The Design and Implementations of Locality-Aware Approximate Queries in Hybrid Storage Systems

Yu Hua, *Senior Member, IEEE*, Bin Xiao, *Senior Member, IEEE*, Xue Liu, *Member, IEEE*, and Dan Feng, *Member, IEEE*

Abstract—Cloud computing applications face the challenges of dealing with a huge volume of data that needs the support of accurate and fast approximate queries to enhance system scalability and improve quality of service. Locality-sensitive hashing (LSH) can support the approximate queries that unfortunately suffer from imbalanced load and space inefficiency among distributed data servers, which severely limits the query accuracy and incurs long query latency between users and cloud servers. In this paper, we propose a novel scheme, called NEST, which offers easy-to-use and cost-effective approximate queries for cloud computing. The novelty of NEST is to leverage cuckoo-driven locality-sensitive hashing to find similar items that are further placed closely through cuckoo-driven method to obtain load-balancing buckets in hash tables. NEST hence carries out flat and manageable addressing in adjacent buckets, and obtains constant-scale query complexity even in the worst case. The benefits of NEST include the increments of space utilization and fast query response. Moreover, due to the salient property of flat addressing in NEST, we implement NEST design in a real hybrid storage system, which consists of DRAM, SSD, and hard disk. The flat addressing allows efficient operations in SSD to improve system performance. We argue that a proper "*division of labor*" among DRAM, SSD, and hard disk in the hybrid and heterogeneous storage hierarchy is desperately needed to strike an optimal balance to remove the indexing bottleneck. Theoretical analysis and extensive experiments (on LANL and Microsoft metadata) in a large-scale cloud testbed demonstrate the salient properties of NEST to meet the needs of approximate query service in cloud computing environments. We have offered open-source codes of NEST for public use.

Index Terms—Hybrid storage systems, approximate queries, locality

1 Introduction

Cloud computing applications generally have the salient property of massive data. The datasets with a volume of Petabytes or Exabytes and the data streams with a speed of Gigabits per second often have to be processed and analyzed in a timely fashion. According to a recent International Data Corporation (IDC) study, the amount of information created and replicated is more than 1.8 Zettabytes in 2011 [1]. Moreover, from small hand-held devices to huge data centers, we are collecting and analyzing ever-greater amounts of information. Users routinely pose queries across hundreds of Gigabytes of data stored on their hard drives or data centers. Some commercial companies generally handle Terabytes and even Petabytes of data everyday [2], [3], [4].

- *Y. Hua is with the Wuhan National Laboratory for Optoelectronics, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, China. E-mail: csyhua@hust.edu.cn.*
- *B. Xiao is with the Department of Computing, The Hong Kong Polytechnic University, Hong Kong. E-mail: csbxiao@comp.polyu.edu.hk.*
- *X. Liu is with School of Computer Science, McGill University, Montreal, Quebec, Canada. E-mail: xueliu@cs.mcgill.ca.*
- *D. Feng is with the Wuhan National Laboratory for Optoelectronics, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, China. E-mail: dfeng@hust.edu.cn.*

Manuscript received 29 Dec. 2013; revised 30 Aug. 2014; accepted 29 Oct. 2014. Date of publication 4 Nov. 2014; date of current version 7 Oct. 2015.
Recommended for acceptance by X. Sun.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2014.2367497

Cloud systems are facing great challenge in handling the deluge of data stemming from cloud computing applications such as business transactions, scientific computing, social network webs, mobile applications and information visualization. How to accurately return the queried results to requests is becoming more challenging than ever to cloud computing systems that generally consume substantial resources to support query-related operations [5], [6], [7]. Cloud computing demands not only a huge amount of storage capacity, but also the support of low-latency and scalable queries [3]. In order to address this challenge, query services have received many attentions in the cloud computing communities, such as query optimization for parallel data processing [4], automatic management of search services [8], similarity search in file systems [9], information retrieval for ranked queries [5], similarity search over cloud data [6], multi-keyword ranked and fuzzy keyword search over cloud data [10], [11], approximate membership query [12] and retrieval for content cloud [13].

Many practical applications in the cloud require real-time *Approximate Near Neighbor (ANN)* query service. Cloud users, however, often fail to provide clear and accurate query requests. Hence, the content cloud systems offer the ANN query to allow users to find the nearest files in distance measures by carrying out a multi-attribute query, such as filename, size, creation time, etc. On the other hand, a cloud system needs to support approximate queries to get particular search results. Consider another example of image protection and spam detection among billions of images in a cloud. A system supporting ANN queries can help identify and detect the modified images, which are often altered by cropping, re-scaling, rotation, flipping, color change or text insertion. Therefore, providing quick and accurate service of ANN query becomes a necessity for cloud development and construction [4].

Despite the fact that LSH [14] can be used to support ANN query due to its simplicity of hashing computation and faithful maintenance of data locality, performing efficient LSH-based ANN query needs to deal with two challenging problems. First, LSH suffers from space-inefficiency and low-speed I/O access because it leverages many hash tables to maintain data locality and a large fraction of data needs to be placed in hard disks. Although the space inefficiency has been partially addressed by multi-probe LSH [15], it decreases space overhead but becomes inefficient to support constant-scale complexity for queries, which makes it not suitable in large-scale cloud computing applications. Second, LSH produces imbalanced load in the buckets of hash tables to maintain data locality. In order to deal with hash collisions, some buckets in a hash table often contain too many items in the linked lists that produce linear searching time. In contrast, other buckets may contain very few or even zero items. Vertical addressing, such as probing data along a linked list within a bucket, further aggregates the negative effect and produces $O(n)$ complexity for n items in a linked list. The high complexity severely degrades the efficiency of query services.

In this paper, we propose a NEST design for cloud applications to support ANN query service and address the above problems of LSH. First, to build a space-efficient structure, we transform conventional vertical addressing of hash tables in LSH into flat and manageable addressing, thus allowing adjacent buckets to be correlated. As a result, we can significantly decrease the number of vacant buckets. Second, to alleviate the imbalanced load in the buckets, we use a cuckoo-driven method in LSH to obtain constant-scale operation complexity even in the *worst* case. The cuckoo method [16] can balance the load among the LSH buckets by providing more than one available bucket.

When facing the challenges of obtaining locality-aware data and achieving load balance in the cloud servers, it is worth noting that performing a *simple* combination of LSH and cuckoo hashing will be inefficient to support ANN query service due to extra frequent "kicking out" operations and high rehashing costs caused by the cuckoo hashing. To overcome such inefficiency, we propose locality-aware algorithms in the NEST design that leverages the adjacent buckets in the cuckoo hashing to manage the overflowed data during the LSH computation.

This paper has made the following contributions.

Locality-Aware Balanced Structure. We propose a novel locality-aware balanced data structure, called NEST, in cloud servers. NEST achieves locality-aware storage by using LSH and load-balanced storage by using the cuckoo-driven method to move crowded items to alternative empty positions. NEST can further significantly decrease the endless loop burden in cuckoo hashing by allocating new items in neighboring buckets, which is perfectly allowed in LSH. The proposed structure provides a locality-aware data management in a tri-tiered heterogeneous storage hierarchy. Specifically, the top-layer DRAM, as a temporary buffer, identifies and aggregates correlated files with the aid of LSH with a complexity of $O(1)$. In order to alleviate the hash collisions in the hash table, we employ a variant of the cuckoo hash [16] and achieve a constant-scale lookup complexity. The middle-layer SSD stores the metadata in the form of key-value pairs. A key is the hashed value of a file ID and the value is the metadata of that file. Correlated files are conducive to sequential operations with a high probability. Finally, the bottom-layer hard disk stores and maintains the correlated files. NEST is able to efficiently utilize the locality of datasets to support sequential operations (e.g., read/write) and data retrieval.

Constant-Scale Worst-Case Complexity. NEST demonstrates salient performance in practical operations, such as item deletion and ANN query, which are bounded by constant-scale worst-case complexity. In essence, we replace conventional vertical addressing, such as a linked list in a bucket, with flat and manageable addressing to a bucket and its limited number of neighbors. NEST has the same constant-scale worst-case complexity for item insertion in most cases, which shows its good scalability. The rehashing event has a very low probability to occur and has little impact on the overall operational performance of NEST. In order to support fast search in hybrid storage systems, we employ a Bloom-filter based hierarchy in both DRAM and SSD [17]. Specifically, in SSD, besides the key-value metadata store, we maintain a Page-level Bloom Filter (PBF) as a metadata index. Thus, PBF can fast answer the requests for membership queries. Since the typical DRAM lookup time is estimated at about 1us, much smaller than around 100us in SSD [18], we maintain a cached Bloom filter in DRAM that only stores "hot spot" files. The Bloom-filter hierarchy is able to efficiently support membership query for data intensive applications with small space overheads.

Practical Implementation. We have implemented the NEST prototype and compared it with the simple combination of "*LSH with cuckoo hashing (LSH-CH)*", and *LSB-tree* [19] for ANN query in a large-scale cloud computing testbed. LSH-CH is a simple combination of LSH and cuckoo hashing, which fails to efficiently handle the increments of hash collisions when data exhibits an obvious locality property. We use recently released real-world traces, i.e. Los Alamos National Lab (LANL) [20] and Microsoft metadata [21] to examine the real performance of the proposed NEST. Comparison results demonstrate performance gains of NEST for its low query latency, high query accuracy and space saving properties.

The rest of the paper is organized as follows. Section 2 shows research backgrounds. Section 3 presents the NEST design and practical operations. We present the experiment configurations and evaluation results respectively in Sections 4 and 5. Section 6 shows the related work. Section 7 concludes our paper.

2 Research Backgrounds

This section shows the research backgrounds of locality sensitive hashing and cuckoo hashing techniques for ANN query.

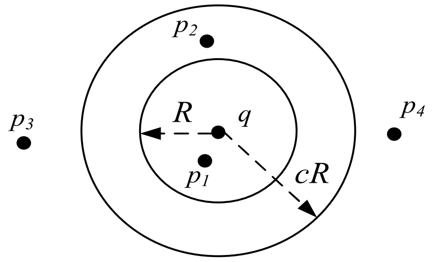


Fig. 1. The LSH scheme for approximately hashing items into the same bucket in hash tables with a high probability.

