Analyzing Costs and Optimizations for an Elastic Key-Value Store on Amazon Web Services

David Chiu Travis Hall Farhana Kabir
Washington State University

Apeksha Shetty and Gagan Agrawal
Ohio State University

Cloud computing has emerged to provide virtual, pay-as-you-go computing and storage services over the Internet, where the usage cost directly depends on consumption. One compelling feature in clouds is *elasticity*, where a user can demand and gain access to resources. However, this feature introduces new challenges in developing application and services. In this paper, we focus on the challenges of elastic data management in cloud environments. Particularly, we consider an elastic *key-value* store, which is used to cache intermediate results in a service-oriented system, and accelerate future queries by reusing the stored values. Such a key-value store can clearly benefit from the elasticity offered by clouds, by expanding the cache during query-intensive periods. However, supporting an elastic key-value store involves many challenges, including selecting an appropriate indexing scheme, data migration upon elastic resource provisioning, and optimizations to remove certain overheads in the cloud.

This paper focuses on the design of an elastic key-value store. We consider three ubiquitous methods for indexing: B$^+$-Trees, Extendible Hashing, and Bloom Filters, and we show how these schemes can be modified to exploit elasticity in clouds. We also evaluate various performance aspects associated with the use of these indexing schemes. Furthermore, we have developed a heuristic to request elastic compute resources for expanding the cache such that instance startup overheads are minimized in our scheme. Our evaluation studies show that the index selection depends on various application and system level parameters that we have identified. And while we confirm that B$^+$-Trees, which pervade many of today's key-value systems, would scale well, we show cases when Extendible Hashing would outperform B$^+$-Trees.

We also conduct an analysis which focuses on cost–performance tradeoffs of maintaining the cache. We have compared several Amazon Web Service (AWS cloud) resources as possible cache placements and found that application dependent attributes such as unit-data size, total cache size, and persistence, have far reaching implications on the cost of cache sustenance. Moreover, while instance-based caches expectedly yield higher cost, the performance that they afford may outweigh lower cost options.

Keywords: cloud data management, elasticity, storage costs, indexing

1. INTRODUCTION

Recent developments in Internet scale and other data-intensive applications have generated considerable interest in cloud computing. The cloud facilitates near instant, paid access to infinite storage and computing, in a model that is tantamount to the familiar utilities, e.g., electricity [Armbrust, *et al.* 2009]. With the cloud, conventional cost and risk barriers for initiating large-scale projects, including high upfront costs for building and maintaining a cluster of high-performance machines, are lifted. Today, many prominent cloud providers have been initiated, and among these, Amazon Web Services (AWS) [1] appear to be the most widely adopted platform.

AWS is classified as an Infrastructure as a Service (IaaS), as it allows users complete control over compute and storage resources. Thus, one important facets of AWS is its notion of *elasticity* — the ability for users to instantly scale their resource allocation up and down according to demands. When exploited optimally, the elastic compute aspect lets users avoid over- or under-provisioning of resources [Armbrust, *et al.* 2009], a highly desirable feature for any enterprise. Although the emergence of the

[1] Amazon Web Services, http://aws.amazon.com

cloud clearly offers new opportunities in computing, its effective use for certain types of applications invokes new challenges. For example, there are currently high levels of interest in accelerating large-scale processes (ubiquitous in data-intensive projects) by employing the cloud. However, systems that haphazardly allocate cloud resources without proper planning may not yield speedups and can lead to significant overhead in terms of usage costs.

The emergence of the elastic paradigm has been timely. Consider a scenario where the demand for various service-oriented applications (e.g., popular Web services, scientific workflows) is not always constant, and certain phenomena could lead to an increase in the number of requests, which would likely reduce availability of the service. With access to the elastic cloud, one way to address this issue would be to dynamically allocate resources and replicate the service application over the newly allocated nodes. In general, while elasticity can be beneficial for many applications and use-scenarios, it also imposes significant challenges in the development of applications or services. Some recent efforts have specifically focused on exploiting the elasticity [Chiu et al. 2010; Das et al. 2009; Lim et al. 2010] for various application classes.

We believe that one prominent issue that has not yet received much attention is data management while *leveraging* elasticity. In this paper, we focus on the challenges of managing an elastic *key-value* store in a cloud environment. While several cloud-based key-value storage systems have been developed recently [Chang et al. 2006; Lakshman and Malik 2009; DeCandia et al. 2007; Fitzpatrick 2004], they have not been considered in the presence of elastic computing. We have developed an elastic key-value cache, which is motivated by the challenges of accelerating *service-oriented* computations on the cloud. Our elastic store caches intermediate results of Web services and uses these results for fulfilling future computations. Such a cache can clearly benefit from elasticity — an unexpected query-intensive period can be responded to by a self-managed expansion of resources.

Self-managed elastic scaling, however, comes at the cost of resource usage price. Thus, another dimension of focus is on evaluating the performance and costs associated with a number of caching and storage options offered by AWS.

Our cache is created by a set of cooperating cloud nodes, and its design adheres to its underlying elastic environment. Maintaining a fast and highly available cache in an elastic environment is challenging, and a major contributor to the performance is the data's indexing scheme used on each of the cooperating nodes. The reasons are two-fold: First, an appropriate indexing scheme would clearly allow for fast random accesses to cached data. Second, as our cache incrementally expands to meet load requirements, portions of resident cached data must be *migrated* to newly acquired cloud nodes. The indexing mechanism, whose performance often relies on the application, must then also be conducive to rendering such migrations efficient.

We have developed data migration algorithms to support the use of three popular indexing schemes: B^+-trees [Bayer and McCreight 1970; Comer 1979], Extendible Hashing [Fagin et al. 1979], and Counting Bloom Filters [Fan et al. 2000; Putze et al. 2009]. We focus on these indexing schemes because they pervade existing systems and are heavily documented in literature. Based on our migration algorithms, we compare the performance obtained from these three methods. Besides developing migration algorithms associated with different indexing schemes, we have also optimized the caching scheme so that it would work as unobtrusively as possible in the cloud environment, with the aim to minimizing the idle time (during the node expansion/migration phase).

These indices were evaluated in terms of total time taken for running an experiment and the time taken to migrate a set of data records upon cache expansion. In cases where querying rates are 50 and 255 queries per time step, the overheads of data migration and node allocation vary on average 44.8% and 30.8% of total execution time, respectively. We also applied a simple heuristic to speculate the prelaunching of AWS nodes as a means to reduce overhead during migration periods. Our evaluation studies confirm that B^+-Trees, which pervade many of todays key-value systems, would scale consistently well. Interestingly, we also observe instances when the Extendible Hashing scheme could outperform B^+-Trees. After optimizing two of our indexing schemes, we observed a $4\times$ and $14\times$ reduction in the overhead, respectively.

Besides indexing and migration performance, we also analyze usage costs. Because AWS offers flexibility in provisioning the resources to store data, weighing the tradeoff between performance and usage costs makes for a compelling challenge. In one approach to store data on AWS, the cooperation of machine instances can be allocated (this scheme is as stated earlier). Data can be stored either on disk or in memory (for faster access, but with limited capacity). The costs of maintaining such a cache would be much higher, as users are charged a fixed rate per hour, per instance allocated. This fixed rate is moreover dependent on the requested machine instances' processing power, memory capacity, and bandwidth. Alternatively, AWS's Simple Storage Service (S3) can also be used store data. It could be a much cheaper alternative, as users are charged a fixed rate per GB stored per month. Data are also persisted on S3, but because of this overhead, we might expect some I/O delays. Depending on the application user's requirements, performance may well outweigh costs or vice versa.

In this paper, we make the following contributions:

(1) We describe our implementation of an elastic cooperative cache using three ubiquitous indexing schemes.
(2) We present algorithms to leverage elasticity through autonomous scaling and data migration. A heuristic for minimizing scaling overheads is also discussed.
(3) We provide an in-depth analysis on the effects of indexing schemes and migration time.
(4) A cost-performance evaluation of deploying such a cache over various AWS service options is also provided. We believe this analysis would be useful in the computing community by offering new insights into deploying general applications onto AWS.

The remainder of this paper is organized as follows: in Section 2, we present the background of our cooperative elastic cache design. We also provide background on the available (at the time of writing) AWS service options. In Section 3, we present the integration of the three indexing schemes in the cache. In Section 4, we present the experimental evaluation. A discussion of related works is given in Section 5. Finally, we conclude our work in Section 6.

2. BACKGROUND

In this section we describe the basic architecture of our chosen key-value store: a cooperative cache managed over consistent hashing. We also present background on AWS resource costs and discuss tradeoffs for their utilization.

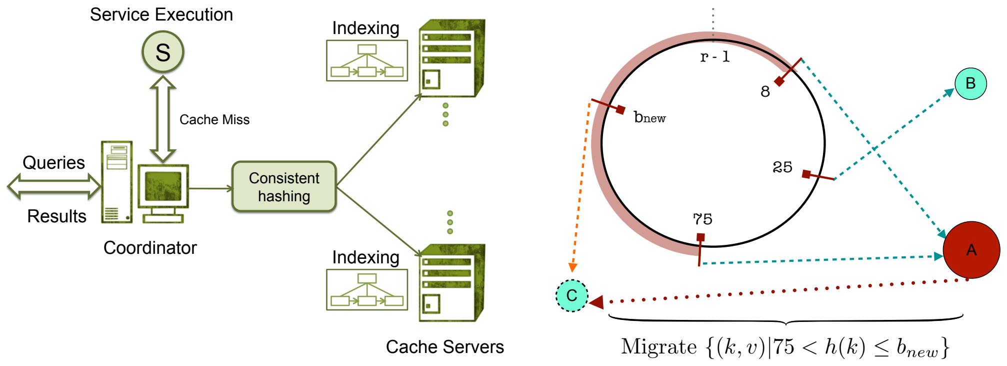
2.1 Elastic Key-Value Cache System

A main goal of the cooperative cache is to provide fast access to the data, and this can be achieved by caching all the data in the main memory. However, because memory is a limited resource, overflowing into disk could cause prohibitively long latencies. Leveraging on the cloud's elasticity, we instead allocate on-demand node instances to handle overflow.

