COST-CONFIGURABLE CLOUD STORAGE SYSTEM ARCHITECTURE DESIGNS

A Thesis
Presented to
The Academic Faculty

by

Hobin Yoon

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
College of Computing

Georgia Institute of Technology
May 2019

COST-CONFIGURABLE CLOUD STORAGE SYSTEM ARCHITECTURE DESIGNS

Approved by:

Dr. Ada Gavrilovska, Advisor
College of Computing
Georgia Institute of Technology

Dr. Ling Liu
College of Computing
Georgia Institute of Technology

Dr. Ymir Vigfusson
Department of Math and Computer Science
Emory University & Reykjavik University

Dr. Kishore Ramachandran
College of Computing
Georgia Institute of Technology

Dr. Karsten Schwan
College of Computing
Georgia Institute of Technology

Dr. Calton Pu
College of Computing
Georgia Institute of Technology

Date Approved: 13 February 2019

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Today's cloud storage systems lack flexible cost-performance trade-offs. For example, (a) in database systems, there are only a limited number of cost-performance options and they are not seamless, (b) in cloud caching systems, there is no flexibility in performance isolation, and (c) in geo-replication systems, the cost-performance trade-off is not optimal to various application types.

In this thesis, we present novel system designs that offer greater flexibility for making finer, online cost-performance trade-offs for data storage systems using (a) data access statistics and (b) models that capture information regarding cost and user experience. We specifically look at ways of achieving better cost-latency trade-offs in the following problem domains: (Mutant) NoSQL database systems, (SpaceLease) cloud caching systems, and (Acorn) geo-replicated, multi-data center systems.

With NoSQL database storage systems, we observe the inflexibility in the cost and performance trade-offs: the trade-offs have limited options and the transition between different cost-performance points are not automatic. We address the inflexibility by proposing Mutant, a NoSQL database storage layer that seamlessly trades off cost and performance. We implemented Mutant by modifying RocksDB, a popular, high-performance NoSQL database, and evaluated with both synthetic and real-world workloads to demonstrate the seamless and automatic cost-performance trade-offs.

With edge cloud caching systems, we observe the unpredictable performance in public cloud cache services: CPs (content providers) pay the same amount of price,

but they get unstable cache hit rate over time. We address the performance unpredictability by proposing SpaceLease, a performance-isolated cache architecture that uses dedicated resource for caching data in the edge cloud platform. We implemented SpaceLease and showed up to 79% reduction in the performance variability with a minimal cost overhead. In addition to the stable performance, SpaceLease also (a) provides a control that trades off cost and hit rate, (b) maximizes the aggregate cache utility across data centers, and (c) adapts quickly to changing workload patterns.

With geo-distributed multi-data center replication systems, we observe that (a) better replication decisions can be made by using the "right" object attribute for each application type, such as *topics* for public video sharing applications and *users* for social network applications, and (b) using the combinations of the attributes and extra random replicas makes better replications under a cost or latency constraint. In response, we developed Acorn, an attribute-based partial geo-replication system, and showed that Acorn delivers up to a 90% cost reduction or a 91% latency reduction.

CHAPTER I

COST-PERFORMANCE TRACE-OFFS IN CLOUD STORAGE SYSTEMS

"We live in a world full of data, and it's expanding at astonishing rates."

– Data never sleeps [62].

The volume and the growth rate of data is extraordinary. First, the data volume is enormous: the total volume of data in data centers is predicted to be as big as 2.6 ZB (zettabytes, 10^{21}) by 2021 [38]. To put the number into perspective, 2.6 ZB is the sum of disk sizes when each and every one in the US buys 8 1-TB disks. Second, the data growth rate is very high. For example, every minute, people post 50 K photos on Instagram and 500 K tweets on Twitter [62]. By 2021, global data center IP traffic is expected to grow in threefolds, with an CAGR (compound annual growth rate) of 25% [38]. Thus, the cost of storing such exploding volume of data would be massive, and it is no surprise that companies strive to find ways to lower storage cost.

At the same time, companies are finding it difficult to distribute data while meeting the key performance indicators, which are often expressed by access latencies. There is evidence that data access latencies are directly related to the level of user experience and company revenue. For example, (a) Google reports that people switch to a competitor website due to a delay as small as 250 ms [75], (b) Amazon reports every 100 ms of delay costs the company 1% in sales [71], and (c) Bing and Google agree that slow pages lose users [99]. Due to the scale of these web service companies, even a small improvement in latency would translate to a huge increase in revenue. Out of the overall service latency, data access time is often the bottleneck in many systems including: NoSQL database systems [117], Big data systems [40], and Data

analytics systems [65].

The two goals – achieving low cost and achieving high performance – are conflicting by nature, and each business has a unique requirement in the cost-performance spectrum. However, **today's cloud storage systems lack flexible cost-performance trade-offs**, and businesses need to settle for suboptimal cost-performance options. We look at the lack of trade-offs in three different system domains: database systems, cloud caching systems, and geo-replication systems.

- **NoSQL Database Systems:** Cloud infrastructure vendors today offer a wide range of storage types on which a NoSQL database system can run: from fast, expensive storage devices to slow, inexpensive ones. The storage options differ in their storage media type (e.g., SSD, HDD), storage system architecture (e.g., locally attached storage or RAID storage attached remotely), and encoding techniques (e.g., different configurations of erasure coding) [80, 56, 9]. The storage options allow building a database with a specific cost-performance trade-off, however **the trade-off options are very limited and the database systems can not make a seamless transition from one trade-off point to another.**

- **Edge Cloud Caching Systems:** Cloud cache providers today offer a pay-as-you-go web caching service to help small to medium businesses deploy web caches without a large upfront investment. However, the cloud cache services are best effort: they neither make any performance isolation guarantees nor they provide any cost-performance trade-off. Thus, **CPs (content providers), which are the clients of the cache services, do not have any control over the performance isolation level or cache hit rate they experience using a cost knob.**

- **Geo-Replicated Storage Systems:** Global web services replicate their data in

geographically distributed data centers for high service availability and fault tolerance [30]. To lower the increasing cost of data replication, partial geo-replication schemes have been proposed. However, **existing partial replications are (a) static in the replication attributes they use and (b) do not provide flexibility in the cost and data access hit rate trade-offs.**

At the core of the each problem domain lies the **data placement problem**: (a) how database records should be placed between fast, expensive storage devices and slow, inexpensive storage devices, (b) how cache resource should be allocated to each edge data center of a CP (content provider), and (c) how data should be replicated into different data centers.

Arriving at an optimal data placement solution in terms of cost and benefits, i.e., the data storage and movement cost and data access performance, requires a deep characterization of (a) the data access patterns and benefit metrics and (b) the infrastructure resource parameters and their costs. Achieving the optimal data placement in practice is hard for the examples mentioned above, because of the inherent variability in their workloads and the complexity of the decision space.

Because of the difficult requirements, people settle for practical solutions with policies that use simple metrics and data placement heuristics when making data placement decisions such as where to place the data in the storage hierarchy, how much storage resource should be allocated in the storage locations, or whether to replicate a particular data item to a data center. Examples of such practical solutions include migrating objects that is older than an age threshold to an inexpensive, cold storage [84], BLOB storage systems triplicating data such as Facebook Haystack [23], and partially geo-replicated database replicating data on third access [63].

Those solutions, while practical, eschew potential opportunities for further optimizations for a given workload to arrive at concrete discrete point in the cost-benefit space. As the volume and value of data continue to grow, ignoring these optimization

opportunities can lead to a significant increase in the incurred data placement costs or in the lost opportunity to provide the desired data access experience to end users. This is further exacerbated by the fact that the types of applications benefiting from these types of data management frameworks continue to increase, thus making it less likely that the default policies are well adjusted for the characteristics of these new workloads.

An immediate question, explored in this thesis, is: **are there practical solutions that can provide finer-grained controls in the achievable cost-benefit trade-off for arbitrary workloads?** In other words, we focus on providing flexible cost-benefit trade-offs, and specifically explore new methods that can improve and customize the trade-offs in a workload-specific manner.

1.1 Thesis Statement

Operational cost is a primary concern for today's tech startups, with the cloud having made a way with the up front capital expenditure and folding these costs into their pay-as-you-go resource prices. Yet today's data processing and storage systems don't treat cost as a design goal, resulting in systems without an explicit cost control knobs. The inflexible-cost system design resulted in cloud applications that are not able to fine-tune their cost to their requirements. As cloud workloads continue to scale in terms of data volume and data demand, **without new fine-grained cost-performance controls, cloud data storage technologies will lead to prohibitive data storage costs for cloud customers, or will fail to deliver desired performance guarantees.**

In this thesis, **we present cost-configurable cloud storage system design principles, where cost is a first-class citizen.** First, we characterize the data access patterns in various dimensions, such as object keyspace, time, and geographic

location, using new models and metrics. Then, we identify the data placement strategies that provide fine-grained trade-offs between the storage system cost and performance. The systems following the data placement strategies provide configurable cost while achieving the highest benefits.

We apply the data placement design principles to several problem domains including (a) data organization in LSM tree-based NoSQL database systems, (b) edge cloud caching systems, and (c) proactive data replication in geo-replicated, multi-data center systems, and show it is possible to achieve fine-grained cost-performance trade-offs in storage systems.

1.2 Contributions

Towards the vision of providing flexible cost-performance trade-offs in the cloud storage systems, I've worked on:

- MUTANT: An LSM tree-based NoSQL database storage layer that provides seamless and automatic cost-performance trade-offs by monitoring the access frequencies of the database storage blocks and partitioning the blocks into different storage device classes (Chapter 2).
- SPACELEASE: A deploy-your-own cloud cache architecture with flexible cost-performance trade-offs that provides CPs a control over the cost, performance isolation level, and cache hit rate (Chapter 3).
- ACORN: A geo-replicated storage system with better cost and user experience trade-offs by making improved geo-replication decisions using object access statistics and object attributes (Chapter 4).

In Chapter 5, we discuss the related work and, Chapter 6 concludes this thesis and present ideas for future research.

CHAPTER II

IN LSM TREE DATABASES

In this chapter, we explore the cost-performance trade-offs in LSM tree-based NoSQL database systems. Traditional databases have limited options in their cost and performance, and making a transition between a cost-performance point to another is manual and time-consuming. We explore the database storage system designs for flexible cost-performance trade-offs and demonstrate how the system designs enable such trade-offs.

2.1 Cost-Performance Trace-Offs in LSM Tree Databases

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[2]. Companies, however, also wish to minimize their database latencies since high user-perceived response times of websites lose users [76] and decrease revenue [71]. Since magnetic storage is a common latency bottleneck, cloud vendors offer premium storage media like enterprise-grade SSDs [9, 81] and NVMe SSDs [32]. Yet even with optimizations on such drives, like limiting power usage [39, 22] or expanding the encoding density [84, 116], 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

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

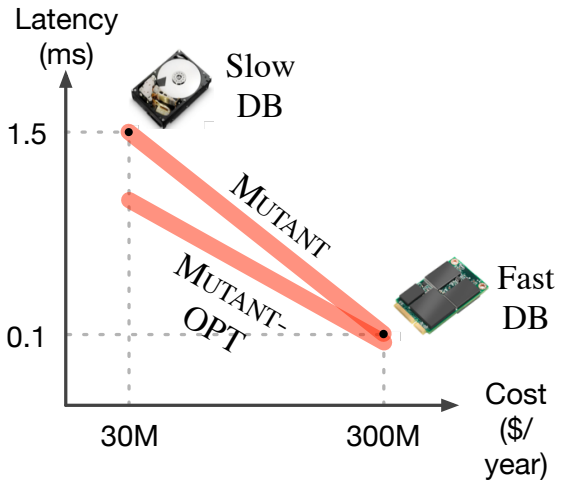


Figure 1: **Mutant provides seamless cost-performance trade-offs** between "fast database" and "slow database", and better cost-performance trade-offs with an extra optimization.

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 applications, a process that can take months as in the case of Netflix [15]. 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 [105] or per-level storage mapping in RocksDB [50], 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

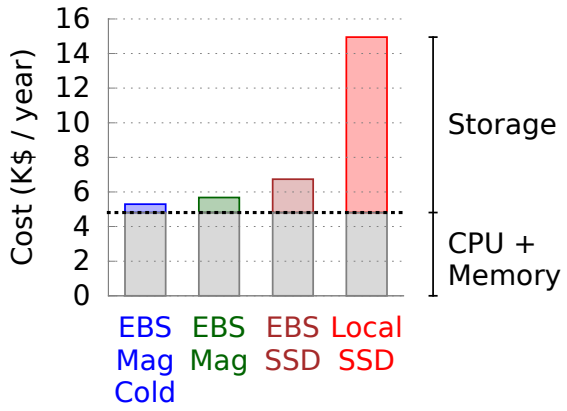


Figure 2: **Fast storage is costly.** Cost to host an IaaS database instance for 1 year using various storage devices. Storage cost can be more than 2× the cost of CPU and memory. Based on the pricing of EC2 `i2.2xlarge` instance (I/O-optimized with 1.6 TiB storage device) [9].

MUTANT: a layer for log-structured merge tree (LSM-tree) data stores [89, 54] 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 [31, 69]. 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 [52]. We evaluated our implementation on a trace from the backend database of a real-world application (the QuizUp trivia app) and the YCSB benchmark [41]. 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.

This chapter 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.

2.2 Data Accesses in LSM Tree Databases

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.

2.2.1 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 [33, 70, 54, 52, 106]. 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 [89].

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 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 [89] that was made popular by LevelDB, and has been adopted by Cassandra and RocksDB [48, 52].

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[1]. 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 [72, 54, 48, 50].

2.2.2 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 [69, 103, 31]. 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 [94]. Figure 3 shows a sharp drop of record accesses as records become old.

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

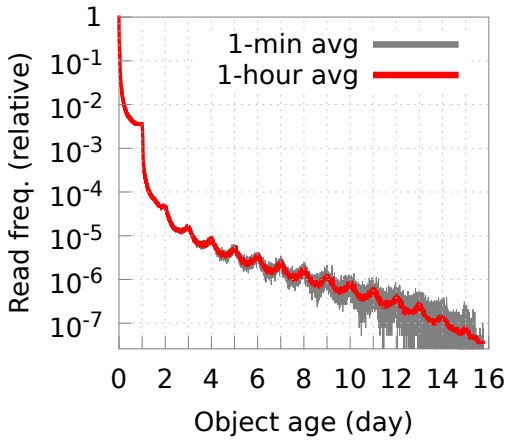


Figure 3: **Record age and its access frequency** from the QuizUp data access trace. Access frequencies are aggregated over all records and normalized.

2.2.3 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.

2.2.4 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 [27] is for quickly skipping an SSTable when the record

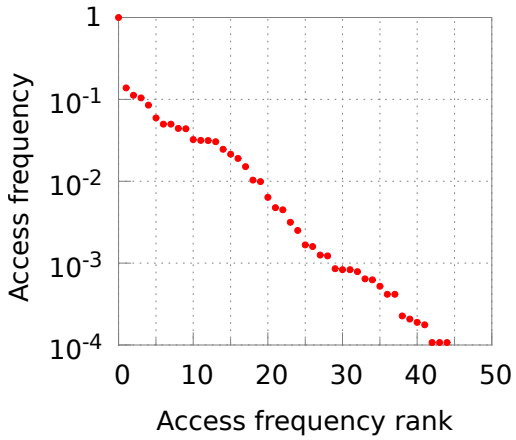


Figure 4: **SSTable access frequency distribution** on day 15 of the 16-day QuizUp trace replay.

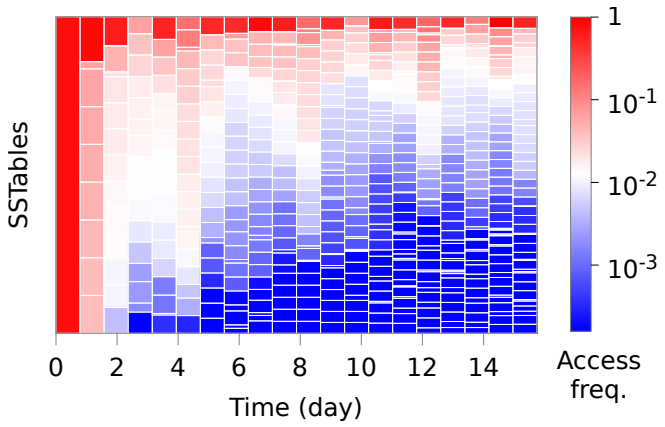


Figure 5: **SSTable access frequencies over time.** Each rectangle represents an SSTable with a color representing its access frequency. SSTable heights at the same time are drawn proportionally to their sizes.

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 (§2.2.1); 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*[1], 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.

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.

2.2.5 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

[1]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.

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.

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

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

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 [102, 9] (§2.5.1). We assume that local SSD volumes are elastic, and inferred its unit price (See §2.5.1). 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.

2.3 System Design for Seamless Cost-Performance Trade-Offs

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.

2.3.1 SSTable Organization

From the highly skewed SSTable access frequencies that we've observed in §2.2.3, 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.

2.3.1.1 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 chapter, 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.

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

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:

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

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, $S_{f,\max}$.

2.3.1.2 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 §2.3.1.3), 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

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

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

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 §2.3.1.3). 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 [67], 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.

2.3.1.3 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 [1]: 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

[1] It was inspired by Newton's law of cooling [25].

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].

2.3.2 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 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 §2.2.4). 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.

2.4 Implementation

We implemented MUTANT by modifying RocksDB, a high-performance key-value store that was forked from LevelDB [52]. 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.

2.4.1 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 2.1). 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.

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 §2.4.3.1.

