

バッチトランザクションに適した ガベージコレクションの効率化

大西 璃奈^{1,a)} 星野 喬^{2,b)} 川島 英之^{1,c)}

概要：Multi-Version Concurrency Control におけるガベージコレクションとして、EPO が知られている。EPO は Version Chain の内部における不要 Version を収集可能であることから、その高効率性が認められている。EPO は収集処理が Write 操作によりトリガされる。本研究ではこのトリガが不適切であるとの考察に基づき、Read 操作に基づく手法 EPO-R を提案する。224 コア環境における実験の結果、EPO-R は EPO に対して最大 3.4 倍のスループット性能を示した。

キーワード：トランザクション処理，並行性制御，ガベージコレクション

Accelerating Garbage Collection for Batch Transactions

RINA OHNISHI^{1,a)} TAKASHI HOSHINO^{2,b)} HIDEYUKI KAWASHIMA^{1,c)}

1. はじめに

1.1 背景

日々行われる振り込みやクレジットカード決済は言うに及ばず、分散ファイルシステムのメタデータ管理や IoT 決済など、数多の社会基盤においてトランザクション処理は行われている。そのため、トランザクション処理の高性能化は社会基盤の効率化に有益である。トランザクション処理の主たる構成要素は並行性制御とリカバリである。並行性制御は複数のトランザクションが同時並行的に処理されても、その結果が Serializable になるように各トランザクションの振る舞いを制御する。近年のアーキテクチャのメニーコア化を受けて様々な並行性制御プロトコルが提案されてきた [1–3]。

並行性制御プロトコルは、単一 Version を用いる Single-Version Concurrency Control (SVCC) と複数 Version を

用いる Multi-Version Concurrency Control (MVCC) の二種類に大別できる。SVCC は Write の際に In-Place Update を行う。そのため SVCC では、Reader と Writer が同時に同一データにアクセスすると、それらの片方はブロックされる。他方、MVCC は Write の際に Append を行う。そのため、MVCC では、Writer の Append 処理と Reader の Scan を Version Chain 上で同時に実行できる。この原理により MVCC は高性能が期待され、PostgreSQL や Oracle などのプロダクションシステムで採用されている。MVCC で多数の Version を管理するには、大規模なメモリが求められる。これはメモリ容量の増加する昨今の傾向に適している。昨今 DRAM 容量は単一サーバで 1TB を超えており、バイト単位でアクセス可能な高記憶密度の NVRAM も普及しつつある。

高性能を期待される MVCC では多数の Version が作成されるため、古くて不要になった Version の回収が必要になる。これをガベージコレクション (GC) という。各 Version の不要性は、アクティブなトランザクションのタイムスタンプの中で、最も古い値 (Active Oldest Transaction Timestamp, AOT) により判定される [4–6]。AOT 未満のタイムスタンプ値を持つ Version は、どのトランザクシ

¹ 慶應義塾大学環境情報学部
Faculty of Environment and Information Studies

² サイボウズ・ラボ
Cybozu Labs

^{a)} rina.onishi2951@keio.jp

^{b)} hoshino@labs.cybozu.jp

^{c)} river@sfc.keio.ac.jp

ンからもアクセスされないことが保証される。従ってそれらは不要であり、回収可能である。GC の起動トリガも重要であり、様々に模索されてきた [2-7]。

1.2 課題

回収指標を AOT に限定すると、バッチトランザクションを含むワークロードでは AOT 更新頻度が低いために、本来不要な Version が長期間回収されず、回収効率が劣化してしまう [4]。この問題を解決すべく Bottcher らは中間 Version を回収可能な Eager Pruning of Obsolete Versions (EPO) 法を提案した [8]。EPO は Version Chain を頻繁に Traverse して不要な Version を検出し、回収効率を向上させる。一方で、この回収 Traversal は性能劣化要因になりえる。Traversal により不要 Version が回収されればメモリ効率が向上する。ところが不要 Version が回収されない Traversal は無駄である。この無駄によって本来行われるべき処理に割かれる CPU 資源が減り、性能劣化を引き起こす可能性がある。無駄な Traversal を減らすにはどうすれば良いのだろうか？

