Necklace: An Efficient Cuckoo Hashing Scheme for Cloud Storage Services

Qiuyu Li* Yu Hua* Wenbo He‡ Dan Feng* Zhenhua Nie* Yuanyuan Sun*
*Wuhan National Lab for Optoelectronics, School of Computer
Huazhong University of Science and Technology, Wuhan, China
{liqiuyu, csyhua, dfeng, niezhenhua, sunyuanyuan}@hust.edu.cn
Corresponding Author: Yu Hua (csyhua@hust.edu.cn)
‡School of Computer Science
McGill University, Montreal, Quebec, Canada
wenbohe@cs.mcgill.ca

***Abstract*—With the rapid growth of data, query performance is an important concern in cloud storage applications. To reduce the query response time, cuckoo hashing via d hash functions has been adopted to achieve O(1) query efficiency. However, in practice the cuckoo hashing consumes a large amount of system resources, since an item insertion may suffer from frequent "kick-out" operations or even endless loops. In order to address this problem, we propose an efficient loop-oblivious scheme, called Necklace, in the cloud. The idea behind Necklace is to minimize data migration among servers and alleviate the collisions in the insertion operation. We identify the shortest path via indexing the auxiliary table, without retrieving the actual storage contents or carrying out the tentative "kick-out" operations. We have implemented Necklace in a real cloud system and examined the performance by using a real-world trace. Extensive experimental results demonstrate the efficiency and efficacy of Necklace.**

I. Introduction

Hash table based structures can support real-time query efficiency, while unfortunately causing low space utilization. Unlike general hashing design, cuckoo hashing [1] improves space utilization without loss of query performance, due to its salient features of flat addressing. The cuckoo hashing has constant-scale query complexity via probing at most d locations in the worst case [2]. In practice, the conventional cuckoo hashing schemes, e.g. a random-walk approach, suffer from intensive migration operations among servers due to unpredictable random selection. Repetitions and endless loops often occur in the random-walk approach due to potential hash collisions. In order to deliver high performance, we need to address three main problems.

Intensive Data Migration. When inserting new data into storage servers via the cuckoo hashing, a kick-out operation may cause data migration among servers [3]. Conventional cuckoo hashing based schemes heavily depend on the timeout to identify an insertion failure. Hence, accurately predicting a failure requires sufficient attempts, which consume substantial system resources and incur large latency in the data migration.

Nondeterministic Performance. Cuckoo hashing generally uses a random walk to identify a vacant bucket to host an inserted item [4]. This scheme overlooks the correlation between the positions and items. Before completing sufficient attempts, it is difficult to identify if an insertion process results in an infinite loop [5]. Moreover, cuckoo hashing provides multiple alternate positions for inserting items, which exacerbates the uncertainty of addressing vacancies when all these candidate positions are occupied (in this case it will execute kick-out operations). The cuckoo path has a variant length, which may lead to loops and repetitions. Multiple concurrent insertion operations may introduce larger variation in operation latency and path lengths.

978-1-4799-4852-9/14/$31.00 © 2014 IEEE

Multiple Migration Of A Single Data Set. The cuckoo hashing schemes based on random walk migrate data at random from the d candidate positions [4]. The random selection will incur repetitions among the augmenting kick-out paths. Hence, multiple migration operations for one data set will consume paramount system resources and cause long latency.

To address these problems, we present a Necklace model to alleviate the actual collisions in the insertion process. With this model, we analyze the occurrence of the collisions in actual procedures. In conventional cuckoo hashing schemes, the cases of $d > 2$ are illustrated as hypergraphs. When the collisions occur in an insertion operation, the shortest path can not be identified accurately. *Necklace* is a simple and easy-to-use model to describe an instance of the case of $d = 3$. In this case our Necklace model uses triangles to denote the inserted items, and the actual storage positions are represented differently from other backup positions. Using the Necklace model, we can clearly present the data migration in a geometric method.

The paper is organized as follows. Section II presents the backgrounds. Section III illustrates our Necklace model. Section IV shows the performance evaluation. Section V presents the related work. We conclude our paper in Section VI.

II. Backgrounds

In this section, we present the research backgrounds of the cuckoo hashing scheme.

