# 高速I/O環境でのOLTP負荷に対するPostgreSQLのCPUスケーラビリティに関する一考察

川田　明良† 　藤田　悦郎† 　日高　東潮†

† 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: †{kawada.akiyoshi,fujita.etsuro,hitaka.toshio}@lab.ntt.co.jp

**あらまし**　高速 I/O 環境での OLTP 負荷に対する PostgreSQL の CPU スケーラビリティの阻害要因はジャーナル (WAL) 編集でのバッファ挿入の直列化であった．本論文では回避方法として PostgreSQL の WAL 導入以前の Write-Through 方式を採用し, 評価した結果を報告する．

**キーワード**　PostgreSQL, スケーラビリティ, OLTP

# A Study for CPU scalability on OLTP in PostgreSQL with high speed I/O devices

Akiyoshi KAWADA†, Etsuro FUJITA†, and Toshio HITAKA†

† NTT Cyber Space Laboratories
1-1 Hikarinooka, Yokosuka, Kanagawa, 239–0847 Japan
E-mail: †{kawada.akiyoshi,fujita.etsuro,hitaka.toshio}@lab.ntt.co.jp

**Abstract**　We found that CPU scalability in PostgreSQL on OLTP with fast I/O devices is suffered from serialization of mutual "exclusive" journal records insertion to buffers, that need to align to obtain a lock. This report is presenting a design, Write-Through, that abandon the whole journal function to avoid the bottleneck in CPU scalability, and the results that we evaluate it.

**Key words**　PostgreSQL, scalability, OLTP

## 1. はじめに

広く使われている RDBMS(relational database management system) は, 製品や OSS(open source software) の PostgreSQL [1] を含め, I/O subsystem が HDD(Hard Disk Drive) の時代の設計であり, SSD(Solid State Drive), RAM Drive に代表される高速 I/O subsystem に必ずしも最適化された設計と言えない.

従来の I/O subsystem である HDD は内部が回転する円盤上を読み取り, 書き込み部が力学的に移動する機構のため, ランダムアクセスでのレスポンスが低速である. そのため, 一般的には RDBMS では HDD へ DB データの同期書き込みを行うとランダムアクセスが大量に発生するため性能が大幅に低下する.

そのためトランザクションのコミットで DB データの書き込みは非同期で行い (チェックポイント処理), 別途ジャーナル (以下 JNL) と呼ばれるデータ更新履歴情報をコミット時に同期で行い順アクセスファイルへ書き込むことで, 高性能化をはかる実装が一般的に行われている. PostgreSQL においても JNL として WAL(Write Ahead Log) と呼ばれるファイルを用意し, 上述のデータ書き込み手法を踏襲している.

しかし近年の SSD に代表される高速 I/O subsystem の登場により, レスポンス性能が HDD と比較して大幅に向上し, ランダムアクセスでのレスポンス性能がシーケンシャルアクセスと比較して遜色なくなってきたことから, JNL を用いたデータ永続化技術の必要性は低くなってきたと考えられる.

また JNL のメモリ上での編集処理において複数トランザクション間の排他制御がマルチコア CPU/マルチプロセッサシステムにおける CPU スケーラビリティの阻害要因となっていることから, JNL 方式に対し, コミット時にデータを即時同期更新する Write-Through 方式の優位性について, PostgreSQL を用いて高速な I/O subsystem 上での検証, 考察を行った.

## 2. 課　　題

近年高速 I/O subsystem として登場してきた SSD システムを DBMS に適用することで HDD システムに比べて大幅に性能が向上すると言われている．実際に PostgreSQL を実行し, ワークロードとして OLTP 系トランザクションを走行させ, HDD システムと比較を行った.

（1） 実験環境

- Xeon(W5590 物理 4 cores)2CPU 構成/96GB RAM
- 8 Intel X25-E(SLC) SSDs, RAID0/HDD は単体
- CentOS-5.4-x86_64/PostgreSQL8.4.4

（2） 試行条件

- TPC-B 相当のベンチマークとして pgbench[2] を使用 (スケールファクタ 1000, 表 2,4 は接続クライアント数 100, それ以外は接続クライアント数 200)
- PostgreSQL の共有メモリサイズ 1024MB

（3） スループット測定結果

- HDD : 24.3 tps
- SSD : 2372.0 tps

上記の通り, 実測でも大幅な性能向上が確認された. その際の CPU リソースの使用状況としては, HDD システムに比べ, SSD システムの方が iowait, idle が小さくなるため, それ以外の user, sys が相対的に増加する (表 1).

