Vol.2017-DBS-165 No.16
Vol.2017-IFAT-128 No.16
2017/9/18

高速 I/O デバイスにおける LSM-Tree

佐藤 克矢[1,a)]　山田 浩史[1]

概要：KVS は高性能さやスケーラビリティの確保のしやすさから多くのサービスで利用されている．KVS の構成要素として *Log-Structured Merge Tree(LSM-Tree)* が広く利用されている．LSM-Tree は write-intensive なワークロードに適しており，近年のアクセスパターンに適しているためである．しかし，既存の LSM-Tree では PCIe SSD ような高速な I/O デバイスを有効に活用することはできない．LSM-Tree では Disk への書き込みはコンパクションをしながら行われるが，その処理の並列性が乏しいため，高速 I/O デバイスの帯域を活用することができないためである．そこで，本論文では PCIe SSD のような高速 I/O デバイスを想定し，LSM-Tree における I/O 帯域利用の高効率化手法を提案する．本論文では，Key の範囲や DB 全体に対してロックを取得せず，関連する各レベルのコンパクションをパイプライン実行する．提案手法を RocksDB(4.10.0) をベースとして実装を行い実験，評価を行った. 既存の LSM-Tree を用いた RocksDB と比較し提案手法において約 10%のスループットの向上を達成した．

キーワード：LSM-Tree, SSD

1. はじめに

Key-value store (KVS) は，その高性能さおよびスケーラビリティの確保のしやすさから，Web サービスを構築する上での必須のコンポーネントとなっている． [1], [9] 実際に Facebook や Amazon などの実際の Web サービスにおいて KVS は利用されている [4], [14]．たとえば，Facebook ではディスクベースのストレージへのアクセスをなるべく抑えるために，Memcached [13] や Redis [11] といった KVS をキャッシュサーバとして用いることで，インターネットからの大量のリクエストを低レイテンシで捌いている．同様に永続 DB としても利用されており，HBase [2], Cassandra [1] などが利用されている．

KVS の構成要素の一つとして，*Log-Structured Merge Tree (LSM-Tree)* が広く利用されている．RocksDB [6] や LevelDB [8] において，この構造が採用されている．LSM-Tree は write-intensive なワークロードにおいて高速に処理を行えるため，書き込みの多いアクセスにおいて適している．一般的に，Web サービスは読み込みリクエストが多かったものの [3]，近年，書き込みの割合が従来よりも増加している．Yahoo [16] によると書き込みの割合が約 50%に近くなってきている．LSM-Tree は書き込み処理を高速に行うために，ディスク書き込みがシーケンシャルになるように設計されている．新しいデータはメモリ上に保存され，一定以上のサイズになるとそれらをソート，マージして一括でディスクに書き込む．LSM-Tree はディスク上のデータを複数レベルが管理しており，あるレベルのサイズが閾値を超えると，次のレベルのファイル中にアイテムをソートしながらマージする．この処理を**コンパクション**と呼ぶ．これにより，メモリに比べて低速なディスク書き込みをシーケンシャルアクセスにすることでディスク書き込みのオーバヘッドを削減している．

しかし，既存の LSM-Tree では PCIe SSD ような高速な I/O デバイスを有効に活用することはできない．LSM-Tree では Disk への書き込みはコンパクションをしながら行われるが，その処理の並列性が乏しいため，高速 I/O デバイスの帯域を活用することができていない．具体的には，別ファイルに対するコンパクションの並列性は実現されているものの，同一 Key スペースに対するコンパクションの並列性は実現されておらず，Key の範囲や DB 全体に対してロックを取得しコンパクションを行っている．実際の KVS への書き込みワークロードは特定のアイテムの更新に偏る (Skew する) ことが知られており [17], [18], [19]，コンパクションが生じるファイルが同一 Key スペースの物になりやすい．そのため，PCIe SSD などの 数 GB/sec オーダーのアクセス帯域を有効に利用できない．それどころかコンパクションのリクエストが上位レベルで滞ってしまい，性能上のボトルネックとなってしまう．これにより，