The API and the interactions among MUTANT, the database, and the client are summarized in Listing 2.2 and Figure 7.

```
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 2.1: **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 2.2: **Mutant API**

2.4.2 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 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.

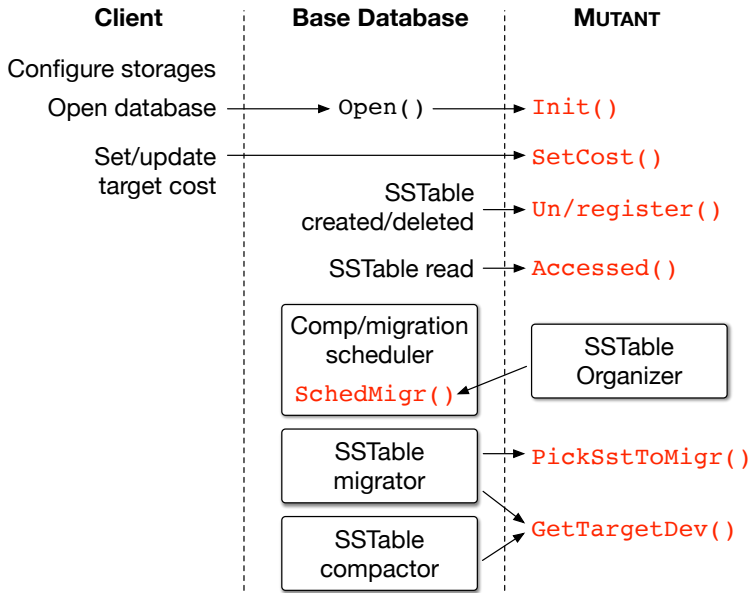


Figure 7: **Interactions among the client, the database, and Mutant.** MUTANT API is depicted in red. Parameters and return values are omitted for brevity.

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

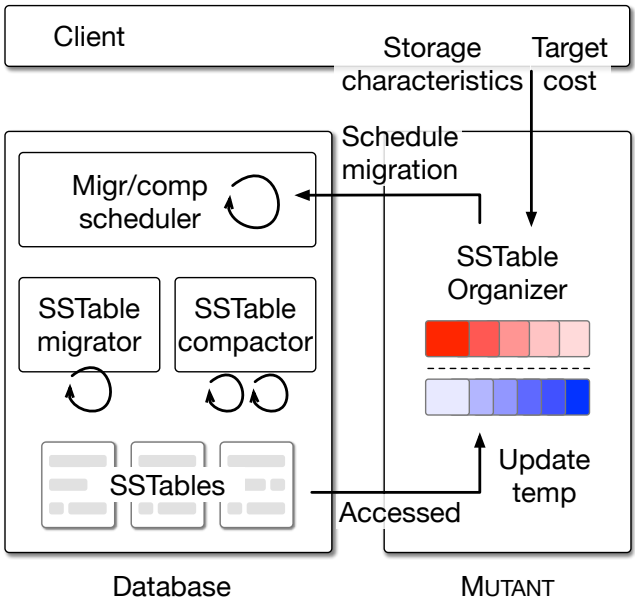


Figure 8: **SSTable Organizer and its interactions with the client and the database.**

existing SSTable.

SSTable Reader Thread: increments an SSTable access counter.

2.4.3 Optimizations

2.4.3.1 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 §2.5.4.3.

SSTable flushes, although similar to SSTable compactions, are not combined with SSTable migrations. Since the newly flushed SSTables are the most frequently accessed (recall §2.2.3), MUTANT always writes the newly flushed SSTables to the fast device, obviating the need to combine SSTable flushes and SSTable migrations.

2.4.3.2 SSTable Migration Resistance

The greedy SSTable organization (§2.3.1.2) 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 affects how well MUTANT meets the target storage cost. We analyze the trade-off in §2.5.4.2.

2.5 Evaluation

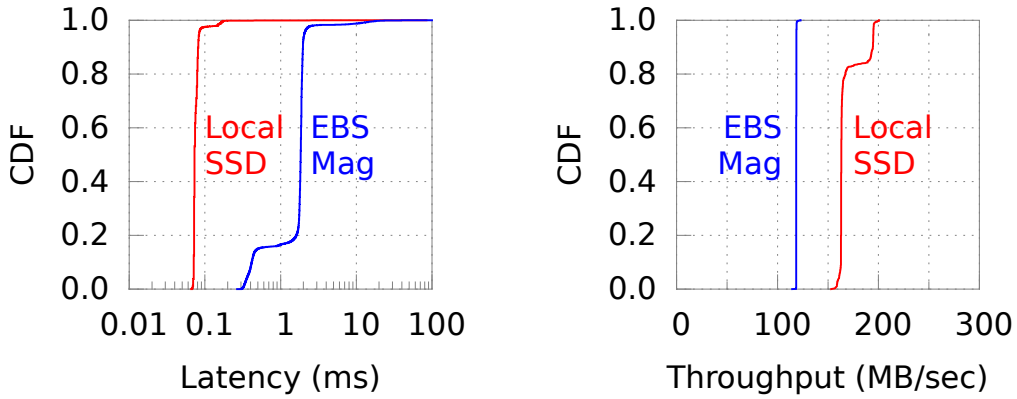
This section evaluates MUTANT by answering the following questions:

- How well does MUTANT meet a target cost and adapt to a change in cost? (§2.5.2.1)
- What are the cost-performance trade-offs of MUTANT like? (§2.5.2.2)
- How much computation overhead does it take to monitor SSTable temperature and calculate SSTable placement? (§2.5.3)
- How much does MUTANT benefit from the optimizations including SSTable component organization? (§2.5.4)

2.5.1 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 [10]. 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 [101].

For the evaluation workload, we used (a) YCSB, a workload generator for microbenchmarking databases [41] 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 [19].



(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.

2.5.2 Cost-Performance Trade-Offs

2.5.2.1 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 §2.5.4.2: a 5% SSTable migration resistance was used in the evaluation. Figure 10(c) shows how MUTANT organized the SSTables among the fast and slow

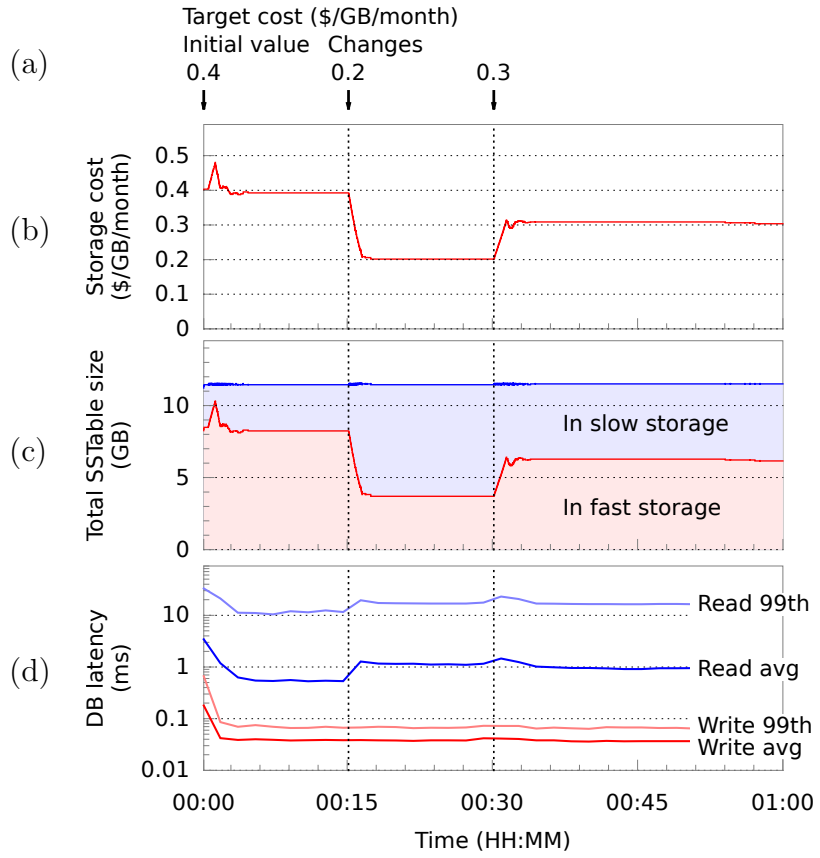


Figure 10: **Mutant makes seamless cost-performance trade-offs.** Target cost changes over time (a), changes in underlying storage cost (b), MUTANT organizes SSTables to meet the target cost (c) and the database latencies (d).

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.

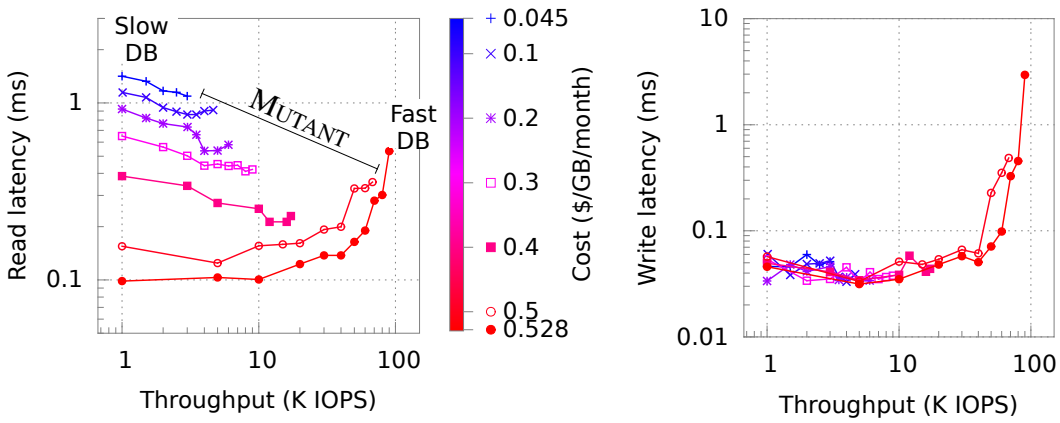


Figure 11: **Cost-performance trade-off spectrum of Mutant.** Read latency (left) and write latency (right) controlling throughput (horizontal axis) by varying the target cost. Colors and symbols represent storage cost.

2.5.2.2 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 [9], rather than the CPU getting saturated. Third, as you increase the throughput, the

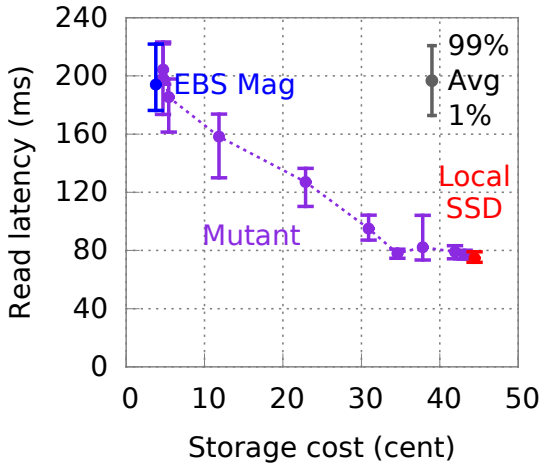


Figure 12: **Cost-latency trade-off with the QuizUp workload trace.** Fast and slow databases are shown in red and blue, respectively. MUTANT with various target costs is shown in purple.

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.

2.5.2.3 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 [50].

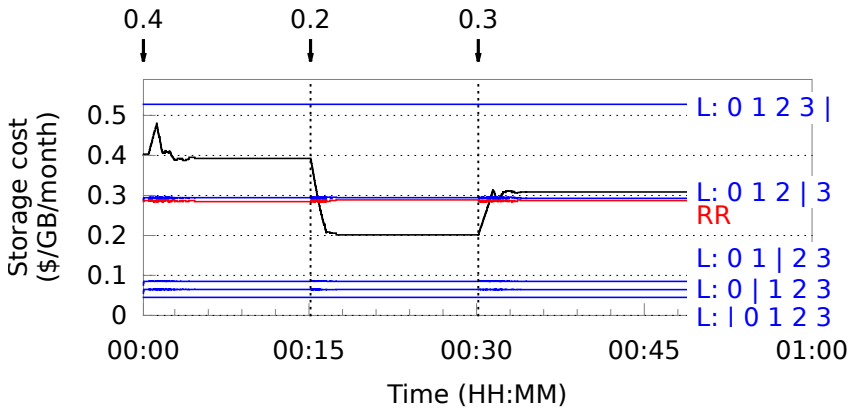


Figure 13: **Storage cost of Mutant and the other SSTable organization strategies.** The blue line (with L) represents RocksDB's leveled organization. SSTables at a level before the symbol | go to the fast storage; SSTables at a level after the symbol go to the slow storage. The red line (with RR) represents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

Starting from 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 [105].

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.

2.5.3 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 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.

2.5.4 Benefits from Optimizations

2.5.4.1 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

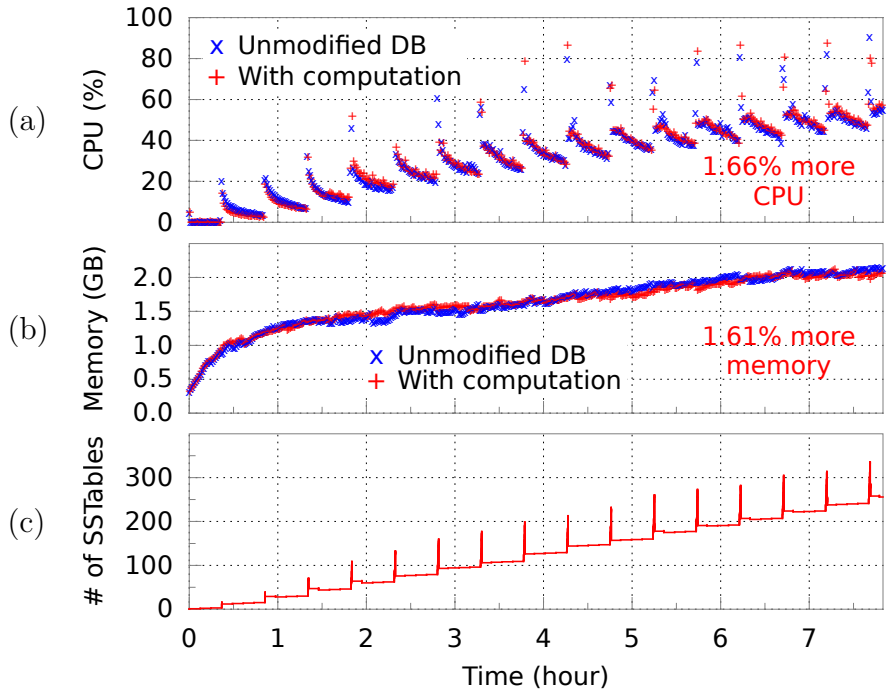


Figure 14: **Computation overhead of Mutant.** Figure (a) and (b) show the CPU and memory usage of the unmodified database (in blue) and MUTANT with SSTable temperature monitoring and SSTable organization calculation on (in red). Figure (c) shows the total number of SSTables.

and tail latencies, as in Figure 15. The latency reduction 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 (§2.5.1).

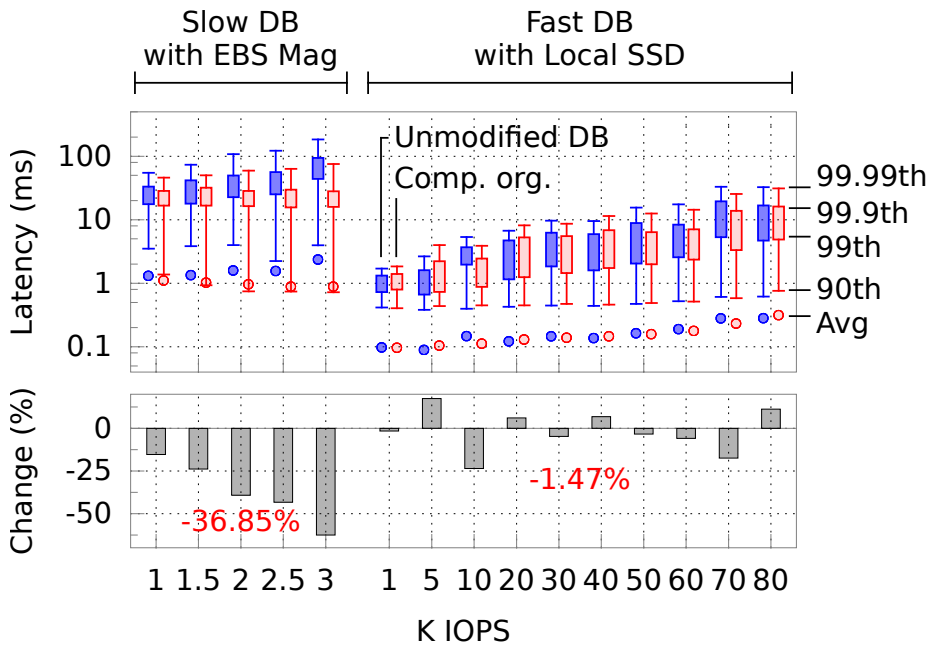


Figure 15: **"Slow database" benefited significantly from SSTable component organization.** A latency comparison of the unmodified database (in blue) and the database with SSTable component organization on (in red).

2.5.4.2 SSTable Migration Resistance

SSTable migration resistance serves as a trade-off knob between the amount of SSTables migrated and the storage cost conformance (§2.4.3.2). 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

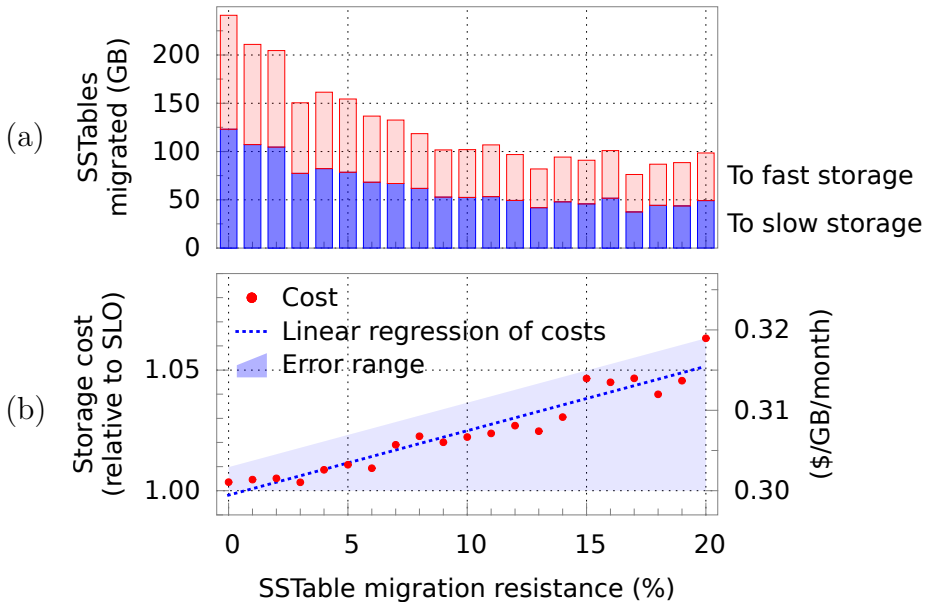


Figure 16: **SSTable migration resistance's effect on the SSTable migrations and cost SLO conformance.** Figure (a) shows how the amount of SSTable migrations changes by SSTable migration resistance, and (b) shows how the storage cost changes.

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.

2.5.4.3 SSTable Compaction-Migration

SSTable compaction-migration integration is an optimization that piggybacks SSTable migrations on SSTable compactions, thus reducing the amount of SSTable migrations (§2.4.3.1). The breakdown of the SSTable compactions 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.

2.6 Summary

In this chapter, 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 and organizes SSTables into different types of storage devices to meet the changing target cost.

CHAPTER III

IN EDGE CLOUD CACHE SYSTEMS

In this chapter, we explore the cost-performance trade-offs in edge cloud caching systems. Today's cloud caching systems have limited to no options when it comes to (a) cost and cache hit level or (b) cost and performance isolation level. We explore the flexible cloud caching system designs that allow such cost-performance trade-offs.

3.1 Cost-Performance Trade-Offs in Edge Cloud Cache Systems

Many small to medium size CPs (content providers) use cloud cache services to deliver their content with a low latency. However, cloud cache services often do not deliver stable performance because of (a) the performance interference among the tenants, (b) the lack of performance SLAs (service level agreements), and (c) the unpredictability of the workloads.

Big CPs, such as Netflix and YouTube, avoid the performance interference issue by building their own CDNs (content delivery networks) [108, 29]. However, building a private CDN is not an option for many small to medium CPs due to the enormous initial investment cost and the low cost efficiency when the data request volume is low.

We propose SpaceLease, an **edge cloud cache with cost-performance control**. SpaceLease is deployed on the edge cloud platform, which the research community and industry leaders envision [93], and uses dedicated computing resource to store and serve cache items. The use of dedicated resource allows CPs (a) to **gain control** over the system cost, thus the performance of the system, and (b) to have **minimal performance interference**, which the multi-tenant, shared cloud

cache services suffer from, consequently removing a dimension of performance unpredictability, and thus costly lag or dissatisfaction from users.

Deploying your own CDN in the edge cloud platform raises architectural and economic questions such as:

- Where should the data be stored and how much cache space should be allocated in each edge data center to maximize the cache utility?
- How scalable or adaptive is the CDN to changing workload patterns?
- How much resource does a CP need from a cloud provider to build a sufficiently good CDN?

We address the questions by identifying cache storage space as the central bottleneck resource for CDNs, and provide an algorithm for identifying how much value can be extracted from a certain amount of cache space for a given workload type. We design SpaceLease with the two principles, **optimality** and **adaptability**. First, SpaceLease optimizes cache utility globally by analyzing the workload locally in each data center. Second, SpaceLease dynamically adapts to changing workload patterns by continuously updating the cache resource allocation. To support optimality and adaptability, we design the core components:

- *Workload Analyzer* calculates the cacheability of workload in each data center using cache item reuse distance analysis.
- *Cache Utility Optimizer* calculates the optimal partition of the cache resource among data centers using the input from Workload Analyzer.
- *Elastic Cache* dynamically resizes the cache resource in each data center to adapt to changing workload patterns.

We implemented a prototype of SpaceLease on Amazon data centers and evaluated it with the synthetic workloads and a 30-day Akamai data access trace. The evaluation results show that SpaceLease removes the performance interference of

the shared cloud cache services, adapts quickly to changing workload patterns, and optimizes the aggregate cache utility of a CP.

This chapter makes the following contributions:

- We identify the performance unpredictability in shared cloud cache services.
- We design a performance-isolated cloud cache architecture that maximizes the cache utility of a CP over all edge data centers and dynamically adapts to changing workload patterns.
- We implement SpaceLease and evaluate using both synthetic and real-world workload and show that SpaceLease (a) reduces the hit rate variability by up to 79% and (b) adapts quickly to changing workload patterns.

3.2 Performance Interference and Inflexible Cost-Performance Trade-Offs

3.2.1 Cloud Cache Services

Low-latency data access is important for web services since it is directly related to user experience level and financial gain. Users switch to a competitor web site with as little as 250 ms of page loading time delay [76]. A 100ms increase in the data access latency translates to 1% revenue loss [71].

To lower the data access latencies, many small to medium size CPs rely on the cloud cache services such as Amazon CloudFront, Google Cloud CDN, and Azure CDN [20, 55, 82]. The cloud cache services provide on-demand cache deployment at large scale alleviating the CPs' burden of having to build their own cache infrastructure.

Table 3: **An example periodic-reuse-distance object request sequence.** Maximum reuse distance is 4 and reuse distance increment is 2. Each shaded rectangle represents a group of object requests with repeating reuse distances.

Request order	0	1	2	3	4	5	6	7	8	9	10	11	12	13	⋯
Object ID	0	0	1	2	0	3	4	1	1	3	2	2	1	0	⋯
Reuse distance	∞	0	∞	∞	2	∞	∞	4	0	2	4	0	2	4	⋯

3.2.2 Cloud Caches Do Not Provide Stable Performance

While cloud cache services are meant to deliver low-latency data accesses, they often deliver unpredictable performance for reasons including unpredictable workload patterns of a CP and the resource contention among the tenants that share the cache space.

To look into the performance variability in cloud cache services, we measured the data access latencies and the cache hit rates of Amazon CloudFront [20], one of the biggest cloud cache services. To evaluate the cache performance, we made a deterministic sequence of object requests using reuse distance, the number of unique objects after the same object was last seen [88]. Specifically, we made periodic-reuse-distance object requests: objects are chosen to have periodic, evenly-spaced reuse distances. In the example Table 3, objects are chosen so that their reuse distances form a repeating sequence of 0, 2, and 4. The first group, from request order 0 to 7, is an exception: it includes some ∞ reuse distances due to the insufficient number of previously-seen objects. We used 1.5 M for the maximum reuse distance with 1000 evenly-spaced reuse distances repeating.