Our system, shown in Figure 1(a), is comprised of a set of cloud nodes, which consist of a coordinator node and cooperating storage nodes, indexed by consistent hashing. Users interface with the coordinator using a simple key-value API. The coordinator is responsible for several mechanisms: upon a given request, it must first determine which server node might contain the data and route the query request to the identified node. During a hit cache on one of the cooperating nodes, it sends the data directly to the user. Conversely, on a miss, the coordinator invokes the service application, S, for execution, then sends the results to both the user and cache.

Initially, it may seem like a simple hashing mechanism would suffice for identifying the node responsible for storing some data (k, v). However, because we are considering our cache under an elastic environment, nodes may incrementally (or decrementally) scale on demand. This dynamism renders many hashing mechanisms useless, as an incredibly large number of key-value pairs would require a rehash upon node membership changes.

To address this problem, consistent hashing [Karger, *et al.* 1997] is being employed on the coordinator due to its ability to quickly adapt to nodes joining and leaving a cooperating system. Consider the example



(a) Client-Server Architecture of the Elastic Cache (b) Consistent Hashing

Figure. 1. System Details

in Figure 1(b), which depicts a consistent hashing system consisting cohort nodes A and B. The hash range is circular from 0 to $r-1$, and consists of several buckets placed randomly or strategically on the ring. Each bucket further stores a pointer to a physical storage node. A simple auxiliary hash function, such as $node = *clockwise\_succ(k \mod r)$ can be used to initially hash a key-value pair (k, v) onto the hash ring. Then a (k, v) pair hashing onto 35, for instance, would follow the ring to the closest *succeeding* bucket in clockwise fashion to node A, referenced by bucket 75.

Aside from interfacing with the user, the coordinator is also responsible for monitoring the cache's status and when appropriate, allocating cloud nodes and scheduling for data splitting and migration from the overflown node to the newly acquired node. In Figure 1(b), we are further showing that node A is overflown, and a new node, C, has been instantiated from the cloud to join the system. Assuming that the range between $(75, 8]$ on the hash ring is crowded with too many keys hashing onto A, we can strategically place a new bucket such that a substantial amount of keys in will be hashed into the new node, C. The new bucket b, for instance, could be placed half way between the two existing buckets: $b_{new} = (75 + \lfloor (r - 75 + 8)/2 \rfloor) \mod r$. To complete the new node membership process, the (k, v) pairs hashing into $(75, b_{new}]$ are finally migrated onto C.

Each cooperating server employs an indexing scheme to facilitate fast searches. The server stores the index together with the cached data in memory to ensure efficient hit times. The specific indexing scheme is application-dependent, however, and we choose to evaluate three among the most widely used indexing schemes. The choice of index on the cache server should impact the node split and migration time, since the support of fast range queries could quickly negotiate the data records that need to be transferred. Moreover, the overall performance of the cache is dependent on the speed of record retrieval and how quickly it can determine a hit or miss. In the following section, we present the background for three ubiquitous index structures: B$^+$-Tree, Extendible Hashing, and Bloom Filters. The performance evaluation of node splitting and data migration in the presence of these popular indexing schemes is a main focus in this paper.

2.2 AWS Usage Costs and Tradeoffs

AWS offers many options for on-demand computing as a part of their Elastic Compute Cloud (EC2) service. EC2 nodes (*instances*) are virtual machines that can launch snapshots of systems, i.e., *images*. These images can be deployed onto various *instance types* (the underlying virtualized architecture) with varying costs depending on the instance type's capabilities.

For example, a Small EC2 Instance (`m1.small`), according to AWS[2] at the time of writing, contains 1.7 GB memory, 1 virtual core (equivalent to a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor), and 160 GB disk storage. AWS also states that the Small Instance has *moderate* network I/O. Another

[2] AWS Instance Types, http://aws.amazon.com/ec2/instance-types

instance type we consider is the Extra Large EC2 instance (`m1.xlarge`), which contains 15 GB memory, 4 virtual cores with 2 EC2 Compute Units each, 1.69 TB disk storage with *high* I/O Performance. Their costs are shown in Table I. We focus on these two contrasting instance types to show a wide spectrum of capabilities, but are careful to note that several other instance types are in fact available in AWS.

AWS Feature	Cost (USD)
S3	\$0.15 per GB-month \$0.15 per GB-out \$0.01 per 1000 in-requests \$0.01 per 10000 out-requests
Small EC2 Instance	\$0.085 per allocated-hour \$0.15 per GB-out
Extra Large EC2 Instance	\$0.68 per allocated-hour \$0.15 per GB-out
EBS	\$0.10 per GB-month \$0.10 per 1 million I/O requests

Table I. Amazon Web Services Costs

Amazon's popular persistent storage framework, Simple Storage Service (S3), provides a key-value store with simple ftp-style API: `put`, `get`, `del`. Typically, the unique keys are represented by a filename, and the values are the data objects, i.e., files. While the objects are limited to 5 GB, the number of objects that can be stored in S3 is unlimited. Aside from the simple API, the S3 architecture has been designed to be highly reliable and available. It is furthermore very inexpensive (see Table I) to use.

Another option for persistent storage is to employ Elastic Block Storage (EBS) in conjunction with EC2 instances. The EBS service is a persistent disk volume that can be mounted directly onto a running EC2 instance. The size of an EBS volume is user defined and limited to 1 TB. Although an EBS volume can only be attached to one instance at any time, an instance can conversely mount multiple EBS volumes. From the viewpoint of the EC2 instance, the EBS volume can be treated simply as a local filesystem.

2.3 Tradeoffs

We now give a brief discussion on the tradeoffs of deploying our cache over the aforementioned cloud resources.

Instance-Memory Option: There are several advantages in supporting our cache over EC2 nodes in terms of flexibility and throughput. Depending on the application, it may be possible to store all cached data directly in memory, which reduces access time. But because small instances contain only 1.7 GB of memory, we may need to dynamically allocate more instances to cooperate in establishing larger capacity. On the other hand, we could also allocate an extra large instance with much more memory capacity to begin with. However, the instance could overfit our cache needs, which would betray cost-effectiveness. Because of these reasons, we would expect a memory-based cache to be the most expensive, but possibly with the highest performance, especially for an abundance of smaller data units.

Instance-Disk Option: In cases where larger amounts of data are expected to be cached, we could store on the instance's disk. Even small EC2 instances provide ample disk space (160 GB), which would save us from having to allocate new instances very frequently for capacity, as we would expect in the previous option. However, disk accesses could be very slow compared to an in-memory cache if request rates are high. Conversely, if the average data size is large, disk access overheads may be amortized over time. We can expect that this disk-based option should be cheaper than the memory-based, with slightly lower performance depending on the average unit-data size.

Persistent Options: The previous two configurations do not account for persisting data. That is, upon node failure, all data is presumed lost even if stored on disk. Moreover, it can be useful to stop and restart a cache, perhaps during peak/non-peak times, to save usage costs.

The simplest persistent method is to directly utilize S3 to store cached data. This avoids any indexing logic from the application developer, as we can subscribe directly to S3's simple API. It is very inexpensive to store data on S3 and more importantly, because S3 is independent from EC2, we further elude instance allocation costs. However, due to S3's reliability and availability guarantees, it implements an architecture which supports replication and consistency, which would likely impact performance. Also, although storage costs are low, the data transfer costs are equivalent to those of EC2 instances, which leads to the expectation that high-throughput environments may not benefit cost-wise from S3.

We noted previously that another persistent method are EBS volumes. One difference between EBS and S3 is that EBS volumes are less accessible. They must first be mounted onto an EC2 instance. But because they are mounted, it alludes to the potential for higher throughput than S3, whose communications is only enabled through high-level SOAP/REST protocols that ride over HTTP. Also in contrast to S3, EBS volumes are not unlimited in storage, and their size must be predefined by users. However, EBS adds a storage and request cost overhead on top of the hourly-based EC2 instance allocation costs.

3. INDEXING BACKGROUND AND ELASTIC INTEGRATION

A distinct indexing service has to be implemented on each node supporting the key-value server. We consider three ubiquitous indexing schemes used in our cache nodes for facilitating key-value storage: B$^+$-Tree [Bayer and McCreight 1970; Comer 1979], Extendible Hashing [Ullman et al. 2001], and Counting Bloom Filters [Fan et al. 2000; Putze et al. 2009]. To facilitate cache elasticity, when a node overflows, we must migrate a subset of its records to another node, which may be preexisting or newly allocated. As the three schemes we have selected are inherently dissimilar in structure and methods of operation, they make compelling candidates for extension to an elastic environment and performance evaluation.

In the rest of this section, we initially present the background on each indexing mechanism, and then describe how we have implemented the migration mechanism over the three indexing schemes upon a node overflow.

3.1 B$^+$-Trees

B-trees and their variant B$^+$-trees are used extensively in many of today's systems. The B$^+$-tree is a multilevel indexing scheme, which automatically adjusts the number of levels depending upon the file size. In terms of access, it is a balanced data structure, where all paths from the root to any leaf have the same length (akin to binary trees, with approximately $\log_2 n$ depth). The leaf nodes of the B$^+$-tree store the records in ascending order from left to right, and all the leaf nodes are linked to the next node, which was specifically designed to accelerate range queries [Elmasri and Navathe 2003; Ullman et al. 2001].

The basic structure of the B$^+$-tree is as follows. Each node contains a set of n keys and $n-1$ child pointers. K_i are the keys and P_i is the pointer to a tree node and Pr_j is the pointer to a record's physical location. All keys in the left branch of the key K are less than or equal to the K and all keys in the right branch are greater than the K. While searching, we follow the appropriate branches based upon the comparison of the key with the entries in the tree. In a process tantamount to searching a binary tree, we start from the root and follow the left path if the key is less than or equal to the root, else we follow the right path, recursively.