表 1 CPU リソースの使用状況 (2 cores の場合)

| | user | sys | iowait | idle |
|---|---|---|---|---|
| HDD(%) | 1.15 | 0.87 | 81.94 | 15.26 |
| SSD(%) | 45.79 | 18.79 | 19.54 | 7.89 |

また, 近年登場してきたマルチコア CPU において, コア数が多くなるに従い, idle の占める割合が増大し, 16 コアでは 50%を超える場合があることが予備実験により判明した (図 1).

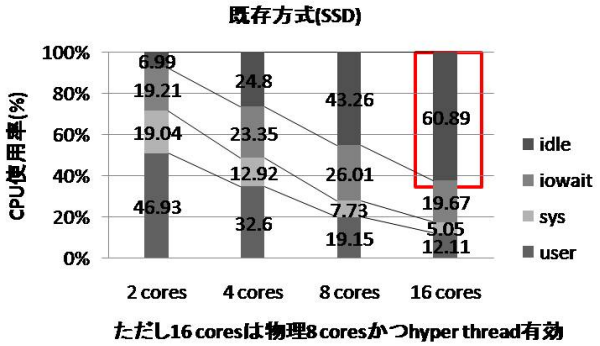


図 1 既存方式の CPU 使用率のコア数依存性

この原因を分析するにあたり, 以下のように検討を行った.

（1） idle の原因としては, 検証を単独サーバで行ったため, 通信待ち合せは考えられず, プログラム内での排他待ち合せによるロック競合が考えられる.

（2） PostgreSQL でのロック関連の関数は全て LWLock-Acquire() という軽量ロック関数でロックを取得, 制御している.

（3） そのため,CPU コア数増大に伴う CPU idle 増大は LWLockAcquire() 関数の呼出しに起因するいずれかの関数で処理時間が増大していると思われる.

そこで Oprofile(CPU のパフォーマンス監視カウンタを利用したプロファイラ) を走行させ, 分析をおこなったところ, CPU コア数を 2 から 16 に増やすに従い, XLogInsert() 関数の CPU 実行時間が増大することが分かった (表 2).

表 2 全 LWLockAcquire() 呼出し関数 (およそ 40) の CPU 実行時間の内訳 (%) 上位

| 関数名 | 2 cores | 16 cores |
|---|---|---|
| XLogInsert | 6.2 | 20 |
| index_getnext | 5.7 | 8.2 |
| XLogFlush | 2.4 | 3 |
| AdvanceXLInsertBuffer | 0.5 | 1.2 |
| 以下省略 | | |

XLogInsert() 関数は, JNL 情報を WAL ファイルへ出力する前に, 走行しているトランザクション群がコミット時に生成する更新履歴情報を共有バッファ内で直列化させ, ファイル出力イメージを生成する処理を行う. そのため, トランザクション実行プロセス間で排他制御を行い, システム内でグローバルな排他ロックを取得し, バッファに更新履歴情報を書き込み, 排他ロックを解放するといった一連の処理を行う [3] [4] [5].

XLogInsert() 内部の何が CPU コアにより処理時間が増大しているか分析するにあたり, Systemtap(リアルタイムでプログラム挙動の動的追跡をするツール) でロック取得待ち時間（区間 A）とロック取得後の処理時間（区間 B）を測定したが (表 3,4), ロック取得後の処理の時間は CPU コア数が変化しても一定値であり, 区間 B が直列化していることが判る.

表 3 XLogInsert() 内で区間 A,B の定義

| | |
|---|---|
| 区間 A | ロック取得 ～ ロック取得成功 |
| 区間 B | ロック取得成功 ～ 内部処理 ～ ロック解放 |

表 4 XLogInsert() 内の区間 A,B の時間計測

| | 2 cores | 4 cores | 8 cores |
|---|---|---|---|
| A ($\mu$sec) | 2793.4 | 2204.7 | 1717.1 |
| B ($\mu$sec) | 7.5 | 7.2 | 8.0 |

ここで, 区間 B の直列化の簡略化した概念を図 2 に示すが, 個々のトランザクションは逐次的に動作するため, 区間 B の後で XLogInsert() 関数からの戻り後に行われる一連の処理は区間 B を追い越して実行することができず, 区間 A を除いた灰色のセルの三角形の箇所で区間 B の実行が待たされることにより, CPU 実行時間に寄与しない CPU idle として計上されてしまう. CPU リソースの使用状況において, core 数が増大するに従い, CPU idle は増大し, 特に図に示す単純化した場合に限れば, CPU idle は 1/2 に漸近することが説明できる.

概念的には, 区間 B の直列化により CPU コア数が増大しても実質 1 コアしか使用できず, 相対的に CPU idle は増大する.