The measurement setup (Figure 17) consists of the three parts: a client, a CloudFront distribution, and a data origin. A client machine, representing end users, makes periodic-reuse-distance object requests with a request rate of 10 requests per second and object size 30 KB. A CloudFront distribution is a unit of cache deployment that is allocated in Amazon's Edge and Regional caches [6]. An object request is served

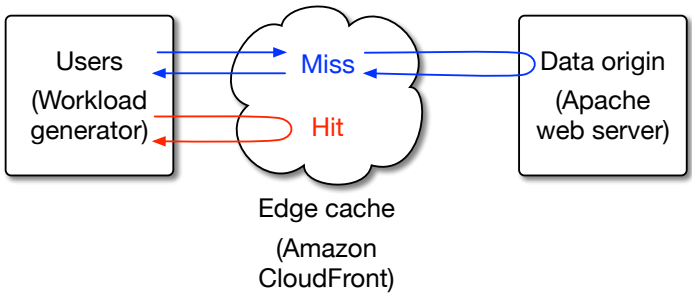


Figure 17: **Measurement setup of a cloud cache service**

by a 2-level cache service: first, a request goes to a closest Edge cache, and, when the object is not found in the Edge cache, the request goes to the Regional cache. When the object is not found in the Regional cache, the request goes to the data origin, an Apache web server serving the objects. The client parses the HTTP response header to see whether the request was served from CloudFront or from the data origin: whether the object was served via a cache miss or a cache hit. We ran the measurement with a request rate of 10 objects/second for 12 days.

We found that there is a high variability in the latency over time (Figure 18). Cache hit rate showed two patterns: **short dips** and **long fluctuations**. To look into the short dips, we recorded the cache server IP addresses to where the requests are routed. We found out that a DNS routing update happened every 1000 requests and the short dips were caused by new cache servers previously unseen by the CP. Since the cache servers are new to the CP, it takes some time before they get warm and the hit rate goes up. The addition of a new server can be from either a new cache server being added to a cache cluster or an abnormal DNS routing that routes requests to a remote region, where the cache is "cold" for the CP. The long fluctuations were likely caused by the resource contention among CPs and the capacity planning of CloudFront. The variability in hit rate over the 12-day measurement period was significant.

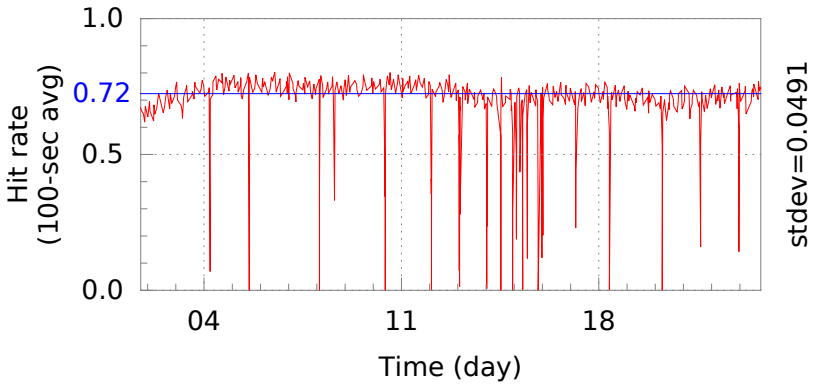


Figure 18: **Cloud cache services have a high performance variability**, which includes long fluctuations and short dips. A stable workload was requested to Amazon CloudFront for 3 weeks. Hit rate over time is shown in red, the average hit rate is shown in blue, and standard deviation are shown on the right.

3.2.3 Resource Distribution Among Data Centers Matters

CPs want to maximize their benefit by maximizing their aggregate *cache utility*, the amount of data served from the cache hits. Aggregate cache utility depends on how the total cache space is split among data centers. To demonstrate how different cache space splits make different aggregate cache utility, we simulated Akamai workload to two Akamai edge data centers, DC A and DC B (Figure 19). We made different splits out of a fixed cache space budget, 347 MB, and ran the workload against an LRU cache in each data center. As we allocate more cache space to DC A and less space to DC B, DC A's cache utility increases fast in the beginning and eventually plateaus and DC B's utility decreases slowly in the beginning and eventually drops fast. The optimal split was when DC A and DC B were allocated about 18% and 82% of the total cache space, which had 5.98% higher cache utility compared to a naive, even split.

As workloads change over time, so do the optimal cache splits. To demonstrate the benefit of dynamic cache space split, we updated the cache space split every day using Summit, which we explain in §3.3.2.1. We assumed a future knowledge of the requests to calculate the optimal split in each time range. With the two data center,

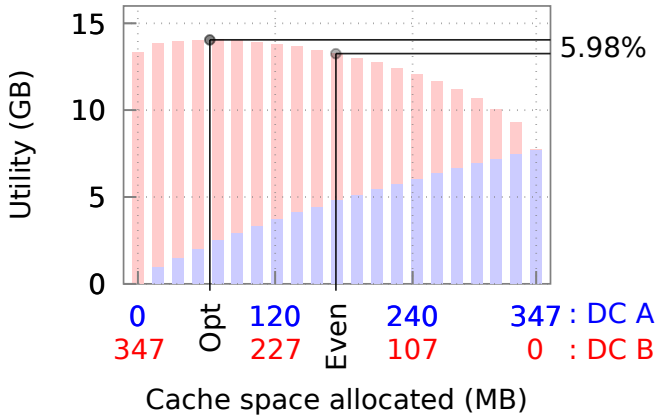


Figure 19: **Different splits of cache space among data centers make different aggregate cache utility.** We ran Akamai workload to two data centers, DC A and DC B, which we allocated different cache spaces as shown in x-axis. An optimal split of the cache space had 5.98% higher cache utility than an even split of the cache space.

DC A and DC B, the optimal dynamic split has 4.43% higher cache utility than the optimal static split (Figure 20). The utility gain is meaningful since the net gain is positive in spite of the cold cache penalty that comes with the cache resizing.

The cache utility benefit of the optimal cache space allocation suggests that a CP with a fixed budget should (a) optimally allocate its cache space among data centers and (b) update the allocation periodically for the changing workload patterns.

3.3 System Design

To solve the cache performance variability problem, we design SpaceLease, a dedicated-space cache architecture with cache state transfer.

Building a dedicated-space caching system faces new challenges including:

- **Optimizing cache space allocation:** Given a cost budget, how much space should be allocated in each of the edge data centers to maximize the aggregate cache utility?
- **Aggregate cache size:** How much aggregated cache space should be allocated for a CP to provide a reasonable performance?

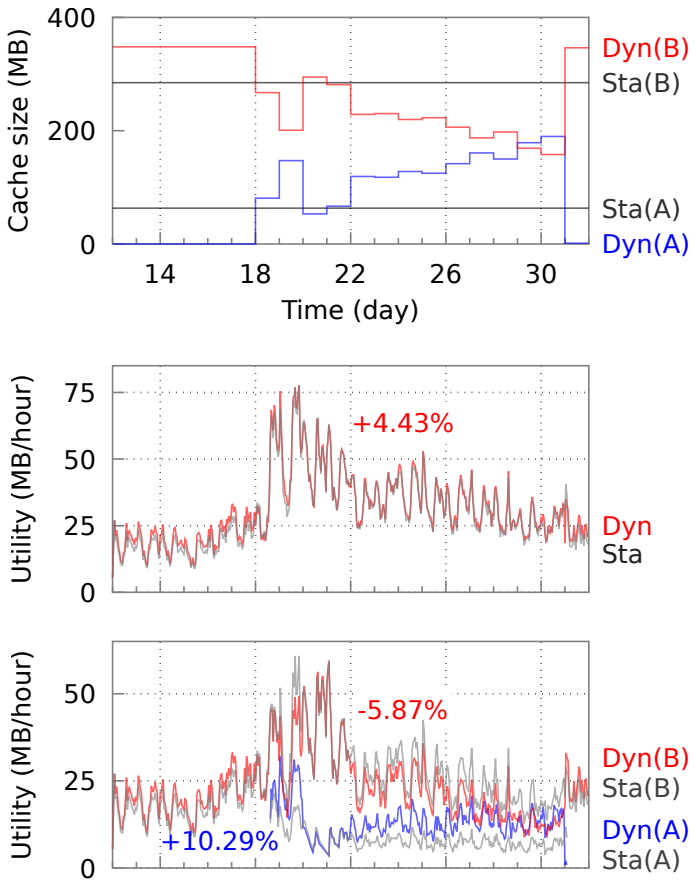


Figure 20: **Dynamic cache space resizing outperforms static cache space partitioning.** The top figure shows the changing cache space of data center A and B, depicted as Dyn(A) and Dyn(B). The static cache split is shown with Sta(A) and Sta(B). The middle figure compares cache utility per hour of dynamic and static split. The bottom figure breaks down cache utility of dynamic and static split by data centers.

- **Elastic cache resizing:** How should the cache be resized to adapt to changing workload patterns?

We contrast SpaceLease and the traditional, shared cache systems in Table 4.

3.3.1 Performance Isolation

SpaceLease provides cache performance isolation by design: we use cloud VMs for hosting the cache service and VM resources are isolated from the other VMs. The degree of performance isolation varies by the metrics and the resource type. Cache

Table 4: **Dedicated- vs. shared-cache architecture**

	Dedicated cache	Shared cache
Performance isolation	Provided by design: the cache VM storage is dedicated to a CP.	Not provided. A CP's performance changes by the other tenants' workloads.
Cache utility optimization	An opportunity to optimize the cache utility custom tailored to a CP's workload.	A CP doesn't have a control over the cache performance in each data center.
Elastic cache resizing	Needs a system component that change the cache space dynamically to changing workload patterns.	Not provided. A change in a CP's workload pattern changes its effective cache size, affecting the performance of the other tenants.

hit rate with SpaceLease is completely isolated form the other VMs, provided the system is not saturated, a case that rarely happens in the web caching systems, which are protected by admission controls. Data access latency can vary by the VM interference, especially when they share a secondary storage such as SSDs [Koh, Jian's]. CPU and memory have relatively minimal interference to the secondary storage. The impact of the latency interference is ignorable compared with that of the cache hit rate interference: with a cache miss, a data access request needs to go to the data source.

3.3.2 Cache Utility Optimization

Since SpaceLease allocates a dedicated cache space in each data center for a CP, the natural follow-up problem is **cache utility optimization**: how much space should be allocated in each data center?

We solve the problem by allocating a small amount of cache space repeatedly to where the "value" of the cache space is the biggest. To quantify the value, we use the metric **cache utility**, which is the amount of data served from cache hits. Since data accesses have locality, a typical cache utility grows fast when the cache size is small and the growth slows down as the cache size becomes bigger. Cache utility curve is a characteristic of the workload arriving at a data center, and can be calculated by monitoring the data accesses at the data center. **Cache utility analyzer**, a

component of SpaceLease, samples the data access requests and builds the utility curves. We present a formal description Summit in the next subsection, the optimal cache allocation algorithm.

3.3.2.1 Summit*: Utility-based Cache Space Allocation*

Let us assume that a CP is considering the space allocation among N different edge data centers, call them $1, 2, \ldots, N$, possibly from different cloud operators.

Costs: A CP is interested in leasing out some amount of cache space in each data center. For the cache space, we focus on the memory (RAM) as the primary, performance-isolated cache item storage. Thus, the cache space cost becomes the VM leasing cost. Big cloud vendors have pricing models that charge proportionally to the amount of memory [AWS, Google Azure].

Benefits: The benefits a CP derive for every cache hit is twofold. First, the end-user gets lower latencies and thus higher level of user experience that directly translates into measurable profit for the CP [Amazon, Bing and Google]. Second, the hit means that content is served out of the edge cache instead of hitting the aggregation point or the wider Internet, which is a saving of the precious backhaul network bandwidth for the network operator. These savings are assumed to be passed on to the CP in some capacity. Together, we can assume that the CP derives some dollar utility from the cache hits.

Objective: Together, the CP is interested in leasing resources on those cache nodes that provide the greatest cache benefit over costs, without exceeding a budget of at most P dollars spent. Formally, let $\vec{x}$ denote the set of space allocations $(x_i)_{i=1}^{N}$ such that $x_i \geq 0$ and

$$\textsc{cost}(\vec{x}) = \sum_{i=1}^{N} c_i x_i \leq B \tag{5}$$

where $c_i > 0$ denotes the per-gigabyte price per time unit that the CP would be

charged for each additional unit of space of cache in data center i. Note that the value of c_i could change every epoch, such as (a) in respond to demand, such as spot pricing [8], and (b) variable VM instance prices in different data centers [12].

The benefit is the financial gain from the amount of data served from the cache hits, or utility, denoted by UTIL:

$$\text{UTIL}(\vec{x}) = \sum_{i=1}^{N} U_i \tag{6}$$

$$U_i = \alpha_i \cdot v_i \cdot h_i \tag{7}$$

where U_i is the cache utility in data center i, which is the product of the data volume-to-dollar cost conversion constant α_i, the requested data volume v_i, and the cache hit rate h_i. Cache utility changes by the amount of cache space x_i. Similar to cache hit rate curve construction, cache utility curve can be estimated through spatial sampling of the request stream [112]. Moreover, U_i is concave for LRU cache-replacement. Methods, such as Talus or SLIDE [24, 111], can be used to force cache replacement schemes to produce concave hit rate curves, even for online streams.

With the $\text{COST}(\vec{x})$ and $\text{UTIL}(\vec{x})$, the objective of the CP is maximizing the $\text{GAIN}(\vec{x})$, the difference of $\text{UTIL}(\vec{x})$ and $\text{COST}(\vec{x})$:

$$\max\{\text{GAIN}(\vec{x}) = \text{UTIL}(\vec{x}) - \text{COST}(\vec{x})\} \tag{8}$$

We propose SUMMIT (algorithm 1), a greedy algorithm that climbs the steepest GAIN curve little by little, while the slope of the curve is positive and the CP's budget doesn't run out. SUMMIT is close to optimal.

Theorem 3.3.1 *When the hit gain curves* $\text{GAIN}(\vec{x})$ *are concave and non-decreasing, the allocation* $\vec{x}$ *by the greedy algorithm algorithm 1 is near-optimal. Specifically,* $\text{GAIN}(\vec{x}) \geq \text{GAIN}(\text{OPT})$ *while* $\text{COST}(\vec{x}) \leq \text{COST}(\text{OPT}) + \max_{i=1,\ldots,N} c_i$.

Input: $\mathbf{gain}(\boldsymbol{i}, \boldsymbol{j})$, $i \in N$**:** GAIN in DC_i, the i-th data center, when the j amount of cache space is allocated. $\boldsymbol{B}$**:** cost budget for the aggregate cache space.

Output: $\boldsymbol{x_i}$**:** cache space allocation in DC_i.

