Optimizing Key-Value Stores for Hybrid Storage Architectures

Prashanth Menon [#], Tilmann Rabl [#†], Mohammad Sadoghi [*], Hans-Arno Jacobsen [#]
[#] *Middleware Systems Research Group, University of Toronto*
[†] *IBM Canada Software Laboratory, CAS Research*
[*] *IBM T.J. Watson Research Center*

Abstract

Flash-based solid state drives (SSDs) are increasingly becoming a popular choice as a storage device within database management systems and key-value stores alike. SSDs offer fast throughput and low latency access to data, but their price-per-byte cost often makes them uneconomical for exclusive use, especially in the era of big data workloads. A common solution to this problem is to augment existing database systems by adding smaller SSDs that target only performance-critical areas. We believe this hybrid approach to be a stop-gap solution.

Rather than simply extending existing systems with SSDs, in this work we completely re-architect how a key-value database operates in a hybrid storage setting with both small but fast SSDs and slower but high-capacity HDDs. We formulate an accurate I/O cost model to study how popular key-value stores behave under several varying representative workloads. Based on these studies and taking a holistic approach, we design a system that dynamically optimizes the data layout and access strategy that leverages the strengths of each available storage medium.

1 Introduction

Traditional database systems were conceived, designed and largely built with the classical storage hierarchy in mind - very fast but limited main memory that sits in front of a large but slow hard-disk drive (HDD). This model has worked as CPU speed, memory speed and hard-disk capacity have grown, but this is no longer the case. CPU and memory speeds have stagnated while new emerging storage mediums like fast flash and non-volatile memory present opportunities for vast improvements to database performance.

Solid-state disk (SSD) offers fast access to persistent data, but their price-per-byte cost continues to make their exclusive use uneconomical for the majority of use-cases, especially in an era of big-data applications. As such, SSDs have often been introduced to the database ecosystem as an extension that targets only performance-critical areas [1, 4, 5]. This means relegating the SSD to operate as a cache or as simple extension to the buffer pool [2, 3].

It is the authors' belief that the extension model for leveraging SSDs and future non-volatile memory solutions is non-optimal. We need to redesign the database architecture in an environment that includes SSDs and presumes the existence of highly multi-core processors and newer emerging non-volatile memory devices.

We constrain this work to key-value stores, specifically BigTable-based systems which are prominently used in industry. First, we describe how such systems operate and construct an I/O cost model that accurately captures the cost of various supported operations. We focus on I/O cost specifically because our interest lies in disk-based databases; main-memory databases will require a vastly different cost model. We use this cost model to reason how LevelDB performs in six different workloads that vary the ratio of reads and writes and vary the key request distributions.

Our contributions are: (I) formulating a cost-model for BigTabe-based systems; (II) identifying common workloads and quantifying their cost under LevelDB; (III) a new storage architectured based on LSM-trees that utilizes SSDs and other emerging storage technologies.

2 LevelDB

LevelDB is an embedded key-value store that bases its design on the popular Log-Structured Merge

(LSM) Trees. Writes into LevelDB are first appended to a commit log and subsequently buffered into an in-memory sorted data-structure called the *memtable*. When the size of the memtable reaches a user-defined threshold, it is flushed to disk in the form of an *SSTable*. SSTables are immutable 2MB files containing a sorted list of key-value pairs, followed by an index to enable efficient searching. LevelDB organizes its SSTables into a series of ordered levels that grow exponentially in size. SSTables within a level are disjoint while SSTables across distinct levels are allowed to overlap in their key ranges. In this scheme, a read operation first probes the memtable for the key of interest. If the probe succeeds, the result is returned. Otherwise, the search continues through the levels beginning at the youngest. Since SSTables within a level are disjoint in the keys they contain, a read operation must only look into at most one SSTable from each level. This process continues until the last and oldest level is reached.