2.1 Locality Sensitive Hashing

Definition 1. (ANN Query). *Given a set S of data points in θ-dimensional space and a query point q, ANN query returns the nearest (or generally ϑ nearest) points of S to q.*

Data points a and b having θ-dimensional attributes can be represented as vectors $\vec{a_\theta}$ and $\vec{b_\theta}$. If their distance is smaller than a pre-defined constant R, we say that they are *correlated*. Correlated items constitute the set of an ANN query result. The distance between two items can be defined in many ways, such as the well known Euclidean distance, Manhattan distance and Max distance.

LSH [14], [22], [23] has the property that close items will collide with a higher probability than distant ones. In order to support ANN query, we need to hash query point q into buckets in multiple hash tables, and furthermore union all items in those chosen buckets by ranking them according to their distances to the query point q. We define S to be the domain of items. Distance functions $||*||_s$ correspond to different LSH families of l_s norms based on s-stable distribution to allow each hash function $LSH_{a,b} : R^\theta \rightarrow Z$ to map a θ-dimensional vector v onto a set of integers.

Definition 2. (LSH Function Family). $\mathbb{H} = \{g : S \rightarrow U\}$ *is called (R, cR, P_1, P_2)-sensitive for any $p, q \in S$*

- If $||p, q||_s \leq R$ then $Pr_{\mathbb{H}}[g(p) = g(q)] \geq P_1$,
- If $||p, q||_s > cR$ then $Pr_{\mathbb{H}}[g(p) = g(q)] \leq P_2$.

The settings of $c > 1$ and $P_1 > P_2$ are configured to support ANN query service. The practical implementation needs to enlarge the gap between P_1 and P_2 by using multiple hash functions. The hash function in $\mathbb{H}$ can be defined as $LSH_{a,b}(v) = \lfloor \frac{a \cdot v + b}{\omega} \rfloor$, where a is a θ-dimensional random vector with chosen entries following an s-stable distribution, b is a real number chosen uniformly from the range $[0, \omega)$ and ω is a constant.

LSH determines the proximate locality between two points by examining their distance in a metric space. If the ball centered at q with radius R covers at least one point, e.g. p_1, as shown in Fig. 1, LSH can provide a point with no more than cR distance to q as query result. We observe that there is an uncertain space in LSH from R to cR distance and the query q will obtain a reply of either point p_1 or p_2, since both points locate within distance cR, i.e. $||p_1, q||_s < cR$ and $||p_2, q||_s < cR$. On the other hand, point p_3 is not close to the queried q due to its distance larger than cR.

We need to configure two main parameters, M, the capacity of a function family $\mathbb{G}$, and ρ, the number of hash tables, to build an LSH. Specifically, given a function family $\mathbb{G} = \{g : S \rightarrow U^M\}$ and $LSH_j \in \mathbb{H}$ for $1 \leq j \leq M$, we have $g(v) = (LSH_1(v), \ldots, LSH_M(v))$ as the concatenation of M LSH functions, where v is a θ-dimensional vector. Furthermore, an LSH consists of ρ hash tables, each of which has a function $g_i (1 \leq i \leq \rho)$ from $\mathbb{G}$.

LSH has been successfully applied in approximate queries of vector space and semantic access. Main variants include entropy-based LSH [24], multi-probe LSH [15], LSBF [12] and LSB-tree [19]. The locality sensitive hashing however has to deal with the imbalanced load in the buckets due to hash collisions. Some buckets may contain too many items to be stored in the linked lists, thus increasing searching complexity. On the contrary, other buckets may contain less or even zero items. We hence take into account the cuckoo hashing technique to obtain constant-scale searching complexity.

2.2 Cuckoo Hashing

The name of cuckoo-driven method comes from cuckoo birds in nature, which kicks other eggs or birds out of their nests. This behavior is similar to the hashing scheme that recursively kicks items out of their positions as needed. Cuckoo hashing uses two or more hash functions for resolving hash collisions to alleviate the complexity of using the linked lists. Instead of only indicating a single position that an item a should be placed, cuckoo hashing can provide two possible positions, i.e., $h_1(a)$ and $h_2(a)$. Hence, collisions can be minimized and a bucket stores only one item. The presence of an item can be determined by probing two positions.

Cuckoo hashing, however, cannot totally eliminate data collisions. An insertion of a new item causes a failure when there are collisions in all probed positions. Even the "kicking out" hashing to make empty room for a new item is likely to produce endless loop. To break the loop, one way is to perform a full rehash if this rare event occurs. Since the item insertion failure in the cuckoo hashing scheme occurs with a low probability, such rehashing has very small impact on the average performance. In practice, the cost of performing a rehashing can be dramatically reduced by the use of a very small additional constant-size space.

Cuckoo hashing is a dynamization of a static dictionary described in [16] and provides a useful methodology for building practical, high-performance hash tables. It combines the power of schemes that allow multiple hash locations for an item with the power to dynamically change the location of an item among its possible locations. Furthermore, the space usage is roughly $2n$ space units, which means that the space usage is similar to that of binary search trees. Cuckoo hashing is very competitive, especially when the dictionary is small enough to fit in cache.

Definition 3. (Standard Cuckoo Hashing). *Cuckoo hashing uses two hash tables, T_1 and T_2, each consisting of m space units, and two hash functions, $h_1, h_2 : U \rightarrow \{0, ..., m-1\}$. Every item $a \in S$ is stored either in bucket $h_1(a)$ of T_1 or in bucket $h_2(a)$ of T_2, but never in both. The hash functions h_i are assumed to behave as independent, random hash functions.*

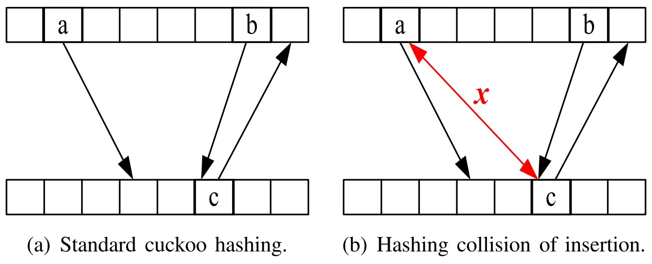

a
b
c
x
(a) Standard cuckoo hashing.
(b) Hashing collision of insertion.

Fig. 2. Cuckoo hashing structure.

Fig. 2 shows an example of cuckoo hashing. Initially, we have three items, a, b and c. Each item has two available positions in hash tables. If either of them is empty, an item will be inserted, as shown in Fig. 2a. When inserting a new item x, both of two available positions have been occupied and item x can "kick out" one existing item that will continue the same operations until all items can find positions as shown in Fig. 2b. If an endless loop takes place, the cuckoo hashing carries out a rehashing operation.

It is shown in [25] that if $m \geq (1+\epsilon)n$ for some constant $\epsilon > 0$ (i.e. two tables are almost half full), and h_1, h_2 are picked uniformly at random from an $(O(1), O(\log n))$-universal family, the probability of failing to arrange all items of dataset S according to h_1 and h_2 is $O(1/n)$.

The d-ary cuckoo hashing further makes an extension and allows each item to have $d > 2$ available positions.

Definition 4. (d-extension). *Each item a has d possible locations, i.e.,* $h_1(a), h_2(a), \ldots, h_d(a)$, *where* $d > 2$ *is a small constant.*

Cuckoo hashing provides flexibility for each item that is stored in one of $d \geq 2$ candidate positions. A property of cuckoo hashing is the increments of load factors in hash tables while maintaining query times bounded to a constant. Cuckoo hashing becomes much faster than chained hashing when increasing hash table load factors [16]. Specifically, performing the relocation of earlier inserted items to any of their other positions demonstrates the linear probing chain sequence upper bounded at d. When an item a is inserted, it can be placed immediately if one of its d locations is currently empty. Otherwise, one of the items in its d locations must be replaced and moved to another of its d choices to make room for a. This item in turn needs to replace another item out of one of its d locations. Inserting an item may require a sequence of item replacement and movement, each maintaining the property that each item is assigned to one of its d potential locations, until no further evictions are needed.

In practice, the number of hash functions can be reduced from the worst-case d to 2 with the aid of popular double-hashing technique. Its basic idea is that two hash functions h_1 and h_2 can generate more functions in the form $h_i(x) = h_1(x) + ih_2(x)$. In the cuckoo hashing, we define the i value belongs to the range from 0 to $d-1$. Therefore, more hash functions do not incur additional computation overheads while helping obtain higher load factors in the hash tables.

The cuckoo hashing is essentially a multi-choice scheme to allow each item to have more than one available hashing positions. The variants of random walk way [26] and d-ary extension [27] further demonstrated the flexibility and efficiency of cuckoo-driven method due to its simplicity. The items can hence "move" among multiple positions to achieve load balance and guarantee constant-scale complexity of operations. However, a simple combination, i.e., utilizing cuckoo hashing in LSH, will result in frequent operations of item replacement and potentially produce high probability of rehashing due to limited available buckets.

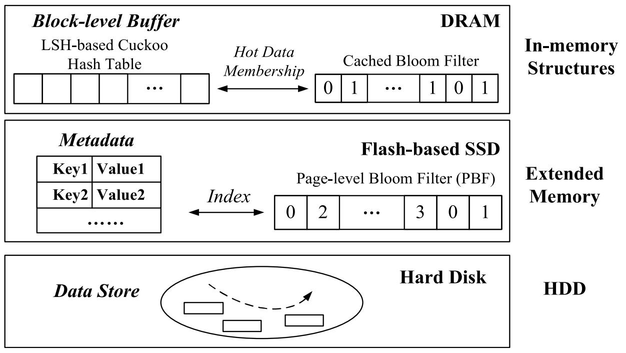


Fig. 3. The hybrid storage architecture.

3 NEST Design

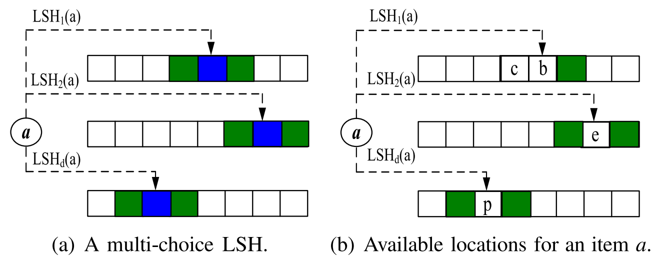
This section presents NEST scheme and illustrates the practical locality-aware operations, including item insertion, deletion and ANN query. The theoretical analysis of the NEST design, say rehash probability, is presented in the conference version [28].