Due to its support for fast range queries, we would expect the B$^+$-tree integration to be particularly auspicious for our consistent hashing-based cache. Such fast accesses to ranges of data should facilitate faster data migration upon node membership. Data migration, in this case, is comprised of deletions of keys in the range from the smallest to half of the overflow node. Since B$^+$-Tree contains the keys sorted in ascending order from left to right, on the leaf level, it is efficient to identify all keys that lie in the range and delete them, as well as their associated data, from the memory of the overflown node. To migrate data from a B$^+$-Tree, both k_{start} and k_{end} are the inputs to the migration procedure, and they denote the limits of the range of keys. We first search to find the leaf containing k_{start}. Because all leaf nodes are linked, we can sweep all leafs until k_{end} has been reached or passed.

3.2 Extendible Hashing

Hash tables are another commonly used form of indexing, which excels at offering $O(1)$ exact-match key search times. The tradeoff, however, is that hash tables are not well-suited to handle range queries.

In most hash implementations, we assume that there exists a hash function $h(k) \in [0, B-1]$, where B is the total number of buckets in hash line. Each bucket contains a set of records stored in memory or a set of pointers to records stored in secondary memory. A hash function should ideally hash each key to a distinct bucket, but this is seldom possible because the key range is often much larger than B. Therefore, the buckets typically allow for storing a set of records, but even so, they can still overflow. To avoid this from happening, hash tables implement some form of collision reconciliation technique. A simple technique is to have overflow chains, where overflown records are stored in a linked list attached to the bucket. The performance of this technique decreases linearly as the load factor (ratio of the number of records stored to the size of bucket) increases.

We can avoid this performance hit by using dynamic hashing [Enbody and Du 1988], where the number of buckets, B, can vary, unlike the aforementioned static hashing. In dynamic schemes, B is increased whenever necessary. We have implemented a form of dynamic hashing, namely, Extendible Hashing [Fagin et al. 1979], which introduces a concept known as a *directory*— an array of pointers to the hash buckets. The buckets themselves contain an additional array of pointers to the records' physical location.

Initially, each directory contains one bucket, but this is allowed to grow whenever required. In Extendible Hashing, the length of the directory is always a power of 2, which translates to doubling the size of the directory size in each growing phase. However, because multiple pointers can point to the same physical bucket, the actual number of buckets can be $\leq$ to the size of the directory. A hash function $h(k)$ computes a binary sequence for each record based on the search key, k, and the first i least significant bits, are used to determine the bucket to which the record belongs. Thus, when a directory contains 2^i number of pointers to buckets the actual number of buckets is $\leq 2^i$.

Searching for a key in an extendible hash table is a two-phase process. First, the least significant i bits from the hash value of the key and determine the bucket it belongs to. Finally, a linear scan within the identified bucket is required to return its position, if it is found. The searching time would expectedly increase as the number of records per bucket increases. Conversely, a higher number of records per bucket would lead to fewer splits and a smaller directory. As we alluded to earlier, while Extendible Hashing offers constant-time exact match queries, range queries will expectedly suffer because the hash function disrupts k's original locality.

To support migration, we implemented Extendible Hashing such that we could dynamically specify the number of records per bucket. Because Extendible Hashtables do not store the records in any particular order, we linearly scan through each bucket and delete keys that lie within the migration range.

The migration procedure inputs k_{start} and k_{end} again to denote the range of keys to be migrated. Initially, we traverse through all directories, denoted by D_x, and for each directory, we follow the pointer, y, to its corresponding bucket. Any key, k, which lies in the range, $[k_{start}, k_{end}]$, is appended along with its data object, v, to $keys$. Finally, $keys$ is returned when all records have been scanned through.

3.3 Bloom Filter

Bloom Filters [Bonomi et al. 2006] are probabilistic data structures used to quickly determine the membership of a record in a set. It consists of a bit array of m bits and a set of j hash functions which hashes each record to j different values. Generally, $m \gg j$, which reduces the probability of the hash functions setting the same bit for a record. Though Bloom Filters are vulnerable to false positives, false negatives are not possible.

Insertions into a Bloom Filter are simple: apply the hash functions to the key and set the corresponding bits. Similarly, we can determine whether a record is present in the set by applying each of the j hash functions to the data item and verifying whether all the corresponding bits are set. If all the corresponding bits are not set, then the data item is not present. However, because false positives are possible, even if all the corresponding bits are set, the record may still not be present, so a scan is required after such a pseudo-hit. Fortunately, the false positive rate has a bound $f = (1 - e^{(-jN/m)})^j$, where j is the number of hash functions, m is the length of the bit array, and N is the number of set bits. Clearly, the false positive rate increases as the number of inserts increases, but choosing a relatively large m and independent hash functions can render f negligible [Kirsch and Mitzenmacher 2008].

Because the same bit could have been set for multiple records, deletion in the traditional Bloom Filter is not possible. Indeed, modifying the bit array could lead to false negatives which are prohibitive. To support deletion, we implemented a variant, Counting Bloom Filters [Fan et al. 2000; Putze et al. 2009]. Each bit in the bit array is associated with a 4-bit counter, which keeps track of the number of records that set the bit, and enables the delete operation.

These structures are quite useful for applications requiring fast tests of record existence (especially for testing non-existence). To search for a record with key k, we first apply the j hash functions to k. Secondly, we AND all bits from the bit array corresponding to the locations $h_i(k)|i = (0, \ldots, j)$. If this result is 1, then the record may be present, and a scan is initiated to retrieve the record. Otherwise, the record is non-existent. Because the linear scan may be required for a hit, it invokes a costly overhead in the case of false positives, but as we had mentioned previously, low false positive rates can be ensured by having a large m and independent hash functions.

The implementation of migration for CBF is also trivial. k_{start} and k_{end} are again the inputs. We start from the minimum threshold and increment till the maximum threshold and search for each key in the range and delete the keys that are present. This makes migration time linear to the amount of keys within $[k_{start}, k_{end}]$, albeit that check for non-existence is fast and guaranteed (no false negatives).

3.4 Elastic Cache Support

We have implemented our system on the Amazon Elastic Compute Cloud (EC2), and will describe our system under its context. EC2 supports Infrastructure-as-a-Service (IaaS), allowing users to allocate nodes on demand. EC2 nodes (*instances*) are virtual machines that can launch snapshots of systems, i.e., *images*. Furthermore, these images can be deployed onto various *instance types* (the underlying virtualized architecture) with varying costs depending on the instance type's capabilities. As a baseline, we have chosen to only employ EC2's `m1.small` instances in our implementation, although it should be noted that instances bearing much greater (or lesser) capabilities are also available.

During system execution, records may be continuously inserted into the cache nodes, and an overflow on any of the nodes can invoke incremental scaling. Recalling from the example given previously in Figure 1(b), this process involves starting a new EC2 node and migrating a subset of the data from the overflown node to the newly allocated node. We can expect two types of overhead when scaling nodes. We refer to the first overhead as *Instance Allocation Time*, which is observed between the time that the command is issued to allocate an EC2 instance to the time the instance is running. We have observed this overhead to take on the order of tens of seconds to several minutes depending on Amazon's load for an `m1.small` instance. The second overhead is *Data Migration Time*, which is invoked after the new EC2 instance has been allocated. It initially identifies the range of data to be migrated from the overflown node, followed by transferring the identified key-value pairs to the new node (migration). While the system is undergoing scaling, it can continue to answer queries from nodes that are not involved in the migration process. Conversely, to ensure consistency, the system waits until after scaling to update.

The main concern in this article is to evaluate how the different indexing schemes (B$^+$-Trees, extendible hashing, and bloom filters) that are used on each EC2 node would affect Data Migration Time. A secondary concern is addressing Instance Allocation Time, which typically dominates Data Migration Time. To this end, we implement a simple speculative heuristic for pre-launching instances when a key-value record threshold has been met. A background thread initiates the pre-launching and eagerly migrates data from the fullest node once the new instance is booted.

Our speculative threshold T is based on the following observations. If the request rate is high, then T should be lowered, as the nodes are likely to fill up faster, and vice versa. Let n denote the node where some (k, v) pair is to be inserted. We use the following to estimate n's threshold:

$$T = c(n)/2 + \delta_H \times (||N|| - R/\delta_L)$$

where δ_L and δ_H are constants representing the lowest and highest *expected* querying rates respectively. R is the current request rate, $c(n)$ is capacity on node n, and $||N||$ is the total number of nodes in our cooperative cache. As the number of nodes, $||N||$, increases the threshold should also increase so as to delay the allocation of new nodes. R/δ_L is used to normalize the current rate, R.

For instance, consider a configuration where $c(n) = 5000$, $\delta_L = 50$, $\delta_H = 250$, and $R = 100$. Then for a cooperative system containing 1, 2, 3, and 4 nodes, the respective thresholds would be 2250, 2500, 2750, and 3000. Hence, T increases linearly by δ_H for each node added in this scenario. Certainly, more robust models can be employed here for speculation, but it is beyond the scope of this work. We use this threshold in our key-value insertion algorithm, shown in Algorithm 1.