[1] 東京農工大学
TUAT, Koganei, Tokyo 183–0054, Japan
[a)] katsuya_sato@asg.cs.tuat.ac.jp

スループットが頭打ちとなり，余分なリソースが必要になる．また，データセンターなどでは余分なハードウェアが必要になりコストや消費電力がかさんでしまうなどの問題が生じる．

そこで，本論文では PCIe SSD のような高速 I/O デバイスを想定し，LSM-Tree における I/O 帯域利用の高効率化手法を提案する．本論文では，Key の範囲や DB 全体に対してロックを取得せず，関連する各レベルのコンパクションをパイプライン実行する．これにより，Skew するワークロード化において，高速 I/O デバイスの帯域を活かしつつ，同一ファイル群に対するコンパクションのボトルネック解消を狙う．

本論文における貢献は以下の通りである．

- 既存の LSM-Tree に関して，高速 I/O デバイス上での帯域利用率が低いこと，および，Skew した書き込みワークロードにおけるコンパクションのボトルネックを明らかにした．
- 上記問題点を解消するべく，コンパクションをパイプライン実行する方式を提案した．具体的には，コンパクション時のファイル更新状況を把握し，安全なタイミングで下位レベルのコンパクションを実行する．
- 提案手法を RocksDB(4.10.0) をベースとして実装を行い実験，評価を行った. 既存の LSM-Tree を用いた RocksDB と比較する．Skewed なワークロードを実行し，スループットを計測する．結果は提案手法において約 10%のスループットの向上を達成することができた．

本論文の構成は以下のようになる．2 章で LSM-Tree の構成及び高速な I/O デバイスにおける LSM-Tree の問題を述べる．3 章では高速な I/O デバイスにおける LSM-Tree を提案する．4 章では提案手法における設計を述べ，5 章では実装，6 章で実験・評価，7 章で関連研究について述べる．8 章では本論文の結論を述べる

2. 背景

2.1 LSM-Tree

LSM-Tree は書き込みの高速な KVS の内部構造としてよく利用されている．書き込みを高速にするための構造として複数のレベルにデータを分割して保存している．図 1 のように，小さいレベルはデータの量が少なく，レベルに比例してデータ量が大きくなるデータ構造である．各レベルは複数のファイルで構成されている．新しいデータは最も小さいメモリ上のレベルに保存される．保存されたデータが各レベルのサイズを超えると 1 つ大きいレベルへデータを移動させる．1 つ大きいレベルへのデータ移動を LSM-Tree ではコンパクションと呼ぶ．これにより，LSM-Tree では書き込みリクエストごとに Disk I/O を発生させないことで高速な書き込みを実現している．