The size of the levels in LevelDB grow by a factor of 10, meaning that a level l_i is 10 times larger than level l_{i-1}. The sizing property of levels allows us to say that, on average, an SSTable on level l_{i-1} overlaps with 10 SSTables on level l_i. To maintain strict size constraints of each of the levels, LevelDB needs to continuously move data from younger levels that are full to older levels that have room to store more data. This process is called *compaction*. Compaction is performed by selecting a candidate SSTable from the overflowing level and finding all SSTables from the subsequent level that overlap in the range of keys they cover. Once the tables are identified, an n-way merge is performed to produce a new set of SSTables on the higher level; the previous SSTables are marked obsolete and scheduled for deletion from the system.

The read algorithm presented previously makes the task of capturing and modelling the read-cost fairly straightforward. Capturing the cost of writes is more tricky. It is tempting to conclude that writes incur no I/O cost aside from that of the commit log since they are buffered in the memtable. Moreover, modern disks can sustain high sequential write speeds and the commit log of BigTable systems have been shown not be a bottleneck [5]. This line of reasoning is incomplete. LevelDB, as with LSM-trees, suffers from write-amplification: the continuous reading and re-writing of insertions, updates and deletions as they propagate through the levels. Write-amplification is manifested through compaction. Compactions place a perpetual burden on I/O resources and thus have a direct impact on the performance of the rest of the system. It is highly desirable to minimize the execution of compaction. Any model that attempts to capture the true costs of operations in a BigTable-based system must account for hidden write-amplification and compaction costs.

3 Cost Model

The primary advantage of the LSM-tree is its ability to convert random writes into purely sequential writes. For this reason, we begin our cost model construction by measuring the random-read (prefetch random-read), sequential read and sequential write performance of modern HDDs and SSDs. Though these are device specific, we believe them to be representative of current offerings. The results are provided in Table 1.

Description	Symbol	Metric
Random Read on HDD	R_H	5.5ms
Random Read on SSD	R_S	0.28ms
Sequential Read/Write on HDD	W_H	150 MB/s
Sequential Read/Write on SSD	W_S	240 MB/s

Table 1: Storage Device Parameters

Using the compaction algorithm described in Section 2, we can derive a cost formula to reflect the cost of compaction in LevelDB. Informally, the cost of compacting an SSTable is the sum of the cost of seeking to the beginning of each SSTable involved, plus the cost of sequentially reading each SSTable as part of the n-way merge and the cost of sequentially writing each output SSTable back out to disk. The structure of levels tells us that, on average, a candidate SSTable from a level overlaps with 10 SSTables on a higher level meaning that a total of $T = 11$ SSTables ($2T$ MB) will be read and re-written. Formally, the average cost of a compaction, C_{comp}, measured in milliseconds can be formulated as

$$C_{comp} = (T \times R) + 2\left(2T \times \frac{1000}{W}\right) \quad (1)$$

The subscripts of R and W have been removed to indicate that the formula is applicable to any storage device that employs the LevelDB storage layout.

Using Formula 1 and the storage parameters from Table 1, we can quantify the I/O cost of a compaction in any LSM-tree based system.

4 Cost Analysis

We consider six workloads that permute three read/write characteristics (read-only, read-write and write-only) and two request distributions (skewed and uniform). For each of the scenarios we consider, we use a workload consisting of 500 million operations on key-value pairs. The composition of reads and writes and their distribution patterns change according to the specific scenario we

evaluate. Every read includes a 16 byte key and every write includes both a 16 byte key and a 200 byte value. These sizes were chosen to produce a working dataset, D, of roughly 100GB (102,400MB). It is easy to see that a workload which generates D MB of data will produce $L = \log_{10} D$ levels. In the analysis below, we ignore any caching behaviour to keep the reasoning simple.

4.1 Read-Only - Uniform

In a read-only workload, the cost is equal to the number of read operations multiplied by the cost of each individual read. Since the cost of a read in LevelDB is dominated by the seek cost of accessing an SSTable, we require only a simple calculation to determine the duration of a read-only workload with a uniform key distribution.

