Breaking Down Memory Walls in LSM-based Storage Systems

Chen Luo
cluo8@uci.edu
University of California, Irvine

ACM Reference Format:
Chen Luo. 2020. Breaking Down Memory Walls in LSM-based Storage Systems. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD'20), June 14–19, 2020, Portland, OR, USA.* ACM, Portland, OR, USA, 3 pages. https://doi.org/10.1145/3318464.3384399

1 INTRODUCTION

The log-structured merge-tree (LSM-tree) [16, 18] is widely used in modern NoSQL systems. Different from traditional update-in-place structures, an LSM-tree first buffers all writes in memory, which are subsequently flushed to disk when memory is full. The on-disk components are usually organized into levels of exponentially increasing sizes, where a smaller level is merged into the adjacent larger level when it fills up. To bound the temporary disk space occupied by merges, modern LSM-tree implementations often range partition a disk component into many fixed-size SSTables.

Efficient memory management is critical for index structures to achieve high performance. Compared to update-in-place systems, where all pages are managed within buffer pools, LSM-trees have introduced additional memory walls. First, the memory component region is isolated from the disk buffer cache. Second, each LSM-tree manages its memory component independently. Since the optimal memory allocation heavily depends on the workload, memory management must be workload-adaptive to maximize the system efficiency. However, such adaptivity is non-trivial due to several problems highlighted below. Existing LSM-tree implementations, such as RocksDB [2] and AsterixDB [12], instead use a static memory allocation scheme due to its simplicity while sacrificing efficiency.

Problem 1: Memory Allocation Among Multiple LSM-trees. An LSM-based storage system often contains multiple LSM-trees representing different datasets and secondary indexes [14]. Each LSM-tree can have different write characteristics, so it is important to allocate and deallocate their memory components properly to maximize system efficiency.

Problem 2: Managing Large Memory Components. Most existing LSM-tree implementations use monolithic B^+-trees or skip-lists to manage their memory components; they are flushed entirely when memory is full. With large memory, it is desirable to flush memory components continuously to minimize blocking and maximize memory utilization. Moreover, since write amplification is determined by the number of on-disk levels [16], the on-disk structure may need to be adjusted to reduce write amplification.

Problem 3: Memory Allocation Between Disk Buffer Cache and Memory Components. Finally, the system must decide how to allocate memory between memory components and the disk buffer cache. This can be formulated as an optimization problem. However, the challenge is to model the workload characteristics accurately, especially when workloads are skewed, and to make sure that the system reacts properly as the workload fluctuates.

In this paper, we present our ongoing project, which attempts to break down the memory walls in LSM-based storage systems. We present a general architecture that enables adaptive memory management. We describe our solution for Problems 1 and 2 and a formulation of Problem 3.

2 BACKGROUND AND RELATED WORK

The importance of efficient memory management has long been recognized for database systems. Various buffer replacement policies [7, 11, 17] have been proposed to minimize disk I/Os. Some commercial DBMSs [3, 9, 19] have offered functionalities to tune the memory allocation among different memory regions. However, these efforts have all focused on storage systems based on in-place updates, i.e., all buffer pages are managed through buffer pools.

There have also been some efforts to optimize memory management for LSM-trees. FloDB [5] presents a two-level memory component structure to mask write latency under large memory. However, FloDB still targets fixed-size memory components and does not optimize on-disk write amplification. Accordion [6] introduces a multi-level memory component structure with in-memory flushes and merges. One

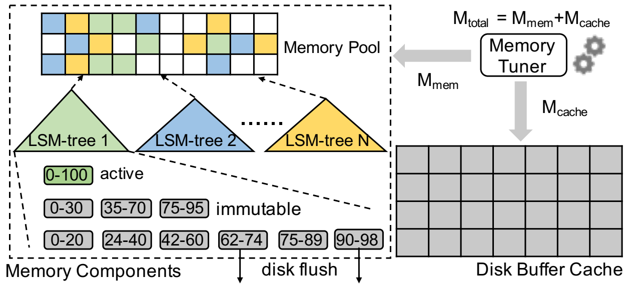

Figure 1: Memory Management Architecture

drawback is that memory components can be very large since they are not range partitioned, resulting in high memory utilization during large in-memory merges. Monkey [10] uses analytical models to tune the memory allocation between memory components and Bloom filters. However, Monkey mainly considers the worst-case I/O cost, which can be very different from real-world workloads due to skews [13].

3 ADAPTIVE MEMORY MANAGEMENT

Figure 1 depicts our proposed memory management architecture, where the overall memory budget M_{total} is divided between the memory component region M_{mem} and the disk buffer cache M_{cache}. It also performs adaptive memory management in the following manner.

The memory components of all LSM-trees are managed through Figure 1's shared memory pool. When an LSM-tree has insufficient memory to store incoming writes, more pages will be requested from the pool. When the memory utilization exceeds a predefined threshold, an LSM-tree is selected to flush its memory component to disk. To bound the recovery time, flushes can also be triggered if the transaction log is too long. A write-heavy LSM-tree can come to have a very large memory component, and flushing it all at once may block incoming writes and cause write stalls [15]. This also reduces memory utilization since a large chunk of memory will be freed at once. To address these problems, we organize a memory component into a multi-level structure, as shown in Figure 1, that can be seen as an in-memory LSM-tree. The active SSTable is used to store incoming writes. When the active SSTable is full, it is flushed as an immutable in-memory SSTable, which can be subsequently merged as well. On receiving a disk-flush request, the LSM-tree selects an SSTable to flush to disk. Disk flushes are performed on a continuous-basis and memory utilization thus stays high. Due to space limitations, we omit further details of memory component management.

