NEST: Locality-aware Approximate Query Service for Cloud Computing

Yu Hua
Wuhan National Lab for Optoelectronics, School of Computer
Huazhong University of Science and Technology
Wuhan, China
csyhua@hust.edu.cn

Bin Xiao
Department of Computing
The Hong Kong Polytechnic University
Kowloon, Hong Kong
csbxiao@comp.polyu.edu.hk

Xue Liu
School of Computer Science
McGill University
Montreal, Quebec, Canada
xueliu@cs.mcgill.ca

***Abstract*—Cloud computing applications face the challenges of dealing with a huge volume of data that needs the support of fast approximate queries to enhance system scalability and improve quality of service, especially when users are not aware of exact query inputs. 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 ease-of-use and cost-effective approximate query service for cloud computing. The novelty of NEST is to leverage cuckoo-driven locality-sensitive hashing to find similar items that are further placed closely 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. Theoretical analysis and extensive experiments in a large-scale cloud testbed demonstrate the salient properties of NEST to meet the needs of approximate query service in cloud computing environments.**

I. 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]–[4].

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]–[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 Locality Sensitive Hashing (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.

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.

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 Scheme.** We propose a novel locality-aware balanced scheme, called NEST, in the 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 the cuckoo hashing by allocating new items in neighboring buckets, which is perfectly allowed in LSH.
- **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.
- **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* [17] 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 a real-world trace 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 II shows research backgrounds and related work. Section III presents the NEST design and practical operations. We give extensive experimental results in Section IV and conclude the paper in Section V.

II. Backgrounds and Related Work

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

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.

A. Locality Sensitive Hashing

Locality Sensitive Hashing (LSH) [14] 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.** $\mathbb{H} = \{h : S \rightarrow U\}$ is called (R, cR, P_1, P_2)-sensitive for any $p, q \in S$

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

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 large constant.

We need to configure two main parameters, M, the capacity of a function family $\mathbb{G}$, and d, 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 $g(v) = (LSH_1(v), \cdots, LSH_M(v))$ as the concatenation of M LSH functions, where v is a θ-dimensional vector. Furthermore, an LSH consists of d hash tables, each of which has a function g_i $(1 \leq i \leq d)$ from $\mathbb{G}$.

LSH has been successfully applied in approximate queries of vector space and semantic access. 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.

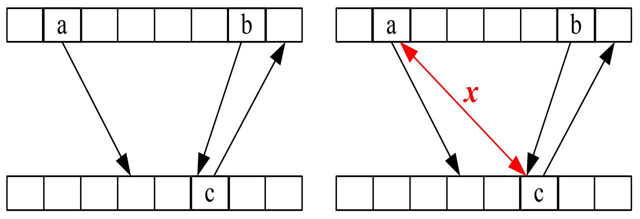
B. Cuckoo Hashing

Cuckoo hashing [16] is a dynamization of a static dictionary and provides a useful methodology for building practical, high-performance hash tables. It combines the power of allowing multiple hash locations for an item with the power of dynamically changing the location of an item among its possible locations.

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, $h_1, h_2 : U \rightarrow \{0,\ldots,m-1\}$. 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.

Figure 1 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 Figure 1(a). 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 Figure 1(b). If an endless loop takes place, the cuckoo hashing carries out a rehashing operation.

It is shown in [18] that if $m \geq (1+\varepsilon)n$ for some constant $\varepsilon > 0$ (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 $O(1/n)$.



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

Fig. 1. Cuckoo hashing structure.

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 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.

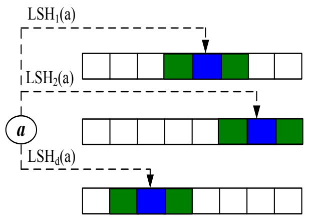
III. NEST Design

This section presents NEST scheme and illustrates the practical locality-aware operations, including item insertion, deletion and ANN query. We also study the rehash probability of the NEST design.