このことから, WAL 情報のバッファ書き込みの前に XLogInsert() 関数が行うトランザクション実行プロセス間排他処理が高速 I/O subsystem において, CPU コア数増大に伴う CPU idle 増大の原因であることが推測される.

次章ではこの問題を解消する方法について述べる.

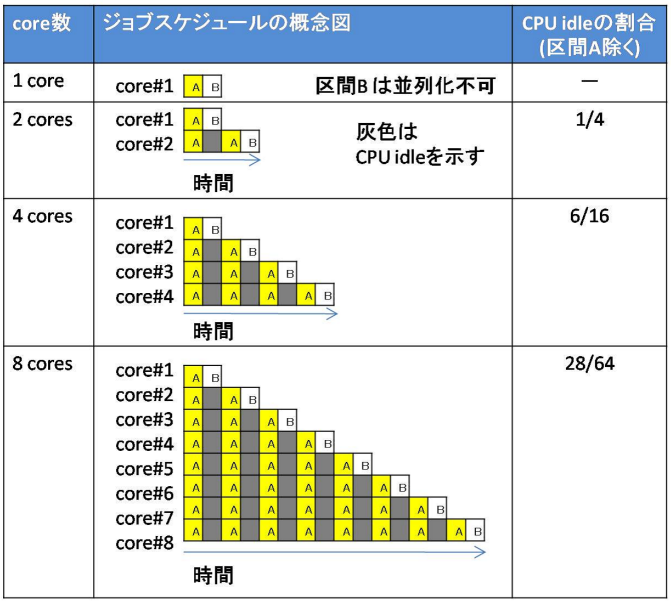


図 2　区間 B の直列化の概念 (区間 A はシステム内でグローバルな排他ロックを取得する)

## 3. 対処方式

PostgreSQL の既存方式では, 更新系 SQL ステートメント毎に JNL 情報を生成し, WAL 編集用バッファ領域に他トランザクションと排他しながら直列化して追記を行い, トランザクションのコミット時に JNL 記憶用ファイルの WAL ファイルへの同期書き込みを行う (図 3).

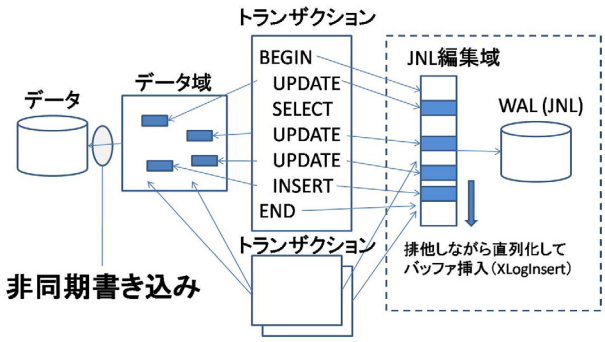


図 3　既存方式

XLogInsert() 関数は従来の JNL 出力方式に準じる限り, 必須となる機能であり, また排他処理を回避するのは困難である. そのため, 本論文では JNL 出力方式自体を回避し, 更新時にデータを即時同期書き込みする Write-Through 方式 (以下 WT 方式) を採用, 検証する. WT 方式のポイントは以下に示す 2 点となる.

（1） XLogInsert() 関数相当の廃止

（2） データ域書き込みの同期処理化

図 4 に WT 方式の概要を示す.

WT 方式により, XLogInsert() 関数での CPU idle 増大は回避できる (図 1 に対する WT 方式の測定結果を図 5 に示すが,

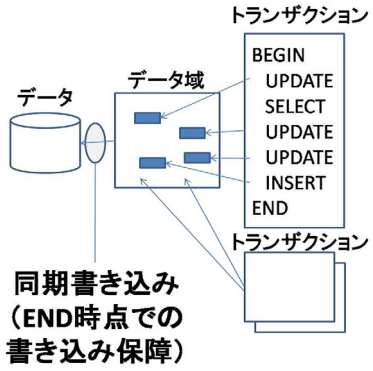


図 4　WT 方式

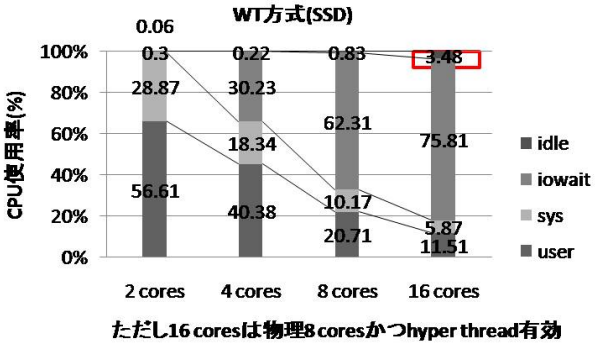


図 5　WT 方式の CPU 使用率のコア数依存性

CPU idle 増大は解消される) が, I/O 出力の方法の変更による影響について考慮する必要がある. そのため, この WT 方式が既存方式に対して有用である状況を考えるにあたり, まず I/O subsystem のレイテンシ, およびランダム・シーケンシャルアクセス特性の差異がどのように影響するのか, モデル化して性能を検証する. なお, 方式による性能差異については更新のみが影響するため, 以降では参照処理については考えないものとする. モデル化にあたり, 定義する変数を表 5 に示す.