In Algorithm 1, k and v are the inputs to the algorithm and they denote the key and value object, respectively. On Lines 1-3, the statically declared inverse hash map, $NodeMap[...]$, the ordered list of buckets B in our consistent hash table, and the auxiliary consistent hash function, h', are brought into

Algorithm 1 Insert(k, v, δ_L, δ_H)

```
1: static $NodeMap[\ldots]$
2: static $B = (\ldots)$
3: static $h' : K \rightarrow [0, r)$
4: $n \leftarrow NodeMap[h'(k)]$
5: $T = c(n)/2 + \delta_H \times (||N|| - R/\delta_L)$
6: if $T < \lceil n \rceil \times 0.1$ then
7:     $T \leftarrow \lceil n \rceil \times 0.1$    ▷ to avoid extremely low values of threshold
8: end if
9: if $T > (\lceil n \rceil \times 0.75)$ then
10:     $T \leftarrow \lceil n \rceil \times 0.95$    ▷ to delay allocation of new nodes
11: end if
12: if $||n|| + sizeof(v) < T$ then
13:     $n$.put($k, v$)   ▷ insert directly on node $n$
14: else if $||n|| + sizeof(v) > T$ then
15:     ▷ Launch threads $t1, t2$ which would execute the Lines 16 - 22
16:     ▷ find fullest bucket referencing $n$
17:     $b_{max} \leftarrow \underset{b_i \in B}{\text{argmax}} ||b_i|| \wedge NodeMap[b_i] = n$
18:     $k^{\mu} \leftarrow \mu(b_{max})$
19:     $n_{dest} \leftarrow n$.migrate($min(b_{max})$, $k^{\mu}$)
20:     ▷ update structures
21:     $B \leftarrow (b_1, \ldots, b_i, h'(k^{\mu}), b_{i+1}, \ldots, b_p) \mid b_i < h'(k^{\mu}) < b_{i+1}$
22:     $NodeMap[h'(k^{\mu}))] \leftarrow n_{dest}$
23: else
24:     ▷ $n$ overflows
25:     ▷ Launch Thread $t3$ if $t1$ and $t2$ are not taking care of $n$ and execute Lines 16-22
26: end if
```

scope. The $NodeMap[b]$ returns the node n pointed by bucket b.

Lines 5-11 are for handling speculative threshold selection. The idle time caused by migration can be reduced to zero if a good threshold is selected, but selecting such a threshold is tricky, as a low value could lead to higher number of instances being launched. This, in turn, would not be optimal, as running extra instances would be costly. Using a higher threshold could lead to higher idle times because the node allocation is being initiated too late. Therefore, the selection of a threshold is a tradeoff between lowering idle times and optimizing the number of instances being initialized.

Returning to the algorithm, Line 5 of Algorithm 1 calculates this threshold. In Lines 6-11, we check for two conditions; the first condition increases the threshold initially to delay launching of instances, whereas the second condition increases the threshold after a certain instant, so that new instances would not be initialized unnecessarily, which helps reduce cost.

In order to reduce idle times we need to parallelize the execution of various parts of the code. It was empirically observed that two instances generally reached the node capacity at the same time, since our experiments issue random workloads. We first introduce two threads, $t1$ and $t2$, that would initialize a new node if required (Lines 15-22). If a node reaches capacity, $\lceil n \rceil$, then a third thread, $t3$, would handle the overflown node if neither of the two threads were handling it already (Lines 24-25). Thus, at any point there could be a maximum of 4 threads running (including the main thread).

The three threads all perform the operation mentioned on Lines 17-22. On line 17, we identify the fullest bucket b_{max} referencing n. On Lines 18-19, we migrate half the keys (from the minimum to the median, k^{μ}) from b_{max}. The migrate method returns a reference to the destination node, n_{dest}, which may be preexisting or newly allocated. Finally on Lines 20-22, the statically declared structures, $NodeMap$ and B, are updated.

Resuming discussion of the cache servers, each cohort node consists of the index, *put()*, *delete()*, *migrate()*, and *search()* methods. At any instance of time, the consistency of the index needs to be maintained, which requires that the methods modifying the structure of the index be synchronized. Hence, *put()*, *migrate()*, and *delete()* to acquire a lock on the cache server instance at the start of the method and release it at the end. Since *search()* is read-only, this method does not acquire any locks.

4. EXPERIMENTAL RESULTS

In this Section, we evaluate the performance of our elastic key-value cache on the Amazon EC2 Cloud. We also compare the performance of the three indexing schemes.

4.1 Experimental Configuration

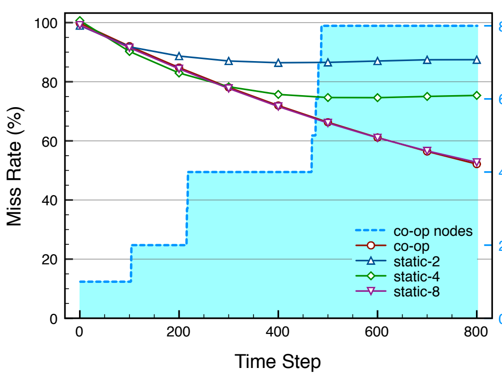
In all experiments, we used *Small EC2 Instances* from the Amazon Cloud (1.7 GB of memory, 1 virtual EC2 core - equivalent to 1.0 -1.2 GHz 2007 Opteron or 2007 Xeon Processor on a 32 bit platform). Each instance was loaded with an Ubuntu Linux Image and a cache server, which contains the indexing logic.

We ran a real service application, *Shoreline Extraction*, a geodetic Web service. Given the location, L, and the time of interest, T, the service retrieves a data file representing the terrain at location L, then interpolates this file with a water level reading, measured at time T. The queries are submitted to the coordinator node, and it tries to locate the results on the cache nodes based on the inputs from the query. If the result is present in the cache, i.e., it is precomputed via some previous request, it is retrieved and returned directly to the caller. In the case of a miss, the shoreline extraction service is invoked. The queries are submitted randomly over 64K distinct possibilities for each service request. Because we know the key range in advance, we have also set r, the consistent hashing modulo function, to 64K.

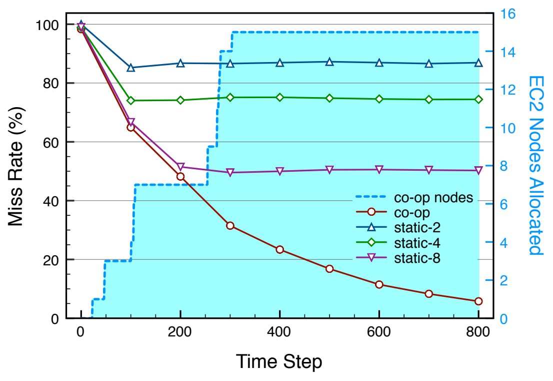
We tested our system under varying query rates. We varied the rate between 50 queries/time step and 255 queries/time step. At each time step, we recorded the average (in seconds) and the number of hits and misses. In order to show the cache's elastic behavior over execution, we submitted R queries per time step. Each time step is a logical iteration, and does not reflect real time. Note that the granularity of a time step in practice, e.g., t seconds, minutes, or hours, does not affect the overall hit/miss rates of the cache. At each time step, we observed and recorded the average service execution time (in number of seconds real time), the number of times a query reuses a cached record (i.e., hits), and the number of cache misses. We analyze the cache behavior in real-time in section 4.3.

4.2 Evaluation of Elastic Cache vs. Static Cache

We compared our cooperative elastic cache (`co-op`) against static versions of our cache. The static caches are fixed at 2, 4, and 8 nodes (`static-2` , `static-4`, and `static-8` respectively), and cannot expand. Therefore, the static versions implement LRU (Least Recently Used) replacement policy to prevent overflow. In this experiment, only the B$^+$-Tree is being considered.



(a) Querying Rate = 50 queries/timestep



(b) Querying Rate = 255 queries/timestep

Figure. 2. Miss Rate

Figure 2(a) and Figure 2(b) represents the results for miss rates for experiments conducted with 50 queries/timestep and 255 queries/timestep respectively. The X-axis represents the time steps elapsed in our experiment (recall from above that a time step is not real time, but a simulated time in which R query requests are sent). The right-hand Y-axis represents the EC2 nodes allocated throughout the experiment.

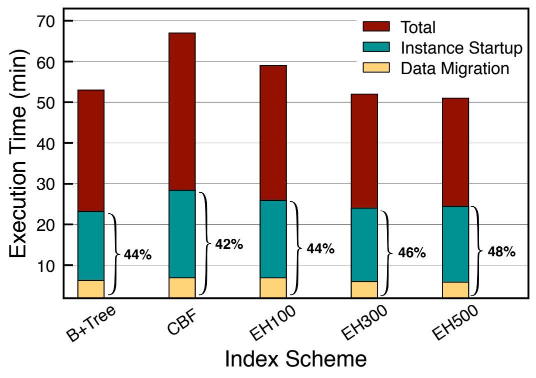
As the experiment proceeds, the miss rates decrease linearly, since requests are submitted at random. `static-2` and `static-4` appear to converge very early in the experiments, while `static-8` seems to perform as well as our system in Figure 2(a). It can also be observed that our system uses a maximum of 8 nodes at the end of the execution, which explains its similar performance to `static-8`. The early convergence of `static-8` can finally be observed in Figure 2(b), where the query rate is increased to 255 queries/time step. Our system can attain near-zero miss rates toward the end of the experiment, however, at the expense of 15 final nodes.

One aspect that is not being shown here is the time taken to split and migrate data when a new node is allocated. This may be a costly overhead that varies depending on the index that is being used on the cohort cache server. We show an evaluation of the impact of indexing schemes, and the migration overheads, next.

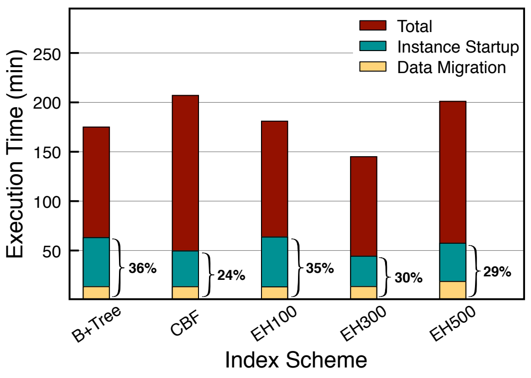
4.3 Cache Server Index Comparison

The three indexing schemes we compared are: B$^+$-Trees (`B+Tree`), and Counting Bloom Filter (`CBF`), and Extendible Hashing with three bucket size configurations: (`EH100`, `EH300`, `EH500`). We evaluate the suitability of these algorithms for our system based on the total time taken to run the experiment, as well as its breakdown on time taken for migration and instance start up. Each experiment was run three times, and we took the average.

In these experiments, we are showing the total time taken to interact with two querying models: 50 queries/timestep and 255 queries/timestep *without* speculative migration. We will show the optimization observed with speculation in a later subsection.



