MUTANT: Balancing Storage Cost and Latency in LSM-Tree Data Stores

Hobin Yoon
Georgia Institute of Technology
hobinyoon@gatech.edu

Juncheng Yang
Emory University
juncheng.yang@emory.edu

Sveinn Fannar Kristjansson
Spotify
sveinn@spotify.com

Steinn E. Sigurdarson
Takumi
steinnes@gmail.com

Ymir Vigfusson
Emory University
& Reykjavik University
ymir@mathcs.emory.edu

Ada Gavrilovska
Georgia Institute of Technology
ada@cc.gatech.edu

ABSTRACT

Today's cloud database systems are not designed for seamless cost-performance trade-offs for changing SLOs. Database engineers have a limited number of trade-offs due to the limited storage types offered by cloud vendors, and switching to a different storage type requires a time-consuming data migration to a new database. We propose MUTANT, a new storage layer for log-structured merge tree (LSM-tree) data stores that dynamically balances database cost and performance by organizing SSTables (files that store a subset of records) into different storage types based on SSTable access frequencies. We implemented MUTANT by extending RocksDB and found in our evaluation that MUTANT delivers seamless cost-performance trade-offs with the YCSB workload and a real-world workload trace. Moreover, through additional optimizations, MUTANT lowers the user-perceived latency significantly compared with the unmodified database.

CCS CONCEPTS

• **Information systems → Record and block layout**; **DBMS engine architectures**; *Cloud based storage*; *Hierarchical storage management*; • **Theory of computation →** *Data structures and algorithms for data management*;

KEYWORDS

LSM-tree data stores, Database storage systems, Cloud database systems, and Seamless cost and latency trade-offs

ACM Reference Format:
Hobin Yoon, Juncheng Yang, Sveinn Fannar Kristjansson, Steinn E. Sigurdarson, Ymir Vigfusson, and Ada Gavrilovska. 2018. MUTANT: Balancing Storage Cost and Latency in LSM-Tree Data Stores. In *SoCC '18: ACM Symposium on Cloud Computing, October 11–13, 2018, Carlsbad, CA, USA*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3267809.3267846

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SoCC '18, October 11–13, 2018, Carlsbad, CA, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-6011-1/18/10…$15.00
https://doi.org/10.1145/3267809.3267846

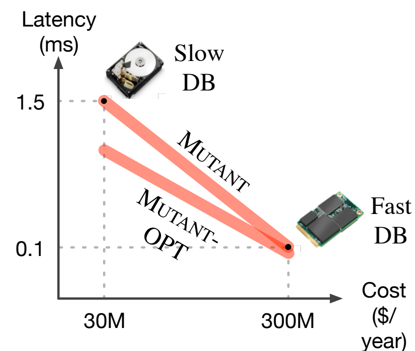


Figure 1: *MUTANT provides seamless cost-performance trade-offs* *between "fast database" and "slow database", and better cost-performance trade-offs with an extra optimization.*

1 INTRODUCTION

The growing volume of data processed by today's global web services and applications, normally hosted on cloud platforms, has made cost effectiveness a primary design goal for the underlying databases. For example, if the 100,000-node Cassandra clusters Apple uses to fuel their services were instead hosted on AWS (Amazon Web Services), then the annual operational costs would exceed $370M[1]. Companies, however, also wish to minimize their database latencies since high user-perceived response times of websites lose users [28] and decrease revenue [26]. Since magnetic storage is a common latency bottleneck, cloud vendors offer premium storage media like enterprise-grade SSDs [8, 30] and NVMe SSDs [15]. Yet even with optimizations on such drives, like limiting power usage [11, 17] or expanding the encoding density [31, 38], the price of lower-latency storage media can eat up the lion's share of the total cost of operating a database in the cloud (Figure 2).

Trading off database cost and latency involves tedious and error-prone effort for operators. Consider, for instance, the challenges faced by an engineer intending to transition a large database system operating in the cloud to meet a budget cut. When the database system accommodates only one form of storage medium at a time, they must identify a cheaper media – normally a limited set of options – while minimizing the ensuing latencies. They then live-migrate data to the new database and ultimately all customer-facing

[1] Conservatively calculated using the price of AWS EC2 c3.2xlarge instance, of which storage size adequately hosts the clusters' data.

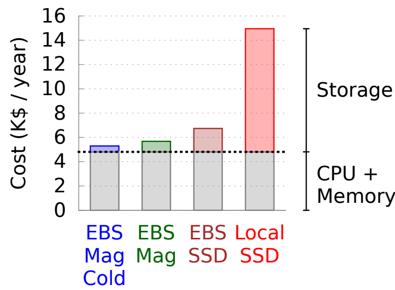


Figure 2: *Fast storage is costly.* *Cost to host an IaaS database instance for 1 year using various storage devices. Storage cost can be more than* 2× *the cost of CPU and memory. Based on the pricing of EC2* `i2.2xlarge` *instance (I/O-optimized with 1.6 TiB storage device) [8].*

applications, a process that can take months as in the case of Netflix [9]. Every subsequent change in the budget, including possible side-effects from the subsequent change in latencies, undergoes a similar laborious process. In the rare cases where the database does support multiple storage media, such as the round-robin strategy in Cassandra [3] or per-level storage mapping in RocksDB [6], the engineer is stuck with either a static configuration and a suboptimal cost-performance trade-off, or they must undertake cumbersome manual partitioning of data and yet still be limited in their cost options to accommodate budget restrictions.

We argue that cloud databases should support seamless cost-performance trade-offs that are aware of budgets, avoiding the need to manually migrate data to a new database configuration when workloads or cost objectives change. Here, we present Mutant: a layer for log-structured merge tree (LSM-tree) data stores [21, 33] that automatically maintains a cost budget while minimizing latency by dynamically keeping frequently-accessed records on fast storage and less-frequently-accessed data on cheaper storage. Rather than the trade-off between cost-performance points being zero-sum, we find that by further optimizing the placement of metadata, Mutant enables the data store to simultaneously achieve both low cost *and* low latency (Figure 1).

The key insight behind Mutant is to exploit three properties of today's data stores. First, the access patterns in modern workloads exhibit strong temporal locality, and the popularity of objects fades over time [14, 24]. Second, LSM-tree designs imply that data that arrives in succession is grouped into the same SSTable, being split off when full. Because of the access patterns, the frequency of which an SSTable is accessed decreases with the SSTable's age. Third, each SSTable of which the database is comprised is a portable unit, allowing them to be readily migrated between various cloud storage media. Mutant combines these properties and continuously migrates older SSTables and, thus, colder data to slower and cheaper storage devices.

Dynamically navigating the cost-performance trade-off comes with challenges. To minimize data access latencies while meeting the storage cost SLO (service level objective), the Mutant design includes lightweight tracking of access patterns and a computationally-efficient algorithm to organize SSTables by their access frequencies. Migrations of SSTables are not free, so Mutant also contains a mechanism to minimize rate of SSTable migrations when SSTable access frequencies are in flux.

We implement Mutant by modifying RocksDB, a popular, high-performance LSM tree-based key-value store [7]. We evaluated our implementation on a trace from the backend database of a real-world application (the QuizUp trivia app) and the YCSB benchmark [18]. We found that Mutant provides seamless cost-performance trade-off, allowing fine-grained decision making on database cost through an SLO. We also found that with our further optimizations, Mutant reduced the data access latencies by up to 36.8% at the same cost compared to an unmodified database.

Our paper makes the following contributions:

- We demonstrate that the locality of record references in real-world workloads corresponds with the locality of SSTable references: there is *a high disparity in SSTable access frequencies.*
- We design an LSM tree database storage layer that provides seamless cost-performance trade-offs by organizing SSTables with an algorithm that has minimal computation and IO overheads using SSTable temperature, an SSTable access popularity metric robust from noise.
- We further improve the cost-performance trade-offs with optimizations such as SSTable component organization, a tight integration of SSTable compactions and migrations, and SSTable migration resistance.
- We implement Mutant by extending RocksDB, evaluate with a synthetic microbenchmarking tool and a real-world workload trace, and demonstrate that (a) Mutant provides seamless cost-performance trade-offs and (b) Mutant-opt, an optimized version of Mutant, reduces latency significantly over RocksDB.