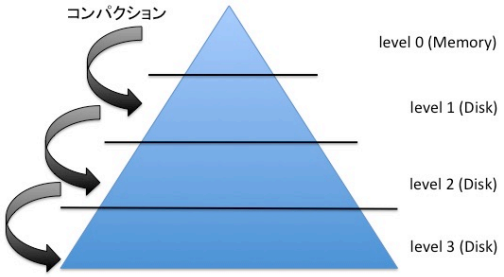

図 1 LSM-Tree のイメージ図

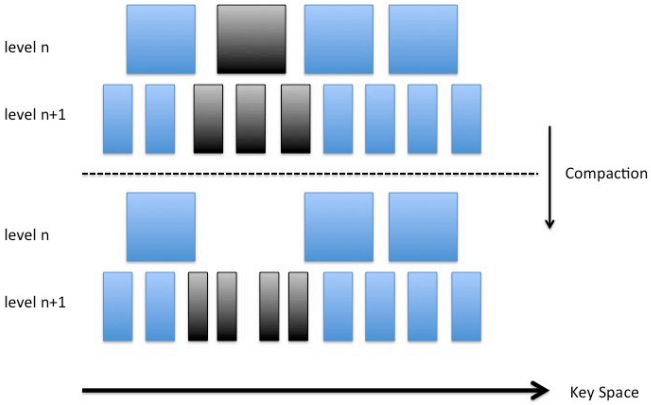

図 2 Compaction におけるマージ処理

コンパクションではデータの移動と同時にデータのマージを行う．LSM-Tree では同じ Key のデータが各レベルごとに存在しうる． LSM-Tree ではデータの更新が行われる際に古いデータをその場では消去しない．古いデータの消去はコンパクションの際に行われる．コンパクションでは，小さいレベルからファイルを 1 つ選び 1 つ大きいレベルへ移動させる．この際に小さいレベルのファイルが含む Key スペースとかぶる次のレベルのファイルとマージする．次のレベルに同じ Key が存在した場合は小さいレベルにある新しいデータのみを書き込むことで古いデータを消去する．例えば，図 2 ではレベル n の黒いファイルをレベル n+1 にコンパクションする．レベル n+1 の同じ Key スペースを含む黒いファイルと共にマージを行いレベル n+1 にマージした結果が保存される．

コンパクションは，バックグラウンドで実行される．コンパクションを行う際には整合性を保つためにロックを取得してから行う．これは，ロックなしだと同じレベルの同じ Key スペースを含むコンパクションを行う可能性があ

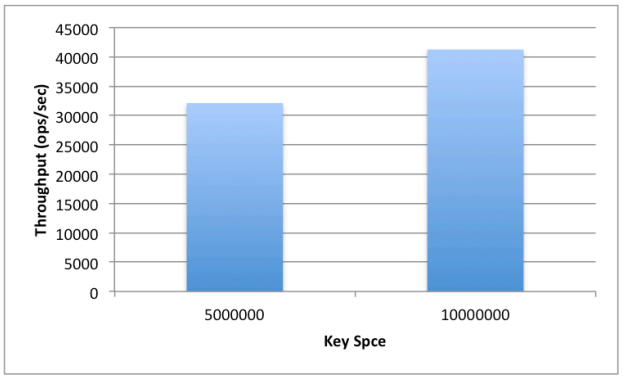

図 3 RocksDB における並行性

り，先に終わったコンパクションのデータを上書きするためである．例えば，LevelDB では DB 全体に対してロックを取得することで同時にコンパクションを行わないようにしている．

2.2 高速な I/O デバイスにおける LSM-Tree

近年，SSD などの高速な I/O デバイスが広く利用されている．[12] これは，高速な I/O デバイスは既存の DB のボトルネックを解消することで性能を大幅に向上できるためである．例えば，PCIe Based SSD などでは数 GB/sec の帯域 [15] を利用することができるため，既存の HDD と比べて約 10 倍の性能を得ることができる．実際に，FaceBook [5] などではストレージとして SSD などを利用している．

しかし，LSM-Tree では高速な I/O デバイスの帯域の使用率が低くなってしまう．これは，LSM-Tree ではコンパクション中に Disk I/O を行わない処理を含むためである．コンパクションではディスクからの読み込み，データのマージ処理，マージ結果の書き込みの順番で処理を行う．データのマージ処理中は Disk I/O が発生しないため帯域が余る．高速な I/O デバイスではコンパクションにおけるマージ処理の割合は増加する． [20] RocksDB ではこれに対して複数のコンパクションを並行に行うことで常に Disk I/O を行うようにしている．他のコンパクションの Disk I/O をマージ処理中に行うことで帯域使用率を向上させている．

実際に RocksDB で Key スペースを変えながらスループットを計測すると図 3 のようになった．このことから，高速な I/O デバイス環境ではコンパクションの並行性を向上させることはスループットを向上させる上で有効であることが確認できる．

特に，書き込みが Skewed した場合には，並行性が著しく下がる． [10] 書き込みが Skewed すると特定の Key スペースのみが頻繁にコンパクションが行われる．既存の LSM-Tree では Key スペースや DB 全体にロックを取得するため並行に行えるコンパクションの数が著しく少なくなる．これにより Disk I/O のない時間が増加するため帯域の使用率が低くなる．