(a) Query rate = 50 Queries/timestep

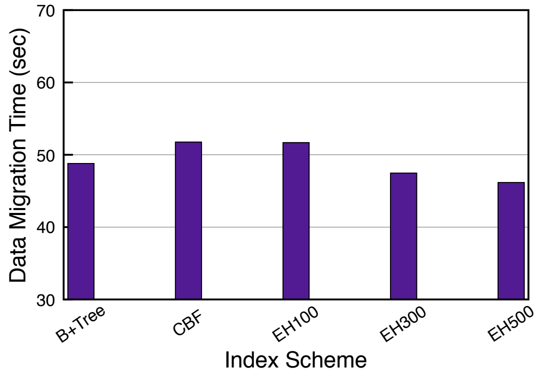


(b) Query rate = 255 Queries/timestep

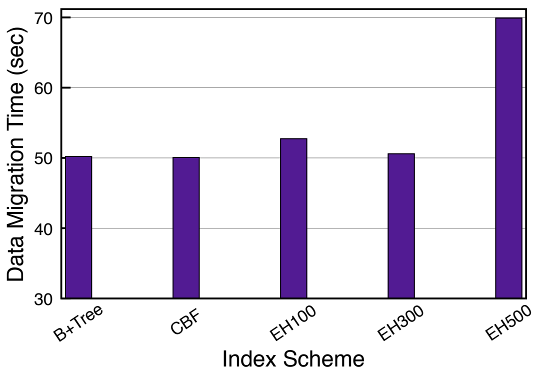
Figure. 3. Execution Time of Indexing Schemes

Figure 3(a) shows the results obtained by running the experiment with 50 queries/timestep, which is in contrast to Figure 3(b) (255 queries/timestep). The total time taken to run the experiments varied between 50 to 70 minutes. As we had alluded to earlier, we observe that instance startup times can vary quite a bit. Combined with data migration, these overheads account for nearly half of the total execution time for 50 queries/timestep. As expected, the migration time for `CBF` performs the worst, but is easily dominated by instance startup overhead. Although these overheads expectedly amortize as we increase the request rate to 255 queries/timestep, they are still quite significant.

In Figure 3(a), it can be observed that `EH500` outperforms the rest, while `CBF` is clearly the worst option. We can observe that, the system parameter (records per bucket) greatly impacts the performance of Extendible Hashing. The B$^+$-tree performs well irrespective of the parameters. This can also be verified by Figure 3(b), where EH300 now performs the best and CBF again performs the worst. However, the performance of EH500 records per bucket has degraded considerably whereas the performance of B$^+$-Tree scales quite well even when the query intensity is increased. Thus, we posit that the performance of Extendible Hashing also depends on the system parameter: querying rate.



(a) Query rate = 50 Queries/timestep, 7× Migration



(b) Query rate = 255 Queries/timestep, 15× Migration

Figure. 4. Migration Time of Indexing Schemes

Focusing now on Figures 4(a) and 4(b), we have averaged over three runs the time taken per data migration, i.e., the time taken to identify and transfer the range of data to be moved from the overflown node to the new node. In total, the 50 queries/timestep rate invoked migration 7 times and the 255 queries/timestep experiment invoked migration 15 times. The results being shown are compelling. We do not observe much variation between the two graphs for B$^+$-Trees, which suggests that it scales well to high requests rates.

The interesting note is that, on average, CBF actually perform better as the number of migrations increases. This is due to the fact that the number of records to be migrated decrease over time across all indexing schemes, as an effect of consistent hashing. The reason for this is because, over time, the ranges on the consistent hashing ring will generally decrease. Recalling that migration on CBF is slightly super-linear due to scanning for false-positives, as the data range decreases over time, false-positives also decrease, rendering this algorithm closer to linear time. As we can see in Figure 4(b), CBF is eventually equivalent to the B$^+$-Trees' linear-time migration algorithm.

The same logic can be applied to explain the degradation in performance for the EH$\star$ schemes, which all perform worse as the number of migrations increase from 7 to 15. Using the worst case, EH500, to exemplify, the average migration times are quite low when we have fewer migrations because there are smaller numbers of buckets to traverse linearly. As the number of migrations increases to 15 times, this would imply that a greater quantity of data is being stored in the index, which translates to not only a larger directory size (which grows exponentially), but also potentially many buckets with data in the range scattered within each. In other words, we begin to observe the inherent problem of hashing-based solutions for handling ranges. The tradeoff is that its $O(1)$ lookup facilitates fast hit/miss indication, which leads to better overall performance for high querying rates.

We summarize by observing that B$^+$-Tree would scale well irrespective of the system parameters, and EH$\star$, with their $O(1)$ exact-match searches, could actually outperform B$^+$-Trees if the parameters are chosen appropriately. However, if the cache system is volatile, and migration is invoked often, EH$\star$ indexing schemes will yield increasingly poor migration performance. CBF should typically be avoided as an indexing scheme for elastic key-value store, but it may scale well for applications relying on space-efficient structures. We also made the interesting observation that CBF migration overheads become better over time.

4.4 Optimizing Instance Allocation Overhead Results

Back in Section 3.4, we described an approach to minimize the idle time when allocating a new EC2 instance. Instances are pre-launched when a threshold T is met and the migration overlaps with normal program execution. We used the following parameter settings for the Insert algorithm: $c(n) = 5000$, $\delta_L = 50$, and $\delta_H = 250$. Specifically, $c(n)$ states that each node can contain 5000 key-value pairs, $\delta_L = 50$ states that 50 requests per second is the lowest expected request rate, and δ_H specifies that 250 requests per second is the highest expected.

Again, we executed a total of three runs and reported the average overhead and total times. We ran these experiments using $R = 50$, i.e., the true request rate being 50 requests per second. We show the results for `EH300` and `CBF` are summarized in Figure 5. On the left side of the figure, we show the original results for `EH300` and `CBF`. The right side of the figure depicts the results after we apply speculative pre-launching.

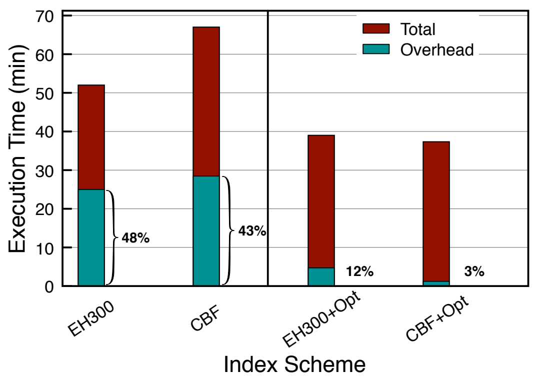


Figure. 5. Instance Allocation Overhead for Query Rate = 50 queries/timestep.

After optimization, we managed to improve the scaling overhead by $4\times$ and $14\times$ respectively in `EH300` and `CBF`. Even with the improvements, there are clearly opportunities for further refinement here, and we propose to develop more robust heuristics in future works.

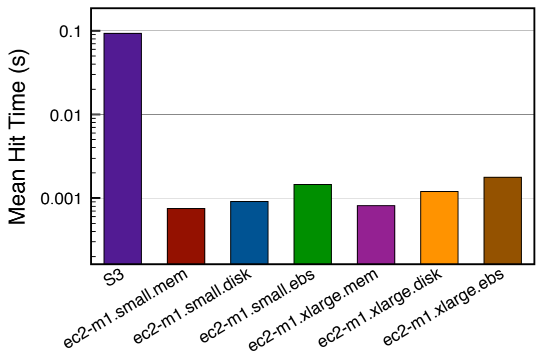
4.5 Performance-Cost Analysis

To analyze the performance and cost tradeoffs, we use only the B$^+$-Tree version and we run experiments over the following configurations:

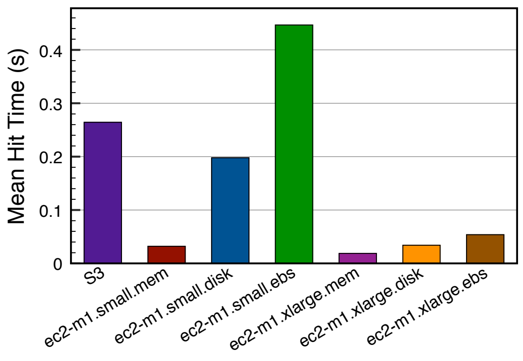
(1) `S3`: Data stored as files directly onto the S3 storage service (persistent).
(2) `ec2-m1.small-mem`: Data stored in memory on Small EC2 instance (volatile, moderate I/O).
(3) `ec2-m1.small-disk`: Data stored as files on disk on Small EC2 instance (volatile, moderate I/O).
(4) `ec2-m1.small-ebs`: Data stored as files on a mounted Elastic Block Store volume on small EC2 instance (persistent, moderate I/O).
(5) `ec2-m1.xlarge-mem`: Data stored in memory on Extra Large EC2 instance (volatile, high I/O).
(6) `ec2-m1.xlarge-disk`: Data stored as files on disk on Extra Large EC2 instance (volatile, high I/O).
(7) `ec2-m1.xlarge-ebs`: Data stored as files on a mounted Elastic Block Store volume on Extra Large EC2 instance (persistent, high I/O).