1.3 貢献

この問題を解決するために、本研究では EPO-R を提案する。EPO における回収 Traversal は、Write 操作が発行されたときに実行される。それに対して EPO-R は Read 操作が発行された時に回収 Traversal を実行する。

GC は並行性制御プロトコルと合わせて使用される。Oracle, HANA, SQL Server などでは用いられる Snapshot Isolation (SI) プロトコルを本研究では対象とする。SI における Read 処理は Version Chain Traverse を必要とする。他方、SI における Write 処理は Version Chain に新規 Version を Append するために Traversal を実行しない。従って回収 Traversal を Read 操作時に起動すれば、回収処理を Read 処理に統合させられる。これにより EPO-R は EPO よりも CPU 資源の無駄を削減できるとの仮説を得た。

この仮説を検証するために EPO と EPO-R の性能比較実験を行った。実験の結果、EPO-R は EPO に対して同等以上の性能を示した。最大性能向上率は 3.4 倍だった。

1.4 構成

本論文の構成は以下の通りである。2 節では準備として MVCC ならびに EPO について述べる。3 節では提案手法である EPO-R を述べる。4 節では EPO-R を実験的に評価する。5 節では関連研究を述べる。6 節では結論を述べる。

2. 準備

2.1 Multi-Version Concurrency Control

MVCC では、あるトランザクションによってデータに更

新が加わる度に、そのタプルの新しい Version を生成する。あるタプルに対する更新処理が複数回実行されると、複数の Version が生成される。そしてそれらはタイムスタンプを Key として並べた Version Chain として保存される。特に、降順に並び、最新の Version が先頭に来るようなレイアウトを N2O (Newest-to-oldest) と呼ぶ。N2O では、最新の Version 取得に Chain の Traversal が必要がない一方で、新規バージョンの追加には、Chain の先頭の張り替えを行う必要がある [9]。

2.2 ガベージコレクション

Version Chain が肥大化するとシステムの性能が劣化する可能性がある。なぜならそれに使われる巨大なメモリが I/O を引き起こしたり、長大な Chain の Traversal に CPU 資源が大量に使用される可能性があるからである。このような事態は不要になった Version により引き起こされる。不要な Version とは、アクティブまたは未来のトランザクションからアクセスされることはない Version である。不要な Version は使用されることはなく、単にメモリ資源の圧迫要因となる。この問題を避けるために、不要な Version はできるだけ速やかに回収することが必要である。これはガベージコレクション (Garbage Collection, GC) と呼ばれる。適切な GC 機構は無駄なメモリ使用を減らすため、スループットを向上させることが期待される。

MVCC における単純な GC は AOT により実現される。AOT はアクティブなトランザクションのタイムスタンプの中で、最も古い値である。各 Version の不要性は AOT により判定される。AOT 未満のタイムスタンプ値をもつ Version は、最早どのトランザクションからもアクセスされないことが保証されるため、GC 機構により回収される。この方法では、GC 機構は最も古いアクティブトランザクション開始時のタイムスタンプのみを保持するに過ぎないため、本来は回収可能な不要 Version を検知できないことがある。

2.3 Steam

不要 Version を検知できない問題を解決するために、Bottcher らは EPO という GC 手法を提案し、Steam の GC 機構に加えた [8]。Steam では、以下の手順で、AOT 値を計算し、バックグラウンドで AOT に基づく GC を行っていた。

- 各ワークスレッドは、タイムスタンプ取得毎に、その値を他のスレッドに共有する。
- 各ワークスレッドは、定期的に、共有されたタイムスタンプをスキャンし、AOT を計算する。

Steam は長期トランザクションと短期トランザクションの両方が含まれる混合ワークロードにおいては性能が劣化する。

この問題を解決するため、Steam の最適化手法として **EPO (Eager Pruning of Obsolete Versions)** が提案された^{*1}。

EPO は AOT 以前の Version のみならず、AOT 以降の Version についても回収を試みる手法である。EPO は AOT 以降であっても、アクティブなトランザクションからアクセス不能な Version を検知する。これは、アクティブなトランザクションのリストを構築することで、実現される。この構築は以下の手順で行われる：