3.1 The Architecture of Hybrid Storage Systems

The conventional DRAM, hard disk, and flash-based SSD have their own advantages and disadvantages. A critical challenge is to find a way to retain their advantages while hiding their disadvantages when managing data in a coordinated approach among the multi-level heterogeneous devices to maximize their performance. This challenge has been addressed routinely by exploiting workload locality. However, simple detection and exploitation of locality has its limitation, particularly with massive volumes of data that lack strong locality or easily detectable locality. We believe that this limitation can be overcome with the exploitation of the correlation among files and data items [29], with very low complexity, say LSH, to support the sequential read/write operations in both SSD and hard disk, while reducing the space overhead in main memory.

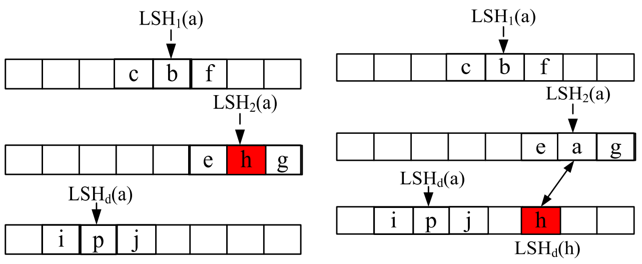
In order to improve storage system performance, we leverage the property of parallel operations in NEST. Specifically, the LSH based hash computation can be completed in parallel in multiple buckets. We exploit this salient property to efficiently support the read and write operations in SSD. To further handle large amounts of data, the NEST design provides a suitable "division of labor" in the hybrid storage hierarchy that consists of DRAM, flash-based SSD and hard disk. The data placement scheme improves the utilization of heterogeneous devices and obtain space savings.

NEST consists of a hybrid storage hierarchy offering semantic-aware data management as shown in Fig. 3. Specifically, in the DRAM, we implement the NEST



(a) A multi-choice LSH. (b) Available locations for an item a.

Fig. 4. The data structure.



(a) Hashing collisions for placing item a. (b) Moving item h to its another location.

Fig. 5. Cuckoo-based solution for hashing collisions.

components, i.e., LSH-based cuckoo hash table, as a block-level buffer to fast and accurately identify the semantic correlation residing in arriving files. LSH is able to exploit the locality of items by mapping similar items into the same hash buckets with a high probability. However, such a locality-aware storage causes load imbalance among the buckets of the hash table, which exacerbates the hash collisions. In order to address the imbalance problem, we leverage a cuckoo-based approach [16] that can balance the load among the LSH buckets via providing more than one available bucket. To deal with potential endless loops that result from cuckoo hashing collisions, the LSH-based cuckoo hash table in NEST serves as a temporary buffer and flushes the metadata into SSD and files into hard disk, when the capacity is full or endless loops occur. In the meantime, the NEST scheme avoids the extra stash used in ChunkStash [18] and hence alleviates the storage overheads.

NEST uses the flash-based SSD to maintain the metadata of semantically-correlated files in the key-value pairs. Since the LSH-based cuckoo hash table has identified and organized correlated files, we then write the metadata of these files into the SSD. The correlated metadata can be placed in the sequential addresses. These correlated files are conducive to sequential operations with a high probability, thus further enhancing the SSD performance. Moreover, in the hard disk, we sequentially write the correlated files into the addresses of hard disk, similar to the operations in the SSD, to achieve a much better performance than the random data layout.

3.2 In-Memory Data Structures

NEST takes into account the case for $d > 2$ due to two main reasons. One is that LSH requires multi-hashing computation to enhance the accuracy of locality aggregation. More hashing functions lead to higher aggregation accuracy. The other reason is that multi-hashing is more important and practical in real-world applications. When $d = 2$, after the first choice has been made to kick out an item, there are no further choices besides the other position. The special case ($d = 2$) appears much simpler. In the literature, the case where $d > 2$ remains less well understood. A natural approach is to use random selection among d choices, like random walk [26], which is adopted in NEST.

3.2.1 Multi-Choice Hashing

NEST structure uses a multi-choice hashing scheme to place items as shown in Fig. 4. It uses LSH to allow each item to have d available positions. The item can select an empty bucket to place. Furthermore, since LSH can faithfully maintain the locality characteristic of data, adjacent buckets exhibit correlation property. If no empty bucket is available, it may choose one from adjacent buckets to reduce or avoid endless loop.

Fig. 4a shows an example of the NEST structure. The blue bucket is the hit position by LSH computation and their adjacent neighboring buckets indicated by green color also exhibit data correlation for ANN query. Once all positions $LSH_i(a)$ are full, the item can choose an adjacent and empty bucket for storage. For instance, in Fig. 4b, if $d = 3$, $LSH_1(a)$, $LSH_2(a)$ and $LSH_3(a)$ have been occupied by other items b, e and p and in this case, the item a may choose the position of the right neighbor of $LSH_2(a)$.

Furthermore, if all neighbors of hit positions are full, we will carry out the "kicking out" operation to make a room for item a. After the probing operations on adjacent neighbors, the probability of endless "kicking out" in NEST is much smaller than the normal cuckoo hashing because we can take advantage of neighboring buckets to solve hash collision, as shown in Fig. 5. In the worst case, if such "kicking out" operation looking for empty position fails, we can carry out the rehashing operation as a final solution. The adjacent probing can significantly reduce or even avoid the occurrence of hash failing. Such scheme works well in NEST, but not in the standard cuckoo hashing. The reason is that items in adjacent buckets in NEST are locality-aware by using LSH computation, while they are uniformly distributed in the standard cuckoo hashing.

3.2.2 Bloom Filter Hierarchy

A standard Bloom filter [17] is a bit array of m bits representing a dataset $S = \{a_1, a_2, \ldots, a_n\}$ of n items. All bits in the array are initially set to 0, where a Bloom filter uses k independent hash functions $\{h_1, \ldots, h_k\}$ to map the set to the bit vector $[1, \ldots, m]$. Each hash function h_i maps an item a to one of the m-array positions equally likely. An item a is considered a member of set S with a very high probability if all $h_i(a)$ $(1 \leq i \leq k)$ are set to 1. Otherwise, a is definitely not in the set S. The membership query in a Bloom filter possibly introduces a false positive, indicating that an item a is a member of set S although it in fact is not. The false positive rate of a standard Bloom filter is $f_{StandardBF} \approx (1 - e^{-\frac{kn}{m}})^k$ when the Bloom filter has m bits and k hash functions for storing n items. This probability is minimized to $(1/2)^k$ or $(0.6185)^{m/n}$ when $k = (m/n) \ln 2$.

In order to support fast membership query and obtain space savings, we build a Bloom filter hierarchy in the

DRAM and SSD layers. Specifically, DRAM maintains a cached Bloom filter that only stores the membership information for the "hot spot" files (i.e., files with very high locality as measured by LSH), incurring very limited space overhead. At the SSD layer, a PBF is stored to maintain the membership information of all files. The page-level organization in PBF facilitates the garbage collection in SSD. Since the in-memory cached Bloom filter can satisfy most membership queries, the hierarchy design can significantly reduce the accesses to SSD. In addition, in order to support deletion operations in Bloom filters, we adopt a form of counting Bloom filter by replacing a bit in the tradition Bloom filter with a 4-bit counter to satisfy most real-world applications [30].

3.2.3 *Practical Operations*

We describe practical locality-aware operations of NEST to support item insertion, ANN query and item deletion. These operations can be efficiently implemented in the hybrid storage architecture via exploring and exploiting the property of parallel operations. Moreover, due to the space savings in NEST, we can further improve the caching and indexing performance.

The insertion operation needs to place items in hashed or adjacent empty buckets to obtain load balance. The recursive insertion algorithm consists of three parts. We need to first find an empty position for the new item a. If no hash collisions occur, this item can be directly inserted. If there is no empty bucket among the positions hit by LSH computation, NEST needs to probe adjacent buckets of $LSH_i(a)$. The third part employs the "kicking out" operation to help item a to find an empty bucket if the first two parts fail to do so.

The key question in item insertion is which item to be moved if d potential positions for a newly inserted item a are occupied. A natural approach in practice is to pick one of the d buckets randomly, replace the item b at that bucket with a, and then try to place b in one of its other $(d-1)$ bucket positions. If all of the buckets for b are full, choose one of the other $(d-1)$ buckets (other than the one that now contains a, to avoid the obvious loop) randomly, replace the item in the chosen bucket with b, and repeat the same process. At each step (after the first), we place the item whenever an empty bucket is found, or else randomly exchange the item with one of $(d-1)$ choices. We refer to this process as the random-walk insertion method for cuckoo hashing.

The ideal scenario of inserting an item is that there is no visit to any hash table bucket more than once. Each item can hence locate in a certain bucket without kicking out other items. Once the insertion procedure returns a previously visited bucket, the behavior may lead to endless loop that requires relatively high-cost rehashing operations. We study the probability of rehashing occurrence. In practice, the rehash occurs if an item insertion cannot stop, i.e. no vacant bucket, after $MaxLoop$ steps. The $MaxLoop$ is a constant to be set application-related. In standard cuckoo hashing, let $MaxLoop = \lambda \log n$ for n items and λ is an approximately chosen constant [16]. We take into account the s-stable distribution in the probability analysis. When $s = 2$, the 2-stable normal distribution has the density function $g(x) = \frac{e^{-x^2/2}}{\sqrt{2\pi}}$.

The ANN query needs to obtain approximate neighbors to a query point q. NEST can complete the ANN query operation in a simple way. The ANN query algorithm allows the query to obtain totally $d \times (2\Delta + 1)$ items, thus requiring accesses to memory for d times. Each access needs to probe $2\Delta + 1$ buckets that are stored and at most $2\Delta + 1$ non-empty buckets provide items. The final set contains correlated data items to satisfy the ANN query request.

In the item deletion, we need to find the item to be deleted and then remove it from the bucket of hash table. Assume that the deletion operation is to remove an existing item. If an item to be deleted does not exist, NEST will return an error.