The cuckoo hashing [1], [6] leverages $d \geq 2$ hash functions, instead of one, to allow each item to get $d-1$ buckets for relocation choice. Hence the hash collisions can be mitigated. Cuckoo hashing can dynamically migrate the existing items and then choose a suitable position for a new item during an insertion. The insertion operation in cuckoo hashing is similar to the behavior of cuckoo birds in nature that cuckoo birds

kick other eggs or birds out of their nests. As a flat addressing solution, the index of cuckoo hashing is horizontal, that is, the query result can be obtained by one probing. Cuckoo hashing ensures constant-scale query time in the worst case and constant amortized time for insertions and deletions. The cuckoo hashing recursively kicks items out of their positions and uses multiple hash functions for resolving hash collisions, which alleviates the complexity of using conventional linked lists.

The placement of inserted items in the cuckoo hashing can be viewed as a graph. There is a theory of random graphs called "Cuckoo Graph", in which an undirected graph can be used to describe the connectivity of storage positions [7]. The vertices in the cuckoo graph indicate the positions in the hash tables. The edges represent the paths can be moved along. The case of $d = 2$ is easy to describe. In the cases of $d \geq 3$,the edges are hyperedges and the cuckoo graph turns into a hypergraph, which is more complicated to consider the relationship between the positions and items. Therefore, we introduce our Necklace model to simplify the complicated situation. We first explain the standard cuckoo hashing represented in the hypergraph.

A. Standard Cuckoo Hashing

Cuckoo hashing is a dynamic variation on multi-choice hashing dictionary. Cuckoo hashing offers the support of fast queries in the constant-scale lookup time due to flat addressing for an item among multiple choices.

Definition 1: **Standard Cuckoo Hashing**. Basic cuckoo hashing uses d hash tables, T_1, T_2, ..., T_d with the length n, and d hash functions $h_1, h_2, \cdots, h_d$: $U \rightarrow \{0, \ldots, n-1\}$. Every item $x \in S$ is kept in one of the d cells: $h_1(x)$ of T_1, $h_2(x)$ of T_2, $\cdots$, or $h_d(x)$ of T_d, but never in all these d cells. The hash functions h_i meet the conditions of independent and random distribution.

We use an example to illustrate the practical execution of standard cuckoo hashing. As shown in Figure 1, we discuss the case of $d = 3$ that we use three hash functions h_1, h_2 and h_3 to locate items into three hash tables T_1, T_2 and T_3. Figure 1(a) demonstrates the item x as a hyperedge, and the available locations of x in three hash tables are $h_1(x)$, $h_2(x)$ and $h_3(x)$, which are represented as hypervertices.

According to the definition of hypergraph, a hypergraph is a generalization of a graph in which an edge can connect any number of vertices. A $k-uniform$ hypergraph is a hypergraph such that all its hyperedges have size k, i.e., a collection of sets of size k. A 2-uniform hypergraph is a graph, and a 3-uniform hypergraph is a collection of unordered triples, and so on. Hence the case of $d = 3$ is a 3-uniform hypergraph which involves many unordered triples.

To demonstrate an insertion in cuckoo hashing, we can observe the hypergraph in Figure 1(b). The pink hyperedge represents the existing item x. The blue hyperedge illustrates a new item y. The candidate positions of y calculated by hash functions are o, p and q. In the hypergraph, p and $h_1(x)$ are in the same location, and the insertion process will incur a collision if the new item y locates in the cell p. However, in standard cuckoo hashing the procedure is likely to put y into $T1$ as a preferential choice. In this case, x will be kicked out and relocates in other two backup cells, i.e., $h_2(x)$ and $h_3(x)$. The kick-out operations will be repeatedly executed if the selected position is still occupied. If the kick-out operations exceed a timeout threshold, the cuckoo hashing will carry out the rehashing operation.

