc.html

[2] Rod Taylor, “Real-Time Vacuum Possibility,” http://archives.postgresql.org/pgsql-hackers/2005-03/msg00518.php, Mar.2005.

[3] Mark Wong at el, “OSDL Database Test 2 (DBT-2),” http://www.osdl.org/lab_activities/kernel_testing/osdl_database_test_suite/osdl_dbt-2/