2 BACKGROUND AND MOTIVATION

The SSTables and SSTable components of LSM-tree databases have significant data access disparity. Here, we will argue that this imbalance is created from locality in workloads, and gives us an opportunity to separate out hotter and colder data at an SSTable level to store on different storage media.

2.1 Preliminaries: LSM-Tree Databases

Our target non-relational databases (BigTable, HBase, Cassandra, LevelDB and RocksDB), all popular for modern web services for their reputed scalability and high write throughput, all share a common data structure for storage: the log-structured merge (LSM) tree [1, 7, 16, 21, 25]. We start with a brief overview of how LSM tree-based storage is organized, in particular the core operations (and namesake) of *log-structured* writes and *merge*-style reads, deferring further details to the literature [33].

Writes: When a record is written to an LSM-tree database, it is first written to the commit log for durability, and then written to the MemTable, an in-memory balanced tree. When the MemTable becomes full from the record insertions, the records are flushed to a new SSTable. SSTable contains a list of records ordered by their keys: the term SSTable originates from sorted-string table. The batch writing of records is the key design for achieving high write throughputs by transforming random disk IOs to sequential IOs. A record modification is made by appending a new version to

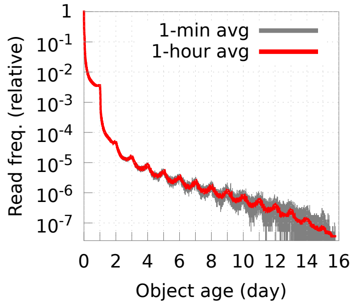


Figure 3: *Record age and its access frequency* *from the QuizUp data access trace. Access frequencies are aggregated over all records and normalized.*

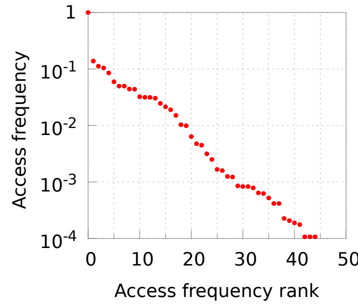


Figure 4: *SSTable access frequency distribution* *on day 15 of the 16-day QuizUp trace replay.*

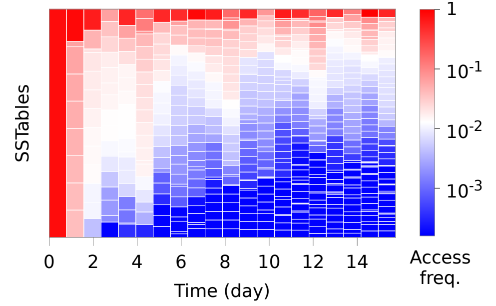


Figure 5: *SSTable access frequencies over time.* *Each rectangle represents an SSTable with a color representing its access frequency. SSTable heights at the same time are drawn proportionally to their sizes.*

the database, and a record deletion is made by appending a special deletion marker, tombstone.

Reads: When a record is read, the database consults the MemTable and the SSTables that can possibly contain the record, merges all matched records, and then returns the merged result to the client.

Leveled compaction: The merge-style record read implies that read performance will depend on the number of SSTables that need to be opened, the number of which is called a read amplification factor. To reduce the read amplification, the database reorganizes SSTables in the background through a process called *SSTable compaction*. In this work, we focus on the *leveled compaction* strategy [33] that was made popular by LevelDB, and has been adopted by Cassandra and RocksDB [7, 19].

The basic version of leveled compaction decreases read amplification by imposing two rules to organize SSTables into a hierarchy. First, SSTables at the same level are responsible for disjoint keyspace ranges, which guarantees a read operation to read at most-one SSTable per level[2]. Second, the number of SSTables at each level increases exponentially from the previous level, normally by a factor of 10. Thus, a read operation in a database consisting of N SSTables only needs to look up $O(\log N)$ SSTables [6, 19, 21, 27].

2.2 Locality in Web Workloads

Modern web workloads have repeatedly been shown to have high temporal data access locality: access frequency drops off quickly with age [14, 24, 37]. We observe that a similar temporal locality exists with database records from the analysis of a real-world database access trace: we gathered a 16 day trace of the key-value stores underlying Plain Vanilla's QuizUp trivia app while serving tens of millions of players [5]. Figure 3 shows a sharp drop of record accesses as records become old.

2.3 Locality in SSTable Accesses

The locality in record accesses, combined with the batch writing of records to SSTables, leads to the locality in SSTable accesses. We confirm the SSTable access frequency disparity by analyzing the SSTable accesses using the QuizUp workload. First, at a given time, only a small number of SSTables are frequently accessed and the others are minimally accessed: the difference between the most and the least frequently accessed ones is as big as 4 orders of magnitudes (see Figure 4). Second, the access disparity persists throughout the time, as shown by the time vs. SSTable accesses in Figure 5. As more records are inserted over time, the number of infrequently accessed SSTables ("cold" SSTables) increases, while the number of frequently accessed SSTables ("hot" SSTables) stays about the same.

2.4 Locality in SSTable Component Accesses

Not only do SSTables have different access frequencies, but also SSTable components have different access frequencies. In this subsection, we analyze how frequently each of the SSTable components are accessed. An SSTable consists of metadata and database records, and metadata includes a Bloom filter and a record index, both of which reduce the read IOs: Bloom filter [13] is for quickly skipping an SSTable when the record doesn't exist in it, and record index is for efficiently locating record offsets.

Component Access Frequencies: To read a record, the database makes a sequence of SSTable component accesses: First, the database checks the keyspace range of the SSTables at each level to find the SSTables that may contain the record. There is at most 1 such SSTable per level thanks to the leveled organization of SSTables (§2.1); in other words, there are at most N SSTables, where N is the number of levels. Second, for each SSTable that passes the keyspace range test, the database checks the Bloom filter to see if the record is in the SSTable or not. Out of the N SSTables, there is only 1 SSTable that contains the record and $N-1$ SSTables that do not. In the former SSTable, the Bloom filter always answers *maybe*[3], in the latter SSTables, the Bloom filter answers *maybe* with a false positive ratio fp. Thus, on average, $1+(N-1)\cdot fp$ read requests go to the next step. The number becomes approximately 1 when fp is very small such as 0.01, which is the case with most of the databases. Third, using the record index, the database locates and reads the record.

[2] Level-0 SSTables are exceptions to the rule, however, databases limit those SSTables to a small number.

[3] Bloom filter tests if an item is in a set or not, and answers a *definitely no* or a *maybe* with a small false positive ratio.

File names (simplified)	Type	Access frequency (relative to Bloom filter)	Size		Access frequency / size
			Bytes	%	
Filter.db	Bloom filter	1	149,848	0.11	869
Summary.db	Index (top)	$\approx \frac{1}{N}$	15,056	0.01	$60/N$
Index.db	Index (bottom)		2,152,386	1.62	
Data.db	Records		130,219,353	98.24	$1/N$

Table 1: *Access frequency and size of SSTable components.* *The sizes are taken from a representative SSTable with 127,167 1-KB records. N is the number of levels.*

Storage type	Cost ($/GB/month)	IOPS
Local SSD	0.528	Varies by the instance type
Remote SSD	0.100	Max 10,000
Remote Magnetic	0.045	Max 500
Remote Magnetic Cold	0.025	Max 250

Table 2: *Cloud vendors provide storage options with various cost and performance.*

Component Sizes: Bloom filter size depends on the false positive ratio, regardless of the number of elements it filters: the smaller the false positive ratio fp is, the bigger the filter becomes. Record index size depends on the number of records in an SSTable and the density of the index entries. Some databases like Cassandra have a top-level, sparse index and a bottom-level, full index, while others such as RocksDB have a single sparse index. Database records account for the majority of the SSTable space.

