

# 分散KVSにおけるアクセス頻度を考慮した コンパクション頻度の動的変更

川浪 大知<sup>1</sup> 川島 龍太<sup>1</sup> 松尾 啓志<sup>1</sup>

**概要:** ビッグデータの普及により、大規模なデータの保存や高速な検索処理が可能な分散キーバリューストアが広く用いられている。なかでも、書き込み性能に優れたストレージ構造として LSM-tree が知られており、Cassandra などのキーバリューストアで用いられている。LSM-tree では定期的に SSTable のコンパクションを実行することで高速な読み出し処理を実現しているが、コンパクション実行中はディスクアクセスが増加するため、頻繁にコンパクションを行うと性能が低下してしまう。本稿ではデータのアクセス頻度の偏りに注目し、アクセス頻度の高いデータに対するコンパクションを優先して行うことで、負荷を抑えつつ、より効果的なコンパクションを行う手法を提案する。提案手法を Apache Cassandra に実装し、アクセス頻度の偏りに Zipf 分布を用いて評価を行った結果、提案手法では SSTable の読み出し数が削減され、平均スループットが最大 7% 向上することが確認された。

**キーワード:** 分散キーバリューストア, コンパクション, アクセス頻度, Cassandra

## 1. はじめに

SNS やクラウドコンピューティングの普及により、大量のデータを高速に扱うストレージシステムが求められている。そこで、スケールアウトによって性能向上が可能な、分散キーバリューストアが注目されている。

書き込み性能を重視した分散キーバリューストアでは Log-Structured Merge Tree (LSM-Tree)[1] と呼ばれるストレージ構造が広く使われている。LSM-Tree を採用している Apache Cassandra[2] において、ストレージは主記憶上の Memtable と、二次記憶上の Sorted String Table (SSTable) の 2 種類から構成される。データの書き込みはまず Memtable に行われ、Memtable のサイズがしきい値を超えると、新規の SSTable として二次記憶上に書き出すフラッシュ処理が行われる。この時、既存の SSTable との間でキーが重複する場合がある。読み出しの際は、要求のあったキーを持つ全ての SSTable を参照するため、SSTable の増加に伴って読み出し性能が低下する。そのため、複数の SSTable をマージしてキーの重複を排除するコンパクション処理が必要になる。しかし、コンパクションの実行中はディスクアクセスの増加によって性能が低下するため [3]、性能向上が見込めるデータに対して優先的にコンパ

クションを行うことが望ましい。

本稿では、キーに対するアクセス頻度の偏りに注目し、アクセス頻度の高いキーを持つ SSTable のコンパクションを優先することで、コンパクションによる負荷を抑えつつ読み出し性能を向上する手法を提案し、Cassandra に実装した。本手法では、キーに対する読み出し要求の数からキーごとのアクセス頻度を求め、フラッシュ時にアクセス頻度の高いキーと低いキーをそれぞれ別の SSTable に分ける。そして、アクセス頻度の高いキーを持つ SSTable のコンパクションを優先して行うことでコンパクションによる I/O 負荷を抑制しつつ、アクセス頻度が高いキーの重複を排除する。

本稿の構成は以下のとおりである。まず、第 2 章では研究背景として、提案手法を実装した分散キーバリューストアである Apache Cassandra の動作とその問題点について説明する。次に第 3 章で提案手法とその実装について説明する。第 4 章で従来手法と提案手法の性能比較と考察を行う。第 5 章で関連研究について述べ、第 6 章でまとめと今後の課題を述べる。

## 2. Apache Cassandra

Apache Cassandra は Facebook によって開発されたオープンソースの分散キーバリューストアである。Amazon Dynamo[4] の分散デザインと Google Bigtable[5] のデータ