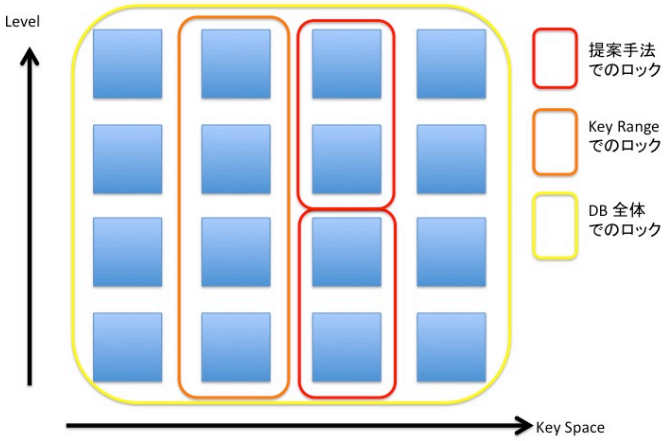

図 4 Lock 範囲

3. 提案

本論文では高速な I/O デバイスにおける LSM-Tree の帯域使用率の向上手法を提案する．I/O デバイスの帯域使用率が向上することにより，既存の LSM-Tree より高速な I/O デバイスを用いたスケールが可能になる．また，スケールが容易になることで少ない台数での目標性能の達成が可能になり消費電力を削減できる．

2.2 で述べたように並行に行えるコンパクションの数を増やすことでより多くの帯域を利用することができる．そこで，本論文ではコンパクションのロックの方向に着目し，レベル方向のロックを考慮する．既存の LSM-Tree では図 4 のように冗長なロックを取得している．LevelDB では DB 全体をロックするためコンパクションは常に 1 つのみである．同様に RocksDB では Key スペースの方向にロックを取得しているため Key の数によって並行なコンパクションの数は変化する．提案手法では key スペース，レベルの両方を必要な数だけロックすることで並行に処理できるコンパクションを増やす．

4. 設計

コンパクションにおけるロックをより細粒度にするために本論文ではレベル方向のロックを追加する．既存の LSM-Tree では図 4 のように DB 全体や Key スペース方向でのロックの取得をしているが，アクセスは行わない．

レベル方向のロックを追加する上で，整合性を保つ必要がある．既存の LSM-Tree では Key スペース方向のみの並行性なので，実行中のコンパクションの Key スペースが被らなければ整合性が保たれる．つまり，実行中のコンパクションが扱っている Key スペースの情報がわかれば良い．新しく行うコンパクションと既に実行中のコンパクション

