2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing

Evaluating and Optimizing Indexing Schemes for a Cloud-based Elastic Key-Value Store

David Chiu
Washington State University
david.chiu@wsu.edu

Apeksha Shetty
Ohio State University
shetty@cse.ohio-state.edu

Gagan Agrawal
Ohio State University
agrawal@cse.ohio-state.edu

***Abstract*—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 be immediately given access to, more (or less) resources based on requirements. However, this feature introduces new challenges in developing application and services. In this paper, we focus on the challenges in data management in Cloud environments, in view of elasticity. 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.

I. Introduction

The Cloud has been hailed as a reliable, scalable, on-demand source of computation and storage provided over the Internet. Among many Cloud providers (including Azure[1] and Google App Engine[2]), Amazon's Elastic Compute Cloud (EC2)[3] has also emerged as a so-called Infrastructure as a Service (IaaS) provider as it allows users complete control over resources. One of the most important facets of EC2 is its ability to allow the users to instantly scale their resource requirement up according to demands. When exploited optimally, this *elastic* compute aspect lets users avoid over- or under-provisioning of resources [1], a highly desirable feature that had eluded conventional distributed environments.

[1]Azure Services Platform, http://www.microsoft.com/azure
[2]Google App Engine, http://code.google.com/appengine
[3]Amazon Elastic Compute Cloud, http://aws.amazon.com/ec2

The emergence of the elastic computing paradigm has been timely. For example, consider a scenario where the demand for various service-oriented applications 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. One way to combat this issue would be to dynamically acquire more computing resources from the Cloud 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 of Clouds [5], [7], [18] for various application classes.

One prominent issue that has not yet received much attention is management of data while leveraging elasticity. In this paper, we focus on the challenges of managing an elastic *key-value* store in a Cloud environment. Though several Cloud-based key-value storage systems have been developed in recent times [4], [17], [8], [13], they have not been considered in the presence of elastic computing. We have developed an elastic key-value store, which is motivated by the challenges toward accelerating *service-oriented* computations on the Cloud. Our elastic store caches intermediate results of services and using them for future computations. Such a cache or key-value store can clearly benefit from elasticity. For instance, an unexpected query-intensive period can be responded to by a self-managed expansion of resources.

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 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 for rendering such migrations efficient.

In this paper, we have developed migration algorithms to support the use of three popular indexing schemes: B+-trees [2], [6], Extendible Hashing [11], and Counting Bloom Filters [12], [21]. We chose to focus on these three schemes because they pervade

978-0-7695-4395-6/11 $26.00 © 2011 IEEE
DOI 10.1109/CCGrid.2011.29

existing systems and are heavily documented in literature. Based on our migration algorithms, we have compared 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 EC2 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.

The remainder of this paper is organized as follows. In Section II, we present the background of our cooperative elastic cache design. In Section III, we present some basic background for the three indexing schemes and discuss their implementation and heuristics for optimization in the elastic cache. In Section V, we and present the experimental evaluation. A discussion of related works is given in Section VI. Finally, we conclude our work in Section VII.

II. Motivating Application: Key-Value Cache Elasticity

In this section, we will describe the basic architecture of the specific key-value store we have considered, which is a cooperative cache. A main goal of the cache we are considering 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, is comprised of a set of Cloud nodes, which consist of a coordinator node and cohort 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, the it must first determine which cohort server node might contain the data and route the query request to the identified cohort node. During a hit cache on one of the cohort 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.

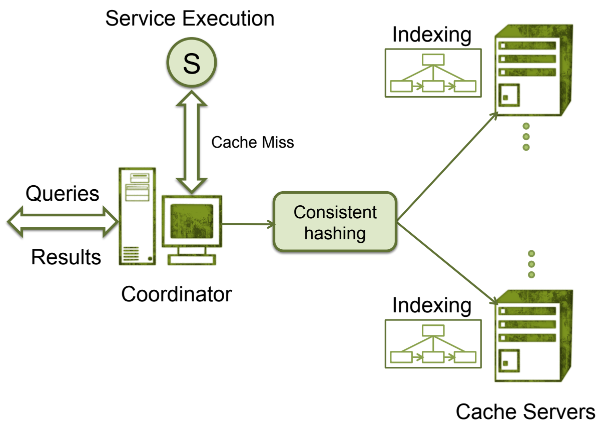


Fig. 1. Client-Server Architecture of the Elastic Cache

Initially, it may seem like a simple hashing mechanism would suffice for identifying the cohort 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 cohort node membership changes.