3.3 The Extended Memory using Flash-based SSD Design

In the storage system implementation, conventional DRAM and hard disk based storage architecture shows the inefficiency to offer large-scale approximate query services. First, conventional file systems generally leverage hard disks for persistent storage, which recently obtain the performance improvements. However, the performance gap between hard disks and other system components, in fact, widens [31]. On the other hand, the availability of memory space and the requirements from real-world applications also build a gap. The use of many DRAM devices is not cost-effective. For example, a 128GB RamSan supports fewer than 2.5 hash-based operations per second per dollar [32]. Moreover, real-world cloud applications need large amounts of space. For example, in identifying and removing duplicate contents, we can maintain the data fingerprints in DRAM. For analyzing 100TB data, the storage of fingerprints alone has to consume more than 320GB memory space, as well as the need to offer 10,000 lookups, insertions and updates per second [33]. Hence, we need to select a proper division of labor in the system components.

In order to bridge the gap, the flash-based SSD is used to obtain a suitable tradeoff between DRAM and hard disk in terms of both performance and costs. SSD has multiple advantages over hard disk in terms of low random read latency and energy consumption, and high physical durability. In the meantime, SSD exhibits space and cost advantages over DRAM. However, due to the unique physical features, we cannot fully replace DRAM or hard disk with the SSD. In practice, existing flash-based SSD devices generally show the little implementation details to consumer devices and use one-size-fits-all Flash Translation Layer (FTL). For example, if the applications are already log-structured, the internal wear-leveling functionality becomes useless. This is also the reason why Google is exploring a design for large data caches by using the FIFO replacement policy, which demonstrates a perfect match for flash memory [34]. Therefore, a reasonable data placement is needed in the flash-based SSD to facilitate its sequential operations and exploit its unique features.

As the extended memory, the flash-based SSD architecture consists of flash chips and FTL as shown in Fig. 6. In a flash chip, SSD maintains data in an array of flash blocks. Each block contains 32-64 pages and a page is the smallest unit of read and write operations. Moreover, read and write

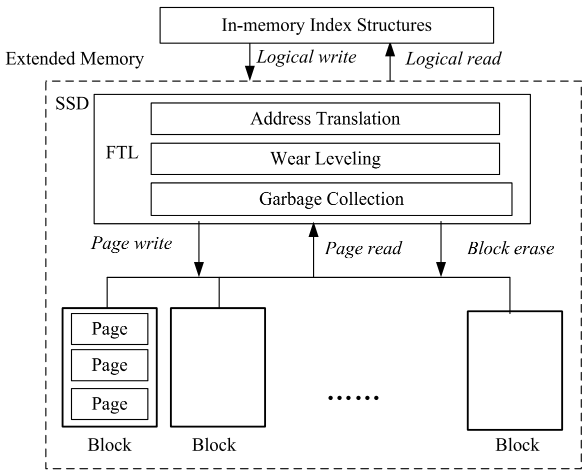


Fig. 6. An SSD architecture for the extended memory.

operations exhibit asymmetric performance, and the write-after-erase property does not support the in-place update operation. A flash page needs to be erased before it is written. Furthermore, read and write operations are performed in the page-level, while erase operation is performed in the block-level. On the other hand, FTL is a block-device software layer that simulates NAND flash as a hard disk. The FTL is able to support address mapping between a logical address in the file system to a physical address in the flash. By offering a disk-like interface, FTL is an intermediate software layer inside an SSD. FTL receives logical read and write commands from the applications and then transforms them to the internal commands in the flash.

In order to provide efficient data management, we need to carefully design the data allocation scheme. The function of allocation scheme is to decide the mapping between physical page and logical page. When a new write request arrives, the allocation scheme needs to select a free physical page by considering the idle/busy states of channels and chips, the erased count of blocks and the priority order of parallelism. Furthermore, there exist static and dynamic allocation schemes. Static allocation assigns a logical page to a pre-defined channel, package, chip, die and plane. Dynamic allocation assigns a logical page to any free physical page in the entire SSD.

In order to reduce write traffic to flash memory and obtain space savings in SSD, NEST eliminates unnecessary duplicate writes, which further improves the efficiency of garbage collection and wear-leveling. The sizes of sequential requests are generally very small and the basic operation unit in flash is a page. The internal management policies in SSDs, say the mapping policy, are also designed in the unit of pages.

Fig. 7 shows the implementation of the SSD design. The SSD design is event-driven, modularly structured, and multi-tiered, thus efficiently supporting the performance evaluation. This prototype serves as an SSD hardware platform and carries out the FTL schemes, allocation schemes, buffer management algorithms and request scheduling algorithms. There are three tiers in the SSD prototype, including the buffer and request-scheduling module at the top, the FTL and allocation module in the middle, and the low-level hardware platform module at the bottom.

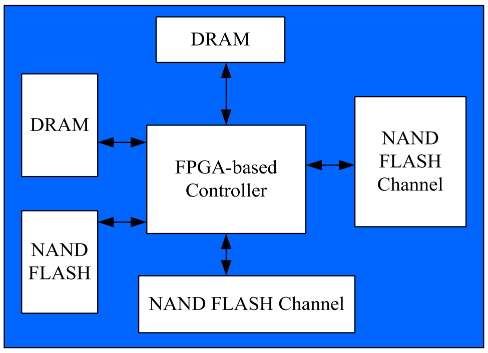


Fig. 7. The implementation of SSD prototype.

Specifically, the top module supports buffer organization and scheduling requests. In the middle module, the FTL sub-module carries out the FTL scheme, and the allocation sub-module supports the allocation between the logical pages and the physical pages. The bottom module supports all the flash operations based on the Open NAND Flash Interface (ONFI) 2.2 Specification [35].

The preliminary design has been used in HAT [36] and SSDsim [37]. In implementing the prototype of NEST, we use an FPGA chip as the controller, multiple flash chips as two independent channels, and multiple 16MB DRAM chips to store the mapping table and data buffer. The hardware prototype supports the buffer management and allocation schemes. Moreover, the data allocation scheme determines how to select free physical page to contain logical page being written to the SSD. To locate a particular physical page, we need to identify the channel address and package address, in addition to the chip address, die address, plane address, block address and page address. We input the configuration parameters and the request streams into the hardware prototype. Hence, we can obtain the performance results, such as waiting time, processing time, response time of each request, total erasure count and buffer hit count.

4 Performance Evaluation Configurations

In this section, we present the experiment setup in terms of system platform, the used traces and configurations.

4.1 Experiment Environments

We implement NEST in a large-scale cloud computing environment that consists of 100 servers, each of which is equipped with Intel 2.0GHz dualcore CPU, 4GB DRAM, 250GB disk and 1000PT quad-port Ethernet network interface card. The prototype is developed in the Linux kernel 2.4.21 environment and all functional components in NEST are implemented in the user space. We make use of two traces, i.e. LANL [20] and Microsoft [21] from real-world applications, to examine the system performance.

We describe the characteristics of real-world traces for our experiments.

- *LANL*. Los Alamos National Laboratory (LANL) recently released multiple sets of data [20]. These data contain a metadata walk of some of the NFS file systems. These metadata demonstrate the information about files and the hierarchical structure in which they archived files were stored. The data set is about 19GB and consists of roughly 112 million lines of archive data and roughly 9 million lines of home/project space data. The attributes of these data include unique ID, file sizes (in bytes), creation time, modification time, block sizes (in bytes) and the paths to files.

- *Microsoft*: From 2000 to 2004, metadata traces [21] have been collected from more than 63,398 distinct file systems that contain 4 billion files. This is the largest set of file-system metadata ever collected. The 92GB-size trace has been published in SNIA [38]. The multiple attributes of data in the traces include file size, file age, file-type frequency, directory size, namespace structure, file-system population, storage capacity and consumption, and degree of file modification. The access pattern studies [21] further show the data locality properties in terms of read, write and query operations.

To investigate the real performance when using two traces, we randomly allocate the available data segments/snapshots among all 100 servers in a round-robin way. Moreover, a client leverages the LSH-based NEST for fast data retrieval. A client captures the system operations from these traces and then delivers the query requests to servers. Both clients and servers use multiple threads to exchange messages and data via TCP/IP. The IP encapsulation technique helps forward the query requests.

Query requests are generated from the attribute space of above typical traces and are randomly selected by considering 1000 uniform or 1000 Zipfian distributions. We set the zipfian parameter H to be 0.75. 2000 query requests constitute the query set and we examine the query accuracy and latency. In practice, ANN query can be interpreted as querying multiple nearest neighbors by first identifying the closest ones to the queried point, and then measuring their distances. If the distance is smaller than a metric, we say the queried point is an approximate member to dataset S. Moreover, in order to construct suitable ANN queries, the methodology of statistically generating random queries in a multi-dimensional space leverages the file static attributes and behavioral attributes that are derived from the available I/O traces [21], [29]. For example, an ANN query in the form of (11:20, 26.8, 65.7, 6) represents a search for the top-6 files that are closest to the description of a file that is last revised at time 11:20, with the amounts of "read" and "write" data being approximately 26.8MB and 65.7MB, respectively. The members in this tuple will be further normalized in the LSH based computation. In addition, due to space limitation, we only exhibits the performance of querying top-6 nearest neighbors. Experiments for querying more nearest neighbors have been done and results show similar observations and conclusions.

The load factor in hash tables can affect the response to queries. Fortunately, cuckoo hashing can have a higher load factor in hash tables without incurring too much delay to queries. It has been shown mathematically that with three or more hash functions and with a load factor up to 91 percent, insertion operations can be done in an expected constant time. We hence set a maximum load factor of 90 percent for the cuckoo hashing implementation. Note that our comparison does not imply, in any sense, that other structures are not suitable for their original design purposes. Instead, we intend to show that NEST is a better scheme for ANN query in large-scale cloud computing applications.