```
/* Initialize cache space at each DC to 0                          */
for i ∈ N do
    x_i ← 0

/* Greedily allocate cache space by following the GAIN curve that
   has the steepest line segment                                   */
while COST(x⃗) < B do
    (i, ΔGAIN) ← max_{i∈N}(GAIN(i, x_i + 1) − GAIN(i, x_i))
    if ΔGAIN ≤ 0 then
        break
    x_i ← x_i + 1
```

Algorithm 1: SUMMIT: Utility-based cache space allocation

3.3.2.2 Adaptability to Changing Workloads

To provide a consistent performance under the changing workloads, SPACELEASE runs SUMMIT periodically and adjusts its cache size to the current workload.

3.4 Evaluation

We evaluate SPACELEASE to answer the following questions:

- Does SPACELEASE provide a stable performance? (§3.4.1) How much does a CP pay for the stable performance? (§3.4.1.1)
- Does SPACELEASE provide configurable cost-performance trade-offs? (§3.4.2)
- How well does SPACELEASE adapts to changing workload patterns? (§3.4.3)

3.4.1 Stable Performance

To evaluate the performance stableness of SPACELEASE, we ran workload `static-1.5M` that we used for measuring the CloudFront performance variability (§3.2.2). against a SPACELEASE edge data center with a total cache size of 37.3 GB and 5 cache nodes.

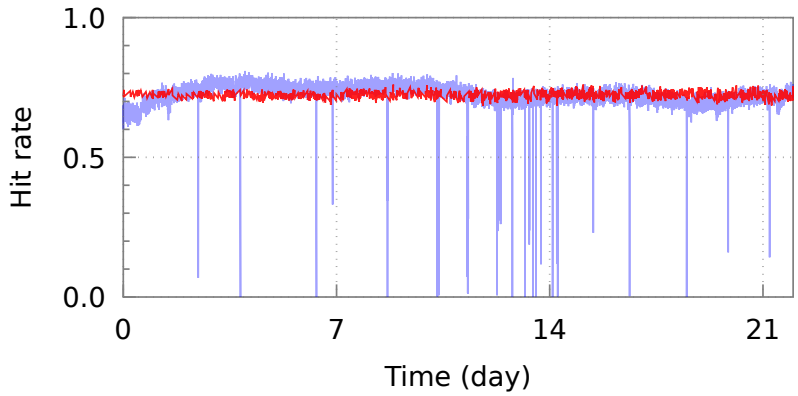


Figure 21: **SpaceLease provides stable hit rate**: no long fluctuations or short dips. The red and blue lines are the hit rates of SpaceLease and CloudFront. The reduction in standard deviation was 79%.

We set the total cache size to 48 GB, which matched the hit rate of the CloudFront evaluation. Since SpaceLease uses the dedicated resource by design, the hit rate stayed stable during the entire evaluation time (Figure 21). There was no short dips or long fluctuations unlike in the CloudFront evaluation. The reduction in standard deviation was 79%: 0.0104 compared with 0.0491 of the CloudFront evaluation.

3.4.1.1 Cost of Stable Performance

To evaluate the cost of the stable performance, we compared the cost of running workload `static-1.5M` on SpaceLease and CloudFront. To our surprise, SpaceLease cost 10.09% less than CloudFront: $51.936 vs. $57.766 (Table 5). The cost difference, however, is arbitrary and depends on the factors including:

- The workload density. Cache node cost does not change by the number of requests or the traffic volume, thus the relative cache node cost becomes lower as the request rate goes higher as long as the cache nodes can handle the requests without performance degradation. Workload `static-1.5M` had 10 30-KB object requests / second and any higher request rate or bigger average object size would have lowered the relative cost.
- Cache node hardware configuration. We used an AWS t2.nano instance for the cache

Table 5: **Cost of SpaceLease and CloudFront** with workload `static-1.5M` for 3 weeks. The costs are in USD in the AWS US East region [14, 12].

	SpaceLease	CloudFront	Exp type
Cache nodes	5.200	-	CapEx
Requests	-	13.608	OpEx
Cache-to-user traff	46.719	44.124	
Cache-to-origin traff	0.017	0.034	
Total	51.936	57.766	

node, which had 1 vCPU, 0.5 GiB of memory [11] and an EBS general purpose SSD volume for storing the cache items [13]. In this example, the resource was underutilized: the CPU usage during the experiment was around 1%. We even tested with the request rate of up to 200 request / second, which is 20× higher than workload `static-1.5M`, but there wasn't a meaningful increase in the CPU usage. Thus, the cost would have been even lower with a further lower-end instance type without sacrificing performance.

To demonstrate how much effect the request rate has on the relative cost of SpaceLease and CloudFront, we compared the costs with various request rates (Figure 22). When the request rate is lower than the break-even point, which is around 5 requests / second, SpaceLease costs more than CloudFront because of the relatively high CapEx (capital expenditure). After the break-even point, SpaceLease costs less than CloudFront, which aligns with why companies with a large traffic volume build their own CDNs [108, 29].

3.4.2 Cost Configurability

To evaluate the cost-performance trade-offs of SpaceLease, we monitored cache hit rate while changing the cost of SpaceLease in the middle of serving the client

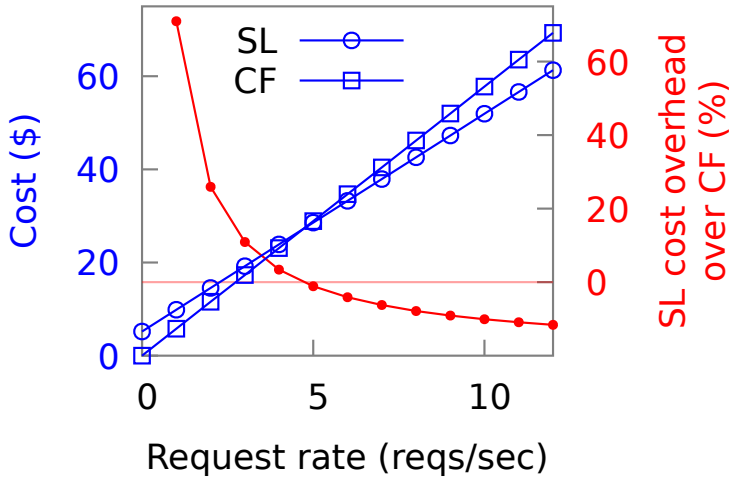


Figure 22: **Cost overhead of SpaceLease over CloudFront becomes smaller as the request rate goes up.** SL and CF on the left y-axis represent the cost of SpaceLease and CloudFront.

requests. For the client requests, we replayed data access traces of an Akamai CP in an edge data center, which had 11 M requests for 20 days. Initially (Phase 1) we set the cost budget to $46.61/month. Then after about 6.5 days (Phase 2), we increase the cost to $84.74, and after another 6.5 days (Phase 3), we decreased the cost down to $21.18 (Figure 23(a)). Cache hit rate followed the cost changes: in Phase 2, when 10 more cache nodes were added, hit rate went up slowly and eventually plateaued when the cache nodes became warmed up, and in Phase 3, when 15 cache nodes were terminated, hit rate sharply dropped and eventually plateaued (Figure 23(b)). The number of cache nodes, which responds to the cost budget, was what drove the cache hit rate changes (Figure 23(c)). There were small delays between the cost changes and the number of cache nodes, which were caused by the cache node VM launch time, cache server initialization time, and the load balancer reconfiguration time. This demonstrates the flexible cost-configurability of SpaceLease in contrast to the single cost-performance point of cloud cache services.

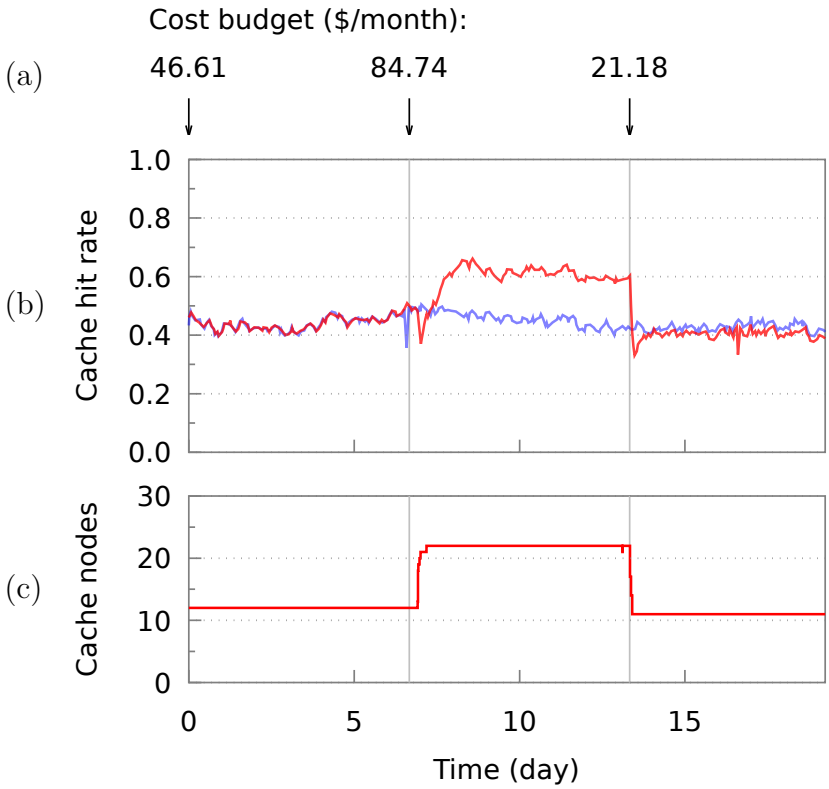


Figure 23: **SpaceLease provides cost-performance trade-offs.** (a) shows the changing cost budget. (b) shows cache hit rate with the changing cost in red and with a fixed cost in blue. (c) shows the number of cache nodes SpaceLease allocated.

3.4.3 Adaptability to Changing Workloads

To evaluate how well SpaceLease adapts to the changing workload to meet the target hit rate, we measure the hit rate while varying the cacheability of the workload. We made requests with randomly generated reuse distances with varying max_rd, maximum reuse distance, in a way that their distribution form a concave curve. The object ID of the i-th request O_i is calculated with the following steps:

- Generate a random number r_i in the range $[0, 1]$ with a uniform random distribution
- Generate the reuse distance RD_i by curving the distribution with the curve parameter k and multiplying it with max_rd: $RD_i = max\_rd * (1 - (1 - d^k)^{1/k})$
- Calculate O_i with by selecting the object ID with reuse distance RD_i from the use order history data structure.

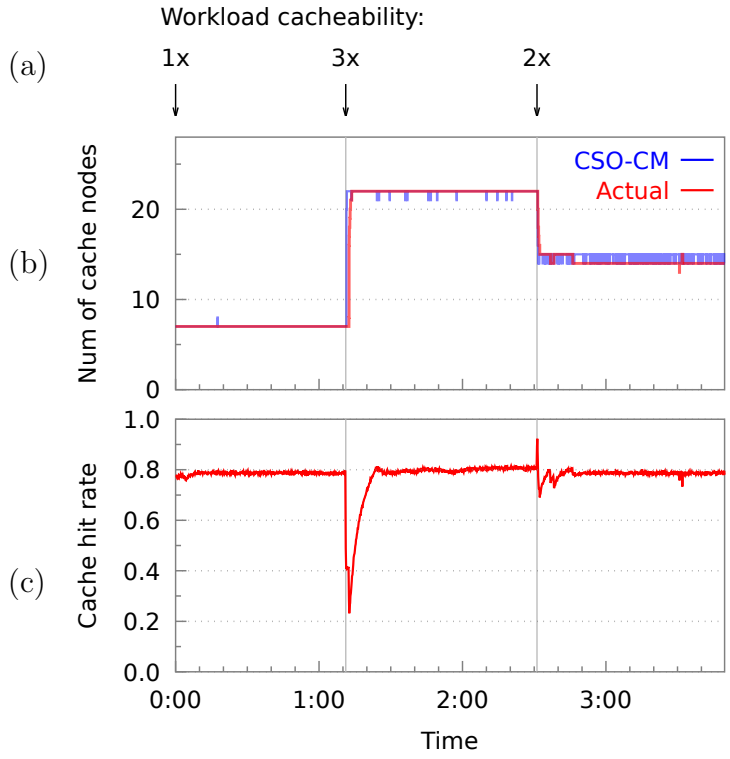


Figure 24: **SpaceLease adapts to changing workload patterns.** (a) shows the changing workload patterns. (b) shows the number of cache nodes: the blue line is what Cache Space Optimizer tells Cache Manager and the read line is the actual number of cache nodes in the data center. (c) shows the aggregated cache hit rate.

We set the target hit rate to 0.8 and varied *max_rd* from 150 K to 450 K down to 300 K, each of which we will call 1×, 3×, and 2× cacheability workloads (Figure 24(a)).

Initially, we set the number of cache nodes to 7 to meet the target cache hit rate, 0.8, with the 1× workload. When we changed the workload to 3×, Cache Utility Analyzer started to generate wider utility curves that would require more cache space to meet the target cache hit rate, Cache Space Optimizer calculated new cache space to meet the target cache hit rate, and Cache Manager started to allocate more cache nodes, making the number of cache nodes to 22. When we changed the workload down to 2×, similar steps happened that eventually made Cache Manager to deallocated some cache nodes, making the number of cache nodes to 14.

There were delays when the workload pattern changed until the number of cache nodes changed, which can be broken down as:

- Cache resize interval: how often Cache Space Optimizer recalculates the optimal cache size and sends new cache sizes to Cache Manager in each data center. Cache resize interval is in sync with *Utility curve report interval*, which states how often a Cache Utility Analyzer reports its utility curve to Cache Space Optimizer. The parameter serves as a control knob that decides the responsiveness of SpaceLease to a workload change and the accuracy of the utility curve, which determines the accuracy of the optimal cache space allocation.
- Delays from exponential filter: the exponential filter smooths out the jitters at the cost of a delayed response to the cache resize command. The exponential parameter of 0.3 smoothed out most of the jitters (Figure 24(b)), but introduced a 2× *Cache resize intervals* of delay.
- Cache node launch time: the time for launching a cache nodes VM and for the cache server, Apache Traffic Server, to get ready to serve requests. This time depends on the Cloud platforms and the instance type. With a small AWS EC2 type (t3.micro), the launch time was 60 seconds on average. The time could be further reduced with containerizaion technologies such as Docker.
- Load balancer reconfiguration time: the time for Load Balancer to update the keyspace mapping to cache nodes. The reconfiguration time was small enough to be ignorable.
- Cache node termination time: the time for terminating a cache node. The time is masked since SpaceLease terminates a cache node in the background after a Load Balancer reconfiguration.

Table 6 summarizes the breakdown of the delays.

Cache hit rate followed the target hit rate well as in Figure 24(c), which shows that the utility curve-, thus reuse distance-, based space allocation is a good metric for non-LRU algorithms as well: Apache Traffic Server uses CLFUS (circular LFU

Table 6: **Breakdown of cache resize time**, how long it takes for a cache size change to take effect. The items in parenthesis are for increasing the cache size.

Cache resize sub task	Time (sec)
Cache resize interval	10
Delay from exponential filter	20
(Cache node launch time)	($\approx$ 60)
Load balancer reconfiguration time	$\approx$ 0.02
Total	$\approx$ 40 ($\approx$ 100)

by sizes) for RAM cache and FIFO for disk cache by default [46, 21]. During the workload transition time, there were temporary dips and spikes. When the workload cacheability increased to 3×, cache hit rate dipped because there was not enough space to cache the requests, started to go up as soon as the new cache nodes were added, and eventually met the target hit rate when the new cache nodes got warmed up. When the workload cacheability decreased down to 2×, hit rate spiked due to the over-provisioning of cache space, started to dip when some of the cache nodes are removed, and eventually went up to meet the target hit rate. The dip was below the target cache hit rate because the remaining cache nodes needed to get warmed up for the newly assigned keyspace ranges that were previously assigned to those just deleted nodes.

3.5 Summary

In this chapter, we have presented SpaceLease, an edge cloud CDN with cost-performance trade-offs, including (a) cost and performance isolation trade-offs and (b) cost and cache hit rate trade-offs, with data structures and algorithms that capture the workload cacheability effectively and a scalable system that resizes the cache resource in each edge data center of a content provider.

CHAPTER IV

IN GEO-REPLICATION SYSTEMS

In this chapter, we explore the cost-performance trade-offs in geo-replication systems. To reduce the increasing cost of geo-replication, companies and research communities have proposed partial geo-replications. However, the existing partial geo-replications are (a) not optimal to all application types and (b) limited in their cost-performance trade-off options. We present a geo-replication system that (a) is flexible in its replication decisions to meet the various data access patterns from applications and (b) makes fine-grained cost-performance trade-offs.

4.1 Cost-Performance Trade-offs in Geo-Replication Systems

Large-scale, data-intensive services (e.g., YouTube, Instagram, Facebook, Twitter, etc.) rely on their ability to provide timely access of continuously evolving content to a globally distributed client base. In support of such functionality, accesses to and manipulations of content are carried out across multiple geographically distributed data centers. Such replicated facilities enable low service latency for data collection and provision, and, through replication, improved fault tolerance in terms of service and data availability.

But replication comes at price: the added storage and network costs grow proportionally in the number of data center replicas and the increasing volume of data being stored [37]. Several geo-replication systems have been recently introduced, each making a different trade-off between costs and benefits [64, 115, 109, 104, 18]. On one extreme of the spectrum are *full replication* systems, such as Cassandra, HBase and MySQL Cluster, which replicate all data in every data center. These systems

thus minimize data access latency while also maximizing replication cost. On the other extreme are *no-active replication* systems where data is stored only in the origin data center and replicated to other data centers only when absolutely needed. These systems minimize cost at the price of added latency, and are the default mode of operation for caching systems and content-distribution networks. Between these two contrasts are *partial replication* systems which seek to lower cost by replicating data only in places where the data is deemed most likely to be consumed based on the statistics of objects and metadata they collect.

The performance of partial replication systems depends critically on how well future data accesses can be predicted. Most partial replication systems, such as caches, assume that future access patterns closely mirror the past, thus exploiting any temporal locality that is exhibited. On one hand, this assumption is crucial for prediction, for instance Brodersen [31] and Kumar [69] show that YouTube videos and Facebook photos remain popular for some time period and then quickly lose their popularity as they age. On the other hand, overreliance on temporal locality misses out on opportunities for more cost-effective geo-replication. A naïve caching system, for instance, built to exploit temporal locality would lose out on the rich geographic locality of data accesses, as illustrated in Figure 25. Here, our trace of YouTube video access traces shows that most videos are accessed only at their site of origin. Replication in cache-like replication systems is necessarily *reactive*: the first user to access new content always endures the full latency of retrieval from the back-end storage. Moreover, client and edge caches already exploit much of the temporal locality exhibited in access traces, forcing the middle layers to replicate data based on seemingly more uniformly random access patterns [58].

We argue that a geo-replication system that simultaneously minimizes cost and latency must be able to base replication decisions on richer information attributes than past access patterns alone. We have seen some

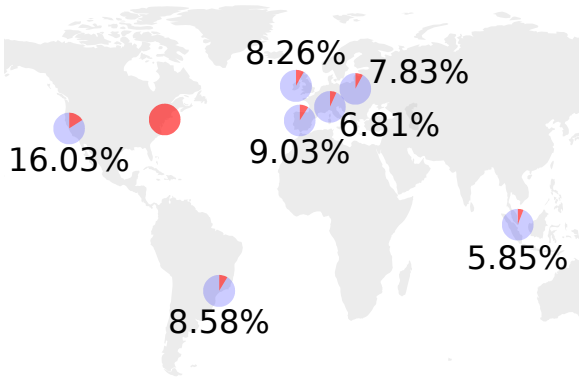


Figure 25: **Spatial locality.** (Sec. 4.4.1) YouTube videos were uploaded to the New York data center and then fully-replicated to all remote data centers. The figure shows the fraction of data accessed in each data center (depicted in red). On average, 91% of the video replicas (depicted in blue) in remote data centers were not accessed.

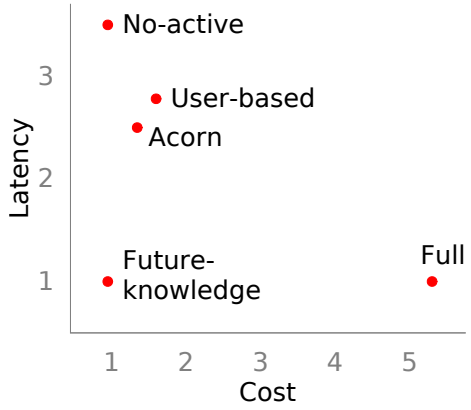


Figure 26: Cost and latency comparison of replication models from the experiments in Section 4.4. "User-based" represents existing, user-based partial replication systems. The horizontal and vertical axes respectively denote the cost and latency overheads.

initial work towards this research direction: several systems leverage additional knowledge about the *user* who posted or consumed a piece of content, either explicitly [64, 115, 109] or indirectly [104, 18]. Whereas user attribute can be useful in private data-sharing applications, such as social network applications, they are less good predictors in public data-sharing applications, such as YouTube and Flickr, where a majority of data is accessed through alternative, non-social channels [31]. Moreover, current replication systems consider at most one attribute at a time, missing opportunities to combine interesting content features for better estimation.