To address this problem, consistent hashing [14] is being employed on the coordinator due to its ability to quickly adapt to nodes joining and leaving a cooperating system (indeed, it is used extensively in highly volatile P2P systems among others). Consider the example in Figure 2, which depicts a consistent hashing system consisting cohort nodes A and B. The hash range is circular (from 0 and $r - 1$), and consists of several buckets placed randomly or strategically. Each bucket further stores a pointer to a physical storage node. An auxiliary static 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 2, we are further showing that node A is overflown, and a new node, C, has been instantiated from the Cloud to incrementally join our 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 $(75, 8]$ will be hashed into the new node, C to alleviate the load on A, e.g., the bucket could be placed at $(75+8+r)/2$. To complete the node membership process, the

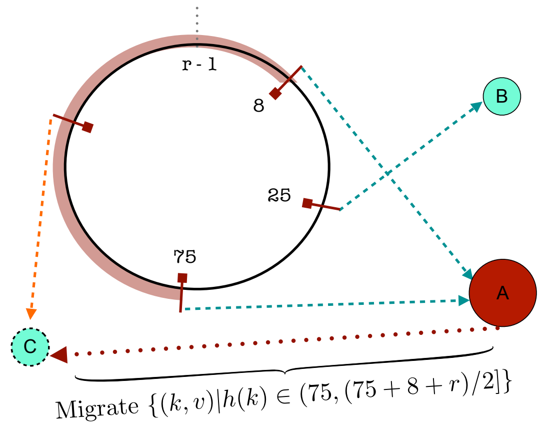


Fig. 2. Consistent Hashing Example

(k, v) pairs hashing into $(75,(75+8+r)/2]$ are finally migrated to C.

Each cohort 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.

III. Indexing Background and Elastic Integration

A distinct indexing service has to be implemented on each cohort node supporting the key-value server. We consider three ubiquitous indexing schemes used in our cohort cache nodes for facilitating key-value storage: B$^+$-Tree [2], [6], Extendible Hashing [24], and Counting Bloom Filters [12], [21]. 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, and thus 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.

A. *B$^+$-Trees*

B-trees and its variant B$^+$-trees are used extensively in many of today's systems. 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 and the previous nodes, which was specifically designed to accelerate range queries [9], [24].

The basic structure of 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. 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.

Algorithm 1 BT_Migrate(k_{start}, k_{end})

```
1: ▷ manipulate B$^+$-tree index and transfer the keys in form of a string, $keys$
2: $end \leftarrow false$
3: ▷ $L$ = leaf initially containing $k_{start}$
4: $L \leftarrow btree$.search($k_{start}$)
5: while ($\neg end \wedge L \neq NULL$) do
6:     ▷ each leaf node contains multiple keys
7:     for all $(k, v) \in L$ do
8:         if $k \leq k_{end}$ then
9:             $keys$.append($k$, $v$)
10:            $btree$.delete($k$)
11:        else
12:            $end \leftarrow true$
13:            break
14:        end if
15:    end for
16:    $L \leftarrow L$.next()
17: end while
18: return $keys$
```

The B+-Tree migration algorithm is shown in Algorithm 1.

Both k_{start} and k_{end} are the inputs to this procedure, and they denote the limits of the range to be migrated (again, typically smallest key, and median key respectively). On Line 4, we 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. Each swept record is appended to a $keys$ set and deleted (Lines 5 - 17). Finally, when a record is $k > k_{end}$, we exit the loop and return the set of keys needing migrated.

B. 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 implementation, 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, all 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 number of records stored to the size of bucket) increases.

We can avoid this performance hit by using dynamic hashing [10], 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 [11], which introduces a concept known as *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, 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.