<sup>1</sup> 名古屋工業大学大学院  
Nagoya Institute of Technology

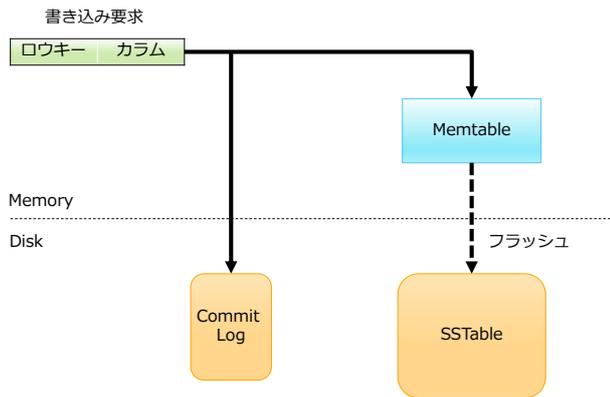


図 1 書き込み処理

モデルを併せ持った設計となっている。Cassandra はストレージエンジンに LSM-Tree 構造を採用をしているため、書き込み性能に優れている。加えて、複数のマシンによる分散処理が可能であり、スケーラビリティ、高可用性、耐障害性を備えている。

## 2.1 書き込み処理

Cassandra における書き込み処理の流れを図 1 に示す。クライアントはロウキーとカラムを指定して書き込み要求を行う。ロウキーとカラムはまず二次記憶上のコミットログに書き込まれる。コミットログへの書き込みが完了したロウは主記憶上の Memtable に書き込まれる。書き込み対象のロウキーが Memtable に存在する場合はデータを上書きし、存在しない場合は Memtable 上に新しく領域を確保する。Memtable のサイズがしきい値を超えると、Memtable は SStable として二次記憶上にフラッシュされる。SStable は、ロウの集合、インデックス、ブルームフィルタ [6] で構成されている。ロウの探索を容易にするため、ロウの集合はフラッシュの際にロウキーでソートされて書き込まれ、この時、ロウの位置に対応するインデックスも作成される。さらにロウの存在確認用のブルームフィルタも同時に作成される。Cassandra ではフラッシュの度に新たな SStable が作成され、既存の SStable に対して上書きを行わないため高速な書き込み処理が可能である。

## 2.2 読み出し処理

次に読み出し処理について説明する。図 2 に示す通り、クライアントはロウキーとカラム名を指定して、カラムの取得を行う。読み出し操作が要求されると Memtable、SStable の順に読み出し対象となるカラムの探索を行う。Memtable にロウキーが存在しない場合や、取り出したロウに要求されたカラムが含まれていない場合は、ロウキャッシュを確認し、カラムが存在しなければ、各 SStable のブルームフィルタを確認して対象カラムの読み出しを行う。この時、複数の SStable に対象カラムが存在する場合はタイムスタンプを元に最新のカラムを選択し、クライアント

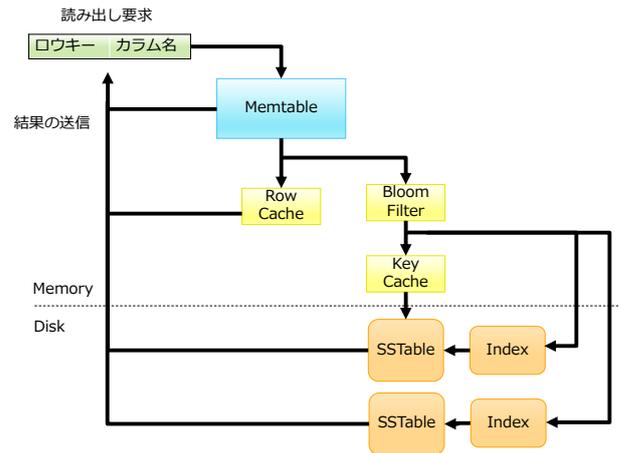


図 2 読み出し処理

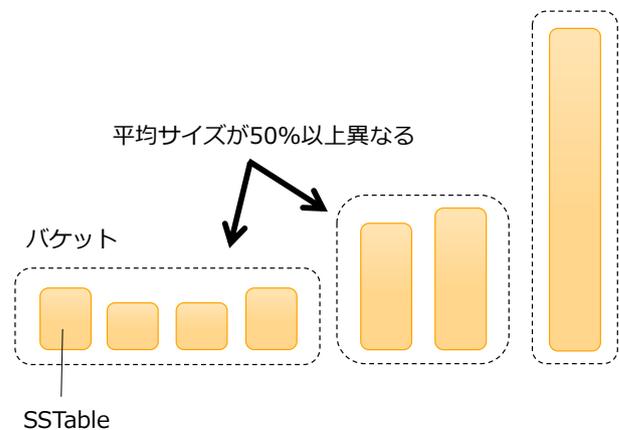


図 3 バケット分類

に応答する。

## 2.3 コンパクション

Cassandra ではフラッシュの度に SStable が作成されるため、同一のロウキーが複数の SStable に存在する。これはディスク使用量の増加や、読み出し性能低下の原因となるため Cassandra では複数の SStable を統合するコンパクションをバックグラウンドで定期的に行う。コンパクションが行われると、複数の SStable から一つの新しい SStable が生成されロウキーの重複が排除される。