We present an access frequency and size breakdown of an SSTable by their components in Table 1. *In the order of Bloom filter, record index, and records, the access frequency decreases and the size increases:* the access frequency to size ratio varies by more than 3 orders of magnitudes with a typical number of levels, 3 or more.

2.5 Cloud Storage Opportunities

With such high access frequency disparities among SSTables and among SSTable components, it would be a waste of resources if we were to keep all data files in fast, expensive devices; likewise, it would be a lost opportunity in performance if we were to keep all files in slow, inexpensive devices. Fortunately, cloud vendors provide various storage options with different cost-performance trade-offs: for example, AWS offers various block storage devices as in Table 2. The pricing is based on block storage in the AWS `us-east-1` region in Feb. 2018 [8, 36] (§5.1). We assume that local SSD volumes are elastic, and inferred its unit price (See §5.1). In addition to that, most of the storages are elastic: you use as much storage as needed and pay for just the amount you used, and there is no practical limit on its size. For a simple storage cost model, we assume the storage cost is linear to its space. Premium storages with complex cost models, such as dedicated network bandwidth between a VM and storage or provisioned I/O models, are not considered in this work.

3 SYSTEM DESIGN

Motivated by the strong access locality of SSTables and SSTable components, we design Mutant, a storage layer for LSM-tree non-relational database systems that organizes SSTables and SSTable components into different storage types by their access frequencies, thus taking advantage of the low latency of the fast storage devices and the low cost of slower storage devices.

3.1 SSTable Organization

From the highly skewed SSTable access frequencies that we've observed in §2.3, it becomes straightforward to store SSTables into different storage types. The strategy for organizing SSTables onto fast and slow storage devices depends on operator intentions, defined as an SLO. A prototypical cost-based SLO could be: "*we will pay no more than $0.03/GB/month for the database storage, while keeping the storage latency to a minimum.*" We focus on the cost-based SLO in this work and leave the latency-based SLO as a future work.

3.1.1 Cost SLO-Based Organization.

The above SLO can be broken down into two parts: the optimization goal (a minimum latency) and the constraint ($3/GB/month). Putting hard bounds on cloud storage latencies is challenging for two reasons. First, the exact latency characteristics are unknown – cloud vendors tend not to make latency guarantees on their platforms due to the inherent performance variability associated with multi-tenant environments. Second, SSTables are concurrent data structures with possible contention that can add non-trivial delays. We relax the latency optimization objective into an achievable goal: *maximize the number of accesses to the fast device.* In this paper, we focus on dual storage device configurations – a database with both a fast storage device and a slow one – and leave the multi-level storage configurations as a future work. For completeness, we convert the size constraint similarly. The high-level optimization problem is now as follows:

Find a subset of SSTables to be stored in the fast storage (optimization goal) such that the sum of fast storage SSTable accesses is maximized, (constraint) while bounding the volume of SSTables in fast storage.

First, we translate the cost budget constraint to a storage size constraint, which consists of the two sub-constraints: (a) the total SSTable size is partitioned into fast and slow storage, and (b) the sum of fast and slow storage device costs should not exceed the total storage cost budget.

$$\begin{cases} P_f S_f + P_s S_s \le C_{\max} \\ S_f + S_s = S \end{cases} \tag{1}$$

where $C_{\max}$ is the storage cost budget, or max cost, S is the sum of all SSTable sizes, S_f and S_s are the sum of all SSTable sizes in the fast storage and slow storage, respectively, P_f and P_s are the unit prices for the two storages media types. Solving Eq 1 for S_f gives the fast storage size constraint as in Eq 2.

$$S_f < \frac{C_{\max} - P_s S}{P_f - P_s} = S_{f,\max} \tag{2}$$

We formulate the general optimization goal as:

$$\begin{aligned} &\text{maximize} \sum_{i \in \text{SSTables}} A_i x_i \\ &\text{subject to} \sum_{i \in \text{SSTables}} S_i x_i \le S_{f,\max} \text{ and } x_i \in \{0,1\} \end{aligned} \tag{3}$$

where A_i is the number of accesses to the SSTable i, S_i is the size of the SSTable i, and x_i represents whether the SSTable i is stored in the fast storage or not.

The resulting optimization problem, Eq 3, is equivalent to a 0/1 knapsack problem. In knapsack problems, you are given a set of items that each has a value and a weight, and you want to maximize the value of the items you can put in your backpack without exceeding a fixed a weight capacity. In our problem setting, the value and weight of an item correspond to the size and access frequency of an SSTable; the backpack's weight capacity matches the maximum fast storage device size, $S_{f,\max}$.

3.1.2 Greedy SSTable Organization. The 0/1 knapsack problem is a well-known NP-hard problem and often solved with a dynamic programming technique to give a fully polynomial time approximation scheme (FPTAS) for the problem. However, this approach for organizing SSTables has two complications.

First, the computational complexity of the dynamic programming-based algorithm is impractical: it takes both $O(nW)$ time and $O(nW)$ space, where n is the number of SSTables and W is the number of different sub-capacities to consider. To illustrate the scale, a 1 TiB disk using 64 MiB SSTables will contain 10,000s of SSTables and have 10^{12} sub-capacities to consider since SSTable sizes can vary at the level of bytes. Moreover, this $O(nW)$ time complexity would be incurred every epoch during which SSTables are reorganized.

Second, optimally organizing SSTables at each organization epoch can cause frequent back-and-forth SSTable migrations. Suppose you have a cost SLO of \$3/record and the database uses two storage devices, a fast and a slow storage that cost \$5/record and \$1/record, respectively. Initially, the average cost/record is $2.71 = \frac{5\times3+1\times(2+2)}{3+2+2}$, with the maximum amount of SSTables in the fast storage while satisfying the cost SLO (Figure 6a at time $t1$). When a new SSTable D is added, it most likely contains the most popular items and is placed on the leftmost side (Figure 6a at time $t2$). We assume that the existing SSTables cool down (as seen in §3.1.3), and their relative temperature ordering remains the same. This results in \$3.40/record, temporarily violating the cost SLO; however, at the next SSTable organization epoch, the SSTables are organized with an optimal knapsack solution, bringing the cost down to \$3.00/record (Figure 6a at time $t2'$). Similarly, when another SSTable E is added, the SLO is temporarily violated, but observed soon after (Figure 6a at time $t3$ and $t3'$). During the organizations, SSTable B migrates back-and-forth between storage types, a maneuver that is harmful to read latencies: the latencies can temporarily spike by more than an order of magnitude.

To overcome these challenges, we use a simple greedy 2-approximation algorithm. Here, items are ordered by decreasing ratio of value to weight, and then put in the backpack one at a time until no more items fit. In our problem setting, an item's value to weight ratio corresponds to an SSTable's access frequency divided by its size, which is captured by SSTable temperature (defined in §3.1.3). The computational complexity of the greedy algorithms is $O(n \log n)$ with $O(n)$ space instead of $O(nW)$ for both with dynamic programming. The log-factor in the time stems from the need to keep SSTable references sorted in-place by temperature. However, the instantaneous worst-case access latency of the SSTables chosen to put into fast versus cold storage can be twice that of the dynamic programming algorithm [23], although we rarely see worst-case behavior exhibited in practice. The algorithmic trade-off thus lies between reducing computational complexity versus minimizing SSTable accesses latency.

3.1.3 SSTable Temperature. So far, we have discussed optimal choices moment-to-moment, but access latencies are dynamical quantities that depend on the workload. MUTANT monitors SSTable accesses with an atomic counter for each SSTable. However, naïvely using the counters for prioritizing popular tables has two problems:

- **Variable SSTable sizes:** The size of SSTables can differ from the configured maximum size (64 MiB in RocksDB and 160 MiB in Cassandra). Smaller SSTables are created at the compaction boundaries where the SSTables are almost always not full. Bigger SSTables are created at L0, where SSTables are compacted to each other with a compaction strategy different from leveled compaction such as size-tiered compaction.
- **Fluctuations of the observed access frequency:** The counters can easily be swayed by temporary access spikes and dips: for example, an SSTable can be frequently accessed during a burst and then cease to receive any accesses, a problem arising when a client has a networking issue, or a higher-layer cache effectively gets flushed due to code changes, faults or maintenance. Such temporary fluctuations could cause SSTables to be frequency reorganized.