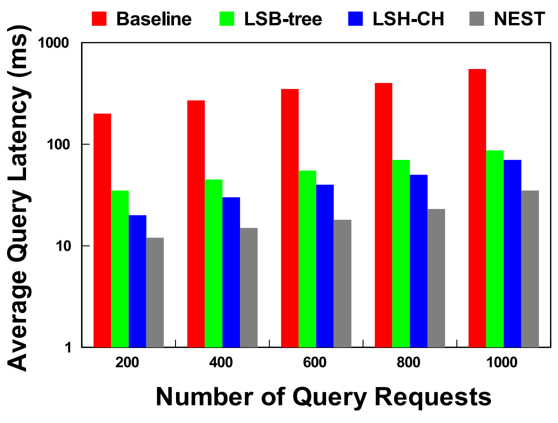
In order to obtain accurate parameters, we use the popular sampling method that is proposed in LSH statement [14], [15] and practical applications [12], [19]. "*Approximate Measure* $\chi = ||p_1^\star - q||/||p_1 - q||$" evaluates the query quality for queried point q, where $p_1^\star$ and p_1 respectively represent the actual and searched nearest neighbors by computing their Euclidean distances. With the aid of this sampling technique, we determine the R values to be 550 and 700 respectively for *LANL* and *Microsoft metadata* traces. In addition, a rehashing in insertion operations may incur the relocation of items. Based on sampling results, we recommend to use 10 LSH functions to obtain a suitable tradeoff between computation complexity and the number of relocation. We also set $\omega = 0.85$, $M = 10$ and $\Delta = 5$ in the experiments to guarantee high query accuracy.

4.2 System Configurations

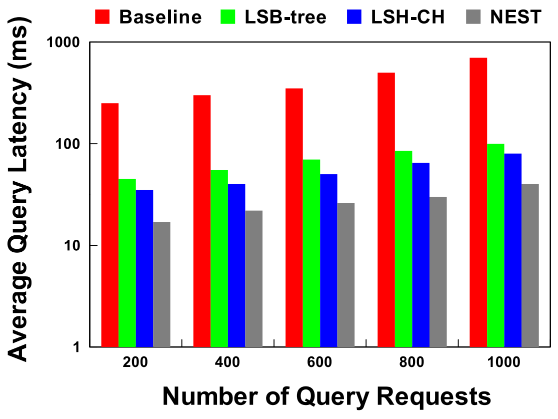
In order to comprehensively evaluate the performance, we leverage two system configurations, i.e., DRAM+HDD and hybrid (DRAM+SSD+HDD). The two configurations use DRAM and HDD (hard disk). Their difference is the use of the SSD implementation prototype as described in Section 3.3. Since the two configurations have different features, we compare their performance with different state-of-the-art schemes.

For the DRAM+HDD configuration, we compare the NEST performance with *LSB-tree* [19], LSH with cuckoo hashing (*LSH-CH*) and *Baseline* schemes. These schemes for performance comparisons are not only state-of-the-art but also their correlation with our work. Specifically, conventional cuckoo hashing techniques can only support exact-matching query, but not approximate query. We hence select the LSB-tree [19] that can support ANN query. LSB-tree is the most recent work that can obtain high-quality ANN query result. It uses Z-order method to produce associated values that are indexed via an auxiliary data structure, i.e., a conventional B-tree. It addresses the endless loop by using an auxiliary data structure as a stash. The *Baseline* approach utilizes the basic brute-force retrieval to identify the closest point in the dataset. It determines an approximate membership by computing the distance between the queried point and its closest neighbor.

For the hybrid (DRAM+SSD+HDD) configuration, we compare NEST with a number of state-of-the-art schemes in terms of hybrid storage systems, including BufferHash [33], Buffer Bloom Filters [39], ChunkStash [18] and Berkeley-DB [40]. The metrics include lookup and insert latencies. In order to comprehensively and fairly evaluate NEST's performance, we consider the following tri-layer system configuration. This settings consist of in-memory buffer with SSD-based metadata index and hard disk based data store. Moreover, both DRAM and SSD execute the Least Recently Used (LRU) algorithm to maintain the most recently visited files. If an item is replaced in DRAM, it will be placed into the

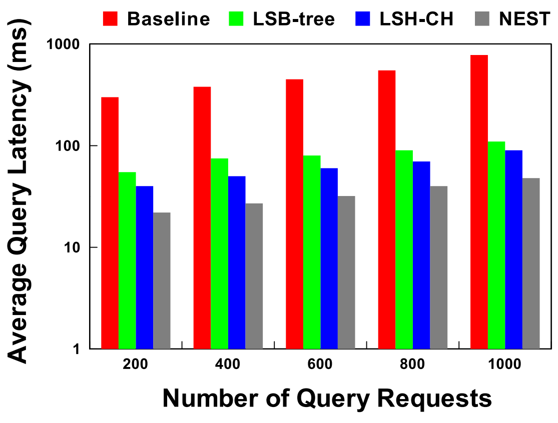


(a) Uniform.

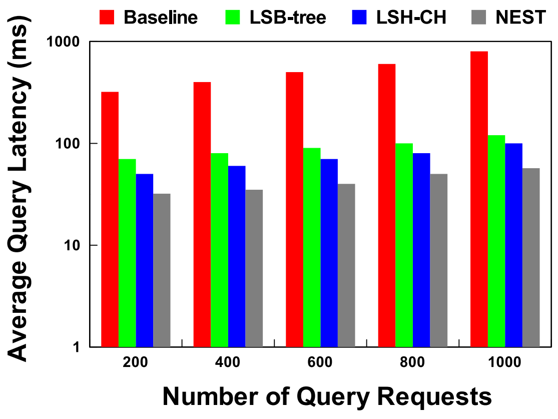


(b) Zipfian.

Fig. 8. Average ANN query latency using LANL trace.



(a) Uniform.



(b) Zipfian.

Fig. 9. Average ANN query latency using Microsoft metadata trace.

SSD. Furthermore, if an item is replaced in SSD, it will be then placed into the hard disk.

5 Performance Results

We present the performance results of using two system configurations, including DRAM+HDD and hybrid storage, in terms of multiple metrics.

5.1 DRAM+HDD

We show advantages of NEST over Baseline, LSH-CH and LSB-tree approaches by comparing their experimental results in terms of query latency, accuracy, space overhead, I/O cost and rehash probability.

5.1.1 ANN Query Latency

Figs. 8 and 9 respectively show the ANN query latency when using LANL and Microsoft metadata traces. We observe that NEST, LSH-CH and LSB-tree obtain significant improvements upon Baseline approach due to hashing computation, rather than linearly brute-force searching. NEST further obtains on average 36.5 and 42.8 percent shorter running time than LSB-tree respectively in uniform and zipfian distributions. Moreover, compared with LSB-tree, LSH-CH obtains on average 8.51 and 9.45 percent latency reduction. The main reason is that LSB-tree needs to run Z-order codes and retrieve a B-tree with $O(\log n)$-scale complexity after the hashing computation.

NEST and LSH-CH can carry out constant-scale complexity even in the worst case. In addition, as described in Section 4.1, since the simple combination of LSH and cuckoo hashing, i.e., LSH-CH, addresses the infinite loop by using an auxiliary stash, the queries in LSH-CH have to navigate the auxiliary space to find possible approximate items, thus incurring a larger latency than NEST.

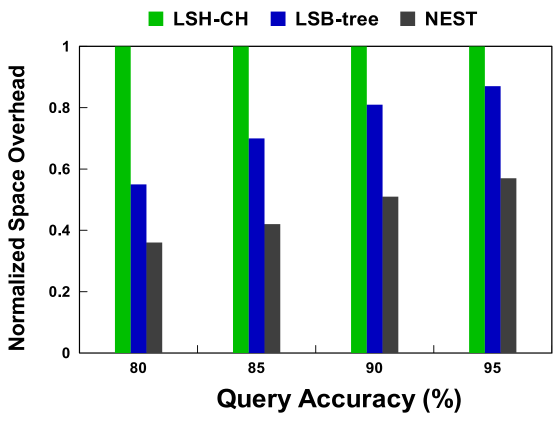
5.1.2 Space Overhead

Fig. 10 shows the space overhead normalized to LSH-CH. We observe that NEST can obtain significant space savings. Compared with the space overhead of LSH-CH that has an auxiliary stash, the average savings from NEST are 51.6 percent in the LANL trace and 47.9 percent in the Microsoft trace.

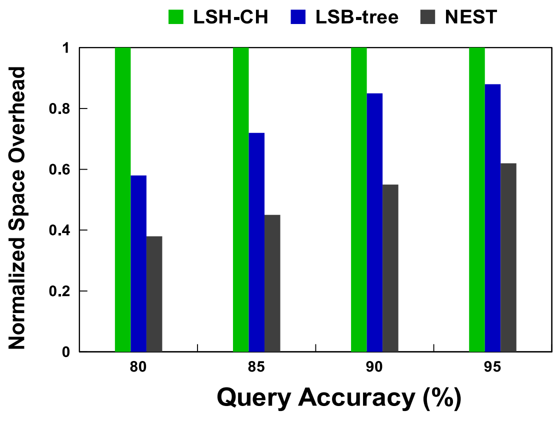
Moreover, LSB-tree needs to keep additional Z-order codes in a B-tree to facilitate ANN query and thus consumes larger space than NEST. The smallest space overhead of NEST is the result of cuckoo hashing usage to achieve load balance among buckets of hash tables. The limited and flat hash-based addressing in NEST also helps to improve the space utilization.

5.1.3 I/O Costs

We take into account I/O costs by examining the access times that include the visits on high-speed memory and low-speed disk. Figs. 11 and 12 respectively illustrate the total I/O costs for approximate queries when using two

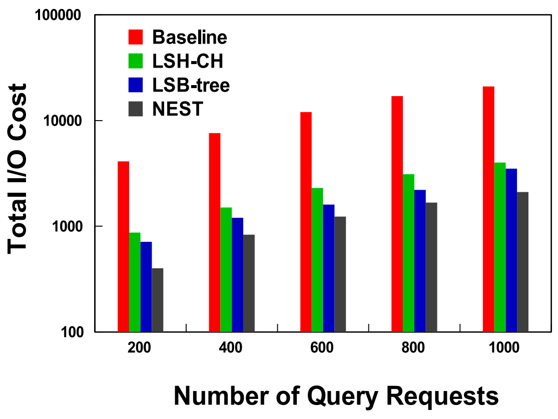


(a) LANL trace.

