並行実行制御手法 TicToc の調査

中村　泰大†　　川島　英之††　　建部　修見††

† 筑波大学 情報学群情報科学類　〒 305–8571 茨城県つくば市天王台 1 丁目 1–1
†† 筑波大学 計算科学研究センター　〒 305–8571 茨城県つくば市天王台 1 丁目 1–1
E-mail: †nakamura@hpcs.cs.tsukuba.ac.jp, ††{kawasima,tatebe}@cs.tsukuba.ac.jp

あらまし　SIGMOD'16 にて、並行実行制御手法 TicToc が提案された。それが報告と一貫した性能を示すのか調査を行った。3 つのマイクロベンチマークを作成し、それぞれ測定したところ、いずれもスケールアップを観察した。この時最大で、280 万 transaction/sec を達成することを観察した。また、Intel Xeon Phi Knights Landing 上で 256 スレッドまでスレッド数を変化させて測定した。この環境でもスケールアップが観察され、最大で 4000 万 transaction/sec を達成することを観察した。

キーワード　トランザクション、並行実行制御、データベースシステム

1. 序　　論

データベースシステムに複数のトランザクションが到着したとき、それらが及ぼす作用の結果がデータベースに非一貫性をもたらさぬよう、データベースシステムは並行実行制御を行う。

SIGMOD'16 では、2013 年に提案された Silo [1] に勝る並行実行制御手法が複数提案 [2], [3] された。それらの手法の中に TicToc [3] がある。TicToc は OCC [4] の弱点であるタイムスタンプ生成コスト [5] を削減し、Silo を上回る性能を発揮することが報告されている。本論文では、TicToc を再実装し、それが報告と一貫した性能を発揮するのか調査を行った。

本報告の構成は以下の通りである。2 節では TicToc とその再実装について述べる。3 節では評価方法と結果を述べる。4 節では評価を基に結論を述べる。

2. TicToc

OCC では、トランザクションのタイムスタンプを得るために共有メモリ上のカウンタを用い、それに対して排他制御や fetch-and-add オペレーションを発行していた。TicToc ではそのボトルネックをなくすために、トランザクションのタイムスタンプは全てワーカーが計算によって求める。

2.1 データ構造

TicToc のタプルは Write Timestamp (wts) と Read Timestamp (rts) を持つ。これらはそれぞれ、そのタプルの値が書き込まれたタイムスタンプと最後に読まれたタイムスタンプである。タプルが wts と rts を持つのは、タプルの値が有効な範囲を示すためである。あるタプル A を読み込んだ時に、タイムスタンプがそれぞれ wts = 3、rts = 6 だったとする。タプル A を読み込んだトランザクションのタイムスタンプが 5 だと算出された場合、タプル A の値はタイムスタンプ 3 から 6 の間で有効であるから、タイムスタンプが 5 の時もタプル A の値は有効だと判断できる。

TicToc は正しく有効な範囲を取得するために、wts と rts をアトミックに読み込む必要がある。そこで TicToc はそれらを 1 つの 64 bit word に格納して保管する。これを Timestamp word(TS_word) と呼ぶ。Timestamp word の内訳は以下に示すとおりである。

TS_word[63]: Lock bit (1 bit).
TS_word[62:48]: delta = rts - wts (15 bits).
TS_word[47:0]: wts (48 bits).

2.2 プロトコル

TicToc のプロトコルは OCC や Silo と同様に Read フェーズ、Validation フェーズ、Write フェーズの 3 つに分割される。トランザクション処理の流れを Algorithm 1 に示す。はじめに Read フェーズを実行し操作の対象となるタプルをワーカーにコピーする。コピーしたタプルに対してオペレーションを実行する。オペレーションの結果が他のワーカーと整合性を保っているかを Validation フェーズで検証する。整合性を保っていることが確認されたのち、Write フェーズでデータの変更をデータベースに反映する。不整合が検知された場合は Read フェーズから再度実行する。

Algorithm 1 execute transaction

```
   Retry point:
1: read_phase()
2: do_operation()
3: if validation_phase() is failure then
4:     abort()
5:     retry()
6: end if
7: write_phase()
```

2.2.1 Read フェーズ

Read フェーズのアルゴリズムを Algorithm 2 に示す。TicToc はタイムスタンプと値の正しい組を得る必要がある。そこで、TicToc はそれらをアトミックに読み込むために TS_word を 2 回読み込む (Algorithm 2 - line 2〜4)。そして TS_word

に変更がなかった場合に、タイムスタンプと値の正しい組を取得できたとし、ワーカースレッドのローカル領域にタプルをコピーする (Algorithm 2 - line 6〜7)。タプルがロックされている場合は、タプルへの操作が中途半端な状態であり、タイムスタンプと値の正しい組を得られない可能性がある (Algorithm 2 - line 5)。従って、ロックが解除されるまで読み込みを繰り返す。

Algorithm 2 Read Phase ([3] より引用)