To address the shortcomings of the existing systems, we propose Acorn, an **A**ttribute-based **CO**ntinuous partial geo-**R**eplicatio**N** system, which achieves lower cost and latency than existing systems. Acorn uses three design principles:

(i) Replicate based on multiple attributes,

(ii) Replicate based on the appropriate attributes for each application, and

(iii) Replicate continuously, even if it implies exploring unknown data.

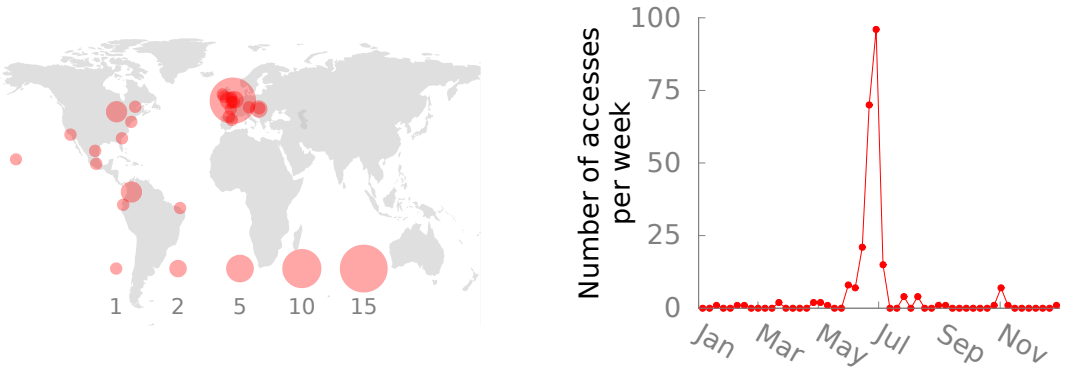
A key idea behind Acorn is that replication decisions should be applied to collections of content objects, which Acorn determines from attributes, rather than reasoning only about individual objects with no shared relationship or history across different objects. Acorn could therefore proactively replicate a new piece of content c in the appropriate data centers because the system was familiar with c's attributes. By design, Acorn is not able to proactively replicate new content that exhibits new attributes previously unseen by the system. Instead, Acorn uses continuous random replication methods to explore such content and selectively replicate it to reduce end-user latencies.

4.2 Design Principles for Better, Fine-grained Cost-Latency Trade-Offs

To achieve low-cost geo-replication while also providing low-latency data accesses, we follow three design principles.

4.2.1 Use appropriate attributes for each type of application

Different types of applications have different data access patterns. In private data-sharing applications, such as Facebook or Snapchat, objects are accessed mostly through friends, thus making "user" the best attribute to monitor and predict future accesses. On the other hand, public data-sharing applications, such as YouTube or Flickr, have diverse sources of accesses. For example, the majority (63%) of



(a) Geographic locality (b) Temporal locality

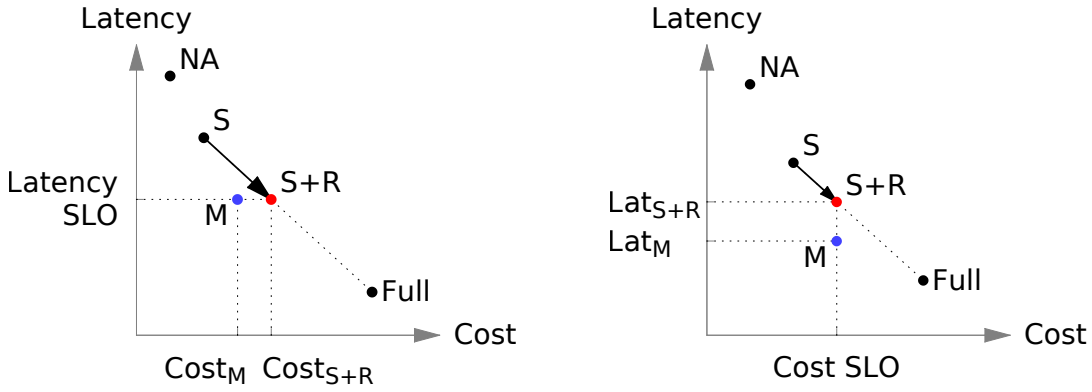
Figure 27: **Data access locality.** (a) Access locations of YouTube videos with the topic "Wimbledon" from Jan. 1st to June 7th, 2014. (§4.4.) The radius of each circle represents the number of accesses in a 5-degree longitude and 5-degree latitude area. "Wimbledon" becomes globally popular after June 7th, because the event commences at the end of June. (b) The number of accesses to the same videos per week in 2014.

YouTube accesses are from non-user-based (non-social) channels, such as search and in-application navigation [31]. For these applications, "topic" is a better attribute for making replication decisions than "user". Figure 27 shows an example of topic's strong data access locality both geographically and temporally. Videos with the topic "Wimbledon" are mostly accessed from North and South America and Europe from Jan. 1st to June 7th, so they don't need to be replicated to Africa, Asia, or Australia.

Although we focus on the type of applications that exhibit data access locality, which is most likely the case with geo-replicated, global-scale applications, we note that there are other types of applications that have different access patterns. For example, restrictions on government and medical data require them to be stored in specific facilities or areas. Static, application-specific replication policies would better serve those applications.

4.2.2 Use multiple attributes

Multi-attribute-based replications have lower cost under a latency SLO (service level objective), or lower latency under a cost SLO, compared to single-attribute-based



(a) Cost comparison (b) Latency comparison

Figure 28: **Multiple attributes are beneficial.** Cost and latency comparison of single- vs. multi-attribute-based replications. S, M, S+R, and NA represent single-attribute based replication, multi-attribute based replication, single-attribute based replication plus extra random replicas, no-active replication, and full replication, respectively.

ones, even when additional extra random replicas are used to remedy any limitations from the predictive use of the single attribute. Using a data attribute, you can make a better replication decision as to where a replica should be placed than making replicas at random places.[1] In the same manner, using two attributes will most likely give you better replication decisions than using one attribute and adding extra random replicas; in Figure 28(a), the multi-attribute replication M has lower cost than the single-attribute with random replication solution ($S + R$). For example, replicating YouTube videos to Atlanta, Georgia, when either (a) they have the topic "tennis"[2] or (b) they are uploaded by the user "John Isner"[3], has a lower cost than replicating them to Atlanta than when (a) they have the topic "tennis" or (c) with an extra 15% probability. Another example is Snapchat prefetching videos of nearby friends, so

[1]This is the rationale behind all history-based prediction systems, including the existing user-based partial replication systems. There could be anti-patterns on which the geographic and temporal locality doesn't hold such as a repeating pattern of an attribute being popular briefly and not accessed at all for the duration of popularity monitor window. However, the anti-patterns rarely happen with user-generated data.

[2]Atlanta has the highest number of USTA members per capita in the US.

[3]Isner played for University of Georgia and is a top-ranked tennis player in the US as of May. 2016.

that users can watch them even without a good network connectivity. Some videos will be watched and others will not. You could increase the prediction accuracy by adding another attribute capturing "interest" (or "topic"), such as "tennis".

In the same manner, using multiple attributes achieves lower latency than using a single attribute under cost SLOs, as shown in Figure 28(b). In general, we expect that M_n, a partial replication system making replication decisions with n attributes, has lower cost and latency than M_{n-1} plus extra random replicas:

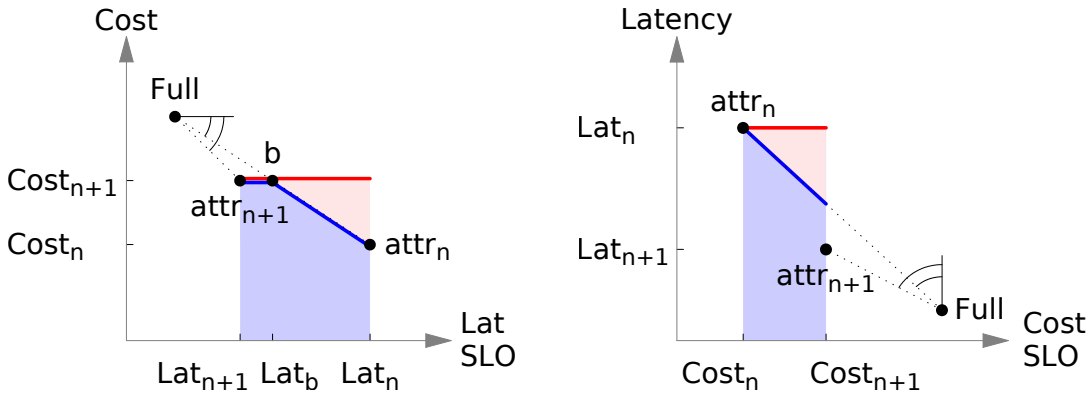
$$\begin{aligned} \text{Cost}_{M_n} &< \text{Cost}_{M_{n-1}+R} \\ \text{Lat}_{M_n} &< \text{Lat}_{M_{n-1}+R}. \end{aligned} \tag{9}$$

The difference either in cost or latency between those systems becomes smaller as n gets bigger.

When you add extra replicas with a uniform probability distribution, the cost increases and the latency decreases mostly linearly, as depicted with the line from S to Full in Figure 28. This is because, regardless of data access patterns, i.e., whether your objects are uniformly accessed or some small amount of your objects are accessed far more often than the others, by adding extra replicas randomly, you are giving each of the objects a same amount of extra chance to be replicated in each local data center. However, we note that it can be off of the straight line, depending on the randomness of extra replica placement and the characteristic of the workload's attributes. Also, this is when you have a sufficient number of objects in each data center. With a small number of objects, cost increase and latency reduction form a step function.

4.2.3 Use continuous replications

The attribute-popularity-based access predictions replicate objects to where they are likely to be accessed in the future; however, not all future accesses can be predicted. For example, accesses to objects with a new attribute or attribute that hasn't appeared for a long time can not be predicted. To achieve further cost reduction under



(a) Cost with latency SLOs (b) Latency with cost SLOs

Figure 29: **Navigating SLO constraints.** Cost and latency reduction of continuous replications under SLO constraints. Attributes are labeled with attr_1 to attr_n by their angles from the line extending full replication along the y-axis. attr_n can be a single or a combination of multiple attributes. One the left, the blue and (red+blue) areas represent the sum of all costs of continuous and non-continuous replication, respectively. On the right, they represent the sum of all latencies.

latency SLO constraints, or latency reduction under cost SLO constraints, you can add extra random replicas. This "continuous" replication allows a replication system to meet the SLOs without having to settle for suboptimal ones. Figure 29 explains this in more detail. In Figure 29(a), when you have a latency SLO Lat_{SLO} between Lat_{n+1} and Lat_n that can be achieved with attributes $attr_{n+1}$ or $attr_n$, respectively, you can use a replication on the broken line $(attr_n, b, attr_{n+1})$, where b is the intersection point of the line $(attr_n, Full)$ and the vertical line extending $attr_{n+1}$. The cost of such replication system is:

$$\text{Cost} = \begin{cases} \text{Cost}_n + \text{Cost}_R, & \text{if } \text{Lat}_{\text{SLO}} \geq \text{Lat}_b \\ \text{Cost}_{n+1}, & \text{otherwise.} \end{cases} \tag{10}$$

Without continuous replication, you would have to use attr_{n+1} that has a higher cost Cost_{n+1} to satisfy Lat_{SLO}. Cost reduction R_{Cost} of continuous replication systems over non-continuous ones is

$$R_{\text{Cost}} = \text{Cost}_{n+1} - \min\left(\text{Cost}_{n+1}, \text{Cost}_n + \text{Cost}_R\right) \tag{11}$$

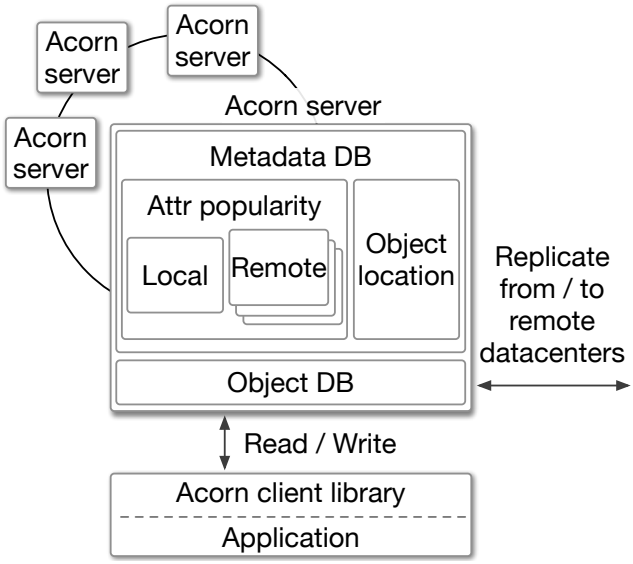


Figure 30: **Overview of the Acorn system architecture.**

, where Cost_R is the cost of the extra random replicas. In the same manner, a latency reduction R_{Lat} of

$$R_{\text{Lat}} = \text{Lat}_n - (\text{Lat}_n - \text{Lat}_R) \tag{12}$$

can be achieved, as shown in Figure 29(b).

4.3 System Design and Implementation

ACORN consists of several ACORN servers – each of which stores a subset of objects and metadata for scalability and fault-tolerance – and a client library that applications link against, as shown in Figure 30.

Metadata DB consists of attribute popularity tables – which store both local and remote popularity of attributes – and object location tables.

4.3.1 Attribute popularity monitor

ACORN servers in each data center monitor the popularity of each attribute independently from other data centers using a sliding time window per each attribute, as shown in Figure 31. For example, ACORN can monitor two attribute popularities;

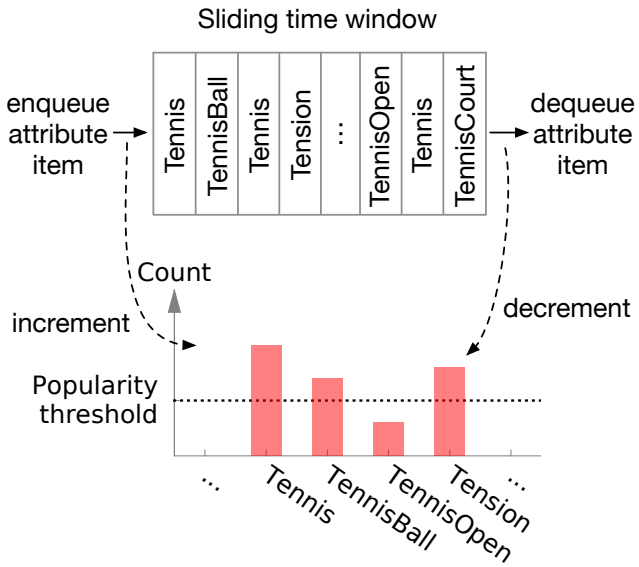


Figure 31: Attribute popularity monitor.

one for "user" and the other for "topic". Popularity of an attribute item is increased as it enters the sliding time window and decreased as it exits the window. The length of the sliding time window is configurable for each application. In general, short windows give adaptability to changing data access patterns, and long windows remember access history better, thus are better for attributes that do not happen very often. Popularity counters are shared by all Acorn servers in a data center. A popularity threshold per each attribute determines whether an attribute item is popular or unpopular. A low threshold detects changes of popularity faster, but suffers from noise; on the other hand, a high threshold is robust from noise, but not adaptive to changes.

4.3.2 Popularity synchronization between data centers

Acorn synchronizes attribute popularity metadata periodically with remote data centers, as shown in Figure 32. An attribute popularity synchronizer node, selected from Acorn servers in a data center, acts as a proxy for the synchronization. This resembles Cassandra's approach of using a coordinator node to aid inter-data center communication [95]. During every synchronization epoch, a synchronizer node calculates changes in popularity items since the previous epoch, i.e., newly popular

items and items that are no longer popular, and broadcasts them to remote data centers with the current synchronization epoch ID. Upon receiving an update from a remote data center, a synchronizer node first checks the synchronization epoch ID of the update to make sure it is getting an incremental update. If the update fails to pass the check, i.e., if the epoch ID value is not bigger than the previous one by one, the receiving node sends a retransmission request with the expected synchronization epoch ID to the sender node. When updates keep failing the check, e.g., from a synchronizer node being unavailable from a network outage for more than one synchronization period, the receiving node sends a full-attribute-popularity-transmit request.

All current and previous snapshots are partitioned and stored across Acorn servers in a data center in the same way objects are stored, as depicted in red color in Figure 32. The previous local attribute popularity snapshot is cached in the main memory of the synchronizer node for a fast calculation of changes at the next synchronization epoch.

The synchronization frequency is configurable to best serve the workload of an application. The frequency is a tradeoff between synchronization overhead and keeping more up-to-date attribute popularity snapshots: the more often you synchronize, the higher synchronization cost you pay, but you obtain the more accurate knowledge of remote attribute popularity. An extreme example is to propagate updates as soon as there is a change in popularity, which is similar to the way Cassandra propagates mutations. As a last step, Acorn compresses the changes to further reduce the inter-data center network traffic.

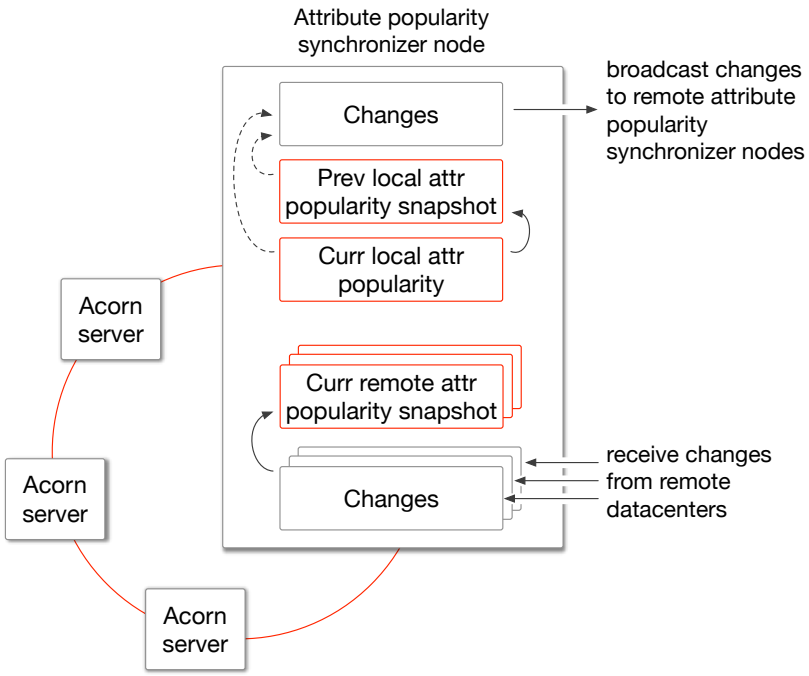


Figure 32: Synchronizing attribute popularity

4.3.3 Partial Replication

Upon a write request, ACORN makes replication decisions independently in each data center based on its attribute popularity snapshots, as shown in Figure 33(a). In contrast to existing work, ACORN does not involve any global component, which usually provides availability and fault-tolerance; global placement manager of SpanStore [115] or global configuration service of Tuba [18].

After writing an object in a data center, ACORN updates the object location metadata, which is looked up when an ACORN server misses the object in the local data center and needs to fetch the object from a remote data center. ACORN uses a combination of eventual and strong consistency when writing to and reading from an object location table. It writes object locations asynchronously, i.e., with *CL* (consistency level) *One*[1], and reads locations of an object from a local data center. When there is no object location information in the local data center, then ACORN

[1]We use Cassandra's *CL* (consistency level) term [43], defined as the number of replicas with successful responses before returning an acknowledgment to client application.

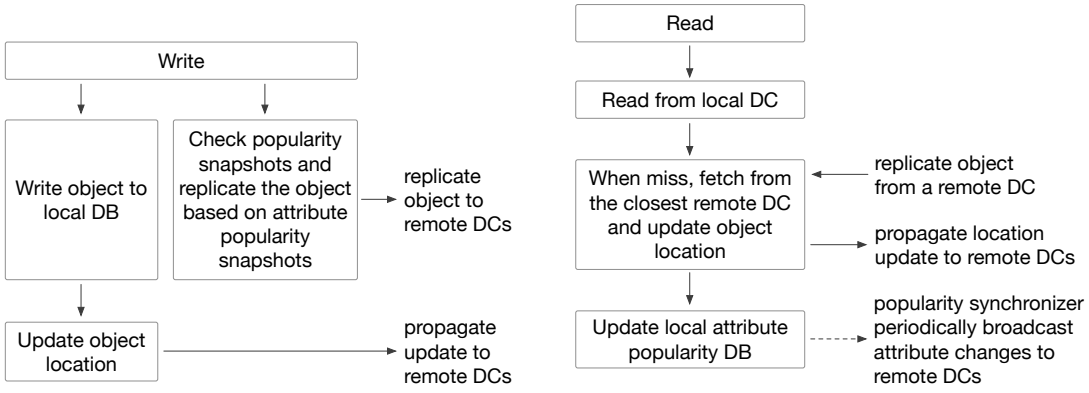


Figure 33: Steps to issue a write operation (left) and read operation (right) in Acorn.

reads from all data centers, i.e., with *CL All*. The protocol allows location table accesses to be predominantly asynchronous or data center-local, resulting in lower latencies.

A read request first looks for an object in a local data center; then, if the request misses, the client then fetches the object from the closest (the one with the lowest network latency) remote data center, as shown in Figure 33(b).

4.3.4 Acorn and Consistency Models

Although Acorn focuses on immutable data, a multi-version concurrency control (MVCC) system can be built on top of Acorn to store mutable data. The consistency model such a system would have depends on how it writes and reads object location metadata:

- Any consistency level weaker than the equivalent of quorum consistency, i.e., writes with *CL Quorum* and reads with *CL Quorum*, gives you an "eventual" knowledge of the object locations, and a MVCC system on top of that will have an eventual consistency.

- With a *CL Quorum* or stronger, you obtain up-to-date object locations, and any consistency model can be built on top of this basis; eventual [45], causal (you have the knowledge of from where to pull objects to meet unmet dependencies)

[73, 74], RMW (read-my-writes) [104], or strong model.

Without additional extensions, the most natural description of ACORN's current model is eventual consistency. Eventual consistency has often been the model of choice in scalable multi-data center environments where data synchronization costs are high due to availability and fault-tolerance constraints. Clients may read stale data and are themselves responsible for reconciling any conflicts [45]. When combined with partial replication, eventually-consistent systems can achieve further cost reduction by minimizing un-accessed replicas. Clients get consistent results eventually when the location metadata of the requested objects are synchronized. A RMW scheme would also be readily compatible with ACORN: since clients rarely change their data centers, partial replication naturally lowers cost with negligible reduction in local data center hit ratio.