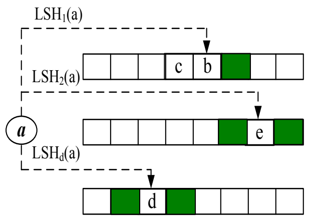
NEST takes into account the case for $d > 2$ due to two main reasons. One is that LSH requires multi-hashing computation to enhance the accuracy of locality aggregation. More hashing functions lead to higher aggregation accuracy. The other reason is that multi-hashing is more important and practical 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, which is adopted in NEST.

A. Structure

NEST structure uses a multi-choice hashing scheme to place items as shown in Figure 2. 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.



(a) A multi-choice LSH.

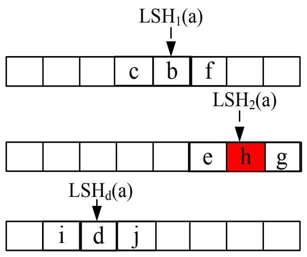


(b) Available locations for an item a.

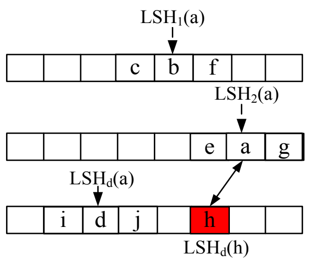
Fig. 2. NEST structure.

Figure 2(a) 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 Figure 2(b), if $d = 3$, $LSH_1(a)$, $LSH_2(a)$ and $LSH_3(a)$ have been occupied by other items b, e and d 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 Figure 3. 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.



(a) Hashing collisions for placing item a.



(b) Moving item h to its another location.

Fig. 3. Cuckoo-based solution for hashing collisions.

B. Practical Operations

We describe practical locality-aware operations of NEST to support item insertion, ANN query and item deletion.

1) Insertion: The insertion operation needs to place items in hashed or adjacent empty buckets to obtain load balance. Figure 4 shows the recursive insertion algorithm for item a. This 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 as described in Figure 5. If there is no empty bucket among the positions hit by LSH computation, NEST needs to probe adjacent buckets of $LSH_i(a)$ as described in Figure 6. 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.

We denote $B[*]$ to be the data in that bucket and use Δ to represent the number of neighbors to be probed, which is an adjustable parameter depending upon locality pattern in real-world applications. In addition, once we test $MaxLoop$ rounds of "kicking out" operation and the insertion fails, we have to execute the rehash operation.

```
Insert(Item a)
1:  DirectInsert(Item $a$)
2:  Adjacent_Probe(Item $a$, Number $\Delta$)
3:  $\eta := 1$
4:  while $\eta \leq MaxLoop$ do
5:      $B[LSH_k(a)] \rightarrow temp$ for some random $k \in \{1, \cdots, d\}$
6:      $a \rightarrow B[LSH_k(a)]$
7:      Insert(Item $temp$)
8:      $\eta{+}{+}$
9:  end while
10: Rehash()
```

Fig. 4. Algorithm for item insertion.

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

```
DirectInsert(Item a)
1: $i := 1$
2: while $B[LSH_i(a)]! = NULL$ and $i \le d$ do
3:     $i++$
4: end while/* an empty position to insert $a$ */
5: if $(i \le d)$ then
6:     $a \rightarrow B[LSH_i(a)]$
7:     Return /* finish the insertion */
8: end if
```

Fig. 5. Algorithm for directly inserting an item without any hash collision.

```
Adjacent_Probe(Item a, Number Δ)
1: $i := 0$
2: while $(i \le d-1)$ do
3:     $i++, j := 1$
4:     while $|j| \le \Delta$ do
5:         if $(B[LSH_i(a)+j] = NULL)$ then
6:             $a \rightarrow B[LSH_i(a)+j]$ /*check right neighbors */
7:             Return /* finish the insertion */
8:         end if
9:         if $(B[LSH_i(a)-j] = NULL)$ then
10:            $a \rightarrow B[LSH_i(a)-j]$ /*check left neighbors */
11:            Return /* finish the insertion */
12:        end if
13:        $j++$
14:    end while
15: end while
```

Fig. 6. Algorithm for probing adjacent buckets.

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 $g(x) = \frac{e^{-x^2/2}}{\sqrt{2\pi}}$.