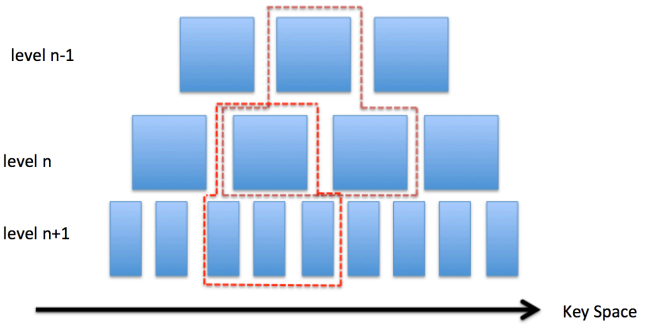

図 5 書き込み先が別のコンパクションの書き込み元と被る場合

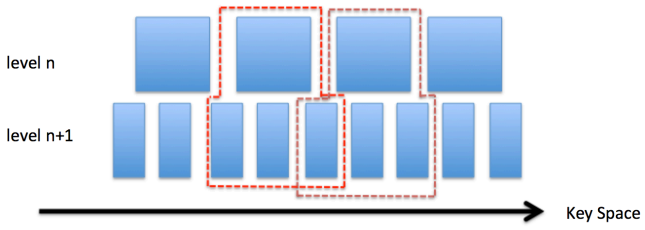

図 6 書き込み先同士が被る場合

の Key スペースと比較することで整合性を維持しながら並行にコンパクションを実行している．

提案手法ではレベル方向に複数のコンパクションが存在する．したがって，レベル方向にコンパクションが被らないようにする必要がある．また，Key スペース方向のみの時と異なり同じ Key スペースでもレベルが被らなければ並行にコンパクションできる．考慮するべき状態は，図 5 のように実行中のコンパクションのレベル，Key スペースが別のコンパクションの書き込み先になる場合，図 6 のように書き込み先が被る場合である．

実行中のコンパクションのレベル，Key スペースが別のコンパクションの書き込み先になる場合は書き込み元のレベルと書き込み先のレベルが一致かつ Key スペースが被る場合である．コンパクションよって別のコンパクション中のデータに書き込むことで別のコンパクションにおける元のデータが変化することで整合性が保てなくなる．

図 2 からもわかるようにコンパクションでは書き込み先のレベルの Key スペースの方が必ず大きくなる．これは，コンパクション元のファイルの Key スペースを含む次のレベルの全ファイルとマージするためである．したがって，コンパクション元のファイルの Key スペースのみではなく次のレベルのコンパクション対象となる全ファイルの Key スペースを考慮する必要がある．

書き込み先が被る場合は Key スペースが被るかつレベルが同じコンパクションにおいて起きる状態である．コンパクションによって同時に行っているコンパクションの書き込みが上書きされることで整合性が保てなくなる．図 6 のようにコンパクション元のファイルの Key スペースでは被らなくてもコンパクション先の Key スペースでは被る．したがって，コンパクション先のレベルが同じかつ同じ Key スペースを含むならばコンパクションを待機させる．

5. 実装

実装は RocksDB(4.10.0) をベースに行った．RocksDB では Key スペース方向での並行なコンパクションを実現しており，実行中のコンパクションの Key スペースを管理している．レベル方向のロックを実装するにあたり実行中のコンパクションのレベル，コンパクション元のファイルの Key スペース情報を実行中のコンパクションを管理するエントリに対して追加を行った．

整合性を維持するために，コンパクションをピックアップする機構における条件を変更した．実行中のコンパクションのレベル，Key スペースが別のコンパクションの書き込み先にならないようにレベルが 1 つ大きいコンパクションの書き込み元の Key スペースと書き込み先の Key スペースが被るならばコンパクションを待機させる．書き込み先が被らないようにレベルが同一かつ書き込み先のレベルが被るならばコンパクションを待機させる．これらの条件に一致しなければ整合性を保持しながら並行にコンパクションが行えるのでスレッドに割り当てを行っていく．

6. 実験

6.1 実験環境

実験環境は Intel(R) Xeon(R) CPU E5-2430 v2 @ 2.50GHz(6 core)，RAM 32GB，Ubuntu 16.04 である．ストレージは fusion io を LSM-Tree 専用のドライブとして用意し，ベンチマークなどは別の SSD 上に保存する．ベンチマークはローカル上で実行する．ベースとなっている RocksDB はライブラリの形で提供されておりネットワーク通信などの機能はないためである．

6.2 性能評価

提案手法による並行性の効果を確認するために実験・評価を行う．提案手法によるコンパクションの並行性向上を確認するためにスループットの計測を行う．ワークロードはコンパクションによる性能変化を確認するために PUT のみかつ Skewed させて行う．データセットはアイテムサイズは 1KB，Key スペースは 100,000,000 のものを用いる．これは全体のレベル数を 5 以上にすることでレベル方向の並行数を 2 以上にするためである．また，事前に全ての Key-Value をランダムな順番で保存することで，レベル方向にコンパクションが起きうる状態にする．