表 5　変数の定義

| 変数 | 定義 |
|---|---|
| $B^{O}_{data}$ | 既存方式のデータ域でアクセスするブロック数 |
| $B^{WT}_{data}$ | WT 方式のデータ域でアクセスするブロック数 |
| $B_{WAL}$ | 既存方式の WAL 書き込みでアクセスするブロック数 |
| $T_{async}$ | 非同期書き込みにおける I/O subsystem 1 ブロックあたりの書き込み折り返し応答時間 |
| $T_{sync}$ | 同期書き込みにおける I/O subsystem 1 ブロックあたりの書き込み折り返し応答時間 |
| $R_{random}$ | ランダムアクセスによる応答時間に対する係数 (次ブロックへのシーク時間の効果) |
| $R_{seq}$ | シーケンシャルアクセスによる応答時間に対する係数 (次ブロックへのシーク時間の効果) |
| $cost^{O}$ | 既存方式の I/O コスト |
| $cost^{WT}$ | WT 方式の I/O コスト |

上記の定義に対し, 1 次近似として既存方式,WT 方式の I/O コスト $cost^{O}$, $cost^{WT}$ は下記の式で記述できる.

$$cost^{O} = B^{O}_{data} T_{async} R_{random} + B_{WAL} T_{sync} R_{seq}$$
$$cost^{WT} = B^{WT}_{data} T_{sync} R_{random}$$

ここで, 既存方式において, WAL への出力情報とデータ域への更新情報は元をたどれば同じ起源であり, 同等の情報量であることから,

$$B_{WAL} \simeq B^{O}_{data}$$

と仮定すれば, I/O コスト差は以下のようになる.

$$cost^{WT} - cost^{O} = B^{O}_{data} T_{sync} R_{random} \times \left\{ \frac{B^{WT}_{data}}{B^{O}_{data}} - \left( \frac{T_{async}}{T_{sync}} + \frac{R_{seq}}{R_{random}} \right) \right\}$$

なお, 導入したレイテンシ (応答時間), ランダム・シーケンシャルアクセスによる応答時間に対する係数については一般に下記の関係が成り立つ.

$$0 < R_{seq}/R_{random} < 1$$
$$0 < T_{async}/T_{sync} < 1$$

もし,SSD システムが十分高速であり, ディスクアクセスのレイテンシ, 並びにランダムアクセスとシーケンシャルアクセスのレイテンシ差が十分小さければ,

$$R_{seq}/R_{random} \simeq 1$$
$$T_{async}/T_{sync} \simeq 1$$

となるので, I/O コスト差は,

$$cost^{WT} - cost^{O} = B^{O}_{data} T_{sync} R_{random} \left( B^{WT}_{data}/B^{O}_{data} - 2 \right) \quad (1)$$

となる. $B^{WT}_{data}$ と $B^{O}_{data}$ は, データの配置状況により差異が生じるが, その差異が小さい場合には, I/O コスト差は式 (1) により以下のようになる.

$$cost^{WT} - cost^{O} = B^{O}_{data} T_{sync} R_{random} (1 - 2) < 0$$

以上より, 今回検証する WT 方式は既存方式に対して十分優位であり得ると推測する.

なお, 今回検証に用いた WT 方式では, JNL 情報がないことから, アーカイブリカバリにおいて問題が生じる可能性があるが, PostgreSQL ではデータ格納構造として MVCC(multi-version concurrency control) 方式を用いているため, データ領域にロールバックに必要な情報が充足していることから, PostgreSQL をはじめとした MVCC 方式に準じた DBMS においては, 今回検証した WT 方式は実用上有効であると考えている. ここで MVCC 方式とは, トランザクションにより更新されたタプル (行データ) を, 他トランザクションが読み出す場合に, トランザクションのコミットによるタプルの更新が確定されるまで読み出しをブロックすることなく, 直ちに更新前のタプルを読み出しさせることで, トランザクション分離レベルを若干犠牲にする代わりに, 同時実行性を向上させる方式である. これにより, 読み出しは書き込みをブロックしなくなり, 書き込みは読み出しをブロックしなくなる. 具体的には, 図 6 に例を示すように, トランザクション ID をトランザクション生成順に単調増大させた場合に, タプル毎にタプルを生成したトランザクション ID とタプルを削除したトランザクション ID を保持し, 各トランザクションからタプルが可視か不可視かどちらであるかを管理する.