Data: read set entry r, tuple t

1: **do**
2: $v1 \leftarrow t$.read_ts_word()
3: read($r.read, t.data$)
4: $v2 \leftarrow t$.read_ts_word()
5: **while** $v1 \neq v2$ or $v1.lock\_bit == 1$
6: $r.wts \leftarrow v1.wts$
7: $r.rts \leftarrow v1.wts + v1.delta$

2.2.2 Validation フェーズ

Validation フェーズのアルゴリズム及びタイムスタンプの算出を Algorithm 3 に示す。タイムスタンプの算出には、Read フェーズでワーカーのローカル領域にコピーしたタプル（ローカルタプル）を用いる。タイムスタンプは、Read セットにあるローカルタプルの wts と Write セットにあるタプルの rts + 1 で最大のものとなる (Algorithm 3 - line 4〜11)。

算出したタイムスタンプを元に、トランザクションが有効であるか、以下のように確認を行う (Algorithm 3 - line 12〜20)。算出したタイムスタンプが、Read セットにあるローカルタプルの rts 以下である場合 (Algorithm 3 - line 13)、wts から rts に間そのタプルは値が変わらないことが保証されているため不整合は起こらない。従って、ローカルタプルを確認するだけでよく、データベース上のタプルに再度アクセスする必要はない。タイムスタンプが rts を超えている場合、データベース上にあるタプルの wts とローカルタプルの wts が等しいか確認する (Algorithm 3 - line 14)。等しい場合、タプルの値は Read フェーズでコピーしてから変更されていないため不整合は発生していない。この時、コミットタイムスタンプまで値が変わらないことを保証するために、タプルの rts をタイムスタンプまで延長する (Algorithm 3 - line 17)。rts の延長には compare-and-swap 命令を用いる。wts が異なる場合、別のワーカースレッドがタプルを変更しているため、このトランザクションをアボートし、トランザクションを再実行する (Algorithm 3 - line 15)。Write セットにあるタプルは、Validation フェーズの初めにロックを獲得するため、確認を行わなくて良い。

2.2.3 Write フェーズ

Write フェーズのアルゴリズムを Algorithm 4 に示す。Validation フェーズを通過し、不整合がないと判断されたトランザクションが Write フェーズを実行する。トランザクションはタプルの値を新しい値に、wts と rts を算出したコミットタイムスタンプに更新する。その後トランザクションは、タプルのロックを解放する。

Algorithm 3 Validation Phase ([3] より引用)

Data: read set RS, write set WS

1: **for** w in sorted(WS) **do**
2: lock($w.tuple$)
3: **end for**
4: $commit\_ts \leftarrow 0$
5: **for** e in $WS \cup RS$ **do**
6: **if** e in WS **then**
7: $commit\_ts \leftarrow \max(commit\_ts, e.tuple.rts + 1)$
8: **else**
9: $commit\_ts \leftarrow \max(commit\_ts, e.wts)$
10: **end if**
11: **end for**
12: **for** r in RS **do**
13: **if** $r.rts < commit\_ts$ **then**
14: **if** $w.wts \neq r.tuple.wts$ or ($r.tuple.rts \leqq commit\_ts$ and isLocked($r.tuple$) and $r.tuple$ not in W) **then**
15: abort()
16: **else**
17: $r.tuple.rts \leftarrow \max(commit\_ts, r.tuple.rts)$
18: **end if**
19: **end if**
20: **end for**

Algorithm 4 Write Phase ([3] より引用)

Data: write set WS, commit timestamp $commit\_ts$

1: **for** w in WS **do**
2: write($w.tuple.value, w.value$)
3: $w.tuple.wts \leftarrow w.tuple.rts \leftarrow commit\_ts$
4: unlock($w.tuple$)
5: **end for**

2.3 再 実 装

提案論文をもとに、C++を用いて TicToc の実装を行った。また、比較対象としてナイーブな OCC の実装を行った。

3. 評 価

再実装した TicToc および OCC についてスレッド数を変化させながらスループットを測定した。ワークロード中の各トランザクションはオペレーションを 10 件含む。データベースの初期タプル数は 1 万件であり、オペレーションの対象となるタプルはランダムに選択される。

ワークロードは 3 種類用意した。一つは全て Read トランザクションであるワークロード（Read only）、一つは Read トランザクションと Write トランザクションが半数ずつあるワークロード（Even)、最後に全て Write トランザクションであるワークロード（Write）である。

3.1 マルチコア環境

評価環境を表 1 に、評価結果を図 1 に示す。ワーカー数を 1 スレッドから 24 スレッドまで変化させて、3 つのワークロードをそれぞれ実行した。トランザクション数は全体で 100 万件とした。図 1 から、TicToc はどのワークロードでも線形にス

ループットが向上していることがわかる。また、TicToc はどのワークロードでも OCC に比べて良いスループットを示していることがわかる。TicToc がスケールアップするのは、TicToc では共有カウンタのボトルネックが排除されているためである。