4.3.5 Implementation Details

We implement ACORN by modifying Apache Cassandra [70], a popular distributed database management system built for high availability, scalability, and fault-tolerance. We modify the write path so that the StorageProxy module writes objects to the local data center and to a subset of remote data centers by checking their attributes against attribute popularity snapshots of remote data centers. Internally, *CL* (consistency level) is modified so that it does not exceed the number of replicas, which is usually smaller than the requested *CL* due to partial replication. This allows a client to wait for the correct number of acknowledgments from remote servers. StorageProxy, after writing an object to local object DB, updates the object location table for future on-demand fetch operations to locate the object. Read serves a requested object from a local data center first, and, when it misses the object, it fetches the object from the closest remote data center. The request is notified to attribute popularity monitor asynchronously.

ACORN's popularity monitor uses Cassandra counters [42] internally to monitor the popularity of each attribute item. They are "eventually" consistent, just like any other writes and reads with their *CL*s less than *Quorum*. Here are the two reasons that the counters serves ACORN's purpose very well. First, they are designed to favor performance and scalability rather than being "strongly" consistent at all times; we want a light-weight counter, and a bit of inconsistency in popularity counters rarely affects whether each popularity item passes the popularity threshold or not. Second, those inconsistencies requiring conflict resolutions rarely happen. Since we use a data center-local popularity monitor, there are no mutation propagations that cross data centers. It makes the counters not suffering from long network delays, which is most often the cause of counter value conflicts.

Similar to Cassandra's multi-data center coordinator nodes [95], ACORN has one attribute popularity synchronizer node per data center, which periodically broadcasts changes in local attribute popularity tables to remote synchronizer nodes. This saves the inter-data center communications, since each ACORN server does not need to talk to multiple remote data center nodes to propagate update when the replication factor of the attribute popularity table is bigger than 1. Whereas any Cassandra node can be a coordinator node for each mutation propagation, ACORN designates a static synchronizer node in each data center.[1] Upon reception of changes from remote data centers, it updates local popularity snapshot tables.

4.4 Evaluation

We evaluate ACORN by comparing it with existing, user-based systems in terms of cost and latency using two types of applications: a public data-sharing application like YouTube or Flickr and a private data-sharing application like Facebook or Snapchat.

[1] This works fine for the purpose of the work. In case the synchronizer node becomes unavailable, a protocol of voting of the synchronizer node and disseminating the information could be designed.

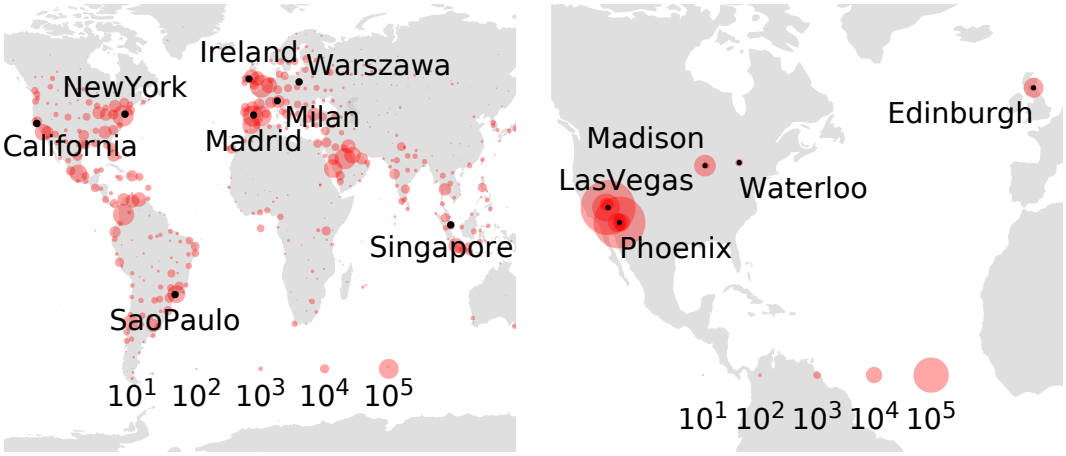


Figure 34: **Data access and data center locations** for a public and private data-sharing application. Red circles represent the access locations, and the size of each circle represents the number of accesses in a 1-degree longitude and 1-degree latitude area. Black dots and labels represent where data centers are placed.

4.4.1 Experimental setup

The workload of the public data-sharing application is gathered from an extensive crawling of Twitter that have both YouTube links and coordinates. We used the Snowball crawling method [98, 59] with periodic random seedings to prevent the crawler from being biased towards nodes with a high number of edges. We assigned the first tweet about each YouTube video to a write (upload) request to the video, and the rest of the tweets to read (view) requests to it. The workload had 833 K users, 2.3 M YouTube videos, and 7.2 M accesses to the videos from Aug. 2010 to April 2015. For the private data-sharing application, we used a Yelp dataset [118], which had 1.1 M reviews, 253 K users, and a 956 K-edge social graph in 5 different regions. From the dataset, we built social network application requests: users check (read requests) the latest reviews (write requests) from their friends, just like Facebook users check the status updates of their friends. It is similar to how SONG [49] generates social network operations.

We test Acorn with both simulated and real data center setups: a multi-data center testbed at Georgia Institute of Technology with real-world network latencies

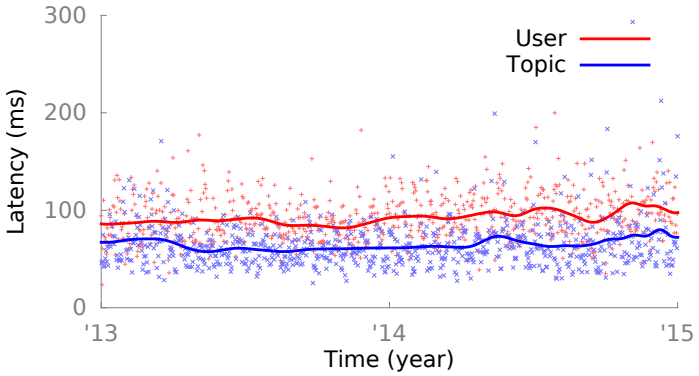


Figure 35: **Sampled latencies of user-based vs. topic-based replications.** in the AWS Singapore data center for the public data-sharing application. For a fair cost comparison, extra 11.82% of random replicas are added to the user-based replication.

among data centers[1] and AWS data centers in 9 different regions. For the purpose of our experiments, we manually placed data centers to evenly balance the workloads, as shown in Figure 34.

We used AWS's pricing model for geo-replication cost, both for storage and inter-data center networking cost.

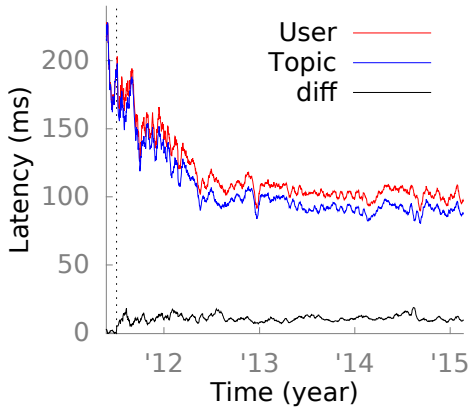


Figure 36: **Latency of the public data-sharing application using different attributes.** The vertical axis is the average latency over all data centers. The latency difference between topic-based and user-based replications becomes consistently noticeable after 2.85% of the experiment time, which is shown in the dotted vertical line.

[1]The values were obtained from the average ping RTTs among AWS EC2 instances in each region, and, in those regions where AWS does not provide services, we used RTTs between the routers of Hurricane Electric's Network Looking Glass [60]. We used tc, a Linux traffic control tool, for inserting the network latencies.

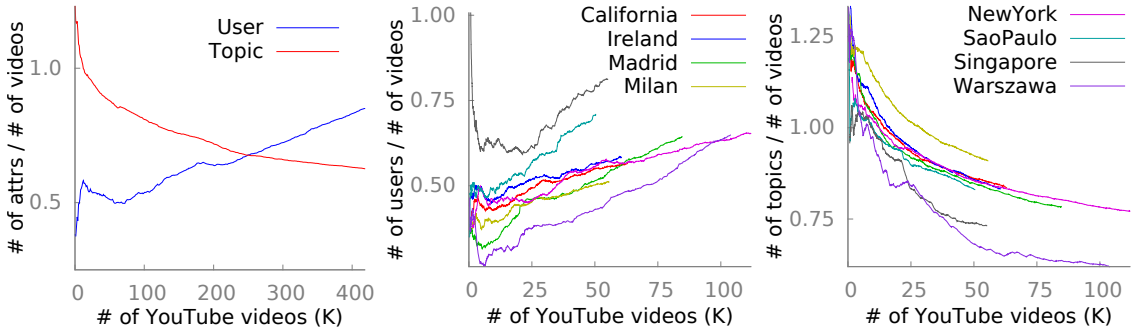


Figure 37: **Growth of attributes in crawl of YouTube accesses**. The number of topics grows slower than the number of users both globally (left) and in each data center (in center and on the right). Different numbers of attributes in each data center can be explained by the different levels of the diversity of interests, languages, or demographics in each region. The horizontal axis is the number of YouTube videos, which grows as the data is being crawled.

4.4.2 Evaluation metrics

To compare replication models independently from application sizes or object sizes, we use cost and latency overhead as evaluation metrics. We define cost overhead CO of a replication model as a relative cost to the minimum cost, which is the cost of the no-active replication model (NA):

$$\mathrm{CO} = (\mathrm{Cost} - \mathrm{Cost}_{\mathrm{NA}})/\mathrm{Cost}_{\mathrm{NA}}. \tag{13}$$

This gives you a relative cost regardless of your application sizes, whether it's a small-scale application or a YouTube-scale one. You can also experiment on a small scale before deploying a full-scale application. In the same manner, we define latency overhead LO as a relative latency to the minimum latency:

$$\mathrm{LO} = (\mathrm{Lat} - \mathrm{Lat}_{\mathrm{Full}})/\mathrm{Lat}_{\mathrm{Full}}. \tag{14}$$

This is independent of your object sizes, whether it's the size of a tweet or the sizes of YouTube videos. The partial replication with future knowledge model has a $\mathrm{LO} = 0$ and a $\mathrm{CO} = 0$, which is every partial geo-replication system's ultimate goal.

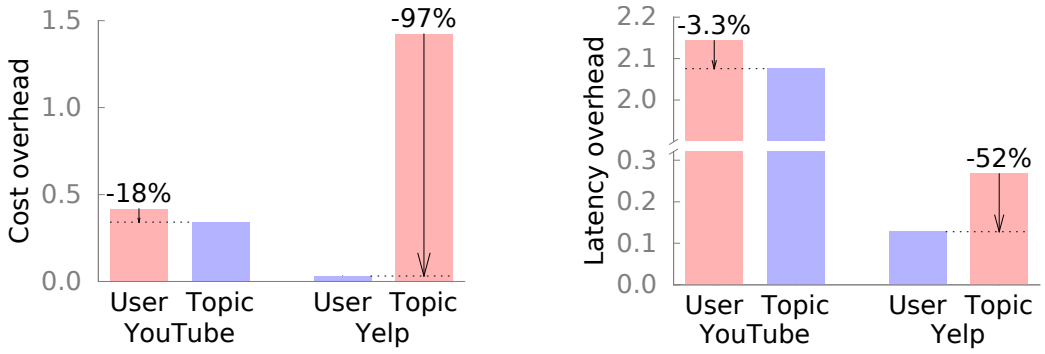


Figure 38: Cost and latency overhead comparisons by attributes and by application types

4.4.3 The right attributes for each application

Different types of applications have different attributes that replication decisions can be made best out of. For the public data-sharing application, topic-based replication consistently outperformed user-based replication, as shown in Figure 35. This is because data is accessed through various channels, and the "user"-based channel is not the best one for making replication decisions. In this case, "topic" captures the popularity of objects better than "user". In the private data-sharing application, where data is shared among friends, "user" popularity is a natural choice.

Figure 38 compares cost and latency overhead using different attributes by different types of applications.

For the private data-sharing application, the result seem to significantly favor "user". The skew can be explained by the strong geometric locality of the friend network graph and the uneven workload. First, most of the inter-data center friendships are between the 2 close data centers (Phoenix and Las Vegas), and the network distance between them is very close: the average network latency between them is only 24 ms, compared to 183 ms of the YouTube data center setup. Second, the majority of the requests come from 2 out of the top 5 regions, as shown in Figure 34(b). Thus, when you replicate objects to where your friends are popular, not only do you make

accurate predictions, you do so without much waste (un-accessed objects), compared to making replications based on "topic" popularity. A workload with a friend network graph reaching out to more distant data centers or a data center setup with longer distances among them would have made more un-accessed objects, resulting in a less favorable result to "user".

A natural follow-up question is: *how do you know which attributes are the best for your application?* Sometimes they are trivial for application designers; for example, in social network applications, where checking status updates of friends is the most-used feature, it is trivial that "user" is the best attribute. Other applications can find their attributes with trial runs of Acorn with some initial part of the data. Figure 36 shows an example of a trial run of less than 3% of the data identifies "topic" as better than "user". It is interesting to observe that the latency decreases over time and stabilizes from around May 2012. The stabilization is because the number of accesses per unit time in our YouTube dataset increases over time; **(a)** Our crawler started from the end of the simulation time period, May 2015, and followed the links of parents and children. It makes newer videos are more likely to be found than older videos. **(b)** YouTube and Twitter have grown over time, and the number of YouTube videos increases over time. The increased video request density allows Acorn to make the proactive replication decisions more accurately. One would think monitoring popularity of "topics" might be impractical since there are an unlimited number of topics. It turns out, with our YouTube workload, monitoring "topics" is easier and more scalable than monitoring "users", since they grow slower both globally and data center wise, as shown in Figure 37. We believe the number of topics has saturates at roughly the total number of words in all languages and the usage of them follows a power law; there are about 1 million English words as of 2016 [107] and, in Twitter, the top 10 languages have an 88% share of all tweets [91].

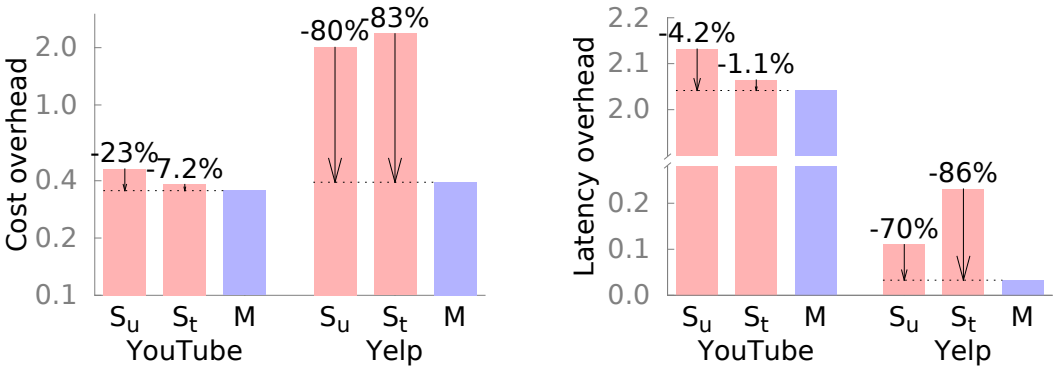


Figure 39: **Cost and latency overheads comparison of single- vs. multi-attribute-based replication.** S_u and S_t use a single-attribute "user" and "topic", respectively, to make replication decisions. M uses both of them.

4.4.4 Multi-attribute based replication

In both public and private data-sharing applications, multi-attribute based replications outperform single-attribute based ones under SLO constraints, as shown in Figure 39: up to 23% and 83% cost overhead reductions in the public and private data-sharing applications, respectively, and up to 4.2% and 86% latency overhead reductions.

Figure 40 also confirms that using multiple attributes almost always outperforms using a single attribute. There is a slight latency inversion in the beginning of 2014, but we think this performance variability is inherent in public, virtualized cloud environments.

4.4.5 Continuous replication

Figure 41(a) compares the cost of a continuous replication system with the cost of a non-continuous replication system under latency SLO constraints. When the latency SLO is high, i.e., when the storage system has a good latency budget, the system can be configured to no-active replication model and run with the lowest cost. As the latency SLO decreases, the system needs to move from a no-active replication model to user-based to topic-based to (user+topic)-based, and finally to full replication model

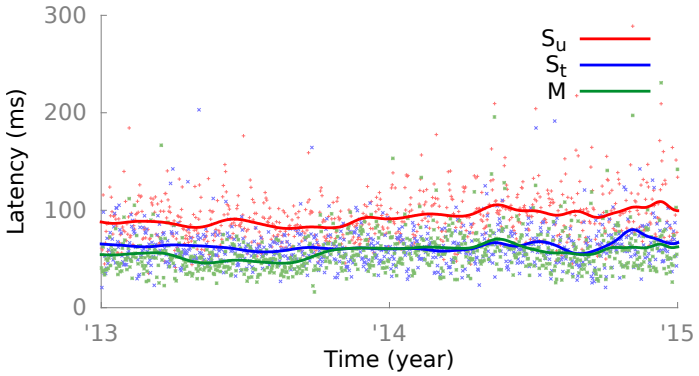


Figure 40: **Sampled latencies of single attribute- vs. multi attribute-based replications** in the AWS Singapore data center for the public data-sharing application. To make a fair cost comparison with the multi attribute-based replication M, 12.29% and 0.54% of extra random replicas are added to the user-based (S_u) and topic-based (S_t) replications.

to meet the decreased latency SLO. A non-continuous replication system needs to follow the red line to meet the decreased latency SLOs, resulting in bigger cost jumps than a continuous replication system, which has smaller cost jumps since it can add as many as needed extra random replicas as the latency SLO decreases. The area under each line represents the sum of all cost under all latency SLOs; the blue area is the sum of all cost of the continuous replication system and the (red+blue) area is the sum of all cost of the non-continuous replication system. Our experiments with the public data-sharing application workload show that the continuous replication system has an average cost reduction of 40.62% over the non-continuous replication system. In the same manner, the continuous replication system has an average latency reduction of 35.00% over the non-continuous replication system under cost SLO constraints as shown in Figure 41(b).

Overall, Acorn achieves up to 54.28% and 89.91% cost overhead reduction and 37.73% and 90.90% latency overhead reduction for the public and private data-sharing applications over existing partial replication systems.

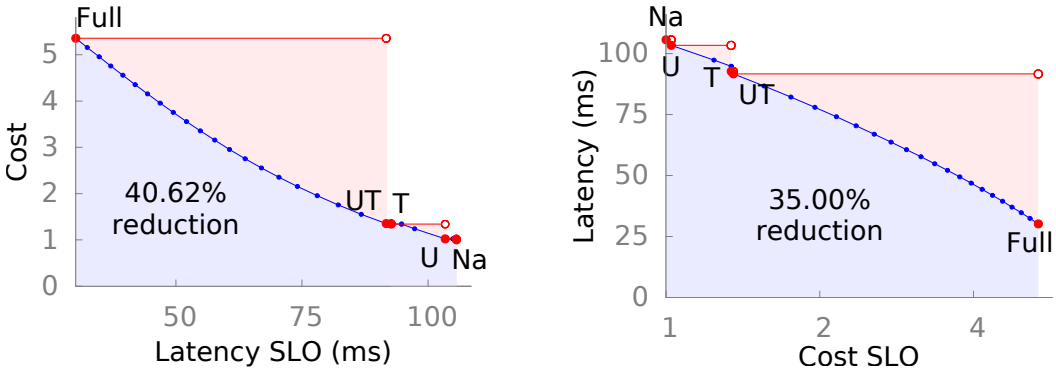


Figure 41: (a) **Cost reduction of the continuous replication system over the non-continuous replication system under latency SLO constraints** using the public data-sharing application workload. Each of the blue and the red line represents the cost of continuous and non-continuous replication system under latency SLOs. Cost in the y-axis is plotted in the relative cost to the no-active replication system for an easy comparison. UT, T, U, and Na at the top of the chart represent (user+topic)-based, topic-based, user-based, and no-active replication, respectively. Each of the blue dots represents a replication system by incrementally adding 5% of extra random replicas from the base replication model. (b) **Latency reduction of the continuous replication system over the non-continuous replication system under cost SLO constraints.** Cost SLO in the x-axis is the relative cost to the no-active replication system and plotted in logs scale to better show the replication models on the left.

4.5 Summary

In this chapter, we have presented Acorn, a partial geo-replication system with better and flexible cost-performance trade-offs. Acorn (a) makes better cost-performance trade-offs for various application types by using the right object attributes when making replication decisions and (b) supports fine-grained cost-performance trade-offs by using extra replicas. Compared to user-based replication systems, we showed that Acorn reduced up to 90% cost overhead and up to 91% latency overhead.

CHAPTER V

RELATED WORK

The work in this thesis was inspired by a wide distribution of areas in the research community. In this chapter, we discuss the main themes of the related work to cost-configurable storage systems in database storage systems and geo-replication systems.

5.1 Cost-configurable Database Storage Systems

LSM Tree Databases: LSM trees, invented by O'Neil [89], 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 [33, 70, 54, 52, 106], 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 [2, 51]. 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 [100], Partitioned Exponential Files that exploits properties of HDD head schedulers [61], WiscKey that separates keys from values to reduce write amplification [77], and work of Lim *et*