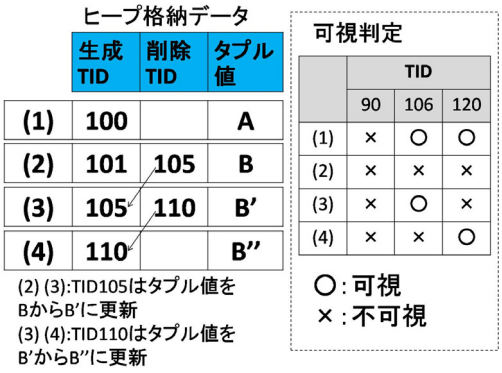


図 6　MVCC 方式

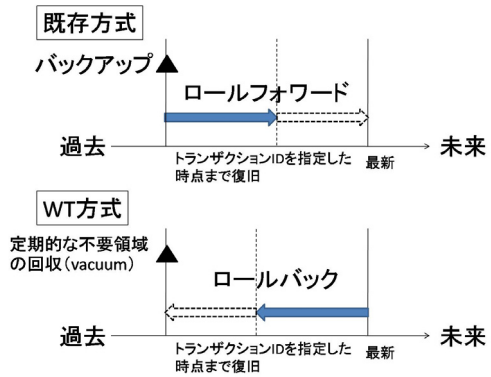


図 7　既存方式と WT 方式でのリカバリ処理

- トランザクション ID が 90 のトランザクションからはタプル (1)(2)(3)(4) は不可視
- トランザクション ID が 106 のトランザクションからはタプル (1)(3) は可視, タプル (2)(4) は不可視
- トランザクション ID が 120 のトランザクションからはタプル (1)(4) は可視, タプル (2)(3) は不可視

(ただし, 実際の可視判定においては, タプルを生成したトランザクション ID, またはタプルを削除したトランザクション ID として書き込まれたトランザクションがコミット済かどうかをチェックし, 未コミットの場合には無効値として扱う) すなわち, WT 方式では, 過去に遡った更新前のタプルの内容は, 定期的な不要領域の回収 (vacuum) が実行されるまで残存し, これをデータ領域のロールバックにおいて適用することが考えられる. 他方, 既存方式では, 過去のある時点のデータ領域のバックアップが存在すれば, それを起点として, JNL を時間順に適用し, ロールフォワードによる復旧が可能である. これは, 図 7 に示すように, トランザクション ID を指定することで, 任意の時点を終点とした復旧が可能と言える.

次章では, 実際に PostgreSQL8.4.4 をベースとして既存方式と WT 方式の比較を行う.

## 4. 実　　験

本章では 2. 章と同様の実験条件, 試行条件において, 既存方式,WT 方式の比較を行った. また, WT 方式の 2 点のポイント (1)XLogInsert() 関数相当の廃止, (2) データ域書き込みの同期処理化, がスループット,CPU リソース使用状況にどのように影響するかを分析するため, XLogInsert() 廃止, すなわち (1) のみ対処した方式 (以下 XL 方式) を用意し, 併せて測定した.

### 4.1 測 定 結 果

測定結果を以下に示す.

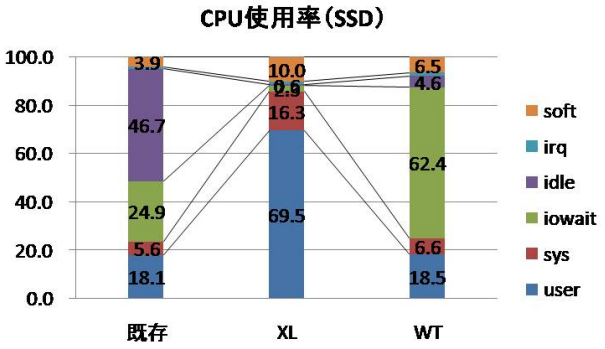


図 8　CPU 使用率 (8 cores)

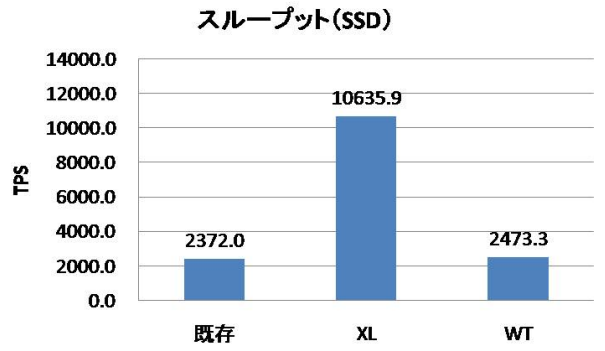


図 9　スループット (8 cores)