Algorithm 2 EH_Migrate(k_{start}, k_{end})

```
1: static $H$ ▷ bring extendible hashtable to scope
2: $keys \leftarrow \{\}$
3: **for** $x \leftarrow 0$ to $2^i - 1$ **do**
4:     $D_x \leftarrow H$.getDirectoryAt($x$)
5:     **for** $y \leftarrow 0$ to $|D_x| - 1$ **do**
6:         $B_y \leftarrow D_x$.getBucketAt($y$)
7:         **for all** $k \in B_y | k \geq k_{start} \wedge k \leq k_{end}$ **do**
8:             $keys \leftarrow keys \cup (k, v)$
9:             $H$.delete($k$)
10:        **end for**
11:    **end for**
12: **end for**
13: return $keys$
```

In Algorithm 2, inputs k_{start} and k_{end} again 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.

C. Counting Bloom Filter

Bloom Filters [3] 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[4]. However, Bloom Filters are vulnerable to false positives, but 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 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

[4]XLattice, http://xlattice.sourceforge.net/components/crypto/

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 [16].

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 [12], [21]. 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 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 the inputs and they again represent the minimum and maximum limits of the migration range. 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).

IV. Elastic Cache Support

We have implemented our system on EC2, and will describe our system under its context. The Amazon Elastic Compute Cloud (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*. 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[5] 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. As a baseline, we have chosen to only employ Small instances in our implementation, although it should be noted that instances bearing much more (or less) capabilities are also available through Amazon.

During system execution, records may be continuously inserted into the cache nodes, and an overflow on any of the nodes can invoke incremental scaling. This process involves starting a new EC2 node and migrating a subset of the data from the overflown node to the newly allocated node using methods specified in the previous sections. There are two phases of overhead in this scheme: (1) instance allocation time, which can take up to several minutes during peak times, and (2) data migration time, which involves identifying the migration range and network transfer time.

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

In our experience, instance allocation dominates data migration time, and we implement a speculative pre-launching of instances when a threshold, T, has been met. A background thread initiates the pre-launching and eagerly migrates data from the fullest node once the new node 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. If n is the node to insert some (k, v) pair, 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: the lowest and highest *expected* querying rate 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 cache insertion algorithm, shown in Algorithm 3. The identifiers used in Algorithm 3 are listed in Table I.

TABLE I
Listing of Identifiers for Algorithm 3

Identifier	Description
k	A queried key
$B = (b_1, \ldots, b_p)$	The list of all buckets on the hash line
$h(k)$	The hash function, which returns the closest upper bucket to k
$N = (n_1, \ldots, n_m)$	The set of all nodes in the cooperative cache
$n \in N$	A cache node
$\|\|n\|\|$	Current size of index on node n
$\lceil n \rceil$	Overall capacity on node n
$\|\|N\|\|$	Number of nodes part of the cooperative cache N
R	Query intensity, i.e. queries per time step

In Algorithm 3, 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, and the auxiliary consistent hash map, h', are brought into scope. The $NodeMap[b]$ returns the node n pointed by bucket b.

Algorithm 3 Speculative-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:    $n$.insert($k, v$)    ▷ 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
27: Speculative-Insert($k, v, \delta_L, \delta_H$)
```

Lines 5-11 deal with 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. Selecting 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.

Retreating discussion back to the side of the cache servers, each cohort node consists of the index, insert(), 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, insert(), 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.

V. 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.

A. 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. Cache server basically contains the indexing logic and for the results presented in this section, only B$^+$- tree was used.

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 cohort 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 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 hash function, to 64K.

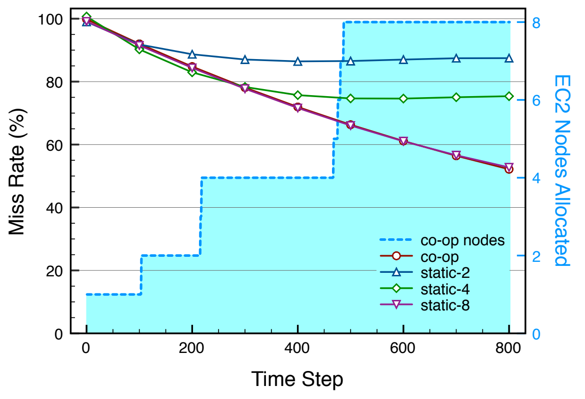
We tested our system under varying query rates, we varied the rate between 50 queries/time step and 255 queries/time step. At each time step, we recorded the average (in seconds) and the number of hits and misses. In order to regulate the integrity in querying rates, we submitted query requests with the following loop: Specifically, we invoke R queries per time step, and thus each time step 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