実験結果は図 7 のようになった．既存の RocksDB と比較して約 10 %の性能向上が確認できた．性能向上が約

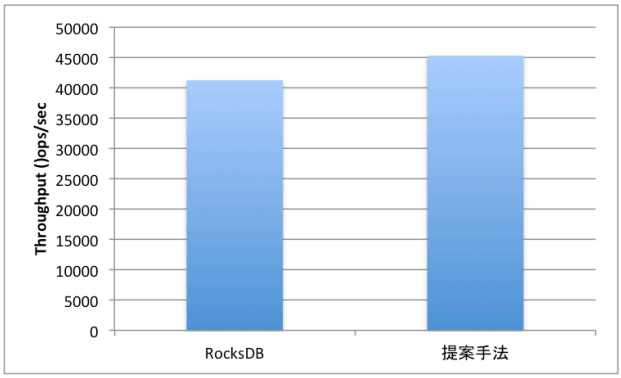

図 7 実験結果

10%となった原因としては Key スペース方向の方が並行数の増え方が大きいことがあげられる．今回の実験では 1 ファイル 2MB で行ったため 1 つのファイルに約 2,000 個の Key-Value が保存される．最大 50,000 個のファイルが 1 つのレベルに対して存在するので，Key スペース方向にも最大 5,000 個の並行なコンパクションが行える．それに対して，レベル方向に発生する最大の並行数が 2 コンパクションであったため，レベル方向の並行性によって可能になったコンパクションは 500 個であり，Key スペース方向の並行数と比較すると小さいため性能向上が約 10%になったと考えられる．

高速な I/O デバイスにおいては既存の LSM-Tree のような Key スペースのみでは帯域を余らせてしまいリソースを有効に利用できない．提案手法を用いることで高速な I/O デバイスにおいては性能向上は 10%ではあるが有効な解決手法であることを示した．

7. 関連研究

Compaction Pipelining [20] はコンパクションにおけるパイプライニングを提案している．高速な I/O 環境においてはコンパクションにおける CPU 処理の割合が増加する．しかし，既存の研究では CPU のリソース効率に着目した研究がない．Compaction Pipelining ではコンパクションにおけるリード，マージ，ライトをパイプライニングすることで CPU の利用効率を向上させる．高速な I/O デバイス (SSD) におけるリソースの使用効率を向上させるという方針は近いが本論文ではさらに高速な I/O デバイス (PCIe based SSD) を対象としている．本論文では Disk I/O の使用効率に着目しているは CPU を対象としている点でも異なっている．

Scaling Concurrent Log-Structured Data Stores [7] はスケールアップマシンにおける並行なメモリアクセスを提案している．スケールアップマシンではメニーコアを用いて高速に演算を行えるが LSM-Tree ではスナップショット，Flush による同期やアクセス順序の管理などによるロックによって並行にメモリ上にアクセスできない．Scaling Concurrent Log-Structured Data Stores ではメモリ上へのアクセスを可能なかぎり non-blocking にすることで並行なメモリアクセスを実現している．本論文とは，平衡性を向上させることで性能を向上させるという手法は近いが，本論文は Disk I/O を対象としておりメモリアクセスという点で異なっている．

既存研究においては LSM-Tree における SSD などの I/O デバイスに着目した研究が多く存在している．しかし，CPU 使用率の向上や，メモリアクセス向上による性能向上などが主であり Disk I/O の使用率に着目した研究がない．ストレージ技術の進歩によって Disk I/O の帯域が向上している上でいかに帯域の使用率をあげるかが重要であると考えられる．

8. 結論

本論文では高速な I/O デバイスにおける LSM-Tree の帯域使用率の向上を行った．既存の LSM-Tree ではコンパクション時のロックを DB 全体や Key スペース方向で取得していたが，本論文ではレベル方向でのロックを考慮した．これにより，レベル方向での並行コンパクションを行う上では整合性を保つためにレベルやコンパクション元やコンパクション先における Key スペースの情報などを管理することで実現した．これらを RocksDB をベースに実装を行った．