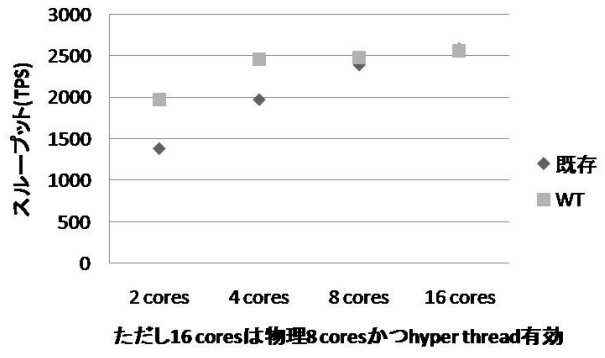


図 10　スループットのコア数依存性

- WT 方式のポイント (1) の観点

既存方式に比べ, XL 方式は XLogInsert() 関数のロック取得後の処理の直列化が解消するため, idle が大幅に減少し, そのため CPU リソース使用内訳のうち user+sys が大幅に増大している. それにより, スループットも大きく向上している.

- WT 方式のポイント (2) の観点

XL 方式に比べ, WT 方式はデータ域の書き込みを非同期から同期に変更したため, iowait が増大し, そのため CPU リソース使用内訳のうち user+sys が減少している. それに伴い, スループットも減少しているが, 既存方式に比べ微増という結果となった.

- CPU スケーラビリティの観点

iowait 増大が原因で, WT 方式はコア数の増大による CPU 資源の増加分を十分に活用することができず, 8 cores 以上ではスループットの伸びが抑えられた (図 10). 結果的に既存方式と WT 方式のスループットは 8 cores 以上では同程度となった.

## 5. 考　　察

SSD システムでは WT 方式は既存方式に対しスループットが上回ると想定したが, 同程度に留まった. XL 方式と WT 方式との間で iowait が増大している点に注目して, ディスク I/O 量を確認すると (図 11) WT 方式ではブロック書き込み数が増

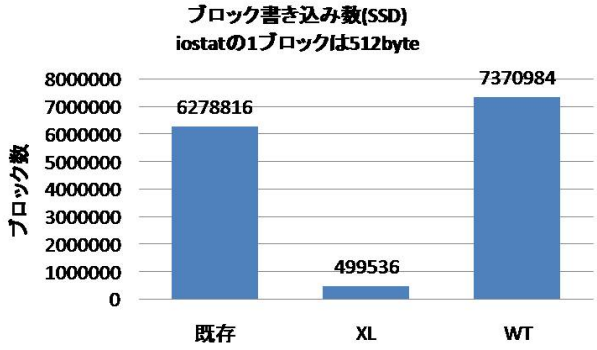


図 11　ブロック書き込み数

大しており, 想定した $B_{data}^{WT} \simeq B_{data}^{O}$ と異なる状況であった.

そこで実際にアクセスするブロック識別子をシステムコール write を呼出す直前においてリスト出力させて調査したところ, 各々のブロック識別子に対する書き込み回数を集計することが出来た. そこでこの書き込み回数を重複数として横軸 (ただし対数スケール), また重複数に該当するブロック数を発生数として縦軸 (同じく対数スケール) に取り, 既存方式 (図 12), XL 方式 (図 13), WT 方式 (図 14) について示す. ここで, 既存方式と XL 方式のリスト出力においては, 非同期書き込みのためディスクに出力されておらずバッファに残留しているデータについては, 最終的にディスクに書き出しが完了するまで待って集計対象とした.

リスト出力の集計結果によれば, 既存方式, XL 方式では, 非同期書き込みのメカニズムにより同一ブロックに対する複数の I/O 出力要求を集約して行うために重複数平均が約 1 なのに対

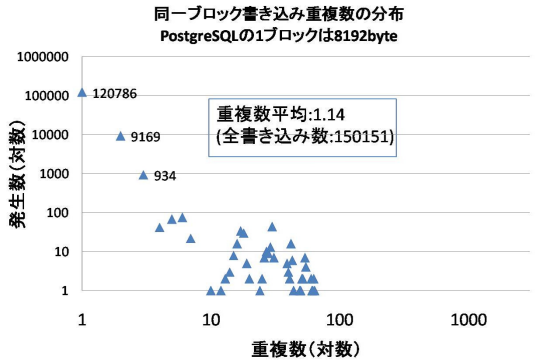


図 12　同一ブロック書き込み重複数の分布 (既存方式)