### 2.3.1 コンパクション対象の選択

SStable のサイズが大きくなるほどコンパクションの実行時間が長くなるため、サイズの大きな SStable のコンパクションは頻繁に行わないほうが良い。そこで、Cassandra はコンパクションを行う際に図 3 のように、複数の SStable をバケットと呼ばれる組に分類する。これは、サイズの近い SStable を同じバケットに入れてバケット単位のコンパクションを行うことで、サイズの大きな SStable が頻繁にコンパクションされることを防止する役割がある。具体的には選択した SStable のサイズと、バケット内の SStable の平均サイズと比較し、差が 50%以内であれば対

象の SSTable をそのバケットに入れ、条件を満たすバケットが存在しない場合は、新しいバケットを生成する。バケット内の SSTable 数が一定数を超えるとコンパクションの対象となるが、対象となるバケットが複数存在する場合は、その中から 1 キーあたりのアクセス頻度が最も高いバケットのみをコンパクションする。

### 2.3.2 コンパクションにおける問題点

上記のコンパクション手法では、あらかじめバケットに分類した後でアクセス頻度を考慮したコンパクション対象の選択が行われる。アクセス頻度の高いロウは、Memtable に存在する可能性が高く、フラッシュされた SSTable やコンパクションで生成された SSTable のほとんどにアクセス頻度の高いロウが存在する。その一方で、アクセス頻度の高いロウがサイズの異なる SSTable に存在する場合、それらが異なるバケットに分類されてコンパクションが行われるため、コンパクション後もアクセス頻度の高いロウキーがサイズの異なる SSTable 間に重複して存在する。その結果、ロウの読み出し時間の短縮につながらないという問題が発生する。また、コンパクションを行う際には I/O が発生するため、コンパクション中のリクエストの処理性能は低下する。そのため、頻繁なコンパクションや、サイズの異なる SSTable のマージは、コンパクションの実行時間の増加につながり、結果として処理性能が低下しまう。

## 3. 提案手法

本章では、アクセス頻度に基づいたコンパクション頻度の動的な変更を提案し、その実装について述べる。提案手法では、ロウキーのアクセス頻度に応じて SSTable を 2 つのグループに分類し、それぞれ異なる頻度でコンパクションを行う。アクセス頻度の高い SSTable のコンパクション頻度を上げることで、アクセス頻度の高いロウキーの読み出し操作は単一の SSTable の読み出しだけで済む。一方でアクセス頻度の低い SSTable のコンパクション頻度を下げることで、コンパクションによる I/O 負荷を緩和させる。従来のフラッシュの動作では、Memtable 上にアクセス頻度の高いロウキーと低いロウキーが存在した場合、同じ SSTable に格納されてしまうため、アクセス頻度の高いロウキーとアクセス頻度の低いロウキーが同一の SSTable に格納されてしまう。そのため、図 4 に示すように、提案手法ではフラッシュの段階で Memtable を、アクセス頻度の高いロウキーを持つ SSTable とアクセス頻度の低い SSTable に分離する。

コンパクションを行う際は、バケットの分類を SSTable のアクセス頻度に基づいて行い、アクセス頻度の近い SSTable 同士が統合されるようにする。アクセス頻度の高いロウキーを持つ SSTable は SSTable 数が少ない場合でもコンパクションを行い、アクセス頻度の低いロウキーを持つ SSTable は多数集まるまでコンパクションを遅らせる。こ