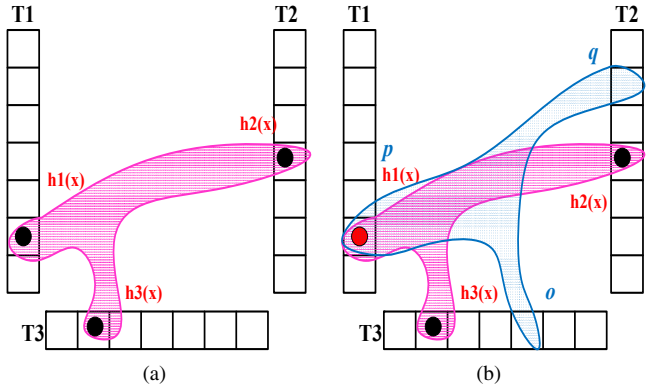


Fig. 1. The standard cuckoo hashing structure.

We can discern that standard cuckoo hashing cannot avoid unnecessary repetitions and endless loops occurring in the random selection of backup locations. Potential hash collisions cause multiple intensive migration operations among servers due to unpredictable random selection. Smart decisions should be made by instructions to identify the shortest augmenting path to a vacancy. We aim to giving advice to relocations when hash collisions occur for alleviating the data migration in practice.

III. Design and Model Analysis

In this section, we present a cost-effective searching method, called Necklace, to identify the shortest augmenting path in the cuckoo hashing.

A. The Necklace Model Design

The idea behind Necklace is to minimize data migration in cloud storage services via judiciously alleviating the hash collisions in the practical insertion procedure. In the case of $d \geq 3$, compared with a random-walk manner using multiple attempts to probe vacant buckets, our design analyzes the relationship among the d candidate storage locations of one item, and distinguishes the actual storage location from the $d-1$ backup ones. Moreover, the Necklace based methods do not suffer from any repetitions in insertion procedures and avoid both unnecessary kick-out operations and endless loops existing on the random-walk augmenting paths. Necklace can identify the shortest augmenting path to a vacancy.

Our proposed Necklace model uses an additional data structure to record concise location information and labels the buckets with unique flags, which consumes a small storage capacity for indicating the shortest path in the kick-out operations. We introduce a Triangular model instead of hypergraph to differentiate the actual storage to other backup positions. Based on the Triangular model, we propose a stereoscopic model consisted of undirected triangular. In order to demonstrate the real hash collisions, we separate undirected triangular in the stereoscopic model from each other, and then get the Necklace model.

In the cases of $d \geq 3$, a cuckoo graph is identified as a hypergraph. Every item is denoted as a hyperedge, and the d candidate buckets are represented as hypervertices. However, the vertices in hypergraphs cannot show the particularity of actual storage. One position can be a backup location of many existing items. It can actually accommodate only one item (actual storage). In order to analyze the principle of how to shape the shortest kick-out path to a vacancy, as a case study, we analyze the situation of $d = 3$. Instead of constructing hyperedges, we use triangles to denote items. The three points of a triangle indicate the three candidate buckets of an item. Due to the particularity of the actual storage bucket, we use a bead to represent the actual storage, as shown in Figure 2. As one bead can be moved along one specific triangle, the triangle and the bead are regarded as a necklace and a pendant, respectively.

1) Undirected Triangular: During the insertion procedure in the cuckoo hashing, an insertion of a new item makes the original inhabitant displaced when a hash collision occurs. One bead would be removed from one position to other $d-1$ spare positions. In the case of $d = 3$, the two-way movement on the edges of triangles allows every item to become an undirected triangular necklace, as shown in Figure 2.

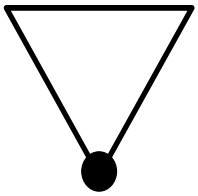


Fig. 2. Undirected triangular model.

Triangular Prism. In the undirected triangular Necklace model, the three points correspond to the buckets in 3 hash tables, as shown in Figure 3. The candidate positions of items a and b are: $h_1(a) = 1$, $h_2(a) = 1$, $h_3(a) = 1$, $h_1(b) = 2$, $h_2(b) = 3$ and $h_3(b) = 2$. One necklace on a triangular prism can denote an item stored in 3 hash tables.

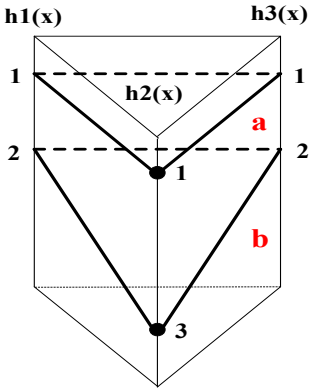


Fig. 3. The Necklaces on the triangular prism.