For a uniform request distribution, the random nature of key requests make it difficult to predict what data can be brought onto the SSD for faster response times. Any sort of LRU or frequency based algorithm will prove to be ineffective.

4.2 Read-Only - Skewed

For a read-only skewed workload, we use an 80/20 request distribution where 80% of the key lookups occur on 20% of the data. In the steady-state case, a skewed read workload mimics the performance of the uniform case because we ignore the impact of caching. In a real setting, the frequently accessed SSTables will be more likely to exist in the operating system page cache or LevelDB's table cache in memory. As such, reads to the 20% "hot-set" will be more likely to be served from main memory. This skew-unawareness presents a large missed opportunity for LevelDB.

4.3 Write-Only - Uniform

Since levels in LevelDB grow exponentially by a factor of 10, 90% of the generated SSTables will reside on the last level and undergo L compactions. There is an additional cost of physically writing D MB of data received from the workload generator into $T = \frac{D}{2}$ SSTables. The I/O cost to execute the workload is captured in Formula 2.

$$C = \left(\frac{D}{W_H}\right) + (0.9T \times L \times C_{comp}) \quad (2)$$

4.4 Write-Only - Skewed

In a skewed 80/20 write-only workload there is significant overwriting of data. Though 100GB of data is sent to the server, only at most 20GB + 20GB = 40GB of unique data will reside on disk after compactions complete when the system stabilizes.

It is useful to visualize each LevelDB level as a histogram of keys where the buckets are the SSTables in a level. Though SSTables have a fixed size, their width in a level's histogram is variable.

The behaviour of the system experiencing a skewed workload depends on the degree of contiguity of the 20% most frequently written keys. Using the histogram technique, if the "hot-set" is contiguous in its range, the width of select buckets in a level will be very narrow. If a bucket (SSTable) is narrow in the range of keys it contains, it will overlap with fewer buckets on a higher level if and when it is chosen for compaction. Therefore, while more compactions may be required, the amount of data read and re-written is smaller than in a uniform case. If the range of keys covered in the "hot-set" is non-contiguous, the histogram will resemble that of a uniform key distribution since a key can only appear once in a given level; duplicates are eliminated through the compaction process. In this case, the compaction cost will approach that of a write-only workload with a uniform key distribution.

4.5 Mixed Read and Write

In a mixed read and write workload, we make no assumptions on the number of read and write operations that characterize the workload. Instead, we let $r < 1$ be the fraction of read operations and $w = 1 - r$ be the fraction of write operations. Now, if O is the total number of operations, then $O_r = rO$ and $O_w = (1 - r)O$ represent the total number of read and write operations, respectively. The total amount of data written to the server is $D_w = O_w \times 0.0002$ MB.

4.5.1 Mixed R/W - Uniform

The cost of executing the workload is equivalent to the cost of serving O_r reads, plus the cost of writing D_w MB, plus the cost of performing all the required compactions. As mentioned previously, the majority of the compaction cost is attributed to the compaction of 90% of the SSTables as they move through each of the L levels. The cost of execution in LevelDB is captured in formula 3.

$$C = (O_r \times R) + \left(\frac{D_w}{S}\right) + (0.9T \times L \times C_{comp}) \quad (3)$$

4.5.2 Mixed R/W - Skewed

We use the same 80/20 distribution model with the added assumption that 80% of the read operations and 80% of the write operations use the same 20% of the key space during execution - meaning they share in their "hot-set" of data.

To calculate the cost of executing the workload in LogStore, we combine the formulations from Section 4.2 and Section 4.4. In essence, since we ignore any caching behaviour, the cost of reads is equivalent to the cost of performing a random read to disk multiplied by the number of read operations.

The cost of writes requires scaling the compaction cost proportional to the contiguity of the "hot-set" of data, since the contiguity alters the degree of overlap of SSTables.