To resolve these issues, we smooth the access frequencies through an exponential average. Specifically, the *SSTable temperature* is defined as the access frequencies in the past epoch divided by the SSTable size with an exponential decay applied [4]: the sum of the number of accesses per unit size in the current time window and the cooled-down temperature of the previous time window. Naïve application of exponential averages would start temperatures at 0, which interferes with the observation that SSTables start out hot. Instead, we set the initial temperature in a manner consistent with the initial SSTable access frequency as follows:

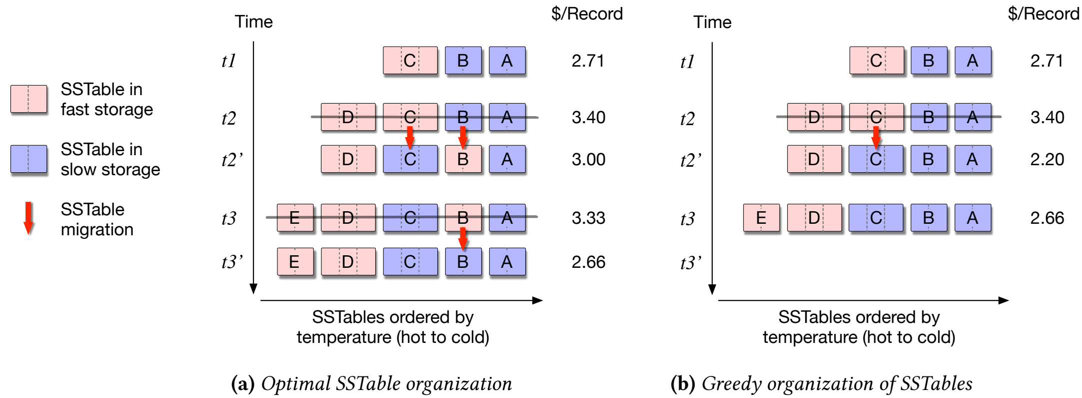
$$T_t = \begin{cases} (1-\alpha) \cdot \dfrac{A_{(t-1,t]}}{S} + \alpha \cdot T_{t-1}, & \text{if } t > 1 \\ \dfrac{A_{(0,1]}}{S}, & \text{if } t = 1 \end{cases} \tag{4}$$

where T_t is the temperatures at time t, $A_{(t-1,t]}$ is the number of accesses to the SSTable during the time interval $(t-1, t]$, S is the SSTable size, and α is a cooling coefficient in the range of $(0, 1]$.

3.2 SSTable Component Organization

We have thus far discussed how MUTANT organizes SSTables themselves by their access frequencies. We discovered that MUTANT can further benefit by considering the *components* of SSTables in

[4] It was inspired by Newton's law of cooling [12].



(a) *Optimal SSTable organization* **(b)** *Greedy organization of SSTables*

Figure 6: ***Greedy SSTable organization reduces SSTable migrations.***

the same light. We observed that the SSTable metadata portion, specifically the Bloom filter and record index, have multiple magnitudes higher access-to-size ratios than the SSTable records portion (see §2.4). Thus, Mutant will strive to keep the metadata on fast storage devices, going so far as to pinning metadata in memory. The trade-off considered here weighs the reduced access latency for metadata because they are always served from memory to the reduced file system cache hit ratio for SSTable records due to the reduced memory available for the file system cache.

The organization of SSTable components depends on the physical layout of an SSTable. On one hand, in databases that store SSTable components in separate files (*e.g.*, Cassandra), Mutant stores the metadata files in a configured storage device such as a fast, expensive one. On the other hand, in databases that store SSTable components all in a single file (*e.g.*, RocksDB), Mutant chooses not to separate out the metadata and records. The latter optimization would involve both implementing a transactional guarantee between the metadata and records, and then rewriting the storage engine and tools. Instead, Mutant keeps the SSTable metadata in memory once it is read. We note that some LSM-tree databases already cache metadata, but only partially: RocksDB provides an option to keep only the L0 SSTable metadata in memory.

4 IMPLEMENTATION

We implemented Mutant by modifying RocksDB, a high-performance key-value store that was forked from LevelDB [7]. The core of Mutant was implemented in C++ with 658 lines of code, and 110 lines of code were used to integrate Mutant with the database.

4.1 Mutant API

Mutant communicates with the database via minimal API consisting of three parts:

Initialization: A database client initializes Mutant with a storage configuration: for example, a local SSD with \$0.528/GB/month and an EBS magnetic disk with \$0.045/GB/month (Listing 1). The storage devices are specified from fast to slow with the (path, unit cost) pairs. A client sets or updates a target cost with `SetCost()`.

SSTable Temperature Monitoring: The database then registers SSTables as they are created with `Register()`, unregisters them as they are deleted with `Unregister()`, and calls `Accessed()` so that Mutant can monitor the SSTable accesses.

```
Options opt;
opt.storages.Add(
  "/mnt/local-ssd1/mu-rocks-stg", 0.528,
  "/mnt/ebs-st1/mu-rocks-stg", 0.045);
DB::Open(opt);
DB::SetCost(0.2);
```

Listing 1: ***Database initialization with storage options***

```
// Initialization
void Open(Options);
void SetCost(target_cost);
// SSTable temperature monitor
void Register(sstable);
void Unregister(sstable);
void Accessed(sstable);
// SSTable organization
void SchedMigr();
sstable PickSstToMigr();
sstable GetTargetDev();
```

Listing 2: ***Mutant API***

SSTable Organization: SSTable Organizer triggers an SSTable migration when it detects an SLO violation or finds a better organization by scheduling a migration with `SchedMigr()`. SSTable Migrator then queries for an SSTable to migrate and to which storage device to migrate the SSTable with `PickSstToMigr()` and `GetTargetDev()`. `GetTargetDev()` is also called by SSTable compactor for the compaction-migration integration we discuss in §4.3.1.

The API and the interactions among Mutant, the database, and the client are summarized in Listing 2 and Figure 7.

4.2 SSTable Organizer

SSTable Organizer (a) updates the SSTable temperatures by *fetch-and-reset*-ting the SSTable read counters and (b) organizes SSTables with the temperatures and the target cost by solving the SSTable placement knapsack problem. SSTable Organizer runs the organization task every organization epoch such as every second. When an

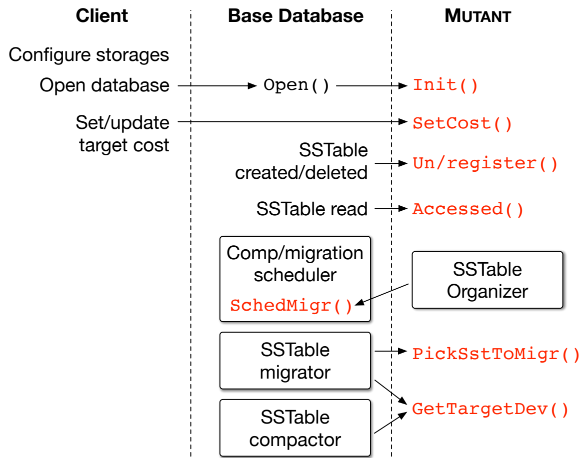


Figure 7: *Interactions among the client, the database, and MUTANT.* *MUTANT API is depicted in red. Parameters and return values are omitted for brevity.*

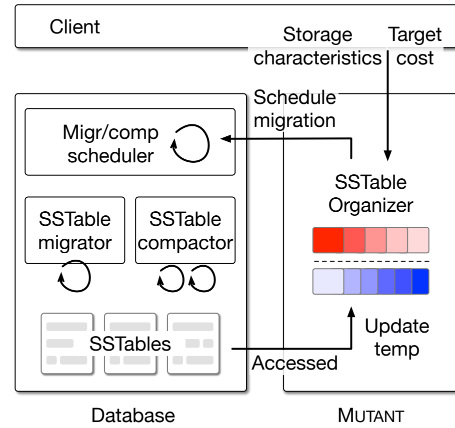


Figure 8: *SSTable Organizer and its interactions with the client and the database.*

SSTable migration is needed, SSTable Organizer asks the database for scheduling a migration. Its interaction with the database and the client is summarized in Figure 8. The two key data structures used are:

SSTable Access Counter: Each SSTable contains an atomic SSTable access counter that keeps track of the number of accesses.