2) ANN Query: The ANN query needs to obtain approximate neighbors to a query point q. NEST can complete the ANN query operation in a simple way. Figure 7 illustrates the ANN query algorithm that 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 *Result* contains correlated data items to satisfy the ANN query request.

3) Deletion: In the item deletion, we need to find the item to be deleted and then remove it from the bucket of hash table. Figure 8 shows the algorithm of deleting an item a from NEST.

```
ANN_Query(Item q)
1: $Result := \emptyset$
2: for $(i := 1, i \le d), i++$ do
3:     for $(j := -\Delta, j \le \Delta, j++)$ do
4:         $Result := Result + B[LSH_i(q) + j]$
5:     end for
6: end for
7: Return $Result$
```

Fig. 7. Algorithm to support ANN query for queried item q.

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.

```
Deletion(Item a)
1: $i := 1, j := -\Delta$
2: while $i \le d$ do
3:     while $B[LSH_i(a)+j]! = a$ and $j \le \Delta$ do
4:         $j++$
5:     end while
6:     if $(B[LSH_i(a)+j] == a)$ then
7:         Delete $a$ from $B[LSH_i(a)+j]$, Return
8:     end if
9:     $i++, j := -\Delta$
10: end while
```

Fig. 8. Algorithm for deleting an item.

C. Rehash Probability

The rehash analysis is based on two reasonable assumptions as follows.

Assumption 1: **Vacant Bucket First**. When inserting an item, if one of its d neighboring buckets is vacant, we will place the item into that vacant bucket without kicking out existing items.

Assumption 2: **No Instant Loop**. An item a kicked out by item b will choose other $(d-1)$ buckets for placement, but not kick out its previous b to avoid an instant loop.

Theorem 1: Given n items following 2-stable normal distribution, each item has d locations with 2Δ adjacent neighbors in NEST. The rehashing probability has an upper bound of

$$\frac{P_1^{(d+2d\Delta)+(MaxLoop-1)(d-1+2d\Delta)}}{d(d+2d\Delta-1)^{MaxLoop-1}} \tag{1}$$

where $P_1 = 1 - 2N_{CDF}(-\omega) - \frac{2}{\sqrt{2\pi}\omega}(1 - e^{-(\omega^2/2)})$ and N_{CDF} is the cumulative distribution function for a random variable following $N(0,1)$.

Proof: Item insertion operations are actually an iterative process by kicking out one of totally $(d + 2d\Delta)$ items if all available buckets have been full in the worst case. For a new item a, it has $(d + 2d\Delta)$ choices and the probability that all buckets are full is $P_1^{(d+2d\Delta)}$, where P_1 is the locality-aware probability in Definition 2.

The item a can randomly choose one item from $(d + 2d\Delta)$ buckets with the probability $1/(d + 2d\Delta)$. The chosen item happens to have all $(d - 1 + 2d\Delta)$ buckets that are also full with the probability of $P_1^{(d-1+2d\Delta)}$ based on the Assumption 2. This iterative process continues until reaching the *MaxLoop* steps in the worst case. We hence have the upper-bound probability of rehashing occurrence

$$\frac{P_1^{(d+2d\Delta)+(MaxLoop-1)(d-1+2d\Delta)}}{d(d+2d\Delta-1)^{MaxLoop-1}} \tag{2}$$

We further study how to obtain the P_1 value that is the probability of hash collisions of two items. First, let $f_s(t)$ be the probability density function of s-stable distribution. According to the conclusion in [19], the probability that items p and $q_i(1 \le i \le n)$ collide in an LSH is

$$P^*(\kappa_i) = P_{a,b}[h_{a,b}(p) = h_{a,b}(q_i)] = \int_0^{\omega} \frac{1}{\kappa_i} f_s(\frac{t}{\kappa_i})(1-\frac{t}{\omega})dt \tag{3}$$

where vector a is drawn from an s-stable distribution, vector b is uniformly drawn from $[0,\omega)$. For a fixed ω, $P^*(\kappa_i)$ decreases monotonically with $\kappa_i = ||p-q_i||_s$. Hence, the probability that an item p collides with the dataset of n items is $\frac{\sum_{i=1}^{n} P^*(\kappa_i)}{n}$.