5 Optimizations

When SSDs are available, we can naturally place some of the younger levels on the SSD to enable faster retrieval without exhausting storage space. However, the rationale of using levels if they exist on the SSD may be void. Levels provide a bounded read time which was necessary on HDD, but an SSD offers almost 50 times faster random-read performance and much better parallelism versus its HDD counterparts. We can use this to our advantage. We propose collapsing the levels on the SSD into a single level and retain the last and largest level on HDD. SSTables on the SSD may overlap, but we can afford to perform *some* redundant lookups on SSD by utilizing inherent parallel access of the SSD and leveraging the fast random-read speed it offers. The write path and compaction algorithm remains the same.

5.1 Read-Only

LevelDB is not a skew-aware system. In our optimized system, we can utilize an LRU-based algorithm to track frequently requested keys and migrate them to SSD. There are currently two proposed techniques to achieve this that vary in the granularity they operate on. In the first solution, if we recognize that a key's access frequency exceeds a threshold (either by using an xLRU or temperature-based technique) and the key resides on HDD, then we *re-insert* only the key into the data store. Along with the key, we add metadata to indicate that the key is a duplicate of one that exists in an older level and where to find it's value. This modification does not impact compaction since no value information exists, but allows fast retrieval since only one level is touched after finding a reinserted key.

The other solution works at the granularity of entire SSTables. If we recognize that a key within an SSTable is accessed frequently enough , we perform an *upward compaction* where we momentarily reverse the direction of compaction in an effort to move data *from* HDD *towards* the SSD that stores the younger levels and where the key can be accessed quicker.

5.2 Write-Only

When we collapse the levels on the SSD and allow overlapping tables, even if we can perform redundant lookups on the SSD, doing so is unmanageable and sub-optimal. A simple solution is to trigger compaction/merging of overlapping SSTables only when the system observes incoming reads. The compaction is efficient since it is performed on SSD and has the benefit of enforcing disjointedness of SSTables to ensure single-seek read latency.

With this optimization, the cost of a write-only workload becomes the time required to initially write all the data to SSD in addition to the compaction cost of of migrating an SSTable from SSD to HDD.

$$C = \left(\frac{D}{W}\right) + (T \times C_{comp}) \tag{4}$$

This cost is a factor of L smaller than that of LevelDB since we only maintain two levels, one on SSD and one on HDD. We can further improve this by optimizing how SSTables are stored on HDD; we leave this to future work our group is currently conducting.

6 Conclusions

In this paper, we constructed an I/O cost model to study how LSM-tree based key-value stores respond to a common set of workloads. Using these results, we outlined the architecture of a new LSM-tree based key-value store that presumes the existence of fast and highly parallel non-volatile storage solutions and makes optimal use of them. We believe that revisiting the design of traditional storage systems in the light of new and emerging persistent storage mediums is required rather than simply extending existing solutions.

References

[1] M. Canim, B. Bhattacharjee, G. A. Mihaila, C. A. Lang, and K. A. Ross. An object placement advisor for DB2 using solid state storage. *PVLDB*, 2(2):1318–1329, 2009.

[2] M. Canim, G. A. Mihaila, B. Bhattacharjee, K. A. Ross, and C. A. Lang. SSD bufferpool extensions for database systems. *PVLDB*, 3(2):1435–1446, 2010.

[3] J. Do, D. Zhang, J. M. Patel, D. J. DeWitt, J. F. Naughton, and A. Halverson. Turbocharging DBMS buffer pool using SSDs. In *SIGMOD*, SIGMOD '11, pages 1113–1124, New York, NY, USA, 2011. ACM.

[4] G. Graefe. The Five-Minute Rule 20 Years Later: and How Flash Memory Changes the Rules . *Communications of the ACM*, 2009.

[5] P. Menon, T. Rabl, M. Sadoghi, and H.-A. Jacobsen. CaSSanDra: An SSD Boosted Key-Value Store. In *ICDE*, 2014.