SSTable-Temperature Map: Each SSTable is associated with a `temperature` object that consists of the current temperature and the last update time. The SSTable-Temperature map is concurrently accessed by various database threads as well as SSTable Organizer itself. To provide maximum concurrency, MUTANT protects the map with a two-level locking: (a) a bottom-level lock for frequent reading and updating SSTable temperature values and (b) a top-level lock for far less-frequent adding and removing the SSTable references to and from the map.
SSTable Organizer is concurrently accessed by a number of database threads including:

SSTable Flush Job Thread: registers a newly-flushed SSTable with MUTANT so that its temperature is being monitored.

SSTable Compaction Job Thread: queries MUTANT for the target storage device of the compaction output SSTables. Similar to what the SSTable flush job does, the newly-created SSTables are registered with MUTANT.

SSTable Loader Thread: registers an SSTable with MUTANT, when it opens an existing SSTable.

SSTable Reader Thread: increments an SSTable access counter.

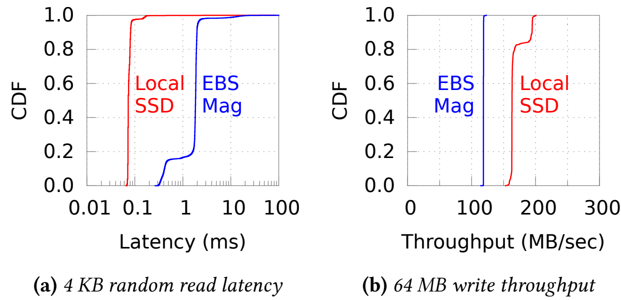
4.3 Optimizations

4.3.1 Compaction-Migration Integration. SSTable compaction and SSTable migration are orthogonal events: the former is triggered by the leveled SSTable organization and the latter is triggered by the SSTable temperature change. However, SSTable compactions cause SSTable temperature changes: when SSTables are compacted together, their records are redistributed among output SSTables by their hashed key order, resulting in similar access frequencies among output SSTables. Consequently, executing SSTable compactions and migrations separately would have caused an inefficiency, the *double SSTable write* problem. Imagine an SSTable in the fast storage is compacted with two SSTables in the slow storage, creating a new SSTable T_a in the fast storage and two new SSTables T_b and T_c in the slow storage. Because their temperatures are averaged due to the record redistribution, T_a's temperature will be low enough to trigger a migration, moving T_a to the slow storage.

Thus, MUTANT piggybacks SSTable migrations with SSTable compactions, which we call *compaction-migration*, effectively reducing the number of SSTable writes, which is beneficial for keeping the database latency low. Note that either an SSTable compaction or an SSTable migration can take place independently: SSTables can be compacted without being moved to a different storage device *(pure compaction)*, and an SSTable can be migrated by itself when SSTable Organizer detects an SSTable temperature change across the organization boundary *(single SSTable migration)*. We analyze how much SSTable migrations can be piggybacked in §5.4.3.

SSTable flushes, although similar to SSTable compactions, are not combined with SSTable migrations. Since the newly flushed SSTables are the most frequently accessed (recall §2.3), MUTANT always writes the newly flushed SSTables to the fast device, obviating the need to combine SSTable flushes and SSTable migrations.

4.3.2 SSTable Migration Resistance. The greedy SSTable organization (§3.1.2) reduces the amount of SSTable churns, the back-and-forth SSTable migrations near the organization temperature boundary. However, depending on the workload, SSTable churns can still exist: SSTable temperatures are constantly changing, and even a slight change of an SSTable temperature can change the temperature ordering. To further reduce the SSTable churns, MUTANT defines *SSTable migration resistance*, a value that represents the number of SSTables that don't get migrated when their temperatures change. The resistance is tunable by clients and provides a trade-off between the amount of SSTables migrated and how adaptive MUTANT is to the changing SSTable temperatures, which



(a) *4 KB random read latency* **(b)** *64 MB write throughput*

Figure 9: *Performance of the storage devices, local SSD and EBS magnetic volumes.* *File system cache was suppressed with direct IO.*

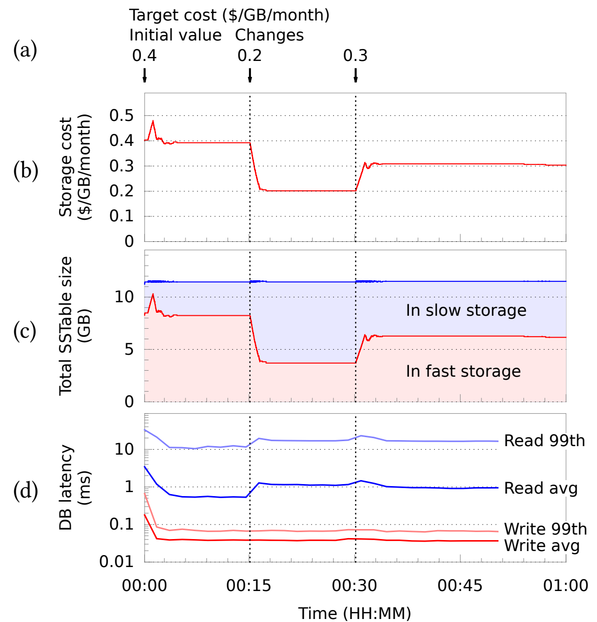


Figure 10: *Mutant makes cost-performance trade-offs seamless.* *Target cost changes over time (a), changes in underlying storage cost (b), Mutant organizes SSTables to meet the target cost (c) and the database latencies (d).*

affects how well Mutant meets the target storage cost. We analyze the trade-off in §5.4.2.

5 EVALUATION

This section evaluates Mutant by answering the following questions:

- How well does Mutant meet a target cost and adapt to a change in cost? (§5.2.1)
- What are the cost-performance trade-offs of Mutant like? (§5.2.2)
- How much computation overhead does it take to monitor SSTable temperature and calculate SSTable placement? (§5.3)
- How much does Mutant benefit from the optimizations including SSTable component organization? (§5.4)

5.1 Experiment Setup

We used AWS infrastructure for the evaluations. For the virtual machine instances, we used EC2 r3.2xlarge instances that come with a local SSD [2]. For fast and slow storage devices, we used a locally-attached SSD volume and a remotely-attached magnetic volume, called EBS st1 type. We measured their small, random read and large, sequential write performances, which are the common IO patterns for LSM tree databases. Compared to the EBS magnetic volume, local SSD's read latency was lower by more than an order of magnitude, and its sequential write throughput was higher by 42% (Figure 9). Their prices were $0.528 and $0.045 per GB per month, respectively. Since AWS did not provide a pricing for the local SSD, we inferred the price from the cost difference of the two instance types, i2.2xlarge and r3.exlarge, which had the same configuration aside from the storage size [35].

For the evaluation workload, we used (a) YCSB, a workload generator for microbenchmarking databases [18] and (b) a real-world workload trace from QuizUp. The QuizUp workload consists of 686 M reads and 24 M writes of 2 M user profile records for 16 days. Its read:write ratio of 28.58:1 is similar to Facebook's 30:1 [10].