Since the write amplification of an LSM-tree is determined by the number of on-disk levels [16], the question of how to adjust the on-disk structure to reduce write amplification remains. One obvious solution is to change the number of on-disk levels as the memory component grows and shrinks. However, this would add implementation complexity, and the LSM-tree must be robust in case of workload fluctuations. Instead, we propose a simple yet effective solution that exploits the property that our flushed SSTables are range partitioned. In the proposed solution, the number of on-disk levels will be determined by the maximum size of the active SSTable. When the memory component is small, this structure behaves like a classical LSM-tree. However, when the memory component becomes larger, its flushed SSTables will have smaller key ranges. It turns out that most SSTables at the first few on-disk levels will then have non-overlapping key ranges, which leads to better write and read performance without impacting space utilization.

To break down the memory wall between memory components and the disk buffer cache, we introduce a memory tuner. Given a memory budget M_{total}, its goal is to find an optimal memory allocation M_{mem} and M_{cache} so that the weighted I/O cost $w \cdot I/O_{write} + r \cdot I/O_{read}$ is minimized. The weights w and r allow us to instantiate the objective function for different use cases. For example, on hard disks, one can set $w = r$ since reads and writes are equally expensive, while on SSDs one can make w larger to reduce write amplification. We are still in the process of designing this memory tuner.

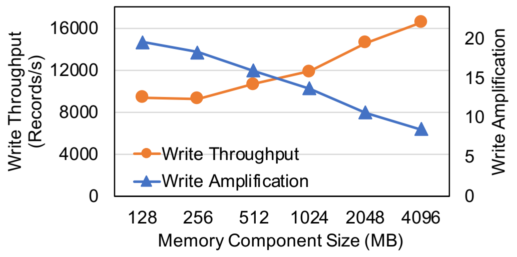

Figure 2: Preliminary Experimental Results

4 RESULTS AND CONTRIBUTIONS

We have implemented our proposed architecture (minus the memory tuner) inside Apache AsterixDB [1, 4]. To evaluate the effectiveness of our proposed solution for exploiting large memory components, we conducted an experiment using the YCSB benchmark [8] with 100GB dataset and uniform updates. Figure 2 shows that larger memory components increase the write throughput by reducing the write amplification. This also reduces the total amount of disk writes and improves the system efficiency.

Acknowledgment. I would like to thank my PhD advisor Michael J. Carey for his support in this work. This work is supported by NSF awards IIS-1838248 and CNS-1925610.

REFERENCES

[1] 2019. AsterixDB. https://asterixdb.apache.org/.

[2] 2019. RocksDB. http://rocksdb.org/.

[3] Sanjay Agrawal et al. 2005. Database tuning advisor for Microsoft SQL Server 2005. In *ACM SIGMOD*. ACM, 930–932.

[4] Sattam Alsubaiee et al. 2014. AsterixDB: A Scalable, Open Source BDMS. *PVLDB* 7, 14 (2014), 1905–1916.

[5] Oana Balmau et al. 2017. FloDB: Unlocking Memory in Persistent Key-Value Stores. In *European Conference on Computer Systems (EuroSys)*. 80–94.

[6] Edward Bortnikov et al. 2018. Accordion: Better Memory Organization for LSM Key-value Stores. *PVLDB* 11, 12 (2018), 1863–1875.

[7] Hong Tai Chou and David J. DeWitt. 1986. An evaluation of buffer management strategies for relational database systems. *Algorithmica* 1, 1 (01 Nov 1986), 311–336.

[8] Brian F. Cooper et al. 2010. Benchmarking Cloud Serving Systems with YCSB. In *ACM SoCC*. 143–154.

[9] Benoit Dageville and Karl Dias. 24–31. Oracle’s Self-Tuning Architecture and Solutions. *IEEE Data Engineering Bulletin* (24–31), 2006.

[10] Niv Dayan et al. 2017. Monkey: Optimal Navigable Key-Value Store. In *ACM SIGMOD*. 79–94.

[11] Theodore Johnson and Dennis Shasha. 1994. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In *VLDB*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 439–450.

[12] Taewoo Kim and et al. 2020. Robust and efficient memory management in Apache AsterixDB. *Software: Practice and Experience* (2020). https://doi.org/10.1002/spe.2799

[13] Hyeontaek Lim et al. 2016. Towards Accurate and Fast Evaluation of Multi-Stage Log-structured Designs. In *USENIX Conference on File and Storage Technologies (FAST)*. 149–166.

[14] Chen Luo and Michael J. Carey. 2019. Efficient Data Ingestion and Query Processing for LSM-Based Storage Systems. *PVLDB* 12, 5 (2019), 531–543.

[15] Chen Luo and Michael J. Carey. 2019. On Performance Stability in LSM-based Storage Systems. *PVLDB* 13, 4 (2019), 449–462.

[16] Chen Luo and Michael J. Carey. 2020. LSM-based storage techniques: a survey. *The VLDB Journal* 29, 1 (2020), 393–418.

[17] Elizabeth J. O’Neil et al. 1993. The LRU-K Page Replacement Algorithm for Database Disk Buffering. *SIGMOD Rec.* 22, 2 (June 1993), 297–306.

[18] Patrick O’Neil et al. 1996. The Log-structured Merge-tree (LSM-tree). *Acta Inf.* 33, 4 (1996), 351–385.

[19] Adam J. Storm et al. 2006. Adaptive Self-tuning Memory in DB2. In *VLDB (VLDB ’06)*. VLDB Endowment, 1081–1092.