```
for time step $i \leftarrow 1$ to ... do
    $R \leftarrow$ current query rate($i$)
    for $j \leftarrow 1$ to $R$ do
        invoke shoreline service(rand_coordinates())
    end for
end for
```

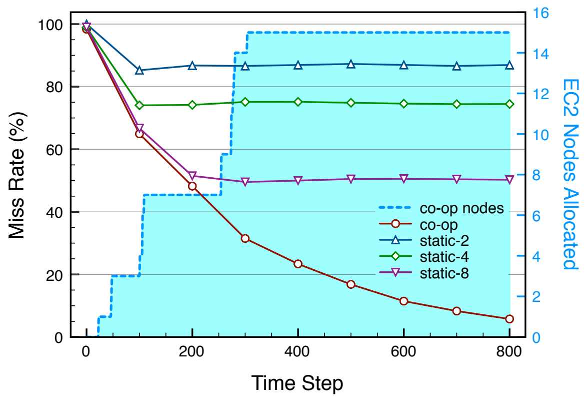
number of seconds real time), the number of times a query reuses a cached record (i.e., hits), and the number of cache misses.

B. 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. For these experiments, all cache servers are utilizing B$^+$-Trees.



(a) Querying Rate = 50 queries/timestep



(b) Querying Rate = 255 queries/timestep

Fig. 3. Miss Rate

Figure 3(a) and Figure 3(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 3(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 3(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.

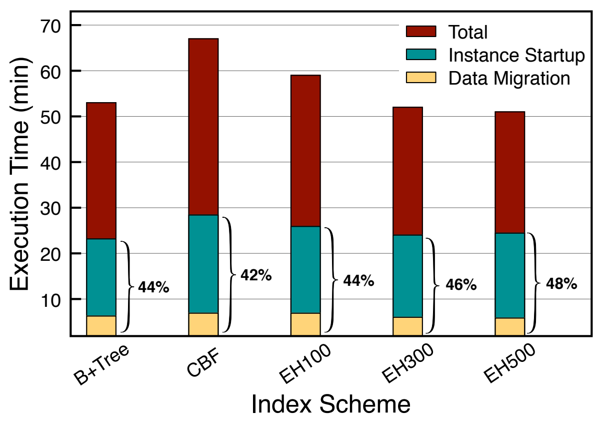
One aspect that is not being shown here is the time taken to split and migrate data when a new node is allocated. This may be a costly overhead that varies depending on the index that is being used on the cohort cache server. We show an evaluation of the impact of indexing schemes, and the migration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

C. Cache Server Index Comparison

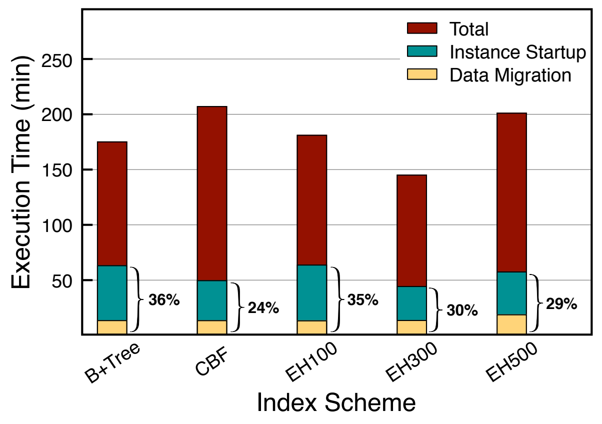
The three indexing schemes we compared are: B+-Trees (`B+Tree`), and Counting Bloom Filter (`CBF`), and Extendible Hashing with three bucket size configurations: (`EH100`, `EH300`, `EH500`). We evaluate the suitability of these algorithms for our system based on the total time taken to run the experiment, the total time taken to migrate records and minimum, maximum and average time taken per migration. 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 speculative execution in a later subsection.

Figure 4(a) shows the results obtained by running the experiment with 50 queries/timestep, which is relatively lower query intensity than Figure 4(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, and 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 4(a), it can be observed that `EH500` outperforms the rest, while `CBF` are 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

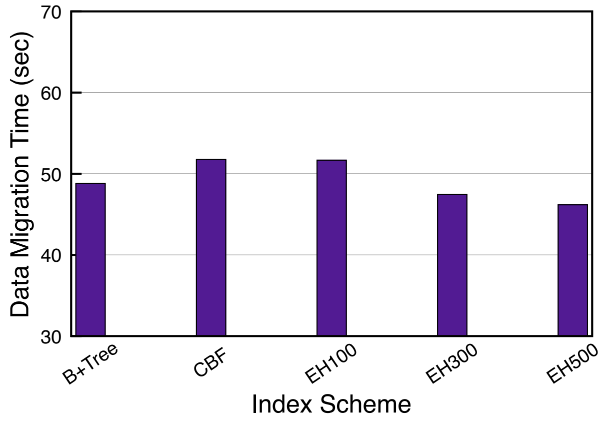


(a) Query rate = 50 Queries/timestep

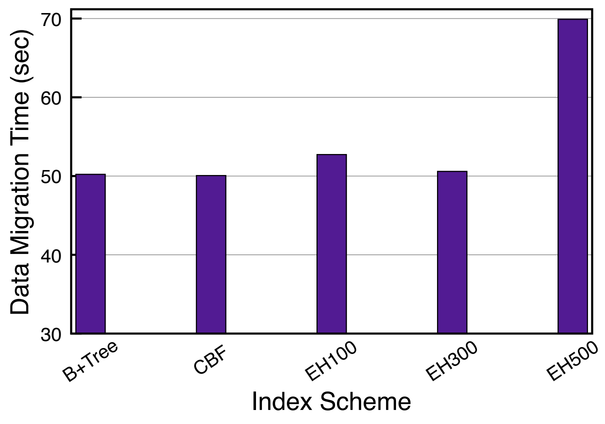


(b) Query rate = 255 Queries/timestep

Fig. 4. Execution Time of Indexing Schemes



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



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

Fig. 5. Migration Time of Indexing Schemes

well irrespective of the parameters. This can also be verified by Figure 4(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.

Focusing now on Figures 5(a) and 5(b), we have averaged 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 5(b), `CBF` is eventually equivalent to the B$^+$-Trees' linear-time migration algorithm.