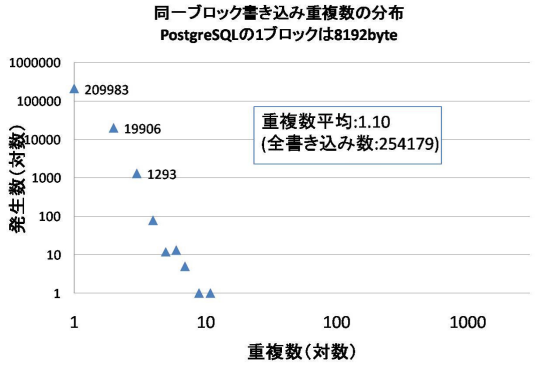


図 13　同一ブロック書き込み重複数の分布 (XL 方式)

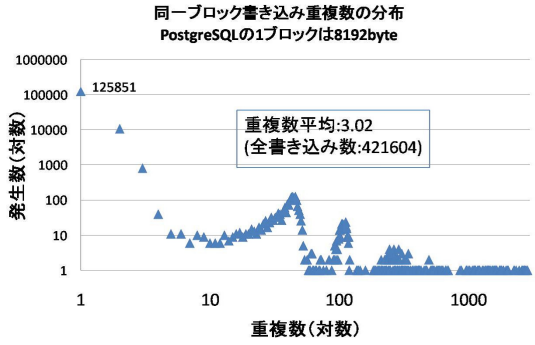


図 14　同一ブロック書き込み重複数の分布 (WT 方式)

し, WT 方式ではこのブロック単位の更新のディスク書き込みの集約効果が失われて, 重複数平均が約 3 となっている. すなわち, PostgreSQL は MVCC 方式により, データ部の追記をベースとしたデータ管理となっていることに起因してデータブロックの終端へ書き込むため, 同一ブロックへの多重書き込みが多数発生していると考えられる.

このことは図 11 で示す I/O 量の差として表れていると考えられる. そこで図 11 を前述の数値モデルに適用すると以下のようになる.

$$\text{既存方式のブロック書き込み数} \ : \ B^{O}_{data} + B_{WAL}$$
$$\simeq 2B^{O}_{data} \simeq 6.3M$$
$$WT \text{ 方式のブロック書き込み数} \ : \ B^{WT}_{data} \simeq 7.4M$$

$$B^{WT}_{data} / B^{O}_{data} > 2$$

これを式 (1) に当てはめると以下のようになる.

$$cost^{WT} > cost^{O}$$

すなわち, WT 方式の I/O コストが既存方式の I/O コストに比べ大きくなり, WT 方式では性能が劣化することが考えられるが, 実際には既存方式よりスループットはわずかに向上している. 従って, この I/O 量を抑止すれば WT 方式で大幅な性能改善が見込めると考えられる.

上記の問題の回避方法としては, MVCC 方式における終端ブロックへの書き込み集中を回避する等により同一ブロック書き込みの集中を抑止し, 既存方式に対する WT 方式の優位性を向上させる余地があると考えられる.

## 6. 周辺研究

PostgreSQL が対象となってはいないが, 同時に発生する複数の JNL のバッファ書き込み要求は, 実は各々の書き込み位置が異なるため, 書き込み位置を取得しさえすればバッファ書き込みは並列処理可能であり (flush pipeline), また競合する複数のバッファ書き込み要求に対しては書き込む処理を集約することにより (consolidation array), JNL のバッファ書き込みの直列化を緩和し, スケーラビリティ向上を報告した研究 [6] があり, JNL の実装見直し, データ部への非同期書き込みというアプローチになっている. 本論文の提案する WT 方式は JNL なし, データ部への同期書き込みという位置付けとなる.

## 7. おわりに

近年登場した高速 I/O subsystem として SSD を使用し, マルチコア CPU システムでの CPU スケーラビリティの阻害要因の分析, 回避方法の提示と実験により, 課題であった CPU idle 増大は大幅に軽減したが, 逆に iowait が増大し, 全体としては CPU コア数が増大した場合の性能向上は微増であることを確認した.

本論文執筆時点で市販されている SSD は HDD と同様, CPU/メモリとは多段のコントローラを介して接続されており, コントローラは HDD を前提とした設計になっているものが多いため, 今後 SSD を前提としたコントローラが登場すれば, よりレイテンシの向上が期待される. その場合には, 今回の検証で得られたデータに比べ, WT 方式でより高い性能が得られる可能性が高いと考えられる.