- 各ワークスレッドは、タイムスタンプ取得毎に、その値を他のスレッドに共有する。
- 各ワークスレッドは、定期的に、共有されたタイムスタンプをスキャンする。
- それらをローカルなリストにソートした上で格納する。このリストを用いてより多くの不要 Version が回収される。

EPO は、他のトランザクションに比べて著しく長いトランザクションが存在するワークロードにおいて、EPO は既存手法よりもトランザクション処理のスループットを向上させることが実験的に示されている。

Algorithm 1 に EPO の動作を示す：

Algorithm 1 EPO Algorithm

Require: active time stamps A (descending order)

Ensure: pruned version chain

```

1: // Write 操作時に起動
2: if Write_FLAG then
3:    $v_{current} \leftarrow retrieveVisibleVersion(a_0, chain)$ 
4:   for  $a_i$  in  $A$  do
5:      $v_{visible} \leftarrow retrieveVisibleVersion(a_i, chain)$ 
6:     // 中間 Version を回収
7:     for  $v$  in  $(v_{current}, v_{visible})$  do
8:        $chain.remove(v)$ 
9:     end for
10:    // Version イテレータを更新
11:     $v_{current} \leftarrow v_{visible}$ 
12:   end for
13:   //  $v_{visible}$  後続の Version を回収
14:    $detachFromVersionChain(prev(v_{visible}), chain)$ 
15: end if

```

図 1 に EPO アルゴリズムの実施例を示す。図中の (a) は、トランザクション T_i が保持する Active Timestamps を指し、(b) はレコード R_i に紐付く Version Chain を指す。 T_i が R_i に対して Write 操作を実行した時に、EPO が起動する。 $v_{current}$ の初期値には、 a_0 にとって Visible な Version が入る。以降、降順に並んでいる Active Timestamps をイテレートしながら中間 Version を回収する。 $a_i := 105$ のとき、 $v_{visible}$ にはタイムスタンプ値 90 を持つ Version が格納される。このとき、中間 Version が無いため Version の回収は為されない。 $v_{visible}$ を $v_{current}$ で更新する。 $a_i := 85$

^{*1} 簡潔性のため、以降では EPO を適用した Steam を EPO と表記する。

のとき、 $v_{visible}$ には、タイムスタンプ値 70 を持つ Version が入る。この時、 $v_{current}$ と $v_{visible}$ の間にある Version を回収する。 $v_{visible}$ を $v_{current}$ で更新する。これで Active Timestamps のイテレートが終了する。最後に $v_{current}$ 後続の Version を回収して EPO が終了する。

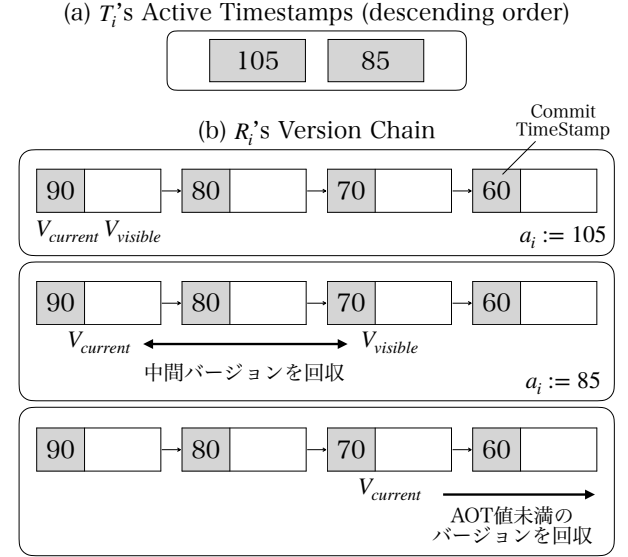


図 1: Version Chain の例

3. 提案: EPO-R

EPO では定期的に不要 Version を特定して、その回収を行う。この処理は Write 操作が起動されるときに限定されている。EPO 原論文にはこの起動が Write に限定される理由は示されていない。