al. that analyzes and optimizes the SSTable compaction parameters [72]. 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 [85] 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 [50]. Cassandra stores SSTables to storages in a round-robin manner to guarantees even usage of storage devices [105].

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.

Hierarchical Storage Systems: The idea of separating working sets into storage subsystems with different characteristics has been explored in various storage systems. In block-storage systems, prominent work includes configurable RAID storage systems such as AutoRAID [113] that organize data into different RAID configurations based on activity, and SSD-HDD hybrid block storage systems such as Hystor [34] that identifies slow or semantically critical blocks and stores them on the faster SSD medium. Work on BLOB (binary large object) storage systems include Facebook's binary separation of photos and videos into "hot" and "warm" storage, specifically HayStack [23] and f4 [84], based on the object types and ages. The diverse cloud storage options, including AWS S3 Standard, Infrequent access, and Glacier [1] and Google Storage's Regional, Nearline, and Coldline [3], can be used as building blocks for hierarchical storage systems. In-memory database systems such as Siberia [47] and Anti-caching [44] offload infrequently accessed records to secondary storage devices in a transactionally safe manner, thus better utilizing memory for "hot" records.

5.2 Cost-configurable Geo-replication Systems

Partial geo-replication systems. Kadambi [64] builds a selective replication system on top of Yahoo!'s PNUTS database and makes per-record replication decisions based on access statistics. It is a natural approach for building partially replicated social network applications where records are structured by users. SpanStore [115] makes user-based replication decisions; however, the assumption that clients have the knowledge of objects access set is unlikely to hold for the class of applications with dynamic temporal and geometric access patterns. It combines multiple cloud service providers to increase the geographic density, thus reducing cost as well as meeting various SLA requirements. Pileus [104] and Tuba [18] select replicas based on clients' consistency-based SLAs. Tuba builds on Pileus and automates system configurations as data access patterns or latencies change. They make replication decisions

based on the access statistics of tablets, which are horizontal partitions of tables as in BigTable [33]. TailGate [109] reduces and flattens network traffic with selective and delayed replications in social network applications. They use a friend graph and time difference in data access patterns in different locations. ACORN makes replication decisions using attributes of contents – "user" is an attribute just as other attributes such as "topic" or combinations of attributes – which delivers lower cost and service latency over static attribute based approaches. This generality enables ACORN to work well with both public and private data-sharing applications. The distributed design of ACORN enables each data center to make local replication decisions based on its local view of the other data centers, unlike the central design of SpanStore's PM (placement manager) or Tuba's CS (configuration service). Thus, there is no need for a global manager or single point of failure.

Partial replication systems. These systems are smaller in scale than partial geo-replication systems but share similar ideas. Wolfson [114] studies an adaptive replication algorithm based on object access statistics in multiprocessor systems. Bestavros [26] studies the speculative data push of web documents by monitoring their inter-dependency. It is suitable for serving a set of static objects. Cimbiosys [96] provides filtered replication of content on mobile devices through P2P synchronization. Replication decisions are made by looking at file metadata and filtering rules users specify. DARE [4] studies a distributed adaptive data replication in a Hadoop cluster. Globus [36] studies replica location service in grid environments.

Database systems. Many RDBMSs and NoSQL databases provide master-slave and master-master replications over multiple clusters but do not support dynamic partial replications; Oracle database provides both master-master (multi-master) and master-slave (materialized view) replications [90]. MySQL has master-slave model [86] and MySQL Cluster provides multi-master replication [87]. Microsoft SQL Server provides transactional and snapshot replication (master-slave) and merge replication

(master-master) [79]. MongoDB and HBase provide only master-slave replications [83, 57]. Cassandra provides the flexibility that clients can effectively configure multi-data center keyspace (database) by specifying the number of nodes in each data center [70], however, the configuration is static and does not adapt to dynamic workload.

Stream processing systems and data center monitoring systems. Large-scale real-time streaming systems, including Spark Streaming [119], Apache Storm [17], or Apache Flume [16], and data center monitoring systems, such as Ganglia [78] and Amazon CloudWatch [7], could be modified to monitor attribute popularity in data centers. ACORN uses Cassandra's counter to monitor attribute popularity, which favors scalability and low latency over strict accuracy. In our experiments, inaccuracy in attribute monitoring was very rare, and even when it happens, clients needed only a small number of additional object pulls.

5.3 Distributed Caching Systems

CDN systems are similar to partial geo-replication systems in that they make local copies of popular, immutable objects by either statically or dynamically analyzing the workload, and a lot of popularity monitoring and management ideas can be borrowed from them. Borst [28] presents distributed lightweight cooperative algorithms. Venkataramani [110] shows prefetching long-lived objects improves hit rates with Zipfian-distributed workload. Chen [35] and Fujita [53] replicate objects with a minimal management overhead by grouping them by their topology or popularity. Push-based strategies have shown possibilities [66, 92]; for instance, Hulu can save a lot of traffic by prefetching popular videos to proxies [68].

ACORN is different from CDN systems in the scale of deployment; ACORN is deployed in "core" data centers, while CDN systems are deployed in "edge" data centers, which are a lot bigger in numbers. It becomes inefficient to do a cooperative metadata management (ACORN's attribute popularity and location databases) as

the overhead per data center grows linearly with the total number of data centers. However, in a small scale deployment, CDN systems could borrow ideas from ACORN's proactive object replication.

Ager [5] reports caching user-generated content is economically infeasible due to the long-tail distribution. Our workload analysis of the public and private data-sharing applications agrees with it.

Facebook's photo caching stack [58] confirms a low local data center cache hit rate, 57.59%, which we calculated from the numbers in the paper. It is a common system design to place partially replicated database systems fronted by caching systems; the former can be placed in hub regions providing authoritative data, while the latter absorbs the load from clients at edge data centers. For instance, Netflix has a Cassandra and Memcached combination [97].

CHAPTER VI

CONCLUSION

In this chapter, we summarize the contributions of this thesis, the lessons we learned, and discuss a variety of future directions.

6.1 Summary of Contributions

In this thesis, we have explored the designs for **cost-configurable cloud storage systems** in 3 problem domains: LSM tree-based NoSQL database systems, edge cloud caching systems, and geo-replication storage systems. To provide cost-configurability in each of the system domains, we analyzed the workloads of the systems, abstracted the workloads to the metrics that best-represent them, and designed systems that re-balance the resource to meet the changing workload patterns and goals in real time.

We identified the fundamental **data placement problems** in each system domain and how the placement of data affects the cost and performance of the systems: (a) cost of cloud NoSQL databases vs. data access latency, (b) cost of edge cloud caching systems vs. cache hit rate and performance isolation level, and (c) cost of geo-replication vs. data access latency. Then, we implemented the systems to demonstrate the efficacy of our ideas and evaluated the systems with both real-world and synthetic workloads.

Exploring the system designs was not without challenges. The challenges we faced include (a) understanding how the state-of-the-art tools solve the problems in different system domains, (b) identifying where the performance bottlenecks are and what functionality lacks in each of the systems, (c) abstracting the problems in efficient ways to design and implement the systems, (d) finding ways to seamlessly integrate

the design into the existing systems, and (e) evaluating the systems extensively on real cloud platforms to show the effectiveness and the limits of the systems.

We overcame the challenges with by (a) thoroughly analyzing how the existing systems solve the problems, (b) measuring the performance with various metrics and examining the functionality, (c) designing time and space efficient data structures and algorithms to abstract the problem, (d) modular design and implementation of the systems and exposing minimal API sets to existing systems and clients of the systems, and (e) using both real world workloads and synthetic workloads that best evaluate the systems. Collaboration was one of the key factors in the success of the research. We continuously sought for collaboration opportunities and tried to expand the collaboration network. Through industry collaboration, we obtained early feedback, insight into real-world scenarios validating the problems and our solutions, and realistic, representative workloads.

6.2 Lessons Learned

In this section, we present a summary of general lessons we leaned while working on this thesis.

- **The need for cost-configurable system design in cloud storage systems.** As modern cloud storage systems grow to Internet scale, the inflexibility in the system cost has become a big problem. Without a system design that considers cost-configurability as a first class citizen, companies would suffer from either a significant amount of cost or a severe performance penalty that leads to a degradation of user experience levels, thus eventually losing their customer base.
- **New enablers for designing cost-configurable cloud storage systems.** It was once believed that providing fine-grained, seamless cost-performance trade-offs is too expensive or complicated to design. First, there were (a) limited storage device types and capacity and (b) limited geographic locations that the storage and

computation is hosted. Second, designing such systems were too complicated because (a) the traditional storage block designs, such as those of MySIAM or InnoDB or even using raw storage device volumes, were too coarse grained to support a fine-grained cost-configurability and (b) optimizing storage resource utility across various geographic locations took examining all storage allocation options exhaustively. New enablers are emerging to make such cost-configurable storage design possible. First, in the near future, edge cloud computing will allow companies to allocate computation and storage resource in fine-grained geographic locations in a elastic manner. Second, new system designs have emerged to allow fine-grained storage block allocations and global resource usage optimizations. To manage modern big data, many storage systems adopted fine-grained storage blocks that are managed in a log-structured manner. Many tools such as calculating data reuse distances have become popular allowing fast calculations on how much cache resource would be allocated to get a specific level of performance.

6.3 Future Opportunities

In this section, we present a variety of next possible steps, both incremental and larger efforts.

6.3.1 Cost-Performance Trade-Offs in NoSQL Database Systems

We have explored the cost-driven cost-performance trade-offs and cost-performance trade-offs in SSTable granularity. Future work includes exploring:

- Data access latency-driven cost-performance trade-offs. The cost-performance trade-offs can be driven by the data access latency as well as the database system cost. However, latency-driven trade-offs, unlike the cost-driven trade-offs, have more challenges, which come from (a) the inherent unpredictability of the system noise level in the shared cloud infrastructure platforms and (b) the feedback loop, which the effort to adjust a latency level affects the latency itself.

- Finer-grained cost-performance trade-offs. In addition to the proposed SSTable-level data organization, database records can also be organized in record level. The data access popularity of a small number of records in an SSTable can change while the popularity of the others remain the same. Techniques such as record-level caching or re-insertion of records to MemTable could be used to achieve such fine-grained cost-performance trade-offs.

6.3.2 Cost-Performance Trade-Offs in Geo-Replication Systems

We have explored the cost-performance trade-offs in partial geo-replications systems. Future work includes exploring:

- Fault-tolerance with partial geo-replications. One of the trade-offs with a reduced replication cost is a lower level of fault tolerance. Thus, it would be exciting to explore how likely it is to loose a data object with a lower number of replicas and looking at the possible solutions to mitigate the risks.
- Offloading unpopular objects. Modern web workloads exhibit highly skewed data access patterns, thus a majority of the object are rarely accessed. As well as reducing the number of replicas, one could explore offloading the object to cold storage such as Amazon Glacier or Google Cloud Coldline storage and look at the trade-offs between the reduced cost and the increased data access latency.
- Partial replication under client-defined constraints. Many businesses have geographic restrictions on where their data is stored. For example, most of government data and medical data require a stronger security and privacy level and is stored in a limited set of data centers, and some company data has restrictions on the countries it can be stored. These restrictions make the cost-performance trade-offs in partial geo-replication more challenging and exciting.

6.3.3 Cost-Performance Trade-Offs in Edge Cloud Caching Systems

We have explored the cost-performance trade-offs in edge cloud caching systems. Future work includes exploring:

- Reducing cold cache misses when resizing the cache cluster. Clients get a lot of cold cache misses when a cache cluster is resized due to the re-balancing of the keyspace ranges each cache node is responsible for. One can explore ways of keeping the cache items when a cache cluster is resized. When a cache cluster is expanding, the cache items that used to belong to old cache nodes can be incrementally moved to new cache nodes. When a cache cluster is shrinking, one can keep the most frequently-accessed cache items from the to-be-decommissioned cache nodes while evicting the less frequently accessed cache items in the to-be-kept cache nodes.
- Balancing the loads among cache nodes. Consistent hashing is almost universally used in distributed systems for the minimal data movement when resizing storage clusters. However, there is an inevitable load imbalance among the cache nodes regardless of the number of virtual nodes in the cluster. It would be exciting to explore different hashing algorithms and the trade-offs such as their performance, computation resource requirements, and scalability.

REFERENCES

[1] "Cloud Storage Classes - Amazon Simple Storage Service (S3)," 2017.

[2] "DSE 5.1 Architecture Guide - How is data maintained?," 2017.

[3] "Storage Classes — Cloud Storage Documentation — Google Cloud Platform," 2017.