5.2 Cost-Performance Trade-Offs

5.2.1 Cost Adaptability. To evaluate the automatic cost-performance configuration, we vary the target cost while running Mutant and analyze its storage cost and database query latency. We used the YCSB "read latest" workload, which models data access patterns of social networks, while varying the target cost: we set the initial target cost to $0.4/GB/month, lowered it to $0.2/GB/-month, and raised it to $0.3/GB/month, as shown in Figure 10(a). We configured Mutant to update the SSTable temperatures every second with the cooling coefficient $\alpha = 0.999$.

Mutant adapted quickly to the target cost changes with a small cost error margin, as shown in Figure 10(b). When the target cost came down from $0.4 to $0.2, about 4.5GB of SSTables were migrated from the fast storage to the slow storage at a speed of 55.7 MB/sec; when the target cost went up from $0.2 to $0.3, about 2.5GB of SSTables were migrated to the other direction at a speed of 34.1 MB/sec. The cost error margin depends on the SSTable migration resistance, a trade-off which we look at in §5.4.2: a 5% SSTable migration resistance was used in the evaluation. Figure 10(c) shows how Mutant organized the SSTables among the fast and slow storages to meet the target costs.

Database latency changed as Mutant reorganized SSTables to meet the target costs (Figure 10(d)). Read latency changes were the expected trade-off as SSTables are reorganized to meet the target costs; write latency was rather stable throughput the SSTable reorganizations. The stable write latency is from (a) records are first written in MemTable not causing any disk IOs, then batch-written to the disk, minimizing the IO overhead and (b) commit log is always written to the fast storage device regardless of the target cost. At the start of the experiment, the latency was high and the cost was unstable because the file system cache was empty and the SSTable temperature needed a bit of time to be stabilized. Shortly after, the latency dropped and the cost stabilized.

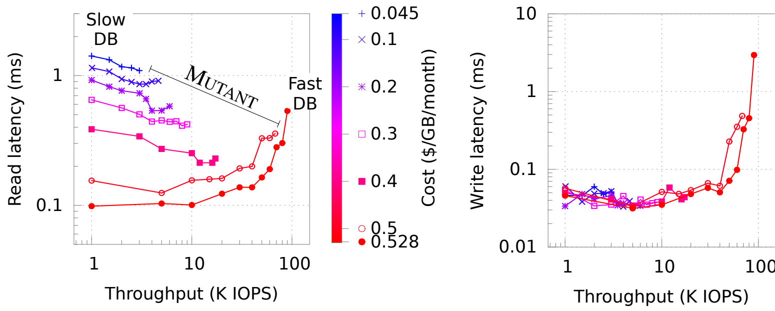


Figure 11: *Cost-performance trade-off spectrum of Mutant.* *Read latency (left) and write latency (right) controlling throughput (horizontal axis) by varying the target cost. Colors and symbols represent storage cost.*

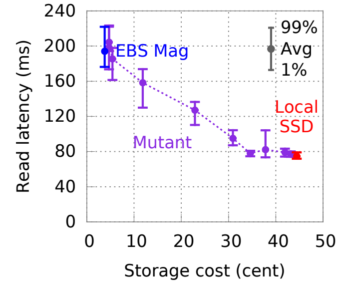


Figure 12: *Cost-latency trade-off with the QuizUp workload trace.* *Fast and slow databases are shown in red and blue, respectively. Mutant with various target costs is shown in purple.*

5.2.2 Trade-Off Spectrum.

We study the cost-performance trade-off spectrum by analyzing both database throughput (in IOPS) and latency as the target cost changes. We first set the baseline points with two unmodified databases: *fast database* and *slow database*, as shown in Figure 11(left). *Fast database* used a local SSD volume and had about 12× higher storage cost, 20× higher maximum throughput, and 10× lower read latency than *slow database* that used an EBS magnetic volume.

The cost-read latency trade-off is shown in Figure 11(left). As we increased the target cost from the lower bound (*slow database*'s cost) to the upper bound (*fast database*'s cost), the read latency decreased proportionally. The throughput-latency curves show some interesting patterns. First, as you increase the throughput, the latency increases: *fast database*. This is because the performance bottleneck was the CPU, and the database saturated when the CPU usage was at 100%. Second, as you increase the throughput, the latency decreases: *slow database*. The latency decrease was from the batching of the read IO requests at the file system layer. The maximum throughput was 3 K IO/sec due to the rate limiting of the EBS volume [8], rather than the CPU getting saturated. Third, as you increase the throughput, the latency initially decreases and then increases: Mutant. The latency changes are the combined effect of the benefit of IO batching in slow storage and the saturation of CPU.

The write latencies stayed about the same throughout the target cost changes (Figure 11(right)). The result was as expected, since the slow storage is not directly in the write path: records are batch-written to the slow storage asynchronously. Figure 11 confirms that Mutant delivers the cost and maximum throughput trade-off: as the target cost increased, the maximum throughput increased.

The evaluation with the QuizUp workload again confirms that Mutant delivers a seamless cost-latency trade-off. Similar to with the YCSB workload, we replayed the QuizUp workload with two baseline databases and Mutant with various cost configurations as shown in Figure 12.

5.2.3 Comparison with Other SSTable Organizations.

We compare the cost configurability of Mutant and the other SSTable organization algorithms, leveled organization and round-robin organization used by RocksDB and Cassandra. RocksDB organizes SSTables by their levels into the storage devices [6]. Starting from

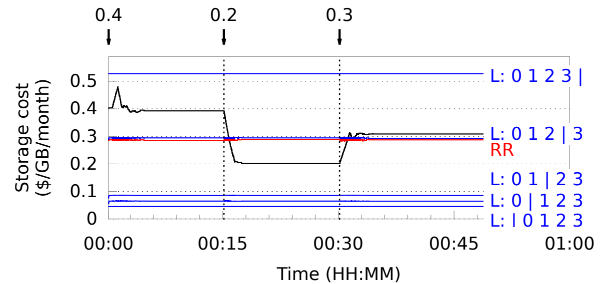


Figure 13: *Storage cost of Mutant and the other SSTable organization strategies.* *The blue line (with L) represents RocksDB's leveled organization. SSTables at a level before the symbol | go to the fast storage; SSTables at a level after the symbol go to the slow storage. The red line (with RR) represents Cassandra's round-robin organization, and the black line represents the seamless organization of Mutant.*

level 0 and the first storage device, RocksDB stores all SSTables at current level in the current storage only if all the SSTables can fit in the storage; if the SSTables don't fit, RocksDB looks at the next storage device to see if the SSTables can fit. Cassandra spreads data to storages in a round-robin manner proportional to the available space of each of the storages [3].

Figure 13 compares the storage cost of Mutant and the other SSTable organization algorithms. First, neither of the algorithms is adaptive to the changing target cost. When the target cost changes, your only option is migrating your data to a database with a different cost-performance characteristic. Second, leveled SSTable organization has limited number of configurations. With n SSTable levels and 2 storage types, SSTables can be split in $n + 1$ different ways. Thus, even when assuming target cost is to be maintained, there is a limited number of cost-performance options.

5.3 Computational Overhead

Computational overhead includes the extra CPU cycles and the amount of memory needed for the SSTable temperature monitoring and SSTable placement calculation. The overhead depends on the number of SSTables: the bigger the number of SSTables, the more CPU and memory are used for monitoring the temperatures and

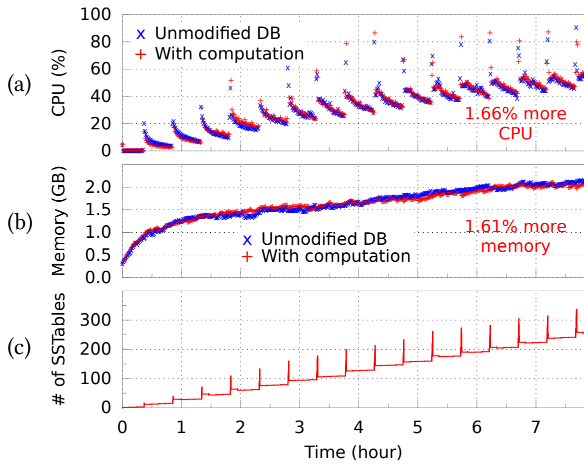


Figure 14: ***Computation overhead of Mutant.*** *Figure (a) and (b) show the CPU and memory usage of the unmodified database (in blue) and Mutant with SSTable temperature monitoring and SSTable organization calculation on (in red). Figure (c) shows the total number of SSTables.*

calculating the SSTable placement. We ran the YCSB workload with 10K IOPS for 8 hours both with and without the computation overhead. For a precise measurement, we disabled SSTable migrations to prevent the filesystem from consuming extra CPU cycles.