3.1 起動トリガ

我々はこの起動トリガは不適切だと考える。なぜなら EPO は並行性制御プロトコル Snapshot Isolation (SI) を対象としており、SI は Read 操作処理時に必ず Version Chain を Traverse する一方、Write 操作処理時には Version Chain の一端のみにアクセスするからである。それゆえ、EPO の起動トリガが Write 操作であり、不要 Version の回収が一切できない場合には、Write 操作は無駄な Scan を行い、その結果として CPU 資源を無駄に用いることになると考えられる。

上記の仮説が正しいならば、並行性制御プロトコルが SI である場合には、EPO の起動トリガを Read 操作とすることにより、トランザクション処理性能が向上すると考える。そこで本研究では EPO の起動トリガを Read 操作とする手法、**EPO-R** を提案する。EPO-R は EPO よりも同等以上のトランザクション処理性能にもたらずと考えられる。なぜなら、EPO-R の処理内容は、EPO の処理内容の部分集合であるからである。また、EPO-R は EPO よりも

トランザクション処理性能を有意に向上させる可能性がある。なぜなら Version Chain が長い程 EPO-R は EPO よりも高速になり、次の 2 つの場合では Version Chain が長くなると考えられるからである。

- トランザクションが長期に渡る場合：
トランザクション T の実行が長期に渡る場合、 T がアクセスしたタプルの AOT は更新されなくなる。そのような場合には、不要 Version が増えなくなる。この場合には Version Chain が長くなる。
- Steam のリスト更新周期が長い場合：
EPO では、Active Timestamps の構築コストを抑制するため、構築を周期的（原論文では 5 ms）で行う。従って次の更新までは、最後に構築した古いリストを使い回す。これにより、回収可能であるにも関わらず、直ちに回収できない Version が生じる。例を示す。Active Timestamps[85, 105] に対して、トランザクション T_i は古い Active Timestamps[85, 65] を保持しているとする。また、あるタプル R_i に [100, 90, 80, 70, 60] の Version Chain が紐付いているとする。今、 T_i が R_i に EPO を実行したならば、[70] のみが回収される。もしも最新のリストを使ったならば、[90, 70, 60] が回収される。このように EPO では、古い Active Timestamps の利用により、回収が見落とされる Version が生じる。その結果、リスト更新周期が長いとき、Version Chain が長くなる。

3.2 Aborted

トランザクション処理の過程で生成される Version の中には、Abort されたために不要になった Version も多数存在する。これらは、Active Transaction List による回収の対象外であったとしても、EPO 実行中に見つけ次第、回収することにした。EPO-R を Algorithm2 に示す：

Algorithm 2 提案手法 EPO-R

Require: active time stamps A (descending order)

Ensure: pruned version chain

```

1: // Read 操作時起動
2: if READ_FLAG then
3:    $v_{current} \leftarrow retrieveVisibleReclaimingAborted(a_0, chain)$ 
4:   for  $a_i$  in  $A$  do
5:      $v_{visible} \leftarrow retrieveVisibleVersion(a_i, chain)$ 
6:     // 中間 Version を回収
7:     for  $v$  in  $(v_{current}, v_{visible})$  do
8:        $chain.remove(v)$ 
9:     end for
10:    // Version イテレータを更新
11:     $v_{current} \leftarrow v_{visible}$ 
12:   end for
13:   //  $v_{visible}$  後続の Version を回収
14:    $detachFromVersionChain(prev(v_{visible}), chain)$ 
15: end if

```

Read 時に起動されることと、Aborted Version を見つけ次第回収する最適化を除いて、動作は既存手法と同じであるため説明を省く。

提案手法を図 2 に示す：

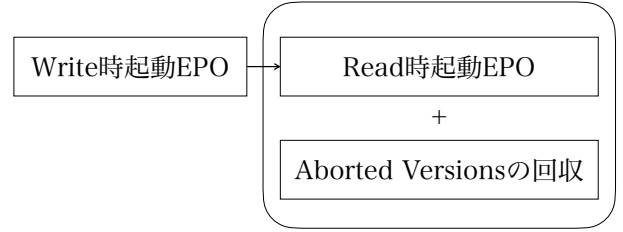


図 2: 提案手法の模式図

3.3 実装

Read 時に EPO を実施する。この中で、Aborted バージョンを見つけた場合、回収する。EPO で用いる Active Transaction List は各スレッドが、一定間隔毎（本論文では 100 ms）に構築する。この構築は、各スレッドが共有された他のスレッドのアクティブなトランザクションの開始タイムスタンプを取得し、それらを降順ソートした上でリストに格納することで行う。