Furthermore, LSH family (R, cR, P_1, P_2) is sensitive for $P_1 = P^*(1)$ and $P_2 = P^*(c)$. The probability density function $f_s(\frac{t}{\kappa_i})$ can help compute P_1 for the s-stable distribution. When considering $s = 2$ Normal distribution, by using a simple calculation, we have

$$P_1 = 1 - 2N_{CDF}(-\omega) - \frac{2}{\sqrt{2\pi}\omega}(1 - e^{-(\omega^2/2)}) \tag{4}$$

where N_{CDF} is cumulative distribution function for random variable following $N(0,1)$.

■

D. Summary

In the NEST design, deletion and ANN query operations can obtain constant-scale complexity even in the worst case. They are bounded by probing at most $O(d \cdot (2\Delta+1))$ buckets, in which parameters d and Δ are small constants (e.g., $\Delta = 1$). The insertion operation can be done in $O(d \cdot (2\Delta+1))$, (i.e., $O(1)$) complexity in most cases. In a few cases, the complexity becomes $O(MaxLoop \cdot d \cdot (2\Delta+1))$, which is $O(1)$ as well. Rarely does the insertion operation need to invoke rehashing. Such low rehash probability for NEST is analyzed in the above section.

IV. Performance Evaluation

In this section, we evaluate the performance of the proposed NEST structure by implementing a prototype in a large-scale cloud computing environment. The evaluation metrics include accuracy and latency of ANN query, I/O costs and space overhead. Another salient feature of NEST, small rehash probability, is also evaluated.

A. Implementation Details

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, 2GB 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 describe the characteristics of the real-world trace for our experiments. From 2000 to 2004, metadata traces [20] have been collected from more than 63,398 distinct file systems that contain 4 billion files. This is one of the largest sets of file-system metadata collected. The 92GB-size trace has been published in SNIA [21]. The multiple attributes of data in the trace 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 [20] further show the data locality properties in terms of read, write and query operations.

In the real cloud system implementation, we partition entire real-world traces into sequential segments that faithfully maintain the original access patterns and data locality. Each cloud server stores one trace segment. A segment, that contains the data with multiple attributes, can be represented as a multi-dimensional vector that consists of their average values. In the same way, a query request from a client can also be represented as a vector. Thus, by using the vectors of segment and query requests, we leverage locality-aware computation to obtain the correlation degree between servers and query requests. If the correlation degree is larger than a threshold, the servers to be queried possibly contain the query results with a high probability. This scheme significantly narrows the searching scope and avoids the brute-force searching operations on all cloud servers. Moreover, both clients and servers use multiple threads to exchange messages and data.

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. The total 2000 query requests constitute a 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 threshold, 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 [20], [22]. 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 size 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 may affect the response to queries. Fortunately, cuckoo hashing has a higher load factor in hash tables without incurring too much delay to queries. It has been shown mathematically that with 3 or more hash functions and with a load factor up to 91%, insertion operations can be done in an expected constant time [16], [18]. We hence set a maximum load factor of 90% for the cuckoo hashing implementation.

In order to obtain accurate parameters, we use the popular sampling method that is proposed in LSH statement [14], [15] and practical applications [17]. "*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 700 for the metadata set. In addition, a rehashing in insertion operation may incur the relocation of items. By analyzing the results of the average numbers of relocation per insertion, we recommend to use $d = 10$ 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.

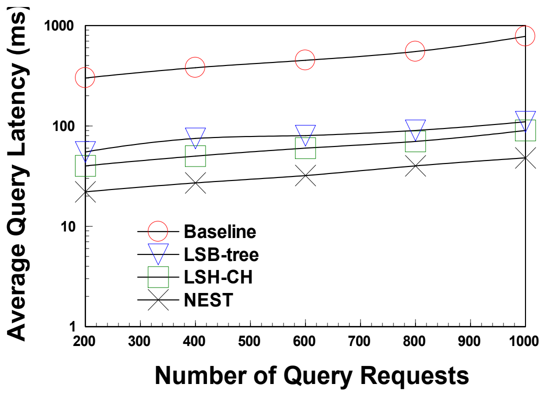
We compare the NEST performance with *LSB-tree* [17], LSH with Cuckoo Hashing (*LSH-CH*) and *Baseline* approaches. Since traditional cuckoo hashing techniques can only support exact-matching query, but not approximate query, we select the state-of-the-art work, LSB-tree [17] that can support ANN query, for performance comparisons. 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 by a conventional B-tree. It addresses the endless loop by using an auxiliary data structure. LSH-CH is a data structure with a simple combination of LSH and cuckoo hashing. 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.