The modest overhead shown in Figure 14 confirms the efficient design and implementation of Mutant: (a) minimal CPU overhead through an atomic counter placed in each SSTable and periodic temperature updates, (b) minimal memory overhead from the use of the exponential decay model in SSTable temperature, and (c) the greedy SSTable organization that keeps both CPU and memory overhead low. Through the experiment, the system consumed 1.67% and 1.61% extra CPU and memory on average. The peaks in the CPU usage are aligned with SSTable compactions that trigger rewrites for a large number of SSTables. There were fluctuations in the overhead over time: the overhead was positive at one minute and negative at the next. Likely explanations include (a) the non-deterministic key generation in YCSB, which affects the total number of records at a specific time between runs, which in turn influences the timing of when SSTable compactions are made and when the JVM garbage collector kicks in and (b) the inherent performance variability in the multi-tenant cloud environment.

5.4 Benefits from Optimizations

5.4.1 SSTable Component Organization. To evaluate the benefit of the SSTable component organization, we measure the latencies of the unmodified database and the database with the SSTable component organization turned on. Since RocksDB SSTables store metadata and records in the same file, we keep the metadata in memory instead of moving the metadata in the file system. SSTable component organization benefited the "slow database" (the database with an EBS magnetic volume) significantly both in terms of the average and tail latencies, as in Figure 15. The latency reduction

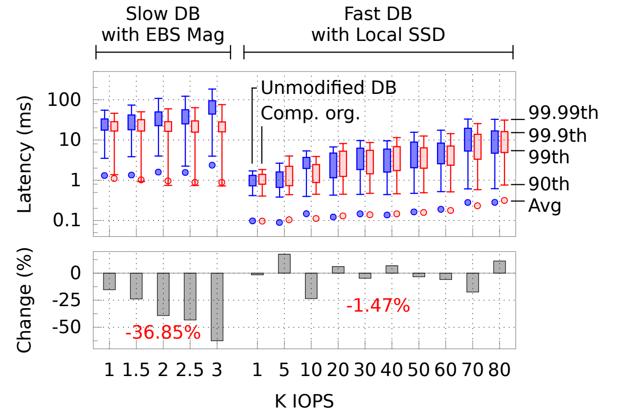


Figure 15: ***"Slow database" benefited significantly from SSTable component organization.*** *A latency comparison of the unmodified database (in blue) and the database with SSTable component organization on (in red).*

came from avoiding (a) reading metadata, the most-frequently accessed data blocks, from the storage device and (b) unmarshalling the data into memory. The metadata caching evaluation was fair in terms of the total memory usage: a trade-off of allocating slightly more memory SSTable metadata and slightly less memory for the file system cache to SSTable records. In the experiment, when the metadata caching was on, the database process used 2.22% more memory on average and the file system cache used 2.22% less memory.

The latency benefit to the "fast database" was insignificant. which, we think, was due to the significantly lesser file system cache miss penalty of the local SSD volume, such as from the DRAM caching provided by many SSDs today, compared to the EBS magnetic volume (§5.1).

5.4.2 SSTable Migration Resistance. SSTable migration resistance serves as a trade-off knob between the amount of SSTables migrated and the storage cost conformance (§4.3.2). We vary SSTable migration resistance and analyze its effect on the trade-off between the SSTable migration reduction and the target cost. As we increased SSTable migration resistance, the number of migrations first decreased and eventually plateaued out. The plateau point depends on the workload: with the YCSB "read latest" workload, the plateau happened at around 13% resistance, as in Figure 16(a). The storage cost increased as the migration resistance increased, as in Figure 16(b). This is because, with modern web workload of which most SSTables are migrated towards the slow storage device, a high resistance makes SSTables stay longer in the fast, expensive storage device than a low resistance. Storage cost increase was linearly bounded to SSTable migration resistance: in the experiment, with a ratio of about 0.08 (relative cost / SSTable migration resistance). One should configure the migration resistance between 0 and the plateau point of the number of SSTables migrated (13% in this example), because there is no more benefit from the SSTable migration reduction beyond the plateau point.

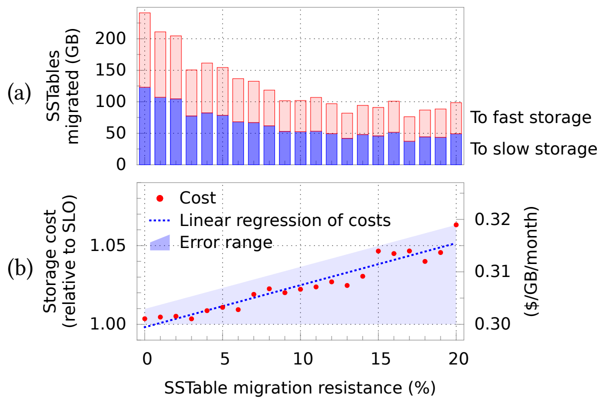


Figure 16: *SSTable migration resistance's effect on the SSTable migrations and cost SLO conformance.* *Figure (a) shows how the amount of SSTable migrations changes by SSTable migration resistance, and (b) shows how the storage cost changes.*

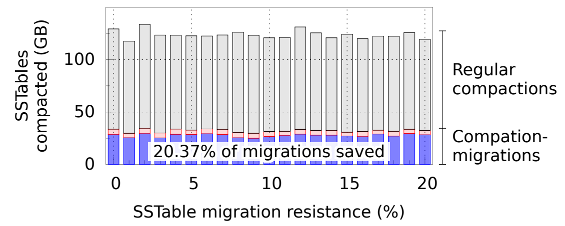


Figure 17: *Breakdown of SSTable compactions by SSTable migration resistance.*

5.4.3 **SSTable Compaction-Migration.** SSTable compaction-migration integration is an optimization that piggybacks SSTable migrations on SSTable compactions, thus reducing the amount of SSTable migrations (§4.3.1). The breakdown of the SSTable compactions in Figure 17 shows that 20.37% of SSTable migrations were saved from the integration. The number of SSTable compactions remained consistent throughout the SSTable migration resistance range, since the compactions were triggered solely by the leveled SSTable organizations, independent of the SSTable temperature changes.

6 RELATED WORK

LSM Tree Databases: LSM trees, invented by O'Neil [33], have become an attractive data structure for database systems in the past decade owing to their high write throughput and suitability for serving modern web workloads [1, 7, 16, 21, 25], These databases organize SSTables, the building blocks of a table, using various strategies that strike different read-write performance trade-offs: (a) size-tiered compaction used by BigTable, HBase, and Cassandra, (b) leveled compaction used by Cassandra, LevelDB, and RocksDB, (c) time window compaction used by Cassandra, and (d) universal compaction used by RocksDB [4, 20]. MUTANT uses leveled compaction for its small SSTable sizes, which allows SSTables to be organized across different storage types with minimal changes to the underlying database. SSTable sizes under leveled compaction are 64 MiB in RocksDB or 160 MiB in Cassandra by default; with the other compaction strategies, there is no upper bound on how much an SSTable can grow.

Optimizations to LSM tree databases include bLSM, which varies the exponential fanout in the SSTable hierarchy to bound the number of seeks [34], Partitioned Exponential Files that exploits properties of HDD head schedulers [22], WiscKey that separates keys from values to reduce write amplification [29], and work of Lim *et al.* that analyzes and optimizes the SSTable compaction parameters [27]. These optimizations are orthogonal to how MUTANT organizes SSTables and can complement our approach.