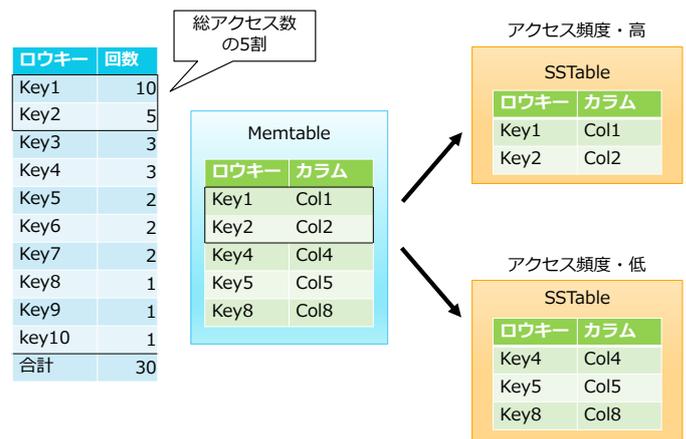


図 4 提案手法のフラッシュ動作図

のようにコンパクションすることでアクセス頻度の高いロウキーは重複が発生しても即座に統合されるため、読み出し時に複数の SSTable を参照する可能性は低下する。

### 3.1 実装

提案手法の実装を Apache Cassandra 3.11 に行った。まず、Memtable 内のロウキーを、アクセス頻度の高いものと低いものに分類する必要がある。そのため、ロウキーごとにアクセス回数を記録するカウンタを用意し、読み出しや書き込み操作が行われた際にカウントする。フラッシュ時はこのカウンタの値を元にアクセス頻度の分類を行う。アクセス回数のカウンタにはロウキーからカウント値へマップを行う連想配列を用い、ロウキーに対する読み出しリクエストの度にカウントを行う。

フラッシュが実行される際、アクセス頻度の高いロウキーを抽出するために、連想配列をカウンタ値の降順でソートする。そしてカウンタ値の多いものから順に合計し、フラッシュ時点での全てのカウンタを合計した値の k 割を占める部分までのロウキーをアクセス頻度の高いロウキーとし、アクセス頻度の高いロウキーのリストを作成する。リストを元に Memtable からアクセス頻度が高いロウキーを SSTable にフラッシュし、その後フラッシュされていないロウキーをアクセス頻度の低いものとして異なる SSTable にフラッシュする。また、アクセス頻度の高い SSTable、アクセス頻度の低い SSTable を区別するためのフラグ SSTable 毎に付けておき、バケットの分類時に使用する。

従来のコンパクションの動作ではバケット分類後にアクセス頻度の高い SSTable を持つバケットをコンパクション対象としている。SSTable 毎にはカウンタが存在し、その SSTable から読み出しが行われる度にインクリメントされる。従来の Cassandra ではこのカウンタを元に SSTable のアクセス頻度を識別している。しかしこの手法では、実際に SSTable がアクセスされるまではアクセス頻度が低い

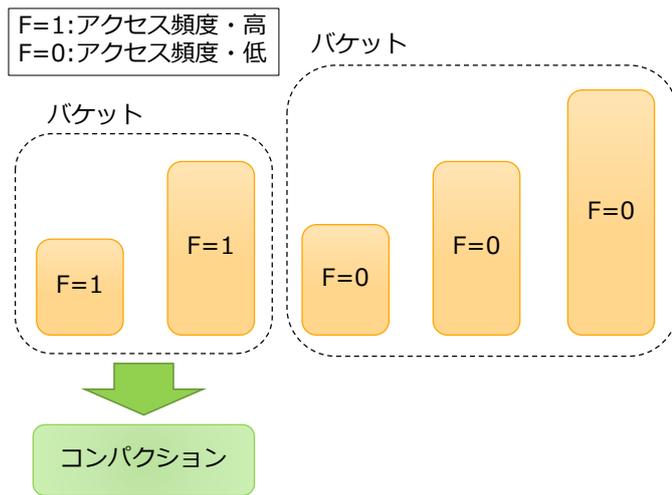


図 5 提案手法におけるコンパクション

SSTable だと識別されてしまうため、提案手法ではフラッシュ時に付与した SSTable のアクセス頻度を表すフラグを用いて、アクセス頻度が高い SSTable のバケットを識別する。従来の Cassandra ではコンパクション候補にするバケット内の SSTable 数に下限を設定しており、バケットに含まれる SSTable 数が一定数をに達するまでコンパクションを行わない。提案手法ではアクセス頻度の高い SSTable, 低い SSTable のそれぞれに異なる下限値を適用することでアクセス頻度の高い SSTable のみがコンパクションされやすくなる。

#### 4. 評価実験

提案手法の有効性を確認するために従来の Cassandra と提案手法を実装した Cassandra に同一のワークロードを実行し、性能の変化を調査した。本章では実験内容と評価結果を示し、評価結果の考察を行う。