各 Record には、Version Chain の先頭を指す 64 ビットのポインタ latest、および、レコードのラッチに使用する lockBit を格納する。Version Chain は単方向連結リストによって実装する。これは、タイムスタンプに対して降順に並ぶ N2O を用いる。

各 Version には、後続のバージョンを指すポインタ prev、値を格納する 64 ビットの value、バージョンのステータスを格納する 8bit の versionStatus、64 ビットの commitTimestamp を格納する。versionStatus には、PENDING、COMMITTED、ABORTED のいずれかを格納する。タイムスタンプの取得には、単調増加するグローバルなカウンタに対してアトミックな加算命令を実行することで行う。これは、トランザクション開始時とコミット時に行う。コミット時には、コミット時に取得したタイムスタンプ値を、新規作成したバージョンの commitTimestamp に書き込む。同時に、それらの versionStatus を COMMITTED に変更する。Read 操作時は、自分の開始時に取得したタイムスタンプ値未満のうち、commitTimestamp が最大のコミット済みバージョンを読み込む。Write 操作時は、新規に作成した PENDING バージョンの prev にレコードの最新バージョンを格納した上で、レコードの最新バージョンを置き換える。レコードに対するアクセス毎に、ラッチを行う。

4. 評価

本節では、提案手法 EPO-R を実験的に評価する。従来手法 EPO にも EPO-R と同様の Aborted Version を回収

する最適化を施して実験を行った。

4.1 実験環境

実験には Intel(R) Xeon(R) Platinum 8276 CPU@2.20 GHz を 4 ソケット搭載したサーバを利用した。各ソケットには物理 28 コア，論理 56 コアがあり，39MB L3 キャッシュを共有していた。それぞれのコアには 32KB private L1d キャッシュ，1024KB private L2 キャッシュが存在した。トータルキャッシュサイズはおよそ 160MB だった。

4.2 ワークロード

実験に用いるワークロードには二つの特性を持たせた。第一の特性は，ワークロードに操作数やスリープ時間の著しく異なるトランザクションを混在させることである。EPO はそのようなワークロードに資するために考案された。第二の特性は，高競合であることである。高競合下では急速に Version Chain が伸び，結果，Traversal 量が増加する。これらの特性を満たすよう，実験パラメタを設定した。その結果を表 1，表 2 に示す。

表 1: 実験パラメタ (全実験共通)

#Threads	224
RecordLength	10,000
Skew	0.8
ReadRatio	50
ActiveTransactionListUpdateInterval[ms]	100