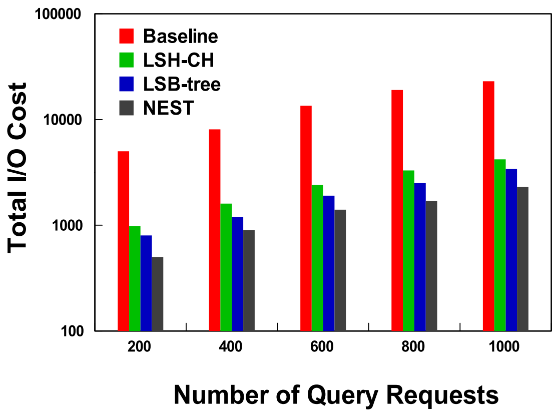


(b) Microsoft metadata trace.

Fig. 10. Normalized space overhead.



(a) Uniform.



(b) Zipfian.

Fig. 11. Total I/O costs for ANN query using LANL trace.

typical traces. The Baseline approach requires the largest number of accesses since it needs to probe the entire dataset. LSH-CH needs to examine the auxiliary space and hence incurs more costs than LSB-tree and NEST.

Furthermore, performing the index on a B-tree makes LSB-tree produce 1.52 and 1.76 times more visits (on the average) than NEST respectively in LANL and Microsoft traces. NEST needs to probe limited and deterministic locations to obtain query results and its operations of constant-scale complexity significantly reduce the costs of I/O accesses.

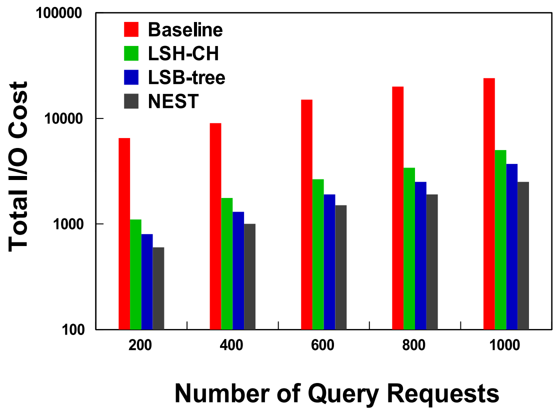
5.1.4 ANN Query Accuracy

We examine query accuracy of NEST and other three approaches by using the metric of average "*Approximate Measure*" in the LANL and Microsoft traces by using uniform and zipfian query requests as shown Figs. 13 and 14. The Baseline uses linear searching on the entire dataset and causes very long query latency, which leads to potential inaccuracy of query results due to stale information of delayed update. Its slow response to update information in multiple servers incurs false positives and false negatives, and hence greatly degrades the query accuracy. The average query accuracy of NEST are 91.2 and 90.5 percent respectively in LANL and Microsoft traces, which are higher than the percentages of 83.6 and 82.7 percent in LSB-tree, and 80.6 and 79.3 percent in LSH-CH. Such improvement comes from the adjacent probing operation in NEST to guarantee query accuracy. Moreover, LSH-CH consumes relatively smaller accuracy than LSB-tree since the stash in the former is not locality-aware for the approximate query. We also observe that the uniform distribution receives higher query accuracy than the zipfian because items in the latter are naturally closer and it is more difficult to clearly identify them.

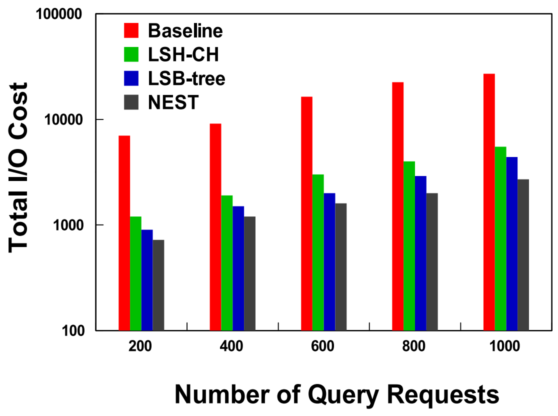
5.1.5 Rehash Probability

Hash collisions are unavoidable for hash functions. Without exception, NEST has a chance for rehashing when hash collisions occur. Surprisingly, the rehashing probability has been reduced significantly. Fig. 15 shows the experimental results by comparing NEST with the standard cuckoo hashing, when we carry out item insertions. An insertion failure means that an endless loop takes place. The average failure probabilities of NEST are very small, 1.72×10^{-6} in the LANL trace and 1.85×10^{-6} in the Microsoft trace. In other words, a failure only occurs when millions of insertions are done. In contrast, the standard cuckoo hashing has a much higher failure probability and we can witness a failure when inserting thousands of items. Such significant decrement of failure rate is because NEST allows items to be inserted into adjacent and correlated buckets.

The extensive experiments demonstrate NEST has great advantages over existing work in terms of query latency, accuracy, space overhead, and rehash probability. In particular, a simple combination of LSH and cuckoo hashing, say LSH-CH, does not work well. The chosen data traces show that the stronger locality a data trace demonstrates, the

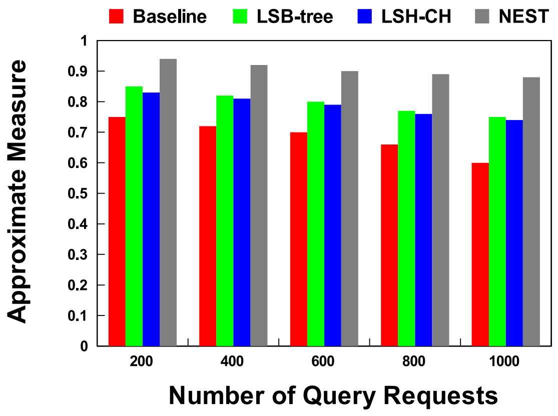


(a) Uniform.

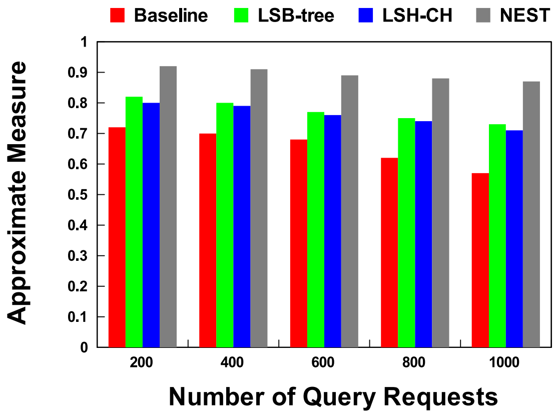


(b) Zipfian.

Fig. 12. Total I/O costs for ANN query using Microsoft trace.



(a) Uniform.



(b) Zipfian.

Fig. 13. ANN query accuracy using LANL trace.

higher probability hash collisions exhibit. NEST can efficiently utilize the locality of datasets to support approximate query and achieve load-balance while significantly alleviate the system performance degradation due to hash collisions by employing locality-aware algorithms.

5.2 The Hybrid Storage (DRAM+SSD+HDD)

Figs. 16 and 17 show the lookup latency under the two workloads respectively. The latency results are shown in the form of cumulative distribution function (CDF) of requests. We observe that NEST can satisfy more than 98 percent query requests within 0.1ms (i.e., 98 percent query requests have a latency of 0.1ms or less), much faster than other schemes. The main reason is that NEST employs a small-size cached Bloom filter in DRAM to maintain the membership of "hot spot" files with an $O(1)$ lookup complexity. Even when a hit miss occurs, the page-level Bloom filter in SSD can provide accurate results by storing the entire dataset. The Bloom filter hierarchy helps significantly improve the lookup performance.

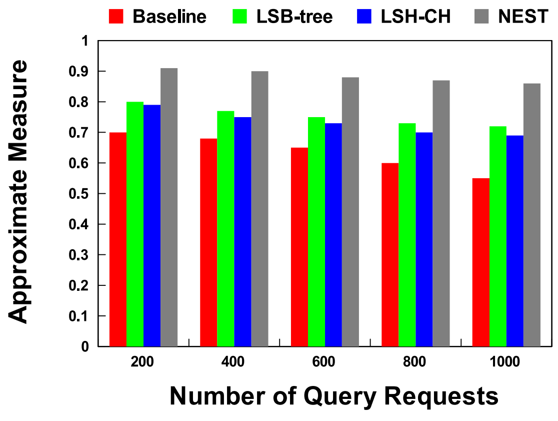
Figs. 18 and 19 show the latency of executing the insert operations. The NEST scheme can complete on average 96.8 percent inserts within 0.65ms and 99.6 percent within 1ms. The latency savings come from many sequential operations in both SSD and hard disk. The LSH-based cuckoo hash table identifies the useful semantic correlation residing among files to aid the organization of sequential operations.

6 Related Work

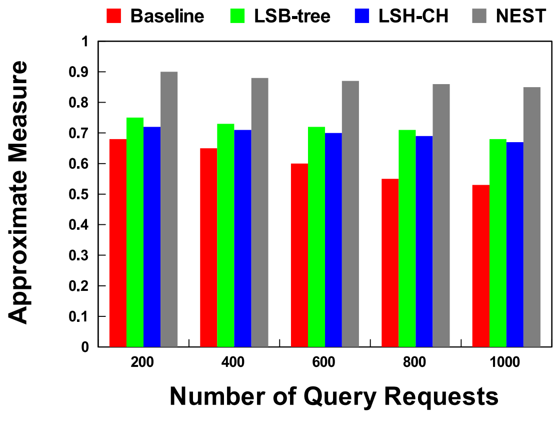
Cloud computing [41], [42] is exhibiting its great potential abilities to provide high quality of services. The cloud computing environments usually contain massive data that critically require fast and accurate retrieval to support intelligent and adaptive cloud services. The ANN query can provide approximate service matching and becomes one of the most critical services in cloud computing [43], [44]. The proposed NEST can support ANN service by using computation with the aid of cuckoo hashing for load balance.

Locality Sensitive Hashing introduced by Indyk and Motwani in [14] has been successfully applied in approximate queries of vector and string spaces. Existing variants include distance-based hashing [45], multi-probe LSH [15] and bounded LSH [9]. Distance-based hashing [45] extends conventional LSH into arbitrary distance measures by taking statistical observation from sample data. Multi-probe LSH [15] checks the hashed buckets more than once to support high-dimensional similarity search and improve indexing accuracy based on statistic analysis. Most of existing LSH-based designs have to consume a large storage space to maintain multiple hash tables to improve the accuracy of approximate queries.