そこで, 将来的にレイテンシが向上した場合の WT 方式の有用性を考察するため, まだ一般的ではないが, PCI Express データバス接続の SSD(Fusion-io, 以下 PCIe SSD), 並びに RAM ディスクシステム環境 (以下 tmpfs) を使用し, 4. 章と同様の評価を参考までに行った. こうして, 図 15,16 に PCIe SSD を使用した測定結果, 図 17,18 に tmpfs を使用した測定結果を示す. これによりレイテンシが向上したストレージでスループットにおいて WT 方式が有利となり, 我々が予想した傾向となることを確認した.

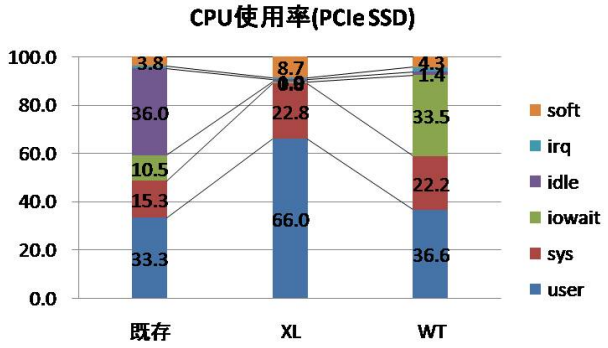


図 15　CPU 使用率 (8 cores)

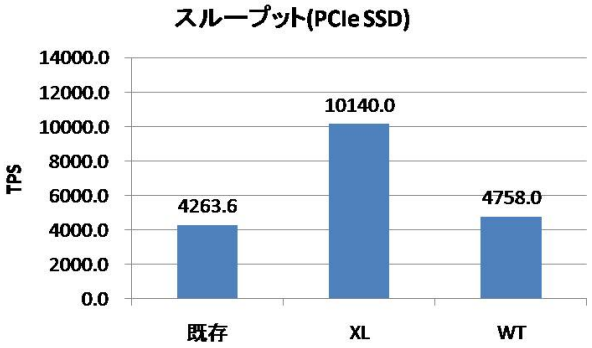


図 16　スループット (8 cores)

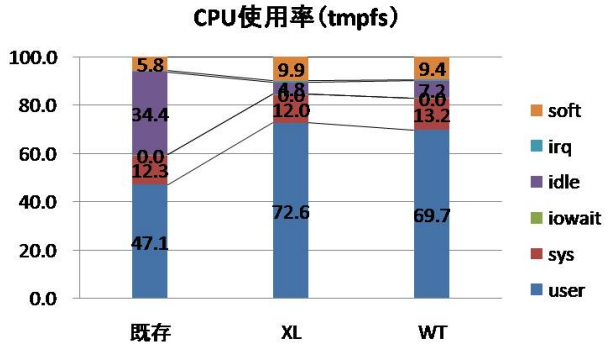


図 17　CPU 使用率 (8 cores)

今後の課題としては MVCC 方式によりデータ領域にロールバックに必要な情報が充足している場合の, 具体的なリカバリ処理時のロールバックの適用方法の検討がある.

## 文　　献


[1] PostgreSQL. http://www.postgresql.org/

[2] pgbench. http://www.postgresql.jp/document/9.0/html/pgbench.html

[3] Ryan Johnson, Ippokratis Pandis, Nikos Hardavellas, Anastassia Ailamaki and Babak Falsafi "Shore-MT: A Scalable Storage Manager for the Multicore Era", Proc. of EDBT, pp. 24-35, 2009.

[4] Ryan Johnson, Manos Athanassoulis, Radu Stoica, Anastasia Ailamaki "A New Look at the Roles of Spinning and Blocking", Proc. of DaMoN, pp. 21-26, 2009.

[5] Maurice Herlihy, Nir Shavit "The Art of Multiprocessor Programming", Morgan Kaufmann, 2008.

[6] Ryan Johnson, Ippokratis Pandis, Radu Stoica, Manos Athanassoulis and Anastasia Ailamaki "Aether: A Scalable Approach to Logging", Proc. of PVLDB 3(1), pp. 681-692, 2010.


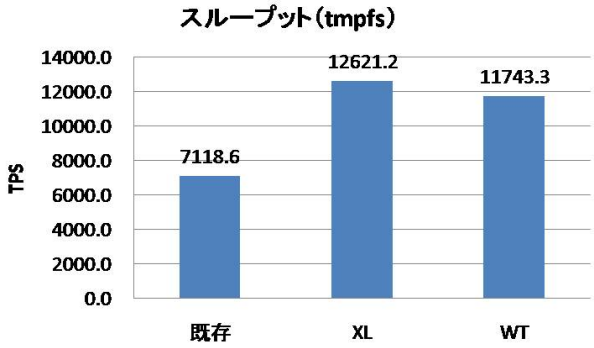


図 18　スループット (8 cores)