表 2: 実験パラメタ

	値 1	値 2	値 3
SleepTime[us]	0	1,000,000	10,000,000
OperationSize	6	100	1,000

4.3 スループット

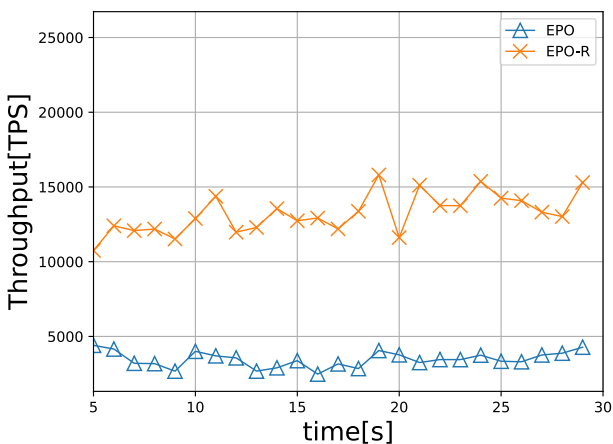


図 3: スループット

図 3 にスループットに関する実験結果を示す。この結果

からは，EPO-R のスループットが EPO のそれより最大で 3.4 倍高いことが観察される。このような性能差が出る原因として，次の 2 つが考えられる。

- GC のオーバーヘッド
- トランザクション中のスリープ時間および操作数

そこでこれらのスループットに対する影響を実験的に調査した。これらの影響を各々 4.4 節と 4.6 節で述べる。

4.4 GC のオーバーヘッドの影響

Version Chain には回収できない Version が存在する。GC におけるこれらへのアクセスは無駄な処理であり，オーバーヘッドである。EPO-R が EPO に優れる理由は，このオーバーヘッドが少ないことである。なぜなら SI の特性により，Read 起動は Write 起動と比して，この無駄な処理を削減できるからである。図 4 は，図 3 における，EPO での回収不能 Version 数を表す。これは最大で 1 秒間に約 2 千万件だった。

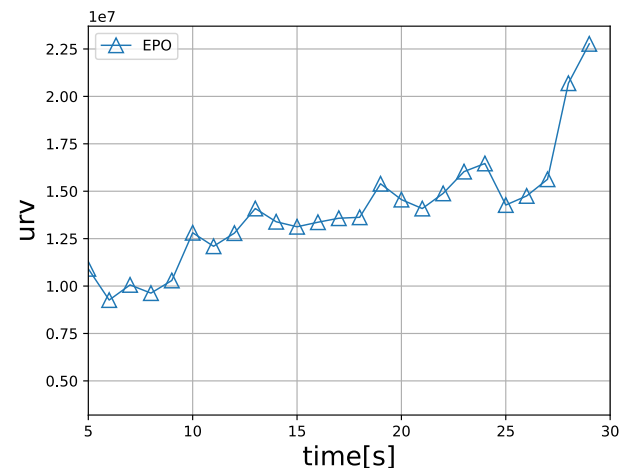


図 4: Write 起動における回収不能 Version 数

4.5 スリープ時間の変化

図 5 はスリープ時間を変えたときのスループットである。便宜上，Short は 224 スレッドのうち三分の一を，残りを Long と表す。ロングのスリープ時間を，図 5a では 0 s，図 5b では 1 s，図 5c では 10 s に変化させた。

予測通り，図 5a では，スリープ時間が均一で，「長さの著しく異なるトランザクションが混在したワークロード」でないため，EPO と EPO-R には差が見られない。一方，図 5b と図 5c を比較すると，スリープ時間が長くなるにつれて，EPO-R が EPO よりもスループットが高くなる傾向が観察された。Long のスリープ時間のみが増加することで，「長さの著しく異なるトランザクションが混在したワークロード」となった。加えて，高競合下では Traversal 量が増加する。さらに，図 4 で見たように，回収不能な Version が多数存在する。このような環境では，Read 時トリガによ