The same logic can be applied to explain the degrade in performance for the `EH*` 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 less migrations because there are smaller numbers of buckets to traverse linearly. As migrations increase to 15 times, this would imply that the much more data is being stored in the index, which translates to not only a larger directory size (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*`, 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*` 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.

D. Optimization Results

Back in Section IV, we described an approach to minimize the idle time when migrating data to newly allocated EC2 nodes. Instances are pre-launched when a threshold is met and the migration overlaps with normal program execution. The results have been summarized in Figures 6. 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 prelaunching.

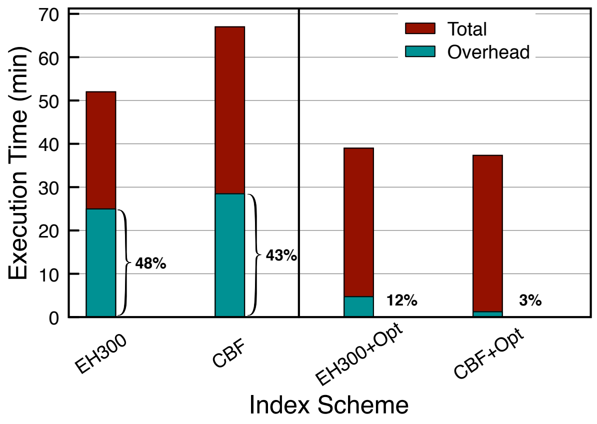


Fig. 6. Optimization Results for Query Rate = 50 queries/timestep.

We executed a total of 3 runs and reported the average overhead and total times. After optimization, we managed to improve overhead by 4× and 14× respectively in `EH300` and `CBF`. Even with the improvements, there are clearly opportunities for improvement here, and we propose to develop more robust heuristics in future works.

VI. Related Works

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 [14] has quickly become the preferred method for specifying data locality. For instance, consistent hashing is currently being employed in many Web caching [15], peer to peer [23], [22], [25], and NoSQL data stores [8], [17], [20].

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* [13] 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 [19] further adds persistence and transaction support to the memcache framework by using BerkeleyDB [20] 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 [4] and the open-source Hadoop-oriented implementation, HBase[6], are distributed column-stores capable of handling very large structured data, capable of scaling to thousands of low-cost machines. Amazon's Dynamo [8] and Facebook's Cassandra [17], 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.

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]http://hbase.apache.org

VII. Conclusion

Clouds are an on-demand source of computational and storage resources, and it 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 was the least suited for our system, in the context of supporting the elastic cache. As expected, B^+-Trees performed well consistently and scaled up well to 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.

Finally, 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.

Acknowledgments

We are extremely grateful to our reviewers for their insightful comments and suggestions for improving this paper. Their constructive input is, as always, *greatly* appreciated.

This work was generously supported by an Amazon Web Services (AWS in Education) Research Award to D. Chiu at Washington State University.

This research was also supported by NSF grants IIS-0916196 and CCF-0833101 and an Amazon Research Award to G. Agrawal at Ohio State University.

References