実験では Skewed な環境における提案手法の評価を行い，既存の LSM-Tree よりも約 10%の性能向上を達成することができた．既存の LSM-Tree よりも最大で事項できる並行数が増えたことにより，Disk I/O を常に行うことで高速な I/O デバイスの使用率を向上させたためだと言える．

これにより，LSM-Tree におけるレベル方向のロックを考慮することで並行数を増加させることの有効性を示すことができた．また，高速な I/O デバイスの帯域をより有効に使えるため少ない台数での目標性能の達成ができるようになり，データセンターなどでの消費電力の削減などが望める．

Future work としてはレベル方向に対して並行数が大きい環境での評価や実際の環境に近いワークロードなどでの有効性を検証する必要がある．

参考文献

[1] Apache. Cassandra. http://cassandra.apache.org.
[2] Apache. Hbase. https://hbase.apache.org.
[3] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload Analysis of a Large-scale Key-vaule Store. In *Proc. of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '12)*, pp. 53–64, 2012.
[4] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani,

Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-value Store. In *Proc. of the 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP '07)*, pp. 205–220, 2007.

[5] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Strum. Optimizing space amplification in rocksdb. In *CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*, 2017.

[6] FaceBook. Rocksdb. http://rocksdb.org.

[7] Guy Golan-Gueta, Edward Bortnikov, Eshcar Hillel, and Idit Keidar. Scaling concurrent log-structured data stores. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pp. 32:1–32:14, New York, NY, USA, 2015. ACM.

[8] Google. Leveldb. https://github.com/google/leveldb.

[9] HyperLevelDB. Hyperleveldb. https://github.com/rescrv/HyperLevelDB, commit 40ce80173a8d72443c5f92e3c072a54ed910bab9.

[10] Christopher Jermaine, Edward Omiecinski, and Wai Gen Yee. The partitioned exponential file for database storage management. *The VLDB Journal*, Vol. 16, No. 4, pp. 417–437, October 2007.

[11] Redis Labs. Redis. https://redis.io.

[12] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Hariharan Gopalakrishnan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Wisckey: Separating keys from values in ssd-conscious storage. *Trans. Storage*, Vol. 13, No. 1, pp. 5:1–5:28, March 2017.

[13] Memcached. Rocksdb. https://memcached.org.

[14] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling Memcache at Facebook. In *Proc. of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI '13)*, pp. 385–398, 2013.

[15] Sandisk. Fusion-io. https://www.sandisk.com/business/datacenter/products/flash-devices/pcie-flash/sx350.

[16] Russell Sears and Raghu Ramakrishnan. blsm: A general purpose log structured merge tree. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pp. 217–228, New York, NY, USA, 2012. ACM.

[17] Peng Wang, Guangyu Sun, Song Jiang, Jian Ouyang, Shiding Lin, Chen Zhang, and Jason Cong. An Efficient Design and Implementation of LSM-tree Based Key-value Store on Open-channel SSD. In *Proc. of the 9th European Conference on Computer Systems (EuroSys '14)*, pp. 16:1–16:14, 2014.

[18] Xingbo Wu, Yuehai Xu, Zili Shao, and Song Jiang. LSM-trie: An LSM-tree-based Ultra-Large Key-Value Store for Small Data Items. In *Proc. of the 2015 USENIX Annual Technical Conference (USENIX ATC '15)*, pp. 71–82, 2015.

[19] Xingbo Wu, Li Zhang, Yandong Wang, Yufei Ren, Michel Hack, and Song Jiang. zExpander: A Key-value Cache with Both High Performance and Fewer Misses. In *Proc. of the 11th European Conference on Computer Systems (EuroSys '16)*, pp. 14:1–14:15, 2016.

[20] Z. Zhang, Y. Yue, B. He, J. Xiong, M. Chen, L. Zhang, and N. Sun. Pipelined compaction for the lsm-tree. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pp. 777–786, May 2014.