2) Stereoscopic Model: In order to analyze the connected graph of the case of $d = 3$, we introduce the stereoscopic model. One item can be represented as a triangle, and many such triangles put together make a solid figure (a connected graph) as shown in Figure 4. In a connected graph, the amount and locations of vacant buckets specify the final results of an insertion, a successful insertion or a failure. For instance, in Figure 4(a), point g is a vacant bucket, which belongs to a triangle with no bead on g. However there is no similar cases like point g in Figure 4(b). Furthermore, a new item can not be inserted successfully to hash tables although there is a vacant bucket. The reason is that there is no augmenting path to a vacancy existing in the connected graph. Hence the model in Figure 4 cannot be used to analyze the specific movements of the beads due to some edges connected to a point, in fact, are not related. We use Necklace model to address this problem.

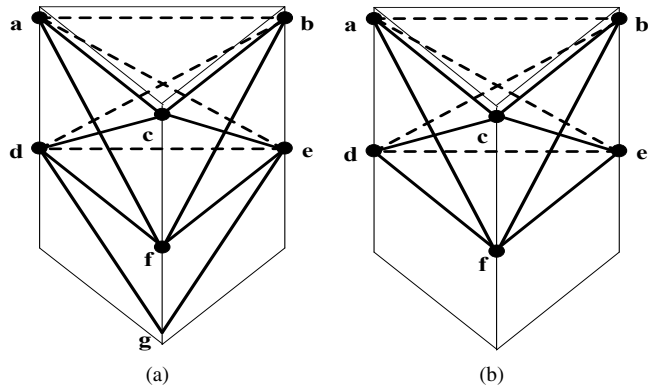


Fig. 4. The Necklaces on triangular trism.

3) The Necklace Model: In the Necklace model, although some edges have the same endpoints, they should be separated from each other because these edges belong to different triangles. We show an example of inserting the 7th item into a connected graph to explain the operating principle in the Necklace model. Figure 5 illustrates the Necklace model related to the example shown in Table I.

There are 7 buckets and 6 items involved in this connected graph, and the only vacant bucket g enables one more new insertion. When a new item is inserted into any of these locations a, b, c, d, e or f, a collision occurs. Due to g is the only vacant bucket involved in this connected graph, identifying an augmenting path to g becomes the sole possible solution to a successful insertion. However addressing paths to g is difficult in the stereoscopic model, as shown in Figure 4(a), since it overlooks the fact that every bead belongs to only one triangle, and cannot move along other triangles. For example, in this case, only the bead on e can be moved to g, but in the stereoscopic model, as shown in Figure 4(a), the beads on both d and e can be moved to g. Moreover, we can discern clearly from Figure 5 that the bead on d can move among the locations of d, e and f. In summary, the Necklace model can predict the real collisions on the insertion kick-out operations.

TABLE I. HASH VALUES OF 6 ELEMENTS IN THE CONNECTED GRAPH

Item	Bucket	h1(Item)	h2(Item)	h3(Item)	Flag
x	b(h3[1])	1	1	**1***	baeg
y	a(h1[1])	**1***	1	2	aeg
z	f(h2[2])	1	**2***	1	faeg
m	c(h2[1])	2	**1***	1	cdeg
n	d(h1[2])	**2***	2	2	deg
j	e(h3[2])	2	3	**2***	eg

B. Vacancy Backtracking

In this section we present the principle of searching the shortest path in the Necklace model via Vacancy Backtracking. The vacancy backtracking uses a method of seeking vacancy

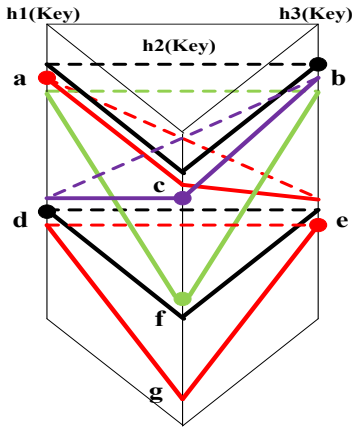


Fig. 5. Necklaces on the triangular prism

retroactively to mark the positions that can successfully locate an item after several kick-out operations.