[1] M. Armbrust, *et al.* Above the clouds: A berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, Feb 2009.

[2] R. Bayer and E. McCreight. 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*, pages 107–141, New York, NY, USA, 1970. ACM.

[3] F. Bonomi, M. Mitzenmacher, R. Panigrah, S. Singh, and G. Varghese. 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*, pages 315–326, New York, NY, USA, 2006. ACM.

[4] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: a distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, pages 15–15, Berkeley, CA, USA, 2006. USENIX Association.

[5] D. Chiu, A. Shetty, and G. Agrawal. Elastic cloud caches for accelerating service-oriented computations. In *Proceedings of SC*, 2010.

[6] D. Comer. The ubiquitous b-tree. *ACM Computing Surveys*, 11:121–137, 1979.

[7] S. Das, D. Agrawal, and A. E. Abbadi. ElasTraS: An Elastic Transactional Data Store in the Cloud. In *Proceedings of Workshop on Hot Topics in Cloud (HotCloud)*, 2009.

[8] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM.

[9] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems, Fourth Edition*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[10] R. J. Enbody and H. C. Du. Dynamic hashing schemes. *ACM Comput. Surv.*, 20(2):850–113, 1988.

[11] R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong. Extendible hashing - a fast access method for dynamic files. *ACM Trans. Database Syst.*, 4:315–344, September 1979.

[12] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Trans. Netw.*, 8(3):281–293, 2000.

[13] B. Fitzpatrick. Distributed caching with memcached. *Linux J.*, 2004:5–, August 2004.

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

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

[16] A. Kirsch and M. Mitzenmacher. Less hashing, same performance: Building a better bloom filter. *Random Struct. Algorithms*, 33(2):187–218, 2008.

[17] A. Lakshman and P. Malik. Cassandra: structured storage system on a p2p network. In *Proceedings of the 28th ACM symposium on Principles of distributed computing*, PODC '09, pages 5–5, New York, NY, USA, 2009. ACM.

[18] H. Lim, S. Babu, and J. Chase. Automated Control for Elastic Storage. In *Proceedings of International Conference on Autonomic Computing (ICAC)*, June 2010.

[19] Memcachedb http://memcachedb.org/.

[20] M. A. Olson, K. Bostic, and M. Seltzer. Berkeley db. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 43–43, Berkeley, CA, USA, 1999. USENIX Association.

[21] F. Putze, P. Sanders, and J. Singler. Cache-, hash-, and space-efficient bloom filters. *J. Exp. Algorithmics*, 14:4.4–4.18, 2009.

[22] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, Nov. 2001.

[23] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM '01 Conference*, San Diego, California, August 2001.

[24] J. D. Ullman, H. Garcia-Molina, and J. Widom. *Database Systems: The Complete Book*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[25] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications*, 22(1):41–53, Jan. 2004.