##### 4.1 実験内容

評価実験では NoSQL 向けのベンチマークソフトである Yahoo! Cloud Serving Benchmark (YCSB)[7] を用いた。使用した計算機の性能を表 1 に示す。クラスタ構成として、Cassandra を動作させる物理計算機を 3 台、YCSB を動作させる物理計算機を 1 台とした。YCSB はリクエスト毎にいずれかの Cassandra ノードを選択し、リクエストを送信する。レプリカ数は 3 とし、Cassandra ノードはいずれか 1 台のノードにデータを書き終えた段階でクライアントに応答を返す。実行したワークロードの設定を表 2 に示す。提案手法ではアクセス頻度の高いキーが繰り返し書き込まれ、多数の SSTable から読み出される状況で性能の向上が見込まれる。そのため Memtable のフラッシュを行うサイズのしきい値を小さく設定することで、SSTable が多数生成させるようにし、キーの重複を発生しやすくした。ロウを、最大 24 Byte のロウキーと 100 byte のカラムを

表 1 計算機の性能

CPU	Intel Core i5-4460 3.2 GHz (4 cores)
Memory	16 GB
Network	1 Gbps
HDD	500 GB, 1 台
OS	Ubuntu 16.04

表 2 ワークロード設定

ロウ数	20,000,000
ロウキーサイズ	最大で 24byte
カラムサイズ	100 B × 10 計 1 KB
オペレーション数	15,000,000
Read:Update 比	50% : 50%

表 3 Cassandra 設定

JVM ヒープサイズ	4 GB
Memtable サイズ	128 MB
レプリケーション数	3

10 個持つ設定とした。初期状態として、20,000,000 ロウをあらかじめ挿入しておき、単一の SSTable として用意した。既存手法と提案手法のどちらでもこの SSTable は他の SSTable と統合されないようにした。ワークロードとして、合計 15,000,000 リクエストを Read:Update 比を 50%:50%にして行った。アクセス頻度の偏りには Zipf 分布を用い、偏り具合を制御するパラメータとして、偏りの大きい  $s=0.99$  と偏りの小さい  $s=0.75$  を用いた。コンパクションを行う SSTable 数の下限値として、既存手法ではデフォルトの 4、提案手法ではアクセス頻度の高い SSTable に対しては 2、アクセス頻度の低い SSTable に対しては 6 を設定した。提案手法では、アクセス頻度の高いキーの分類に用いる割合のパラメータ  $k$  を 0.3, 0.5 の 2 パターンで評価を行った。

評価項目として、平均スループット、平均レイテンシを測定し、頻度の高いカラムに対する SSTable の読み出し数が削減されているかを確認するためにアクセス頻度毎の平均 SSTable 読み出し数を測定した。

##### 4.2 評価結果

アクセス頻度の偏りが最も大きい  $s=0.99$  の場合の平均スループットの時間変化を図 6 に示す。縦軸が単位時間あたりのリクエスト処理量で横軸はワークロードの経過時間を表している。1 リクエストの応答のために読み込まれた平均 SSTable 数の時間変化を図 7 に示す。ワークロードの開始から 2000 秒にかけては SSTable がキャッシュされるためスループットが増加した。その後スループットが徐々に低下し、ワークロードの終了時に 0 となった。従来手法と提案手法 ( $k=0.5$ ) のスループットは同程度を示し、提案手法 ( $k=0.3$ ) は従来手法よりスループットが平均 15% 低下した。平均 SSTable 読み出し数は提案手法のほうが少な

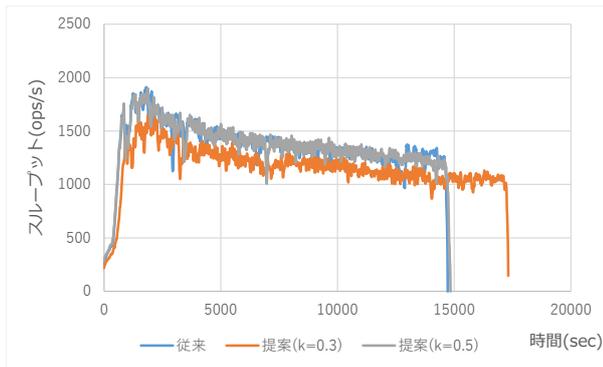


図 6 平均スループット (s=0.99)