In both `m1.small` (32-bit) and `m1.xlarge` (64-bit) systems, we employ the Ubuntu Linux 9.10 Server image provided by Alestic.[3]

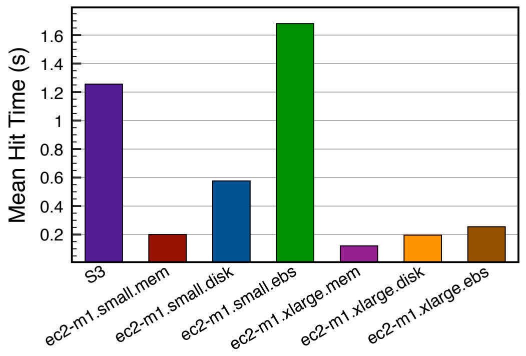
[3]Alestic, http://alestic.com/



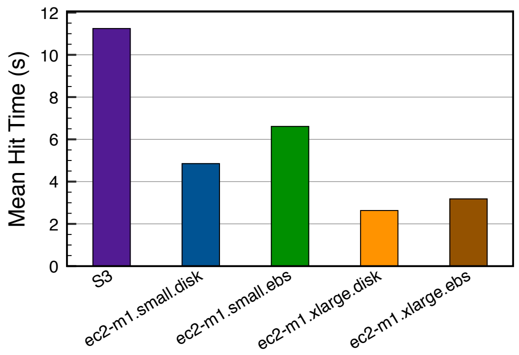
(a) Data Size = 1 KB



(b) Data Size = 1 MB



(c) Data Size = 5 MB



(d) Data Size = 50 MB

Figure. 6. Mean Cache Hit + Retrieval Time

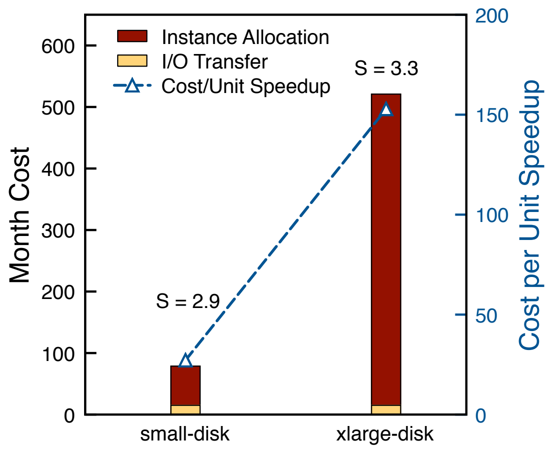
To analyze cache and storage performance, which can be affected by memory size, disk speed, network bandwidth, etc., we varied the sizes of cached files: 1 KB, 1 MB, 5 MB, 50 MB. One such file is output from one execution, and over time, we would need to store a series of such files in our cache. These sizes allow for scenarios from cases where all cached data can fit into memory (e.g., 1 KB, 1 MB) to cases where in-core containment would be infeasible (e.g., 50 MB), coercing the need for disk or S3 storage. The larger data will also amortize network latency and overheads, which increases throughput.

We submitted queries to hot caches, which guarantees a hit on every query, and we are reporting the mean time in seconds to search the cache and retrieve the relevant file *on one cache node*. Figures 6(a), 6(b), 6(c), and 6(d) show the average *hit* times for each cache configuration.

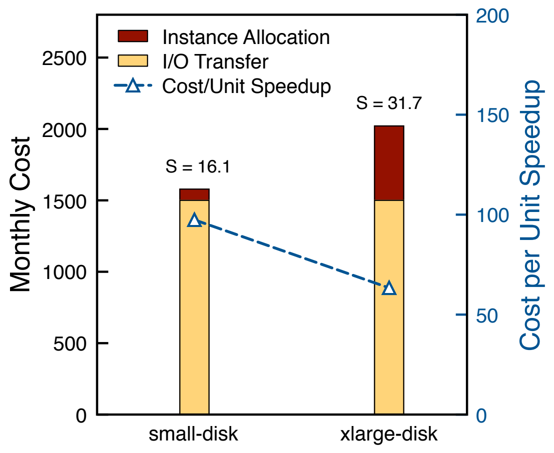
Figure 6(a) shows that using `S3` for small files eventually exhibits slowdowns by 2 orders of magnitude. In the other figures, we observe the justification for using memory-bound configurations, as they exhibit for the lowest mean hit times. Also, we observe consistent slowdowns for `ec2-m1.small-disk` and `ec2-m1.small-ebs` below `S3` in the 1 MB and 5 MB cases. Finally, using the results from Figure 6(d), we can conclude that these results again support our belief that disk-bound configurations of the small instance types should be avoided for such mid-sized data files due to disk access latency. Similarly for larger files, `S3` should be avoided in favor of `ec2-m1.xlarge-ebs` if persistence is desirable. We have also ascertained from these experiments that the *high* I/O that is promised by the extra large instances contributes significantly to the performance of our cache.

We now present an analysis on cost for the instance configurations being considered. The costs of the AWS features evaluated in our experiments were summarized earlier in Table I. While in-cloud network I/O is currently free, in practice, we cannot assume that all users will be able to compute within the same cloud. We thus consider the worst case where all data is transferred *outside* of the AWS network, but in practice, the costs might be much lower depending on the system configuration. The settings from the previous set of experiments are repeated, so an average unit data size of 50 MB will yield a total cache size of 25 GB of cloud storage (assuming there are 500 distinct request keys). We are furthermore assuming a fixed rate of $R = 2000$ requests per month. To show the impact of I/O, we have also extrapolated $R = 200000$ requests. Due to space constraints, we only show the analysis for 50 MB unit sizes. A complete analysis of all data sizes can be found in [Chiu 2010].

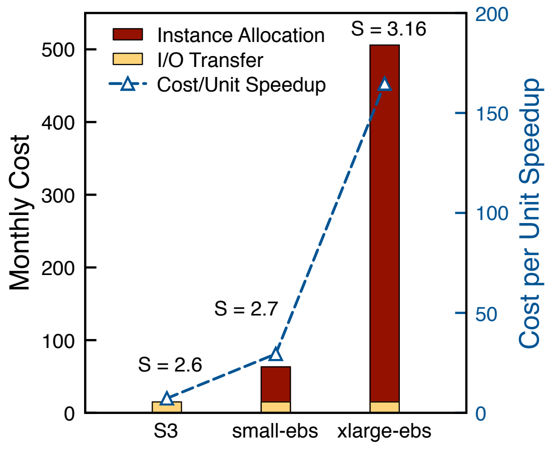
In Figures 7(a) and 7(b), we show the monthly costs of sustaining volatile caches on `m1.small-disk` and `m1.xlarge-disk` instances for $R = 2000$ and $R = 200000$ requests respectively. To qualify the monthly costs, we also display the speedups achieved (shown as S in the figures) for the 2000th and 200000th request. To hold the large cache data in its entirety, we only use disk-based instance types here. Here, the total cost can be computed as $C = C_{Alloc} + C_{IO}$ where $C_{Alloc} = h \times k \times c_t$ denotes allocation cost, impacted by the number of hours h needed to allocate k nodes for instance type t. $C_{IO} = R \times d \times c_{io}$ accounts for transfer costs, where R transfers were made per month, each involving d GB of data per transfer, multiplied by the cost to transfer per GB, c_{io}.



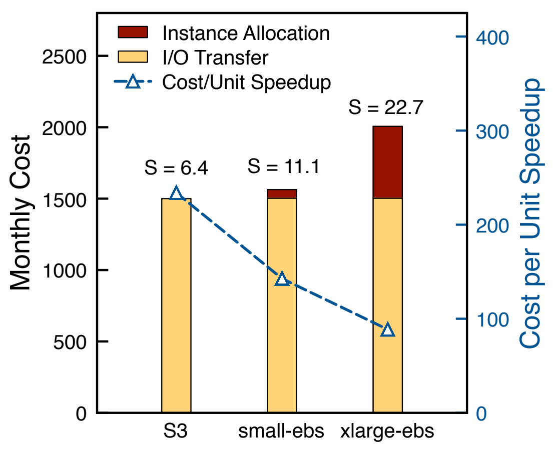
(a) Volatile, $R = 2000$



(b) Volatile, $R = 200000$



(c) Persistent, $R = 2000$



(d) Persistent, $R = 200000$

Figure. 7. Cache Cost Analysis: 50 MB Unit Size (25 GB Total Cache Size)

Let us first consider Figure 7(a). On the left axis, the monthly costs are shown. Clearly, the total costs are dominated by instance allocation time due to small amounts of I/O. The cost per unit speedup (dotted line shown against the right axis), suggests that `m1.xlarge` instances may not be cost effective for applications with low request rates. In Figure 7(b), we show the same configuration with the higher amount of requests, $R = 200000$. Due to the large amount of requests and 50 MB per transfer, the costs now become dominated by I/O. As can be seen, the significant gap between the speedups obtained by `m1.small` and `m1.xlarge` are now revealed. The costs per speedup are also showing the reverse trend, suggesting that `m1.xlarge` should be used when R is expected to be large.

Next, we focus on the persistent storage options, shown in Figures 7(c) and 7(d). The equations for computing total costs are changed as follows. For `ebs`-based instances, the costs can be computed the same as before, $C_{EBS} = C_{Alloc} + C_{Store} + C_{Req} + C_{IO}$ with the addition of C_{Store} and C_{Req}. C_{Store} is the monthly cost to store the data, and C_{Req} is the cost per 10000 out-bound requests. C_{Alloc} and C_{IO} remain the same as before. For `S3`, the cost is $C_{S3} = C_{Store} + C_{Req} + C_{IO}$. Because C_{Req} and C_{Store} are typically so low, they are not shown in the figures. For instance, C_{Store} is only \$3.75 per month for storing 25 GB on `S3`. This number drops to \$2.50 per month for `ebs` storage. C_{Req} is almost zero cost for our amount of requests per month.

Initially, in Figure 7(c) we can clearly observe that `S3`, devoid of costly instance allocation, is by far the most inexpensive option. Furthermore, when R is small, its performance is comparable to the considerably more costly `ebs` instance types. This results in an extremely high payoff of having low cost per unit speedup. However, for a large request rate R this payoff is amortized due to the I/O transfer costs. Moreover, the speedups observed by `ebs` also overcome `S3` significantly when given the high number of requests.

While this results in lower cost per unit speedup on `ebs`-backed instances, the instance `ebs` allocation costs are still quite high in practice (e.g., `xlarge-ebs` would still cost \$500 more per month over `S3`). Furthermore, as we mentioned previously, not all applications will require out-of-cloud transfers, which gives more rise towards using `S3` as an inexpensive and viable storage option.