Cuckoo hashing [16], [46] can provide constant-scale complexity in the worst case. Random walk cuckoo hashing [26] demonstrated that for sufficiently large d with high probability the graph structure of the resulting cuckoo graph is such that, regardless of the staring vertex, the



(a) Uniform.



(b) Zipfian.

Fig. 14. ANN query accuracy using Microsoft trace.

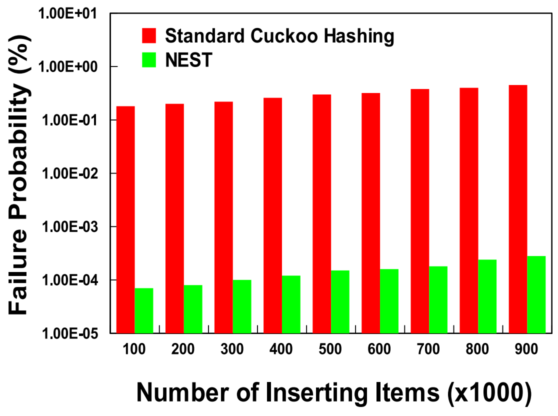


Fig. 15. Insertion failure probability due to endless loops.

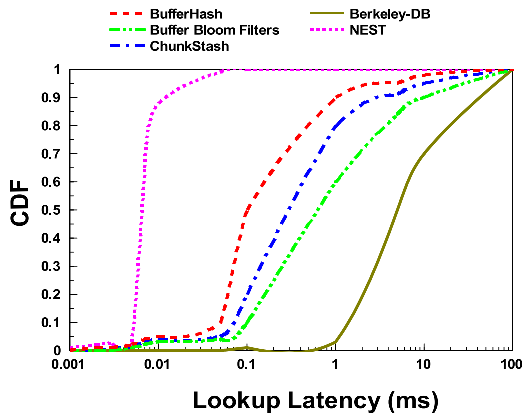


Fig. 16. Lookup latency under the LANL workload.

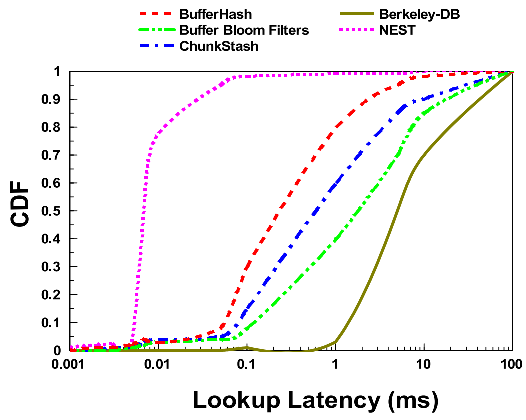


Fig. 17. Lookup latency under the Microsoft workload.

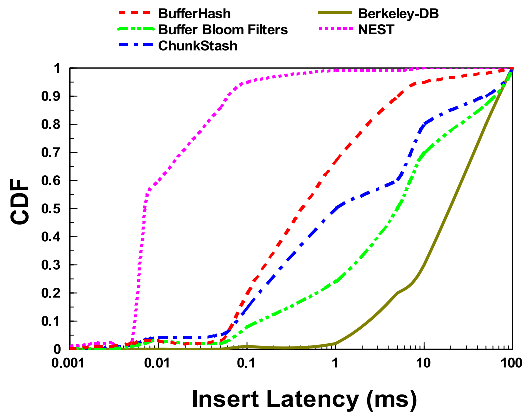


Fig. 18. Insert latency under the LANL workload.

random-walk insertion method will reach a free vertex in polylogarithmic time with high probability. Authors in [47] present the methods of greatly improving the failure probability bounds for a large class of cuckoo hashing variants by using only a constant amount of additional space. The d-ary Cuckoo Hashing [27] further generalizes the cuckoo hashing to show how to yield a simple hash table data structure that stores n elements in $(1+\epsilon)n$ memory cells, for any constant $\epsilon > 0$. This structure indicates that cuckoo hashing can obtain better performance in practical situations. For example, at $d = 4$, it achieves 97 percent space utilization and at 90 percent space utilization, insertion requires only about 20 memory probes on the average. In history-independent cuckoo hashing [48], the memory representation at any point in time yields no information on the specific sequence of insertions and deletions that led to its current content, other than the content itself. An improved cuckoo hashing [49] uses tables of different size or granted both hash functions with access to the whole table, thus enhancing the probability that the first hash function hit an empty cell.

Existing solutions to data-index bottleneck problem, such as BufferHash [33], Buffer Bloom Filters [39], BloomFlash [50], ChunkStash [18], Multiple Bloom Filters [51] and Berkeley-DB [40], attempt to address the above challenges from the viewpoints of data structures (e.g., Bloom filters in SSD). Different from these state-of-the-art solutions, NEST exploits the semantic correlation among files by using a locality-aware data management scheme in the hybrid storage

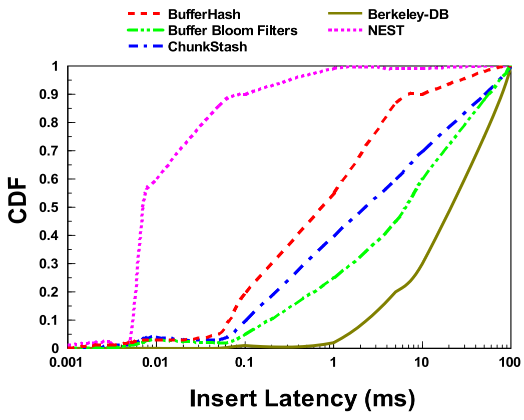


Fig. 19. Insert latency under the Microsoft workload.

hierarchy to facilitate sequential operations and obtain space savings. NEST is able to carry out a near-optimal data placement in system implementations and support a proper "division of labor" among the heterogeneous devices to obtain significant performance improvements.

The above existing work motivates our design of NEST that makes further improvements upon them. The NEST is a novel structure to use LSH to map approximate items into the same or adjacent buckets and obtain load balance by using cuckoo hashing without compromising query performance.

7 Conclusion

This paper presented a novel locality-aware hashing scheme, called NEST, for large-scale cloud computing applications. The new design of NEST provides solutions to two challenges in supporting approximate queries, namely, locality-aware and balanced storage among cloud servers. NEST uses an enhanced LSH to store one item in one bucket, exploited by cuckoo hashing to achieve load-balance. The LSH in NEST, in turn, can significantly reduce the probability of the loop in cuckoo hashing by allowing adjacent buckets to be locality-aware and correlated items to be placed closely with a high probability. We then obtain a fast and limited flat addressing, which is $O(1)$ complexity even in the worst case for ANN query, while conventional vertical addressing structures (e.g., the linked lists) for LSH have $O(n)$ complexity. NEST hence can efficiently support ANN query service in large-scale cloud computing applications. NEST achieves a proper division of labor in the heterogeneous storage hierarchy that consists of DRAM, SSD and hard disk. The NEST design supports fast queries with the aid of a Bloom filter hierarchy and is able to organize sequential operations in both SSD and hard disk to obtain performance improvements. NEST is practical and easy to implement due to its simplicity.

The future work will consider possible use in real-world industrial applications. The source codes of NEST are available for public use at the website http://cs.hust.edu.cn/stlab/csyhua/nestcode.zip.

Acknowledgments

This work was supported in part by National Natural Science Foundation of China (NSFC) under Grant 61173043, National Basic Research 973 Program of China under Grant 2011CB302301, NSFC 61025008, and NSERC Discovery Grant 341823, US National Science Foundation Award 1116606. The authors greatly appreciate Los Alamos National Lab (LANL) and Microsoft for providing large-scale datasets. This is an extended version of our manuscript published in the Proceedings of the 32nd IEEE International Conference on Computer Communications (INFOCOM), 2013, pages: 1327-1335.

References

[1] J. Gantz and D. Reinsel, "2011 digital universe study: Extracting value from chaos," *Int. Data Corp.*, Jun. 2011.
[2] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "A view of cloud computing," *Commun. ACM*, vol. 53, no. 4, pp. 50–58, 2010.
[3] S. Bykov, A. Geller, G. Kliot, J. Larus, R. Pandya, and J. Thelin, "Orleans: Cloud computing for everyone," in *Proc. ACM Symp. Cloud Comput.*, 2011.
[4] S. Wu, F. Li, S. Mehrotra, and B. Ooi, "Query optimization for massively parallel data processing," in *Proc. ACM Symp. Cloud Comput.*, 2011.
[5] Q. Liu, C. Tan, J. Wu, and G. Wang, "Efficient information retrieval for ranked queries in cost-effective cloud environments," in *Proc. INFOCOM*, 2012, pp. 2581–2585.
[6] C. Wang, K. Ren, S. Yu, and K. Urs, "Achieving usable and privacy-assured similarity search over outsourced cloud data," in *Proc. INFOCOM*, 2012, pp. 451–459.
[7] Y. Hua, B. Xiao, and J. Wang, "BR-tree: A Scalable prototype for supporting multiple queries of multidimensional data," *IEEE Trans. Comput.*, no. 12, pp. 1585–1598, Dec. 2009.
[8] F. Leibert, J. Mannix, J. Lin, and B. Hamadani, "Automatic management of partitioned, replicated search services," in *Proc. ACM Symp. Cloud Comput.*, 2011.
[9] Y. Hua, B. Xiao, D. Feng, and B. Yu, "Bounded LSH for similarity search in peer-to-peer file systems," in *Proc. ICPP*, pp. 644–651, 2008.
[10] N. Cao, C. Wang, M. Li, K. Ren, and W. Lou, "Privacy-preserving multi-keyword ranked search over encrypted cloud data," in *Proc. INFOCOM*, 2011, pp. 222–233.
[11] J. Li, Q. Wang, C. Wang, N. Cao, K. Ren, and W. Lou, "Fuzzy keyword search over encrypted data in cloud computing," in *Proc. INFOCOM*, 2010, pp. 1–5.
[12] Y. Hua, B. Xiao, B. Veeravalli, and D. Feng, "Locality-sensitive Bloom filter for approximate membership query," *IEEE Trans. Comput.*, no. 6, pp. 817–830, Jun. 2012.
[13] M. Bjorkqvist, L. Y. Chen, M. Vukolic, and X. Zhang, "Minimizing retrieval latency for content cloud," in *Proc. INFOCOM*, 2011, pp. 1080–1088.
[14] P. Indyk and R. Motwani, "Approximate nearest neighbors: Towards removing the curse of dimensionality," in *Proc. Symp. Theory Comput.*, pp. 604–613, 1998.
[15] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li, "Multi-probe lsh: Efficient indexing for high-dimensional similarity search," in *Proc. VLDB*, pp. 950–961, 2007.
[16] R. Pagh and F. Rodler, "Cuckoo hashing," in *Proc. ESA*, pp. 121–133, 2001.
[17] B. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, 1970.
[18] B. Debnath, S. Sengupta, and J. Li, "ChunkStash: Speeding up inline storage deduplication using flash memory," in *Proc. USENIX ATC*, 2010.
[19] Y. Tao, K. Yi, C. Sheng, and P. Kalnis, "Quality and efficiency in high-dimensional nearest neighbor search," in *Proc. SIGMOD*, 2009.
[20] Los Alamos National Lab (LANL). File System Data. [Online]. Available: http://institute.lanl.gov/data/archive-data/
[21] N. Agrawal, W. Bolosky, J. Douceur, and J. Lorch, "A five-year study of file-system metadata," in *Proc. FAST*, 2007.
[22] A. P. de Vries, N. Mamoulis, N. Nes, and M. Kersten, "Efficient k-NN search on vertically decomposed data," in *Proc. ACM SIGMOD*, 2002, pp. 322–333.
[23] J. Gan, J. Feng, Q. Fang, and W. Ng, "Locality-sensitive hashing scheme based on dynamic collision counting," in *Proc. ACM SIGMOD*, 2012, pp. 541–552.
[24] R. Panigrahy, "Entropy based nearest neighbor search in high dimensions," in *Proc. Symp. Discrete Algorithms*, pp. 1186–1195, 2006.