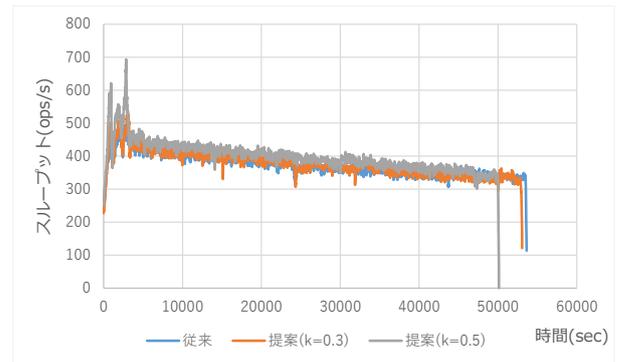


図 9 平均スループット (s=0.90)

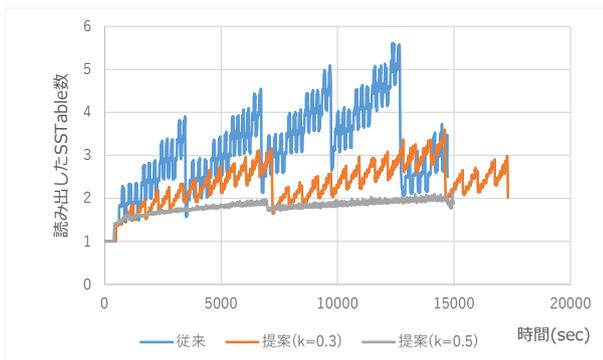


図 7 1 リクエストの平均 SSTable 読み出し数 (s=0.99)

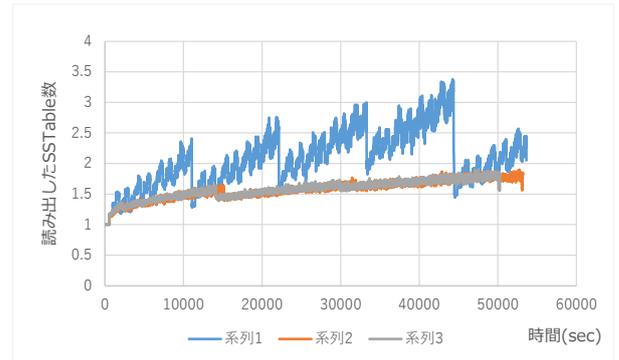


図 10 1 リクエストの平均 SSTable 読み出し数 (s=0.90)

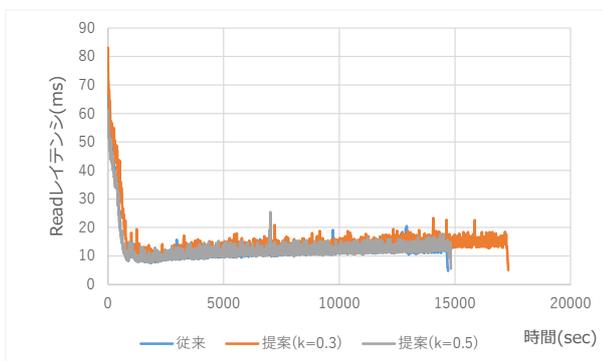


図 8 平均 Read レイテンシ (s=0.99)

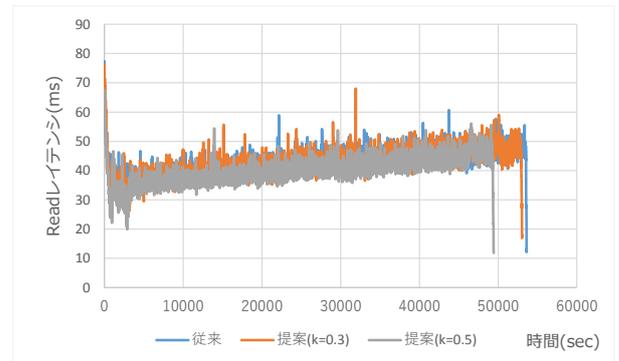


図 11 平均 Read レイテンシ (s=0.90)