4.6 Discussion

The experiments demonstrate some interesting tradeoffs between cost and performance, the requirement for persistence, and the average unit-data size. We summarize these options below, given parameters d = average unit-data size, T = total cache size, and R cache requests per month. Although we could not present the cost evaluation for all unit data sizes in this article (see [Chiu 2010]), we offer a summary of our observations here.

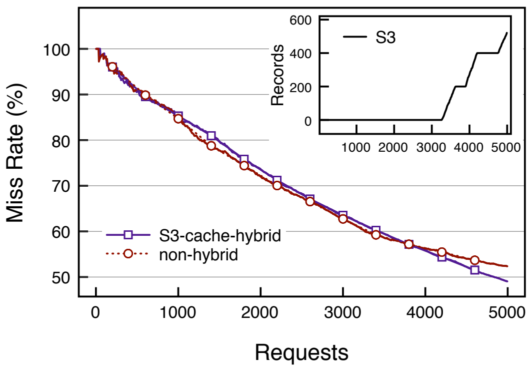
For smaller data sizes, i.e., $d \leq 5$ MB, and small total cache sizes $T < 2$ GB, we posit that because of its affordability, `S3` offers the best cost tradeoff when R is small, even for supporting volatile caches. `m1.small.mem` and `m1.small.disk` also offer very good cost-performance regardless of the request rate, R. This is due to the fact that the entire cache can be stored in memory, together with the low cost of `m1.small` allocation. Even if the total cache size, T, is much larger than 2 GB, then depending on costs, it may still even make sense to allocate multiple small instances and still store everything in memory, rather than using one small instance's disk – we showed that, if request rate R is high, and the unit-size, d, is small, the speedup for `m1.small.disk` is eventually capped two orders of magnitude below the memory-bound option. If $d \geq 50$ MB, we believe it would be wise to consider `m1.xlarge`. While it could still make sense to use a single small instance's disk if R is low, we observed that performance is lost quickly as R increases, due to `m1.small`'s lower-end I/O.

If data persistence is necessary, `S3` is by far the most cost-effective option in most cases. However, it also comes at the cost of lower throughput, and thus `S3` would be viable for systems with less expectations for high amounts of requests. The cost analysis also showed that storage costs are almost negligible for `S3` and `EBS` if request rates are high. If performance is an issue, it would be prudent to consider `m1.small-ebs` and `m1.xlarge-ebs` for smaller and larger unit-data sizes respectively, regardless of the total cache size. Of course, if cost is not an a pressing issue, `m1.xlarge` with or without EBS persistence should be used achieve the highest performance.

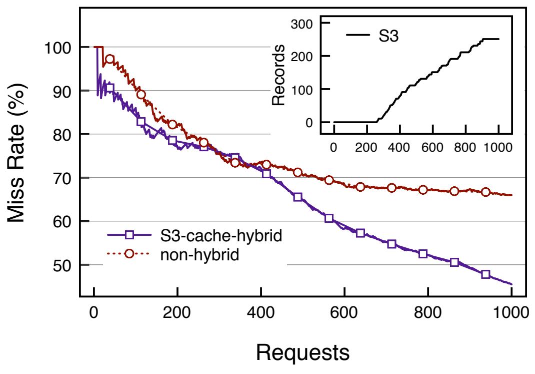
4.7 Hybrid Cache Evaluation

From these observations, we believe it would be cost effective to create a hybrid cache, which uses only *one* EC2 node and evicts records into S3. This cache variant is in contrast to previous versions, which grow incrementally without eviction. We believe this hybrid configuration would offer a cost-effective way to store the same amount of keys, but at a slight cost of performance of having to query S3.

The key search function would initially query the EC2 node, and upon miss, looks in S3. We built this cache hybrid, and ran the following experiments on one `m1.xlarge` node. Using 5 MB unit size, we again randomly submitted 3000 unique keys over 5000 requests. We also submitted 1000 requests over 500 unique keys for 25 MB unit size. We measured their miss rates, shown in Figures 8(a) and 8(b) respectively. The subgraphs within each figure show the amount of records that have been evicted into S3. As can be seen in both figures, the hybrid cache offers higher miss rates, which would expectedly lower average request times. This pattern is more prominent and occurs sooner in Figure 8(b) due to a lower key range being used in this experiment. Due to space constraints, we do not offer a deep performance and cost analysis on this version of the cache against previous versions. We plan to disseminate these results as future work.



(a) Data Size = 5 MB



(b) Data Size = 25 MB

Figure. 8. Miss Rates for S3-Hybrid Cache

5. RELATED WORKS

[Juve et al. 2010] examined application performance in the context of scientic workflow data deployed on various storage options, namely S3, NFS, GlusterFS, and PVFS. Their findings proved that the cost of running workows on EC2 is not at all prohibitive when a single instance is used, and S3's performance and cost-effectiveness exceeds all others when data reuse is necessary in an application. [Dejun et al. 2009] addressed the issue of performance homogeneity, imperative for predicting future resource requirements for maintaining the SLA in service-oriented applications, in the context of Amazon EC2's resource provisioning, and observed heterogeneous performance behavior among instances. However, small EC2 instances, spread across different zones, can be classified into three or four clusters with similar performance measures. [Evangelinos and Hill 2008] concluded that EC2 has huge potential in parallel HPC applications if coupled with the right parallel middleware (MPI) and high-performance interconnect (e.g., Myrinet or Infiniband). [Hill and Humphrey 2009] performed a quantitative evaluation of 64-bit Amazon EC2 as a replacement of Gigabit Ethernet Commodity Clusters for small-scale scientic applications. Their results echo the findings of other similar research. EC2 is not the best platform for tightly coupled synchronized applications with frequent communication between instances because of high network latency. However, its on-demand capabilities with no queuing front-end (unlike traditional HPC environment) makes for a compelling environment for scientists looking to quickly debug and compute small scale applications, without the queue and wait time.

Distributed data and cache storage systems are abundant, and they differ in usage expectations which define their functionalities. As today's data store solutions typically seek to avoid a centralized architecture, consistent hashing [Karger, *et al.* 1997] has quickly become the preferred method for specifying data locality. For instance, consistent hashing is currently being employed in many Web caching [Karger, *et al.* 1999], peer to peer [Stoica et al. 2001; Rowstron and Druschel 2001; Zhao et al. 2004], and NoSQL data stores [DeCandia et al. 2007; Lakshman and Malik 2009; Olson et al. 1999].

Due to the simplicity in their APIs, key-value stores (or so-called NoSQL data stores) have become increasingly popular in supporting today's applications. *Memcached* [Fitzpatrick 2004] is a popular distributed key-value caching system which originally aimed to accelerate dynamic Web applications by eluding expensive unnecessary queries to back-end databases. It stores serialized data objects in memory up to a fixed size, but it is typically assumed small. The memcached servers utilize an LRU and TTL eviction policy and requires manual scaling. MemcacheDB [memcached] further adds persistence and transaction support to the memcache framework by using BerkeleyDB [Olson et al. 1999] as a back-end. In contrast, our system has no restrictions on data size and allows the servers to expand on-demand when reaching capacity. We use consistent hashing on the client to route requests and furthermore provision data migration capabilities to avoid cache misses upon node expansion. Due to memcache's lack of migration, we can expect significant amounts of misses on certain range of keys during manual scaling. We are working on comparing our work against memcached in an elastic cloud environment.

Google's BigTable [Chang et al. 2006] and the open-source Hadoop-oriented implementation, HBase[4], are distributed column-stores capable of handling very large structured data, capable of scaling to thousands of low-cost machines. Amazon's Dynamo [DeCandia et al. 2007] and Facebook's Cassandra [Lakshman and Malik 2009], are highly available and reliable key-value stores for structured data. Like our system, both Cassandra and Dynamo allow for incremental scaling of nodes through exploiting consistent hashing to handle data partitioning and migration. The above efforts in key-value stores put forth focus on supporting features required in transactional databases, including replication, persistence, and consistency. While enabling such support is a necessity for persistent data applications, it expectedly leads to a degrade in performance. The data cache presented in this paper is far more ephemeral and lightweight in nature. Our cache does not focus on persistence and thus avoid these such requirements.

Other works on indexing multi-dimensional data in the cloud [Wang et al. 2010] have focused on using Content-Addressable Networks (CAN) and R-Trees. There they propose an overlay structure called RT-CAN that is capable of scaling up with demand. However, there is an assumption that the overlay is a static network, meaning once a node is brought online it remains so until the hardware fails. This preclused the ability to scale down on-demand. By comparison, our system makes no such assumption and, indeed, is built with the expectation that nodes will be joining and leaving frequently as demand requirements change.

Our proposed cache system has been tailored for cloud environments and is capable of incrementally grow to flexibly adapt to increasing workloads, which are prevalent in compute- and query-intensive environments. Moreover, the main contribution in this paper is analyzing data migration overheads given various pervasive key-indexing schemes in elastic cloud environments.

6. CONCLUSION

Clouds are an on-demand source of computational and storage resources and supports dynamic scaling of these resources. This property of the cloud motivated us to implement a cooperative elastic cache which has been deployed onto Amazon EC2. We showed through experiments that elasticity can be leveraged incrementally to reduce cache miss rates to near-zero values in our application. Moreover, the system achieved the same performance as the static node versions (found in traditional cluster environments), but utilized fewer nodes in the process, which is important in terms of cost.

Secondly, we evaluated the performance of B$^+$-tree, Extendible Hashing and Counting Bloom Filters. Counting Bloom Filters consistently performed poorly and were the least suited for our system, in the context of supporting the elastic cache. As expected, B$^+$-Trees performed well consistently and scaled

[4]http://hbase.apache.org

well with change in query intensity. The performance of Extendible Hashing was dependent on its parameter (number of records per bucket) and the system environment (query intensity). Thus, Extendible Hashing outperformed all other indexing schemes, on choosing the right parameters, which was not initially within our expectations. Another interesting observation we made was the Bloom Filter's increasing performance in data migration, as nodes were being added. In the end, however, this resurgence will not overtake B^+-Trees' overall performance. We attempted to optimize the system by minimizing wait time (idle time) through speculative prelaunching of instances. This was achieved by pre-loading instances when a threshold was met and by introducing multi-threading. The threshold varied dynamically and depended on the node capacity, number of nodes allocated and the query intensity.