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 an elegant scheme for ANN query in large-scale cloud computing applications.

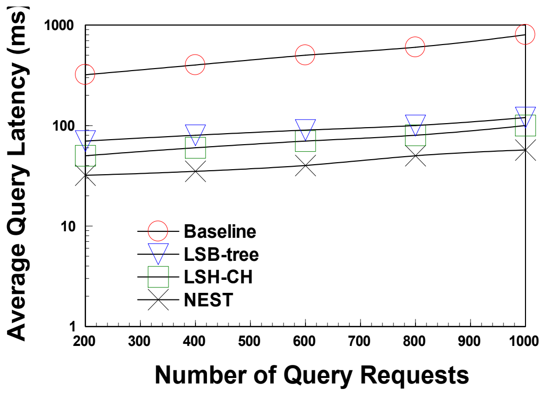
B. Performance Results

We show the 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.

1) ANN Query Latency: Figure 9 shows the ANN query latency when using metadata trace. 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% 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% latency reduction. The main reason is that LSB-tree needs to run Z-order codes and retrieve a B-tree 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 IV-A, since the simple combination of LSH and cuckoo hashing, i.e., LSH-CH, addresses the loop by using an auxiliary structure, the queries in LSH-CH have to navigate the auxiliary storage space to find possible approximate items, thus incurring a larger latency than NEST.



(a) Uniform.



(b) Zipfian.

Fig. 9. ANN query latency.

2) Space Overhead: Figure 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 structure, the average savings from NEST are 47.9% in the 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 the buckets of hash tables. The flat hash-based addressing in NEST can improve the space utilization.

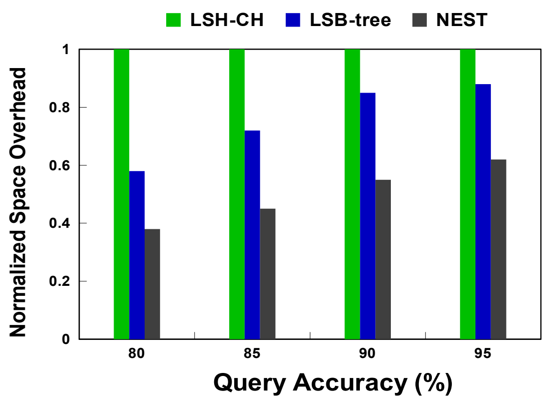


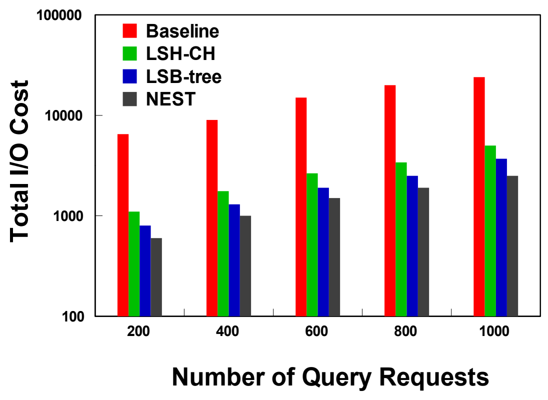
Fig. 10. Normalized space overhead.

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. Figure 11 illustrates the total I/O costs for approximate queries. 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.