い傾向にあり、 $k$  の値が大きいほど SSTable の読み出し数が削減された。また各手法における Read レイテンシの時間変化を計測すると図 8 に示す結果となり、いずれの手法も 2000 秒以降は、ほぼ一定のレイテンシを示した。提案手法 ( $k=0.3$ ) において読み出す SSTable 数が削減されているにもかかわらずスループットが低下した原因として、ワークロードの実行中に書き込まれたデータがディスクキャッシュに格納されたことが挙げられる。SSTable からの読み出しにディスクアクセスを伴わなかったため、Read レイテンシは一定となり、コンパクションによってスループットが向上しなかったと考えられる。アクセス頻度の偏りが小さくなった  $s=0.90$  でのスループットの変化は図 9、SSTable の読み出し数の変化は図 10、レイテンシは図 11

となった。スループットは提案手法 ( $k=0.5$ ) の場合に最大となり、従来手法と比較すると平均スループットは 7% 向上した。これはアクセス頻度の偏りが小さくなったことで、アクセス頻度の高い SSTable がキャッシュに乗りづらくなったため  $k=0.5$  ではキャッシュに乗らなかった SSTable がコンパクションされたためと考えられる。よりアクセス頻度の偏りが小さい  $s=0.75$  では、スループットの変化は図 12 となり、提案手法 ( $k=0.3$ ) は従来手法と比較して平均スループットは 4% 向上した。

## 5. 関連研究

Bigtable, Cassandra, HBase[8], LevelDB[9] といった多くのデータベースで LSM-tree が採用されている。重複し

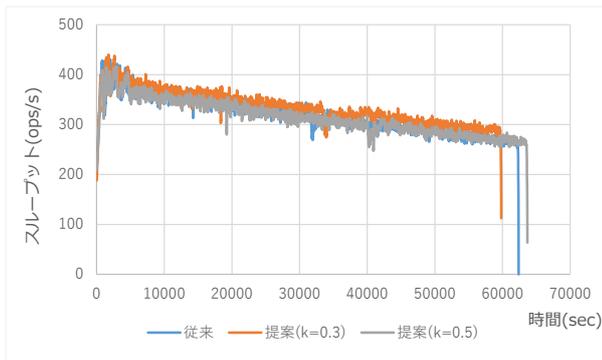


図 12 平均スループット (s=0.75)

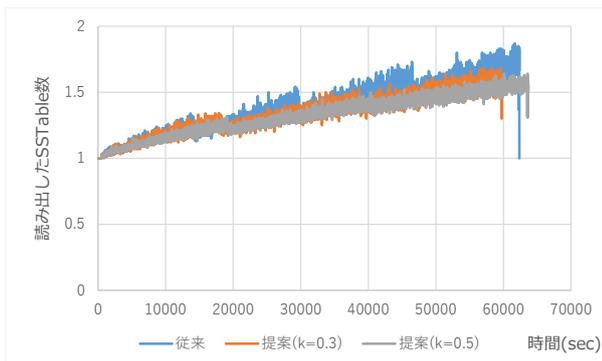


図 13 1 リクエストの平均 SSTable 読み出し数 (s=0.75)

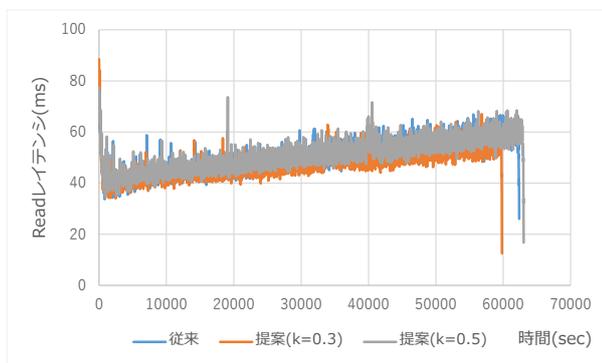


図 14 平均 Read レイテンシ (s=0.75)

たデータを取り除くためにコンパクションは必須であるが、コンパクションによる I/O 負荷は大きいので、コンパクションによる負荷を低減させる手法が考案されている。HBase や LevelDB では Leveled Compaction を使用しており、Cassandra でも選択可能である。Leveled Compaction では SSTable をレベルと呼ばれる複数の階層に分け、同一レベル内では異なるキーレンジを持つ複数の SSTable に分けている。コンパクションは隣接するレベル間でキーのレンジが重複する SSTable だけを対象とするため、コンパクション時間は短くなる。アクセス頻度を考慮した SSTable の分類を行っていないので、アクセス頻度の低いキーもコンパクション対象となる可能性がある。文献 [3] では、コンパクションサーバを導入して規模の大きなコン