Multi-Storage, LSM Tree Databases: Several prior works use LSM tree databases across multiple storages. Time-series databases such as LHAM [32] splits the data at the the component (B+ tree) boundaries, storing lower level component in slower and cheaper storages. RocksDB organizes SSTables by levels and store lower-level SSTables in slower and cheaper storages [6]. Cassandra stores SSTables to storages in a round-robin manner to guarantees even usage of storage devices [3].

In comparison, the cost-performance trade-offs of these approaches lack both configurability and versatility. First, databases are deployed based on a static cost-performance trade-off, independent of the database's lifetime. Any modifications and adjustments involve laborious data migration. Second, the trade-offs are limited in options. Both with LHAM and RocksDB's leveled SSTable organization, the data is split in a coarse-grained manner. LHAM partitions data at the the component (B+ tree) boundaries, leading to only a small number of components since the components grow exponentially in size. Leveled SSTable organization, which partitions data at the level boundaries, typically produces at most 4 to 5 levels. Cassandra's round-robin organization provides only one option, dividing SSTables evenly across storages.

These multi-storage, LSM tree database storage systems share the same idea as MUTANT: separating data into different storages based on their cost-performance characteristics. To the best of our knowledge, however, MUTANT is the first to provide a seamless cost-performance trade-off, by taking advantage of the internal LSM tree-based database store layout, the data access locality from modern web workloads, and the elastic cloud storage model.

7 CONCLUSIONS

We have presented MUTANT: an LSM tree-based NoSQL database storage layer that delivers seamless cost-performance trade-offs with efficient algorithms that captures SSTable access popularity, organizes SSTables into different types of storage devices to meet the changing target cost. Future work includes exploring (a) latency SLO enforcement in addition to the cost SLO enforcement and (b) finer-grained storage organization that addresses microscopic, record-level access frequency changes.

ACKNOWLEDGMENT

We would like to thank Avani Gadani, Helga Gudmundsdottir, and the anonymous reviewers for their valuable suggestions and comments. This work was supported by AWS Cloud Credits for Research program and NSF CAREER #1553579.

REFERENCES

[1] 2016. Apache HBase. https://hbase.apache.org
[2] 2017. Amazon EC2 Instance Types. https://aws.amazon.com/ec2/instance-types
[3] 2017. Cassandra source code - Directories. https://github.com/apache/cassandra/blob/684e250ba6e5b5bd1c246ceac332a91b2dc90859/src/java/org/apache/cassandra/db/Directories.java#L368
[4] 2017. DSE 5.1 Architecture Guide - How is data maintained? https://docs.datastax.com/en/dse/5.1/dse-arch/datastax_enterprise/dbInternals/dbIntHowDataMaintain.html
[5] 2017. QuizUp - The Biggest Trivia Game in the World. http://research.quizup.com
[6] 2017. RocksDB source code - Compaction picker. https://github.com/facebook/rocksdb/blob/e27f60b1c867b0b60dbff26d7a35777e6bb9f14b/db/compaction_picker.cc#L1286
[7] 2018. RocksDB - A persistent key-value store for fast storage environments. http://rocksdb.org
[8] Amazon Web Services. 2018. Amazon EBS Volume Types. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/EBSVolumeTypes.html
[9] Siddharth Anand. 2015. How big companies migrate from one database to another. https://www.quora.com/How-big-companies-migrate-from-one-database-to-another-without-losing-data-i-e-database-independent.
[10] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS Performance Evaluation Review*, Vol. 40. ACM, 53–64.
[11] Shobana Balakrishnan, Richard Black, Austin Donnelly, Paul England, Adam Glass, David Harper, Sergey Legtchenko, Aaron Ogus, Eric Peterson, and Antony IT Rowstron. 2014. Pelican: A Building Block for Exascale Cold Data Storage. In *OSDI*. 351–365.
[12] Theodore L Bergman and Frank P Incropera. 2011. *Fundamentals of heat and mass transfer*. John Wiley & Sons.
[13] Burton H Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* 13, 7 (1970), 422–426.
[14] Anders Brodersen, Salvatore Scellato, and Mirjam Wattenhofer. 2012. Youtube around the world: geographic popularity of videos. In *Proceedings of the 21st international conference on World Wide Web*. ACM, 241–250.
[15] Brian Bulkowski. 2017. Amazon EC2 I3 Performance Results. https://www.aerospike.com/benchmarks/amazon-ec2-i3-performance-results
[16] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. 2008. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)* 26, 2 (2008), 4.
[17] Dennis Colarelli and Dirk Grunwald. 2002. Massive arrays of idle disks for storage archives. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*. IEEE Computer Society Press, 1–11.
[18] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*. ACM, 143–154.
[19] Jonathan Ellis. 2011. Leveled Compaction in Apache Cassandra. http://www.datastax.com/dev/blog/leveled-compaction-in-apache-cassandra
[20] Facebook. 2017. RocksDB Wiki - Compaction. https://github.com/facebook/rocksdb/wiki/Compaction
[21] Sanjay Ghemawat and Jeff Dean. 2017. LevelDB. https://github.com/google/leveldb
[22] Christopher Jermaine, Edward Omiecinski, and Wai Gen Yee. 2007. The partitioned exponential file for database storage management. *The VLDB Journal-The International Journal on Very Large Data Bases* 16, 4 (2007), 417–437.
[23] Jon Kleinberg and Eva Tardos. 2006. *Algorithm design*. Pearson Education India.
[24] Sanjeev Kumar. 2014. Efficiency at Scale. *International Workshop on Rack-scale Computing* (2014).
[25] Avinash Lakshman and Prashant Malik. 2010. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review* 44, 2 (2010), 35–40.
[26] Jim Liddle. 2008. Amazon found every 100ms of latency cost them 1% in sales. *The GigaSpaces* 27 (2008).
[27] Hyeontaek Lim, David G Andersen, and Michael Kaminsky. 2016. Towards Accurate and Fast Evaluation of Multi-Stage Log-structured Designs. In *FAST*. 149–166.
[28] Steve Lohr. 2012. For Impatient Web Users, an Eye Blink Is Just Too Long to Wait. http://www.nytimes.com/2012/03/01/technology/impatient-web-users-flee-slow-loading-sites.html
[29] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2016. WiscKey: Separating Keys from Values in SSD-conscious Storage. In *FAST*. 133–148.
[30] Microsoft Azure. 2017. High-performance Premium Storage and managed disks for VMs. https://docs.microsoft.com/en-us/azure/virtual-machines/windows/premium-storage
[31] Subramanian Muralidhar, Wyatt Lloyd, Sabyasachi Roy, Cory Hill, Ernest Lin, Weiwen Liu, Satadru Pan, Shiva Shankar, Viswanath Sivakumar, Linpeng Tang, et al. 2014. f4: Facebook's warm BLOB storage system. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. 383–398.
[32] Peter Muth, Patrick O'Neil, Achim Pick, and Gerhard Weikum. 2000. The LHAM log-structured history data access method. *The VLDB Journal-The International Journal on Very Large Data Bases* 8, 3-4 (2000), 199–221.
[33] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Informatica* 33, 4 (1996), 351–385.
[34] Russell Sears and Raghu Ramakrishnan. 2012. bLSM: a general purpose log structured merge tree. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. ACM, 217–228.
[35] Amazon Web Services. 2017. Amazon EC2 Pricing. https://aws.amazon.com/ec2/pricing
[36] Amazon Web Services. 2018. Amazon EBS Pricing. https://aws.amazon.com/ebs/pricing
[37] Linpeng Tang, Qi Huang, Amit Puntambekar, Ymir Vigfusson, Wyatt Lloyd, and Kai Li. 2017. Popularity Prediction of Facebook Videos for Higher Quality Streaming. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association.
[38] Mingyuan Xia, Mohit Saxena, Mario Blaum, and David Pease. 2015. A Tale of Two Erasure Codes in HDFS. In *FAST*. 213–226.