[4] Abad, C. L., Lu, Y., and Campbell, R. H., "Dare: Adaptive data replication for efficient cluster scheduling," in *Cluster Computing (CLUSTER), 2011 IEEE International Conference on*, pp. 159–168, Ieee, 2011.

[5] Ager, B., Schneider, F., Kim, J., and Feldmann, A., "Revisiting cacheability in times of user generated content," in *INFOCOM IEEE Conference on Computer Communications Workshops , 2010*, pp. 1–6, March 2010.

[6] Amazon CloudFront, "How CloudFront Delivers Content," 2018.

[7] Amazon Web Services, "Amazon CloudWatch - Cloud & Network Monitoring Services," 2016.

[8] Amazon Web Services, "Amazon EC2 Spot Instances," 2018.

[9] Amazon Web Services, "Amazon EBS Volume Types," 2019.

[10] Amazon Web Services, "Amazon EC2 Instance Types," 2019.

[11] Amazon Web Services, "Amazon EC2 Instance Types - Amazon Web Services," 2019.

[12] Amazon Web Services, "Amazon EC2 Pricing," 2019.

[13] Amazon Web Services, "Amazon Elastic Black Store (EBS) - Amazon Web Services," 2019.

[14] Amazon Web Services, "Content Delivery Network Pricing - Amazon CloudFront," 2019.

[15] Anand, S., "How big companies migrate from one database to another," 2015. `https://www.quora.com/How-big-companies-migrate-from-one-database-to-another-without-losing-data-i-e-database-independent`.

[16] "Apache Flume," 2016.

[17] "Apache Storm," 2016.

[18] Ardekani, M. S. and Terry, D. B., "A self-configurable geo-replicated cloud storage system," in *Symp. on Op. Sys. Design and Implementation (OSDI)*, pp. 367–381, 2014.

[19] Atikoglu, B., Xu, Y., Frachtenberg, E., Jiang, S., and Paleczny, M., "Workload analysis of a large-scale key-value store," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 40, pp. 53–64, ACM, 2012.

[20] AWS, "Amazon CloudFront," 2018.

[21] Bakhtiyari, S., "Performance evaluation of the apache traffic server and varnish reverse proxies," Master's thesis, University of Oslo, 2012.

[22] Balakrishnan, S., Black, R., Donnelly, A., England, P., Glass, A., Harper, D., Legtchenko, S., Ogus, A., Peterson, E., and Rowstron, A. I., "Pelican: A building block for exascale cold data storage," in *OSDI*, pp. 351–365, 2014.

[23] Beaver, D., Kumar, S., Li, H. C., Sobel, J., Vajgel, P., and others, "Finding a needle in haystack: Facebook's photo storage," in *OSDI*, vol. 10, pp. 1–8, 2010.

[24] Beckmann, N. and Sanchez, D., "Talus: A simple way to remove cliffs in cache performance," in *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, pp. 64–75, IEEE, 2015.

[25] Bergman, T. L. and Incropera, F. P., *Fundamentals of heat and mass transfer*. John Wiley & Sons, 2011.

[26] Bestavros, A., "Speculative data dissemination and service to reduce server load, network traffic and service time in distributed information systems," in *Data Engineering, 1996. Proceedings of the Twelfth International Conference on*, pp. 180–187, IEEE, 1996.

[27] Bloom, B. H., "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[28] Borst, S., Gupta, V., and Walid, A., "Distributed caching algorithms for content distribution networks," in *Proceedings of the 29th Conference on Information Communications*, INFOCOM'10, (Piscataway, NJ, USA), pp. 1478–1486, IEEE Press, 2010.

[29] Böttger, T., Cuadrado, F., Tyson, G., Castro, I., and Uhlig, S., "Open connect everywhere: A glimpse at the internet ecosystem through the lens of the netflix cdn," *ACM SIGCOMM Computer Communication Review*, vol. 48, no. 1, pp. 28–34, 2018.

[30] Brewer, E., "Cap twelve years later: How the "rules" have changed," *Computer*, vol. 45, no. 2, pp. 23–29, 2012.

[31] Brodersen, A., Scellato, S., and Wattenhofer, M., "Youtube around the world: geographic popularity of videos," in *Proceedings of the 21st international conference on World Wide Web*, pp. 241–250, ACM, 2012.

[32] Bulkowski, B., "Amazon EC2 I3 Performance Results," 2017.

[33] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E., "Bigtable: A distributed storage system for structured data," *ACM Transactions on Computer Systems (TOCS)*, vol. 26, no. 2, p. 4, 2008.

[34] Chen, F., Koufaty, D. A., and Zhang, X., "Hystor: making the best use of solid state drives in high performance storage systems," in *Proceedings of the international conference on Supercomputing*, pp. 22–32, ACM, 2011.

[35] Chen, Y., Qiu, L., Chen, W., Nguyen, L., Katz, R. H., and Katz, Y. H., "Efficient and adaptive web replication using content clustering," *IEEE Journal on Selected Areas in Communications*, vol. 21, pp. 979–994, 2003.

[36] Chervenak, A. L., Schuler, R., Ripeanu, M., Amer, M. A., Bharathi, S., Foster, I., Iamnitchi, A., and Kesselman, C., "The globus replica location service: design and experience," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 20, no. 9, pp. 1260–1272, 2009.

[37] Cisco, "Cisco Global Cloud Index: Forecast and Methodology, 2014-2019," 2015.

[38] Cisco, "Cisco Global Cloud Index: Forecast and Methodology, 20162021," 2018.

[39] Colarelli, D. and Grunwald, D., "Massive arrays of idle disks for storage archives," in *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pp. 1–11, IEEE Computer Society Press, 2002.

[40] Conley, M., Vahdat, A., and Porter, G., "Achieving cost-efficient, data-intensive computing in the cloud," in *Proceedings of the Sixth ACM Symposium on Cloud Computing*, pp. 302–314, ACM, 2015.

[41] Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., and Sears, R., "Benchmarking cloud serving systems with YCSB," in *Proceedings of the 1st ACM symposium on Cloud computing*, pp. 143–154, ACM, 2010.

[42] Datastax, "Whats New in Cassandra 2.1: Better Implementation of Counters," 2014.

[43] Datastax, "Apache Cassandra - Configuring data consistency," 2015.

[44] DeBrabant, J., Pavlo, A., Tu, S., Stonebraker, M., and Zdonik, S., "Anti-caching: A new approach to database management system architecture," *Proceedings of the VLDB Endowment*, vol. 6, no. 14, pp. 1942–1953, 2013.

[45] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W., "Dynamo: amazon's highly available key-value store," in *ACM SIGOPS Operating Systems Review*, vol. 41, pp. 205–220, ACM, 2007.

[46] dev@trafficserver.apache.org, "Cache Storage - Apache Traffic Server 9.0.0 Documentation," 2019.

[47] Eldawy, A., Levandoski, J., and Larson, P.-Å., "Trekking through siberia: Managing cold data in a memory-optimized database," *Proceedings of the VLDB Endowment*, vol. 7, no. 11, pp. 931–942, 2014.

[48] Ellis, J., "Leveled Compaction in Apache Cassandra," 2011.

[49] Erramilli, V., Yang, X., and Rodriguez, P., "Explore what-if scenarios with song: social network write generator," *arXiv preprint arXiv:1102.0699*, 2011.

[50] Facebook, "RocksDB source code - Compaction picker," 2017.

[51] Facebook, "RocksDB Wiki - Compaction," 2017.

[52] Facebook Open Source, "RocksDB - A persistent key-value store for fast storage environments," 2019.

[53] Fujita, N., Ishikawa, Y., Iwata, A., and Izmailov, R., "Coarse-grain replica management strategies for dynamic replication of web contents," *Comput. Netw.*, vol. 45, pp. 19–34, May 2004.

[54] Ghemawat, S. and Dean, J., "LevelDB," 2017.

[55] Google Cloud, "Google Cloud CDN - Low Latency Content Delivery," 2018.

[56] Google Cloud Plaform, "Storage Options," 2017.

[57] HBase, "HBase 2.0.0. Cluster Replication," 2016.

[58] Huang, Q., Birman, K., van Renesse, R., Lloyd, W., Kumar, S., and Li, H. C., "An analysis of facebook photo caching," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pp. 167–181, ACM, 2013.

[59] Huguenin, K., Kermarrec, A.-M., Kloudas, K., and Taïani, F., "Content and geographical locality in user-generated content sharing systems," in *Proceedings of the 22nd international workshop on Network and Operating System Support for Digital Audio and Video*, pp. 77–82, ACM, 2012.

[60] Hurricane Electric, "Network Looking Glass," 2016.

[61] Jermaine, C., Omiecinski, E., and Yee, W. G., "The partitioned exponential file for database storage management," *The VLDB Journal-The International Journal on Very Large Data Bases*, vol. 16, no. 4, pp. 417–437, 2007.

[62] Josh James, "Data Never Sleeps 6.0," 2018.

[63] Kadambi, S., Chen, J., Cooper, B. F., Lomax, D., Ramakrishnan, R., Silberstein, A., Tam, E., and Garcia-Molina, H., "Where in the world is my data," in *Proceedings International Conference on Very Large Data Bases (VLDB)*, 2011.

[64] Kadambi, S., Chen, J., Cooper, B. F., Lomax, D., Ramakrishnan, R., Silberstein, A., Tam, E., and Garcia-Molina, H., "Where in the world is my data," in *Proceedings International Conference on Very Large Data Bases (VLDB)*, 2011.

[65] Kambatla, K. and Chen, Y., "The truth about mapreduce performance on ssds.," in *LISA*, pp. 109–118, 2014.

[66] Kangasharju, J., Roberts, J., and Ross, K. W., "Object replication strategies in content distribution networks," *Comput. Commun.*, vol. 25, pp. 376–383, Mar. 2002.

[67] Kleinberg, J. and Tardos, E., *Algorithm design*. Pearson Education India, 2006.

[68] Krishnappa, D. K., Khemmarat, S., Gao, L., and Zink, M., "On the feasibility of prefetching and caching for online tv services: a measurement study on hulu," in *Passive and Active Measurement*, pp. 72–80, Springer, 2011.

[69] Kumar, S., "Efficiency at scale," *International Workshop on Rack-scale Computing*, 2014.

[70] Lakshman, A. and Malik, P., "Cassandra: a decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.

[71] Liddle, J., "Amazon found every 100ms of latency cost them 1% in sales," *The GigaSpaces*, vol. 27, 2008.

[72] Lim, H., Andersen, D. G., and Kaminsky, M., "Towards accurate and fast evaluation of multi-stage log-structured designs," in *FAST*, pp. 149–166, 2016.

[73] Lloyd, W., Freedman, M. J., Kaminsky, M., and Andersen, D. G., "Don't settle for eventual: scalable causal consistency for wide-area storage with cops," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pp. 401–416, ACM, 2011.

[74] Lloyd, W., Freedman, M. J., Kaminsky, M., and Andersen, D. G., "Stronger semantics for low-latency geo-replicated storage.," in *NSDI*, pp. 313–328, 2013.

[75] Lohr, S., "For impatient web users, an eye blink is just too long to wait," *New York Times*, no. February 29, 2012.

[76] Lohr, S., "For Impatient Web Users, an Eye Blink Is Just Too Long to Wait," 2012.

[77] Lu, L., Pillai, T. S., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H., "Wisckey: Separating keys from values in ssd-conscious storage," in *FAST*, pp. 133–148, 2016.

[78] Massie, M. L., Chun, B. N., and Culler, D. E., "The ganglia distributed monitoring system: design, implementation, and experience," *Parallel Computing*, vol. 30, no. 7, pp. 817–840, 2004.

[79] Microsoft, "Microsoft SQL Server 2014. Types of replication," 2016.

[80] Microsoft Azure, "Disk storage," 2017.

[81] Microsoft Azure, "High-performance Premium Storage and managed disks for VMs," 2017.

[82] Microsoft Azure, "Content Delivery Network (CDN)," 2018.

[83] MongoDB, "MongoDB 3.0. Replication Introduction," 2016.

[84] Muralidhar, S., Lloyd, W., Roy, S., Hill, C., Lin, E., Liu, W., Pan, S., Shankar, S., Sivakumar, V., Tang, L., and others, "f4: Facebook's warm blob storage system," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pp. 383–398, 2014.

[85] Muth, P., O'Neil, P., Pick, A., and Weikum, G., "The lham log-structured history data access method," *The VLDB Journal-The International Journal on Very Large Data Bases*, vol. 8, no. 3-4, pp. 199–221, 2000.

[86] MySQL, "MySQL 5.7 Reference Manual. Replication," 2016.

[87] MySQL, "MySQL Cluster Replication: Multi-Master and Circular Replication," 2016.

[88] Niu, Q., Dinan, J., Lu, Q., and Sadayappan, P., "Parda: A fast parallel reuse distance analysis algorithm," in *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pp. 1284–1294, IEEE, 2012.

[89] O'Neil, P., Cheng, E., Gawlick, D., and O'Neil, E., "The log-structured merge-tree (lsm-tree)," *Acta Informatica*, vol. 33, no. 4, pp. 351–385, 1996.

[90] Oracle, "Oracle Database 12c. Introduction to Advanced Replication," 2016.

[91] Orcutt, M., "The Many Tongues of Twitter," 2013.

[92] Pallis, G. and Vakali, A., "Insight and perspectives for content delivery networks," *Communications of the ACM*, vol. 49, no. 1, pp. 101–106, 2006.

[93] Panda, A., McCauley, J. M., Tootoonchian, A., Sherry, J., Koponen, T., Ratnasamy, S., and Shenker, S., "Open network interfaces for carrier networks," *ACM SIGCOMM Computer Communication Review*, vol. 46, no. 1, pp. 5–11, 2016.

[94] Plain Vanilla Games, "QuizUp - The Biggest Trivia Game in the World," 2017.

[95] Planet Cassandra, "Data Replication in NoSQL Databases Explained," 2015.

[96] Ramasubramanian, V., Rodeheffer, T. L., Terry, D. B., Walraed-Sullivan, M., Wobber, T., Marshall, C. C., and Vahdat, A., "Cimbiosys: A platform for content-based partial replication," in *Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, pp. 261–276, 2009.

[97] Ruslan Meshenberg, Naresh Gopalani, L. K., "The Netflix Tech Blog: Active-Active for Multi-Regional Resiliency," 2013.

[98] Scellato, S., Mascolo, C., Musolesi, M., and Latora, V., "Distance matters: geo-social metrics for online social networks," in *Proceedings of the 3rd conference on Online social networks*, pp. 8–8, 2010.

[99] Schurman, E. and Brutlag, J., "The user and business impact of server delays, additional bytes, and http chunking in web search," in *Velocity Web Performance and Operations Conference*, 2009.

[100] Sears, R. and Ramakrishnan, R., "blsm: a general purpose log structured merge tree," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pp. 217–228, ACM, 2012.

[101] Services, A. W., "Amazon EC2 Pricing," 2017.

[102] Services, A. W., "Amazon EBS Pricing," 2018.

[103] Tang, L., Huang, Q., Puntambekar, A., Vigfusson, Y., Lloyd, W., and Li, K., "Popularity prediction of facebook videos for higher quality streaming," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, USENIX Association, 2017.

[104] Terry, D. B., Prabhakaran, V., Kotla, R., Balakrishnan, M., Aguilera, M. K., and Abu-Libdeh, H., "Consistency-based service level agreements for cloud storage," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pp. 309–324, ACM, 2013.

[105] The Apache Software Foundation, "Cassandra source code - Directories.java," 2017.

[106] The Apache Software Foundation, "Apache HBase," 2019.

[107] The Global Language Monitor, "Number of Words in the English Language," 2016.

[108] Torres, R., Finamore, A., Kim, J. R., Mellia, M., Munafo, M. M., and Rao, S., "Dissecting video server selection strategies in the youtube cdn," in *Distributed Computing Systems (ICDCS), 2011 31st International Conference on*, pp. 248–257, IEEE, 2011.

[109] Traverso, S., Huguenin, K., Trestian, I., Erramilli, V., Laoutais, N., and Papagiannaki, K., "Social-aware replication in geo-diverse online systems," *Parallel and Distributed Systems, IEEE Transactions on*, vol. PP, no. 99, pp. 1–1, 2014.

[110] Venkataramani, A., Yalagandula, P., Kokku, R., Sharif, S., and Dahlin, M., "The potential costs and benefits of long-term prefetching for content distribution," *Comput. Commun.*, vol. 25, pp. 367–375, Mar. 2002.

[111] Waldspurger, C., Saemundsson, T., Ahmad, I., and Park, N., "Cache modeling and optimization using miniature simulations," in *Proceedings of USENIX ATC*, pp. 487–498, 2017.

[112] Waldspurger, C. A., Park, N., Garthwaite, A. T., and Ahmad, I., "Efficient MRC Construction with SHARDS," in *FAST*, pp. 95–110, 2015.

[113] Wilkes, J., Golding, R., Staelin, C., and Sullivan, T., "The hp autoraid hierarchical storage system," *ACM Transactions on Computer Systems (TOCS)*, vol. 14, no. 1, pp. 108–136, 1996.

[114] Wolfson, O., Jajodia, S., and Huang, Y., "An adaptive data replication algorithm," *ACM Transactions on Database Systems (TODS)*, vol. 22, no. 2, pp. 255–314, 1997.

[115] Wu, Z., Butkiewicz, M., Perkins, D., Katz-Bassett, E., and Madhyastha, H. V., "Spanstore: Cost-effective geo-replicated storage spanning multiple cloud services," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pp. 292–308, ACM, 2013.

[116] Xia, M., Saxena, M., Blaum, M., and Pease, D., "A tale of two erasure codes in hdfs," in *FAST*, pp. 213–226, 2015.

[117] Xu, Q., Siyamwala, H., Ghosh, M., Suri, T., Awasthi, M., Guz, Z., Shayesteh, A., and Balakrishnan, V., "Performance analysis of nvme ssds and their implication on real world databases," in *Proceedings of the 8th ACM International Systems and Storage Conference*, p. 6, ACM, 2015.

[118] Yelp, "Yelp Dataset Challenge," 2016.

[119] Zaharia, M., Das, T., Li, H., Hunter, T., Shenker, S., and Stoica, I., "Discretized streams: Fault-tolerant streaming computation at scale," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pp. 423–438, ACM, 2013.