3.2 メニーコア環境

評価環境を表 2 に、評価結果を図 2 に示す。ワーカー数を 1 スレッドから 256 スレッドまで変化させて、Read only ワークロードと Even ワークロードを実行した。トランザクション数は、ワーカーごとに 200 万件とした。図 2 から、どちらのワークロードでもスケールアップしていることがわかる。Read only では、65 スレッド、129 スレッド、および 193 スレッド時に大きく性能が劣化している。これは、各コアで動くスレッド数が均一でなくなることでロードインバランスが起こっていることが原因であると考えられる。

表 1　マルチコアの評価環境

プロセッサ	Intel(R) Xeon(R) CPU E5-2695 v2 @ 2.40GHz
コア	24
メモリ	64GB
OS	CentOS release 6.8 (Final)

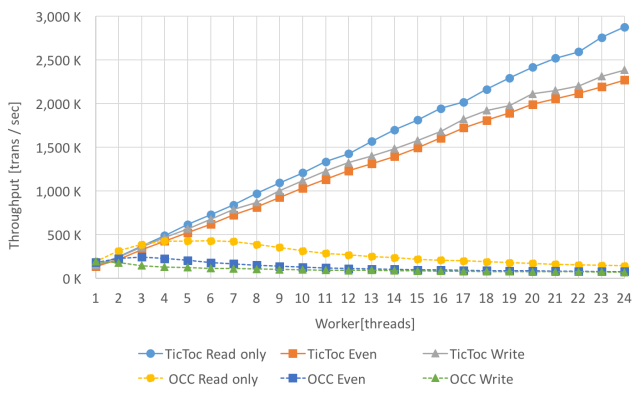

図 1　マルチコア環境での各ワークロードのスループット

表 2　メニーコアの評価環境

プロセッサ	Intel(R) Xeon Phi(TM) CPU 7210 @ 1.30GHz
コア	64 (256 スレッド)
メモリ	MCDRAM: 16 GB DRAM: 96 GB
OS	CentOS Linux release 7.2.1511 (Core)

4. 結　　論

TicToc はマルチコア環境とメニーコア環境のどちらにおいてもスケールする結果が得られた。ピーク性能として 4000 万 transaction/sec (tps) を達成できることが観察された。

原論文 [3] では永続化については具体的に検討されておらず、本論文でも永続化については考慮していない。トランザクション処理において、永続化は必要不可欠 [6] [7] である。今後は TicToc に永続化処理を施す手法を探求する。

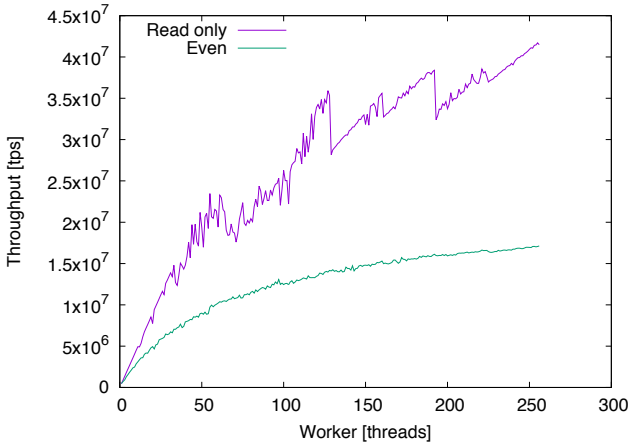

図 2　メニーコア環境でのワークロードのスループット

謝　　辞

本研究の一部は、JST CREST「ポストペタスケールデータインテンシブサイエンスのためのシステムソフトウェア」、JST CREST「EBD:次世代の年ヨッタバイト処理に向けたエクストリームビッグデータの基盤技術」、JST CREST「広域撮像探査観測のビッグデータ分析による統計計算物理学」、科研費「#16K00150」による。

文　　献

[1] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pp. 18–32, New York, NY, USA, 2013. ACM.

[2] Kangnyeon Kim, Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. Ermia: Fast memory-optimized database system for heterogeneous workloads. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pp. 1675–1687, New York, NY, USA, 2016. ACM.

[3] Xiangyao Yu, Andrew Pavlo, Daniel Sanchez, and Srinivas Devadas. Tictoc: Time traveling optimistic concurrency control. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pp. 1629–1642, New York, NY, USA, 2016. ACM.

[4] H. T. Kung and John T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, Vol. 6, No. 2, pp. 213–226, June 1981.

[5] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. Staring into the abyss: An evaluation of concurrency control with one thousand cores. *Proc. VLDB Endow.*, Vol. 8, No. 3, pp. 209–220, November 2014.

[6] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, Vol. 17, No. 1, pp. 94–162, March 1992.

[7] Jim Gray. The transaction concept: Virtues and limitations (invited paper). In *Proceedings of the Seventh International Conference on Very Large Data Bases - Volume 7*, VLDB '81, pp. 144–154. VLDB Endowment, 1981.