パクションをオフロードすることで、フォアグラウンドのサービスへの影響を軽減しているがコンパクション対象となる SSTable の選択方法には関与していない。[10] では、Cassandra のストレージに HDD と SSD を使用し、アクセス頻度の高いデータを SSD に保存することで性能の向上を図っているが、コンパクションの頻度を変更してはいない点が本手法と異なる。[11] では、単一検索と範囲検索における処理時間の差に着目し、高速に実行可能なクエリを優先的に実行するようクエリのスケジューリングを行うことで、平均応答時間を短縮しているが、コンパクションによる性能の低下は考慮していない。

## 6. まとめと今後の課題

本稿では分散キーバリューストア Apache Cassandra 上でアクセス頻度に応じて SSTable のコンパクション頻度を動的に変更する手法を提案した。フラッシュの際にアクセス頻度の高いロウキーを持つ SSTable と、アクセス頻度の低いロウキーを持つ SSTable に分離し、アクセス頻度の高い SSTable のコンパクション発生のしきい値を小さく設定することで、アクセス頻度の高い SSTable のコンパクションをより多く発生させる手法を提案した。アクセス頻度の偏りや Memtable を分割する割合を変化させて評価を行ったところ、提案手法で平均スループットが最大で 7% 向上することが確認された。

今後の課題としては、アクセス頻度の偏りが大きかった場合でもスループットが向上させるために、ディスクキャッシュを考慮してアクセス頻度による分類やコンパクションの頻度を決定することが挙げられる。また、今回の実験ではフラッシュ回数を多くするために Memtable のサイズを小さく設定したが、実用的な環境での効果を考慮した実験を行う必要がある。

謝辞 本研究の一部は、科研費基盤研究 (C)24500113, (C)15K00168 による。

## 参考文献

- [1] Patrick O'Neil, Cheng, E., Gawlick, D., Elizabeth O'Neil: The log-structured merge-tree (LSM-tree), *Acta Informatica*, Vol. 33, No. 4, pp. 351-385 (1996).
- [2] Lakshman, A. and Malik, P.: Cassandra: a decentralized structured storage system, *ACM SIGOPS Operating Systems Review(OSR)*, Vol. 44, No. 2, pp. 35-40 (April 2010).
- [3] Ahmad, M. Y. and Kemme, B.: Compaction Management in Distributed Key-value Datastores, *In Proc. VLDB Endow.*, Vol. 8, No. 8, pp. 850-861 (2015).
- [4] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P. and Vogels, W.: Dynamo: Amazon's Highly Available Key-value Store, *In Proc. Twenty-first ACM SIGOPS Symposium on Operating Systems Principles, SOSP '07*, ACM, pp. 205-220 (2007).
- [5] Chang, F., Dean, J., Ghemawat, S., Hsieh, W., Wallach,

- D., Burrows, M., Chandra, T., Fikes, A. and Gruber, R.: Bigtable: A distributed structured data storage system, *7th OSDI*, Vol. 26, pp. 305–314 (2006).
- [6] Bloom, B. H.: Space/Time Trade-offs in Hash Coding with Allowable Errors, *Commun. ACM*, Vol. 13, No. 7, pp. 422–426 (1970).
- [7] Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R. and Sears, R.: Benchmarking Cloud Serving Systems with YCSB, *In Proc. the 1st ACM Symposium on Cloud Computing*, SoCC '10, New York, NY, USA, ACM, pp. 143–154 (2010).
- [8] Apache HBase. <https://hbase.apache.org/>.
- [9] LevelDB. <http://leveldb.org/>.
- [10] 鴨下将成, 川島龍太, 松尾啓志: 分散キーバリューストアにおけるアクセス頻度を考慮した階層化ストレージ手法の提案, 技術報告 16, 名古屋工業大学, 名古屋工業大学, 名古屋工業大学 (2016).
- [11] 福田 諭, 川島龍太, 齋藤彰一, 松尾啓志: 検索範囲を考慮したクエリスケジューリングによる Cassandra の応答性能向上, 情報処理学会論文誌, Vol. 56, No. 2, pp. 492–502 (2015).