In terms of costs, we offered an extensive evaluation of supporting such a cache over Amazon's EC2 and S3 services. While S3 is certainly a viable option, for high performance applications, users might prefer the more costly memory-based store. We designed a hybrid version of this cache, which stores the most-recently used records in EC2 nodes, and evicts records into S3, as a cost-effective alternative.

We plan to extend our elastic key-value store such that all the subtlety regarding resource allocation in the cloud is abstracted from the user. We propose a fully autonomous cache engine that would intelligently control cloud resources based on users' expectation of cost and performance. Our cache engine would control two aspects of the data organization: micro-level and macro-level. At the micro-level, each cache server is concerned with where data should be placed: in memory, on disk, or in persistent storage. As such, we plan to outfit each server node with efficient strategies for classifying data and promoting or demoting to different storage options (or evicting in entirety) based on usage patterns, data size, and other factors. At the macro-level, the organization engine is mostly concerned with managing the size of each tier in the hierarchy, and thus is also concerned with costs. The application users have the option to input the following: (1) a cost constraint parameter, C, (2) a cost-priority parameter, P_c ($0<P_c<1$), and (3) a list of cloud resource usage cost. The objective is to maximize cache performance given these constraints. The cost-priority parameter allows users to configure the importance of performance versus cost, i.e., a high value of P_c implies that the cache manager should strive to keep costs as low as possible. The macro component is also responsible for capacity planning, and upon any danger of overflowing, it would initiate the allocation of new instances (or upgrade to a larger machine), if within budget. Conversely, it would take necessary steps to consolidate instances if under-utilization is predicted.

Acknowledgments

We are grateful to the reviewers for their helpful and constructive feedback for this article.

This work is supported by an AWS Research Grant and a New Faculty Research Grant to D. Chiu at Washington State University. This research was also supported by NSF grants IIS-0916196 and CCF-0833101 and an AWS Research Grant to G. Agrawal at Ohio State University.

REFERENCES

ARMBRUST, *et al.*, M. 2009. Above the clouds: A berkeley view of cloud computing. Tech. Rep. UCB/EECS-2009-28, EECS Department, University of California, Berkeley. Feb.

BAYER, R. AND MCCREIGHT, E. 1970. Organization and maintenance of large ordered indices. In *SIGFIDET '70: Proceedings of the 1970 ACM SIGFIDET (now SIGMOD) Workshop on Data Description, Access and Control*. ACM, New York, NY, USA, 107–141.

BONOMI, F., MITZENMACHER, M., PANIGRAH, R., SINGH, S., AND VARGHESE, G. 2006. Beyond bloom filters: from approximate membership checks to approximate state machines. In *SIGCOMM '06: Proceedings of the 2006 conference on Applications, technologies, architectures, and protocols for computer communications*. ACM, New York, NY, USA, 315–326.

CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. 2006. Bigtable: a distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*. USENIX Association, Berkeley, CA, USA, 15–15.

CHIU, D. 2010. Auspice: Automatic service planning in cloud/grid environments. Ph.D. thesis, The Ohio State University, Columbus, OH.

CHIU, D., SHETTY, A., AND AGRAWAL, G. 2010. Elastic cloud caches for accelerating service-oriented computations. In *Proceedings of Supercomputing (SC'10)*.

COMER, D. 1979. The ubiquitous b-tree. *ACM Computing Surveys 11*, 121–137.

DAS, S., AGRAWAL, D., AND ABBADI, A. E. 2009. ElasTraS: An Elastic Transactional Data Store in the Cloud. In *Proceedings of Workshop on Hot Topics in Cloud (HotCloud)*.

DECANDIA, G., HASTORUN, D., JAMPANI, M., KAKULAPATI, G., LAKSHMAN, A., PILCHIN, A., SIVASUBRAMANIAN, S., VOSSHALL, P., AND VOGELS, W. 2007. Dynamo: amazon's highly available key-value store. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*. SOSP '07. ACM, New York, NY, USA, 205–220.

DEJUN, J., PIERRE, G., AND CHI, C.-H. 2009. Ec2 performance analysis for resource provisioning of service-oriented applications. In *Proceedings of the 3rd Workshop on Non-Functional Properties and SLA Management in Service-Oriented Computing*.

ELMASRI, R. AND NAVATHE, S. B. 2003. *Fundamentals of Database Systems, Fourth Edition*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

ENBODY, R. J. AND DU, H. C. 1988. Dynamic hashing schemes. *ACM Comput. Surv. 20,* 2, 850–113.

EVANGELINOS, C. AND HILL, C. 2008. Cloud computing for parallel scientific hpc applications: Feasibility of running coupled atmosphere-ocean climate models on amazon's ec2. In *Cloud Computing and Its Applications*. Chicago, IL.

FAGIN, R., NIEVERGELT, J., PIPPENGER, N., AND STRONG, H. R. 1979. Extendible hashing - a fast access method for dynamic files. *ACM Trans. Database Syst. 4*, 315–344.

FAN, L., CAO, P., ALMEIDA, J., AND BRODER, A. Z. 2000. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Trans. Netw. 8,* 3, 281–293.

FITZPATRICK, B. 2004. Distributed caching with memcached. *Linux J. 2004*, 5–.

HILL, Z. AND HUMPHREY, M. 2009. A quantitative analysis of high performance computing with amazon's ec2 infrastructure: The death of the local cluster? In *Proceedings of the 10th IEEE/ACM International Conference on Grid Computing*. Banff, Alberta, Canada.

JUVE, G., DEELMAN, E., VAHI, K., MEHTA, G., BERRIMAN, G. B., BERMAN, B. P., AND MAECHLING, P. 2010. Data sharing options for scientific workflows on amazon ec2. In *SC*. 1–9.

KARGER*, et al.*, D. 1997. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *ACM Symposium on Theory of Computing*. 654–663.

KARGER*, et al.*, D. 1999. Web caching with consistent hashing. In *WWW'99: Proceedings of the 8th International Conference on the World Wide Web*. 1203–1213.

KIRSCH, A. AND MITZENMACHER, M. 2008. Less hashing, same performance: Building a better bloom filter. *Random Struct. Algorithms 33,* 2, 187–218.

LAKSHMAN, A. AND MALIK, P. 2009. Cassandra: structured storage system on a p2p network. In *Proceedings of the 28th ACM symposium on Principles of distributed computing*. PODC '09. ACM, New York, NY, USA, 5–5.

LIM, H., BABU, S., AND CHASE, J. 2010. Automated Control for Elastic Storage. In *Proceedings of International Conference on Autonomic Computing (ICAC)*.

memcached. Memcachedb http://memcachedb.org/.

OLSON, M. A., BOSTIC, K., AND SELTZER, M. 1999. Berkeley db. In *Proceedings of the annual conference on USENIX Annual Technical Conference*. USENIX Association, Berkeley, CA, USA, 43–43.

PUTZE, F., SANDERS, P., AND SINGLER, J. 2009. Cache-, hash-, and space-efficient bloom filters. *J. Exp. Algorithmics 14*, 4.4–4.18.

ROWSTRON, A. AND DRUSCHEL, P. 2001. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*. 329–350.

STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, M. F., AND BALAKRISHNAN, H. 2001. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM '01 Conference*. San Diego, California.

ULLMAN, J. D., GARCIA-MOLINA, H., AND WIDOM, J. 2001. *Database Systems: The Complete Book*. Prentice Hall PTR, Upper Saddle River, NJ, USA.

WANG, J., WU, S., GAO, H., LI, J., AND OOI, B. C. 2010. Indexing multi-dimensional data in a cloud system. In *SIGMOD*. Indianapolis, Indiana.

ZHAO, B. Y., HUANG, L., STRIBLING, J., RHEA, S. C., JOSEPH, A. D., AND KUBIATOWICZ, J. D. 2004. Tapestry: A resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications 22,* 1 (Jan.), 41–53.

David Chiu is an Assistant Professor of Computer Science at Washington State University in Vancouver, WA. He received his Ph.D. from the Ohio State University in 2010 and his B.S. and M.S. from Kent State University in 2002 and 2004, respectively. His research interests are in scientific workflows, distributed computing, and data management.

Travis Hall is currently pursuing a Masters in Computer Science at Washington State University, Vancouver, where he also received a B.S. in Computer Science. His primary research area is in cloud computing under the tutelage of Dr. David Chiu. Most recently, he has been working under Google's Summer of Code program on a web service designed to provide students with course- and assignment-based code repositories. Web software (and design), cloud computing, and Computer Science education are some of his main interests.

Farhana Kabir is currently pursuing a Masters in Computer Science at Washington State University, Vancouver. Her research area is Cloud Computing, under the supervision of Dr. David Chiu. She received a B.S. in Computer Science (with highest distinction) from Purdue University. In the past she has worked as a software developer for Oracle Corp. More recently she was a graduate intern at Intel's Digital Home Group, working on the security controller software for the Intel CE media processor. Cloud Computing, Information Retrieval, Networking, and Security are among the technologies she is most excited about and wishes to work on in the future.

Apeksha Shetty has been working as R&D at Epic Systems since July, 2010. She graduated with an M.S. in Computer Science and Engineering from the Ohio State University. She was advised by Dr. Gagan Agrawal, and her thesis was on the Evaluating and Optimizing Indexing Schemes for an Elastic Cache in a Cloud Environment. Apeksha's interests include data mining, cloud computing, web services, web technology, and software engineering.

Gagan Agrawal is a Professor of Computer Science at the Ohio State University. He received his BS degree from IIT Kanpur in 1991, and MS and PhD degrees from University of Maryland, College Park in 1994 and 1996, respectively. His research interests include parallel and distributed computing, compiler and runtime systems, and data mining/integration. He has published more than 175 papers in these topics.