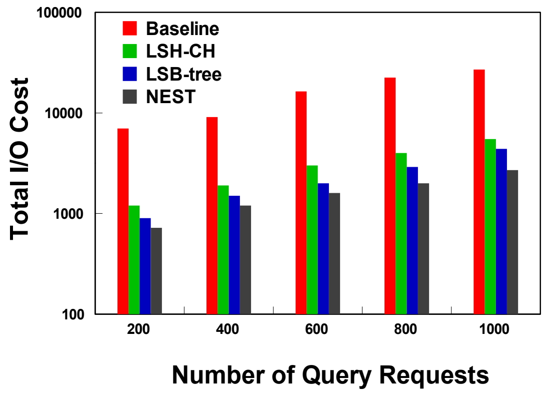
Furthermore, performing the index on a B-tree makes LSB-tree produce 1.76 times more visits (average values) than NEST in the trace. NEST needs to probe limited and deterministic locations to obtain query results and its operations of constant-scale complexity significantly reduce the costs of I/O accesses.

4) *ANN Query Accuracy:* We examine query accuracy of NEST and other three approaches by using the metric of average "*Approximate Measure*" in the trace by using uniform and zipfian query requests as shown in Figure 12. 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 is 90.5% in the trace, which is higher than the percentages of 82.7% in LSB-tree, and 79.3% 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 auxiliary structure 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) *Rehash Probability:* Hash collisions often appear in the computation of hash functions. Without exception, NEST has a chance for rehashing when hash collisions occur. Surprisingly, the rehashing probability has been reduced significantly. Figure 13 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



(a) Uniform.



(b) Zipfian.

Fig. 11. Total I/O costs for ANN query.

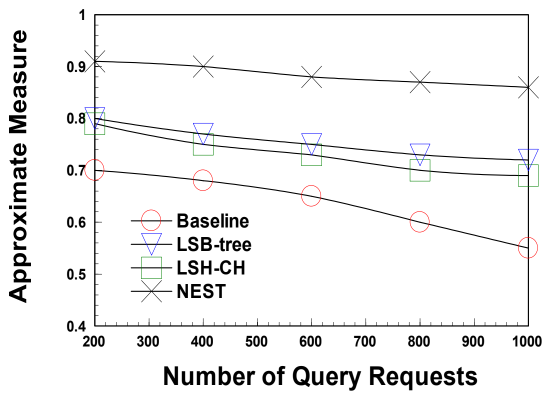
very small in the trace. In other words, a failure only occurs when millions of insertions are done. In contrast, the standard cuckoo hashing has a much higher failure probability and we can observe a failure when inserting thousands of items. Such significant decrement of failure rate is because NEST allows items to be inserted into adjacent and correlated buckets.

C. Summary

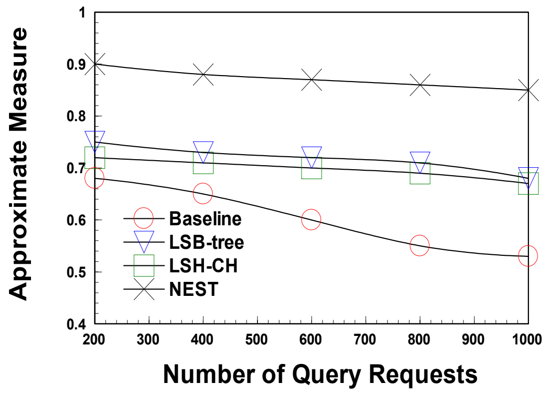
The extensive experiments demonstrate NEST has great advantages over existing work in terms of query latency, accuracy, space overhead, and rehash probability. In particular, a simple combination of LSH and cuckoo hashing, say LSH-CH, does not work well. NEST can efficiently exploit and leverage the locality of datasets to support approximate query in a cloud environment. It achieves load-balance in its stored data structure, while significantly alleviates the system performance degradation due to hash collisions by employing locality-aware algorithms.

V. 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,



(a) Uniform.



(b) Zipfian.

Fig. 12. ANN query accuracy.