[25] R. Pagh, "On the cell probe complexity of membership and perfect hashing.," in *Proc. Symp. Theory Comput.*, 2001.
[26] A. Frieze, P. Melsted, and M. Mitzenmacher, "An analysis of random-walk cuckoo hashing," in *Proc. Approx., Randomization Comb. Optim., Algorithms Techn.*, pp. 490–503, 2009.
[27] D. Fotakis, R. Pagh, P. Sanders, and P. Spirakis, "Space efficient hash tables with worst case constant access time," *Theory Comput. Syst.*, vol. 38, no. 2, pp. 229–248, 2005.
[28] Y. Hua, B. Xiao, and X. Liu, "NEST: Locality-aware approximate query service for cloud computing," in *Proc. INFOCOM*, 2013, pp. 1303–1311.
[29] Y. Hua, H. Jiang, Y. Zhu, D. Feng, and L. Tian, "SmartStore: A new metadata organization paradigm with semantic-awareness for next-generation file systems," in *Proc. SC*, 2009.
[30] L. Fan, P. Cao, J. Almeida, and A. Broder, "Summary cache: A scalable wide-area web cache sharing protocol," *IEEE/ACM Trans. Netw.*, vol. 8, no. 3, pp. 281–293, Jun. 2000.
[31] E. Elefheriou, R. Haas, J. Jelitto, M. Lantz, and H. Pozidis, "Trends in storage technologies," *Data Eng.*, vol. 33, no. 4, pp. 4–13, 2010.
[32] J. Bromley and W. Hutsell. Improve application performance and lower costs. Texas Instruments. [Online]. Available: http://texmemsys.com/files/f000240.pdf
[33] A. Anand, C. Muthukrishnan, S. Kappes, A. Akella, and S. Nath, "Cheap and large CAMs for high performance data-intensive networked systems," *Proc. 7th USENIX Conf. Netw. Syst. Design Implementation*, 2010, p. 29.
[34] D. Andersen and S. Swanson, "Rethinking flash in the data center," *IEEE Micro*, vol. 52, no. 4, pp. 52–54, Aug. 2010.
[35] Open NAND Flash Interface SpecificaRion. revision2.2 *http://onfi.org/wp-content/uploads/2009/02/ ONFI 202_2 20 Gold.pdf*.
[36] Y. Hu, H. Jiang, D. Feng, L. Tian, S. Zhang, J. Liu, W. Tong, Y. Qin, and L. Wang, "Achieving page-mapping FTL performance at block-mapping FTL cost by hiding address translation," in *Proc. IEEE Mass Storage Syst. Technol.*, 2010, pp. 1–12.
[37] Y. Hu, H. Jiang, D. Feng, L. Tian, H. Luo, and S. Zhang, "Performance impact and interplay of ssd parallelism through advanced commands, allocation strategy and data granularity," *Proc. ACM ICS*, pp. 96–107, 2011.
[38] Storage Networking Industry Association (SNIA). [Online]. Available: http://www.snia.org/
[39] M. Canim, G. A. Mihaila, B. Bhattacharjee, C. A. Lang, and K. A. Ross, "Buffered Bloom filters on solid state storage," in *Proc. ADMS*, 2010.
[40] M. Olson, K. Bostic, and M. Seltzer, "Berkeley DB," in *Proc. USENIX Annu. Techn. Conf.*, 1999.
[41] M. Armbrust, A. Fox, R. Griffith, A. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zahaira, "Above the clouds: A Berkeley view of cloud computing," Dept. Electr. Eng. Comput. Sci., Univ. California, Berkeley, CA, USA, Tech. Rep. UCB/EECS-2009-28, 2009.
[42] R. Buyya, C. Yeo, S. Venugopal, J. Broberg, and I. Brandic, "Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility," *Future Gen. Comput. Syst.*, vol. 25, no. 6, pp. 599–616, 2009.
[43] K. Birman, G. Chockler, and R. van Renesse, "Toward a cloud computing research agenda," *ACM SIGACT News*, vol. 40, no. 2, pp. 68–80, 2009.
[44] B. Hayes, "Cloud computing," *Commun. ACM*, vol. 51, no. 7, pp. 9–11, 2008.
[45] V. Athitsos, M. Potamias, P. Papapetrou, and G. Kollios, "Nearest neighbor retrieval using distance-based hashing," in *Proc. Int. Conf. Data Eng.*, 2008, pp. 327–336.
[46] R. Pagh, and F. Rodler, "Cuckoo hashing," *J. Algorithms*, vol. 51, no. 2, pp. 122–144, 2004.
[47] A. Kirsch, M. Mitzenmacher, and U. Wieder, "More robust hashing: Cuckoo hashing with a stash," in *Proc. ESA*, 2008.
[48] M. Naor, G. Segev, and U. Wieder, "History-independent cuckoo hashing," in *Proc. Autom., Lang. Programming*, pp. 631–642, 2010.
[49] R. Kutzelnigg, "An improved version of cuckoo hashing: Average case analysis of construction cost and search operations," *Math. Comput. Sci.*, vol. 3, no. 1, pp. 47–60, 2010.
[50] B. Debnath, S. Sengupta, J. Li, D. J. Lilja, and D. Du, "Bloomflash: Bloom filter on flash-based storage," Tech. Rep. MSR-TR-2010-161, EMC Corp., Santa Clara, CA, USA, Mar. 2010.
[51] D. Park, and D. Du, "Hot data identification for flash memory using multiple Bloom filters," TR-10-026, Univ. Minnesota, Minneapolis, MN, USA, Oct. 2010.

Yu Hua received the BE and PhD degrees in computer science from the Wuhan University, Wuhan, China, in 2001 and 2005, respectively. He is currently an Associate Professor at the Huazhong University of Science and Technology, Wuhan, China. His research interests include computer architecture, cloud computing and network storage. He has more than 50 papers to his credit in major journals and international conferences including *IEEE Transactions on Computers* (TC), *IEEE Transactions on Parallel and Distributed Systems* (TPDS), USENIX ATC, INFOCOM, SC, ICDCS, ICPP. He has been on the program committees of multiple international conferences, including INFOCOM and ICPP. He is a senior member of the IEEE, and a member of ACM and USENIX.

Bin Xiao received the BSc and MSc degrees in electronics engineering from Fudan University, Shanghai, China, in 1997 and 2000 respectively, and the PhD degree in computer science from University of Texas at Dallas, TX, in 2003. Currently, he is an Associate Professor in the Department of Computing of the Hong Kong Polytechnic University, Hong Kong. His research interests include distributed computing systems, data management, secured communication networks, focusing on wireless sensor networks and RFID systems. Currently he is the associate editor of the *International Journal of Parallel, Emergent and Distributed Systems*. He is a member of the IEEE computer society and a senior member of the IEEE.

Xue Liu received the BS degree in mathematics and the MS degree in automatic control, respectively, both from Tsinghua University, Beijing, China. He received the PhD degree in computer science from the University of Illinois at Urbana-Champaign, IL, in 2006. He is currently an Associate Professor in the School of Computer Science at McGill University, Montreal, QC, Canada. His research interests are in computer networks and communications, smart grid, real-time and embedded systems, cyber-physical systems, data centers, and software reliability. His work has received the Year 2008 Best Paper Award from IEEE Transactions on Industrial Informatics, and the First Place Best Paper Award of the ACM Conference on Wireless Network Security (WiSec 2011). He serves as an associate editor of the *IEEE Transactions on Parallel and Distributed Systems* and editor of the *IEEE Communications Surveys & Tutorials*. He is a member of the IEEE and the ACM.

Dan Feng received the BE, ME, and PhD degrees in computer science and technology from the Huazhong University of Science and Technology (HUST), Wuhan, China, in 1991, 1994, and 1997, respectively. She is currently a Professor and Vice Dean of the School of Computer Science and Technology, HUST. Her research interests include computer architecture, massive storage systems, and parallel file systems. She has more than 100 publications to her credit in journals and international conferences, including *IEEE Transactions on Parallel and Distributed Systems* (TPDS), *Journal of Computer Science and Technology*, USENIX ATC, FAST, International Conference on Distributed Computing Systems, High-Performance Parallel and Distributed Computing, SC, ICS and International Conference on Parallel Processing. She servers as the program committees of multiple international conferences, including SC and Mass Storage Systems and Technologies. She is a member of the IEEE.