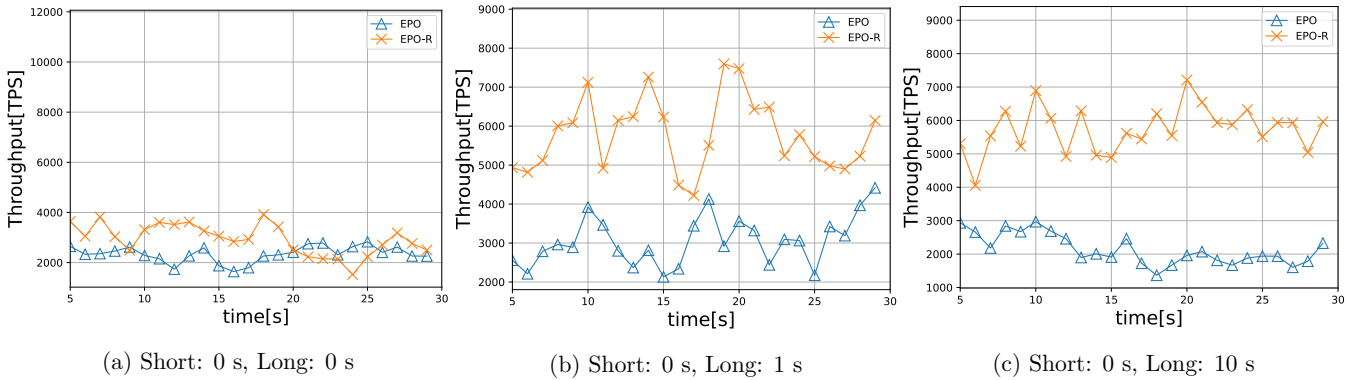


図 5: スリープ時間を変化させたときのスループット比較

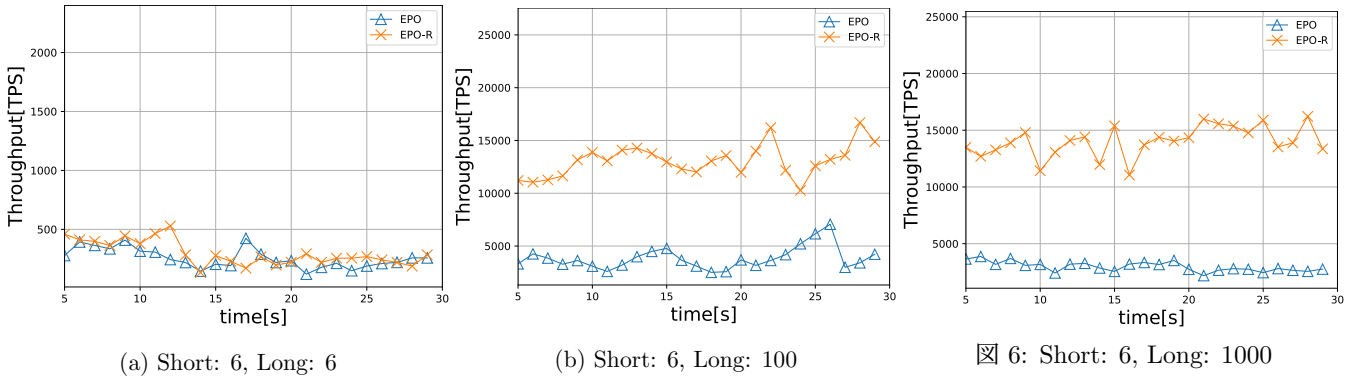


図 7: 操作数を変化させたときのスループット比較

て、Write 時トリガの GC による CPU 資源の無駄を抑制できたため 2 手法のスループットに差異が生じたと考えられる。

4.6 操作数の変化

操作数を変化させたときのスループットを図 7 に示す。ロングの操作数を、図 6a では 0、図 6b では 1、図 6c では 10 に変化した。

予測通り、図 6a では、操作数が均一で、「長さの著しく異なるトランザクションが混在したワークロード」とはならないため EPO と EPO-R の差が見られない。一方、図 6b と図 6c を比較すると、操作数が増えるにつれて、提案手法の方がスループットが高くなる傾向が観察される。Long の操作数のみが増加することで、「長さの著しく異なるトランザクションが混在したワークロード」となった。加えて、高競合下では Traversal 量が増加する。さらに、図 4 で見たように、回収不能な Version が多数存在する。このような環境では、Read 時トリガによって、Write 時トリガの GC による CPU 資源の無駄を抑制できたため 2 手法のスループットに差異が生じたと考えられる。

5. 関連研究

言語処理系用の GC 機構は、到達可能性 (Reachability) によってオブジェクトの不要性を判定する。一方で、MVCC 用の GC 機構では、バージョンの可視性 (Visibility) に基

づいて、不要性を判定する。Visibility の定義は、MVCC プロトコルによって異なる。SI では、アクセスした Version Chain のうち、トランザクション開始時に取得したタイムスタンプ未満の中で、最大の値を持つ Version がそのトランザクションにとっての Visible Version となる。従って、アクティブかつ未来のトランザクションから Visible でないバージョンは全て不要と判断し回収することができる。MVCC 用の GC 機構の実装としては、Deuteronomy [2], Hekaton [5], Cicada [6] があり、これら全て Version の不要性判定に最低見積り AOT 値に基づき回収を行なっている。一方で、SAP HANA に搭載されている Interval-Based GC や、EPO では、アクティブなトランザクションを追跡し、任意のトランザクションから Visible な Version を特定することで、より多くの Visible でない Version を回収を行う。