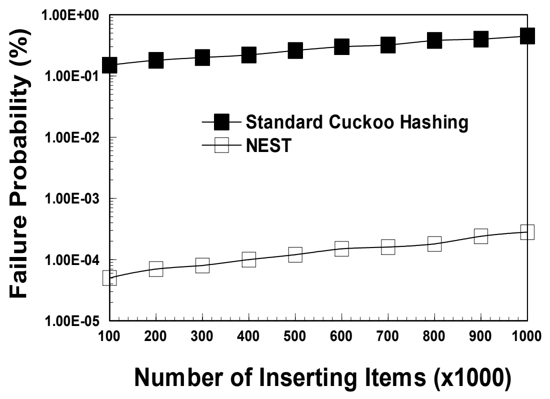


Fig. 13. Insertion failure probability due to the loops.

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, which is also verified by our extensive experiments in a real cloud implementation.

Acknowledgements

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, and NSERC Discovery Grant 341823, US National Science Foundation Award 1116606.

References

[1] J. Gantz and D. Reinsel, "2011 Digital Universe Study: Extracting Value from Chaos," *International Data Corporation (IDC)*, June 2011.
[2] M. Armbrust, A. Fox, R. Griffith, A. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, *et al.*, "A view of cloud computing," *Communications of the 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," *Proc. ACM Symposium on Cloud Computing*, 2011.
[4] S. Wu, F. Li, S. Mehrotra, and B. Ooi, "Query optimization for massively parallel data processing," *Proc. ACM Symposium on Cloud Computing*, 2011.
[5] Q. Liu, C. Tan, J. Wu, and G. Wang, "Efficient information retrieval for ranked queries in cost-effective cloud environments," *Proc. INFOCOM*, 2012.
[6] C. Wang, K. Ren, S. Yu, and K. Urs, "Achieving usable and privacy-assured similarity search over outsourced cloud data," *Proc. INFOCOM*, 2012.
[7] Y. Hua, B. Xiao, and J. Wang, "BR-tree: A Scalable Prototype for Supporting Multiple Queries of Multidimensional Data," *IEEE Transactions on Computers*, no. 12, pp. 1585–1598, 2009.
[8] F. Leibert, J. Mannix, J. Lin, and B. Hamadani, "Automatic management of partitioned, replicated search services," *Proc. ACM Symposium on Cloud Computing*, 2011.
[9] Y. Hua, B. Xiao, D. Feng, and B. Yu, "Bounded LSH for Similarity Search in Peer-to-Peer File Systems," *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," *Proc. INFOCOM*, 2011.
[11] J. Li, Q. Wang, C. Wang, N. Cao, K. Ren, and W. Lou, "Fuzzy keyword search over encrypted data in cloud computing," *Proc. INFOCOM*, 2010.
[12] Y. Hua, B. Xiao, B. Veeravalli, and D. Feng, "Locality-Sensitive Bloom Filter for Approximate Membership Query," *IEEE Transactions on Computers*, no. 6, pp. 817–830, 2012.
[13] M. Bjorkqvist, L. Y. Chen, M. Vukolic, and X. Zhang, "Minimizing retrieval latency for content cloud," *Proc. INFOCOM*, 2011.
[14] P. Indyk and R. Motwani, "Approximate nearest neighbors: towards removing the curse of dimensionality," *Proc. ACM symposium on Theory of Computing*, 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," *Proc. VLDB*, pp. 950–961, 2007.
[16] R. Pagh and F. Rodler, "Cuckoo hashing," *Proc. ESA*, pp. 121–133, 2001.
[17] Y. Tao, K. Yi, C. Sheng, and P. Kalnis, "Quality and Efficiency in High-dimensional Nearest Neighbor Search," *Proc. SIGMOD*, 2009.
[18] R. Pagh, "On the Cell Probe Complexity of Membership and Perfect Hashing.," *Proc. STOC*, 2001.
[19] A. Andoni, M. Datar, N. Immorlica, P. Indyk, and V. Mirrokni, "Locality-sensitive hashing using stable distributions," *Nearest Neighbor Methods in Learning and Vision: Theory and Practice, MIT Press*,, 2006.
[20] N. Agrawal, W. Bolosky, J. Douceur, and J. Lorch, "A five-year study of file-system metadata," *Proc. FAST*, 2007.
[21] Storage Networking Industry Association (SNIA), "http://www.snia.org/,"
[22] 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," *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2009.