Version 不要性判定の方式の違いに加え、GC 起動のトリガも GC 機構によって異なる。Deuteronomy では、ソフトとハードの二種類のテーブルサイズの閾値によって、GC を起動する。HyPer [3] ではコミット毎に GC を起動する。Cicada では 10 us 置きに各スレッドが静止状態を宣言し、全てのスレッドの宣言を確認次第リーダースレッドが GC 起動の合図を各スレッドに送る。

6. 結論

本論文では、長さが著しく異なるトランザクションが混

在するような環境において、EPOでのGCをRead時起動とする手法EPO-Rを提案した。EPO-RがEPOよりも優れた性能を示すことを実験的に示した。

MVCCにおけるGCは現実には即したワークロードや高負荷環境において、性能ボトルネックになり得る。GCの起動トリガやアルゴリズムを慎重に選択することで、高負荷環境におけるGCのオーバーヘッドのための性能劣化を抑制可能であることを示した。

7. 謝辞

この成果は、科研費JP19H04117ならびに国立研究開発法人新エネルギー・産業技術総合開発機構(NEDO)の委託業務(JPNP16007)の結果得られたものです。

参考文献

- [1] Kimura, H.: FOEDUS: OLTP Engine for a Thousand Cores and NVRAM, *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, New York, NY, USA, Association for Computing Machinery, p. 691–706 (online), DOI: 10.1145/2723372.2746480 (2015).
- [2] Levandoski, J., Lomet, D., Sengupta, S., Stutsman, R. and Wang, R.: High Performance Transactions in Deuteronomy, *Conference on Innovative Data Systems Research (CIDR 2015)*, (online), available from (<https://www.microsoft.com/en-us/research/publication/high-performance-transactions-in-deuteronomy/>) (2015).
- [3] Neumann, T., Mühlbauer, T. and Kemper, A.: Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems, *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, New York, NY, USA, Association for Computing Machinery, p. 677–689 (online), DOI: 10.1145/2723372.2749436 (2015).
- [4] Lee, J., Shin, H., Park, C. G., Ko, S., Noh, J., Chuh, Y., Stephan, W. and Han, W.-S.: Hybrid Garbage Collection for Multi-Version Concurrency Control in SAP HANA, *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, New York, NY, USA, Association for Computing Machinery, (online), DOI: 10.1145/2882903.2903734 (2016).
- [5] Diaconu, C., Freedman, C., Ismert, E., Larson, P.-A., Mittal, P., Stonecipher, R., Verma, N. and Zwilling, M.: Hekaton: SQL Server's Memory-Optimized OLTP Engine, *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, New York, NY, USA, Association for Computing Machinery, (online), DOI: 10.1145/2463676.2463710 (2013).
- [6] Lim, H., Kaminsky, M. and Andersen, D. G.: Cicada: Dependably Fast Multi-Core In-Memory Transactions, *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, New York, NY, USA, Association for Computing Machinery, (online), DOI: 10.1145/3035918.3064015 (2017).
- [7] Faleiro, J. M. and Abadi, D. J.: Rethinking Serializable Multiversion Concurrency Control, *Proc. VLDB Endow.*, Vol. 8, No. 11, p. 1190–1201 (online), DOI: 10.14778/2809974.2809981 (2015).
- [8] Böttcher, J., Leis, V., Neumann, T. and Kemper, A.:

Scalable Garbage Collection for In-Memory MVCC Systems, *Proc. VLDB Endow.*, Vol. 13, No. 2, p. 128–141 (online), DOI: 10.14778/3364324.3364328 (2019).

- [9] Wu, Y., Arulraj, J., Lin, J., Xian, R. and Pavlo, A.: An Empirical Evaluation of In-Memory Multi-Version Concurrency Control, *Proc. VLDB Endow.*, Vol. 10, No. 7, p. 781–792 (online), DOI: 10.14778/3067421.3067427 (2017).