Beads and Chains: In the Necklace model, we use "bead" and "chain" respectively to represent the actual storage location and the scope of moving an item. The collisions of the beads denote the kick-out operations in a cuckoo hashing insertion. As every collision has $d-1$ selectable directions, in the case of $d=3$, there are two options to relocate this item. In this prepared Necklace model, every position may have "shortest path to a specific vacancy" and "shortest path to the nearest vacancy". Our work in the vacancy backtracking addresses the problem of how to identify "shortest path to the nearest vacancy" by using an additional structure.

Shortest path to a specific vacancy: A connected graph may have more than one vacancy. Given one specific vacancy in a connected graph, some positions can address their shortest paths to this vacancy.

Shortest path to the nearest vacancy: Based on different vacancies, one position may have multiple "shortest paths to a specific vacancy". In these paths, the best path has the smallest amount of steps to identify any vacancies. We illustrate the best path as "shortest path to the nearest vacancy".

We explicate the vacancy backtracking insertion by using an instance in the Necklace model shown in Figure 5. g is the only vacancy belonging to the connected graph. Hence we study the shortest path without the interferences of other vacancies. In Table I, '$*$' denotes the actual storage location of an item. From the vacancy g, the bead on e can be moved to g in one step. e obtains a flag eg to construct a path $e \rightarrow g$. After this movement, e becomes vacant. In the related chains of e (a and d), the beads on a and d can be shifted to e. a and d can be labeled as aeg and deg, that is $a \rightarrow e \rightarrow g$ and $d \rightarrow e \rightarrow g$ can represent the paths to the vacancy g. In other word, the insertion procedures can identify the vacancy g from a and d by two steps. Moreover, we consider the buckets which can arrive at g in three steps. After detecting a and d, the buckets which can be reached by 2 steps, we analyze the buckets whose beads can be moved to a and d. Since e and g have been labeled before, after eliminating the labeled buckets e and g, b and f can drive their beads to a, and the bead on c can be shifted to d. The flags of b, f and c are $baeg$, $faeg$ and $cdeg$. The path becomes $b \rightarrow a \rightarrow e \rightarrow g$, $f \rightarrow a \rightarrow e \rightarrow g$ and $c \rightarrow d \rightarrow e \rightarrow g$. The flagging actions can be completed via reaching a labeled bucket (i.e. e and g), meaning that this bucket has addressed a shorter path than before. After these operations, all the buckets in this connected graph have already been labeled as denoted in Table I.

C. Flag Formation in Auxiliary Table

We detect the paths with an auxiliary table. We use this indexing to illustrate the relations between the beads and chains in the Necklace model, and to study the practical collision. The labeled rows in the auxiliary structure indicate that these items can definitely identify a vacancy. The path flag conducts the direction of practical kick-out operations. Figure 6 illustrates the flag formation in the auxiliary table of the Necklace model in Figure 5.

First, we can find a vacancy in $h2(j)$ and then $h3(j)$ turns to a vacancy after moving j to $h2(j)$. Hence $h3(j)$ can successfully store a new item through one movement. Second, other positions which can relocate their current storage to $h3(j)$ are one step away from $h3(j)$. We find y and n have the same hash value as $h3(j)$ in the same column. So $h1(y)$ or $h1(n)$ turns to a vacancy if y or n moves. Third, we continue the procedure do the same, and then we label $h3(x)$, $h2(z)$ and $h2(m)$. The end condition of labeling operation (one vacancy) is identifying a labeled row, like j can be found (in the third step) because it has the same value ($h1(j)$) with $h1(n)$, but the row $h3(j)$ has been labeled. A labeled row represents an item having already searched a shorter path, because the program is a breath-first search and a former path must be shorter than a later one. If the operation successfully identifies one of any candidate bucket of the new item, the whole labeling operation (multiple vacancies) is finished. This end condition of multiple vacancies avoids searching other useless buckets with longer paths, hence Vacancy Backtracking Insertion alleviate the labeling time.

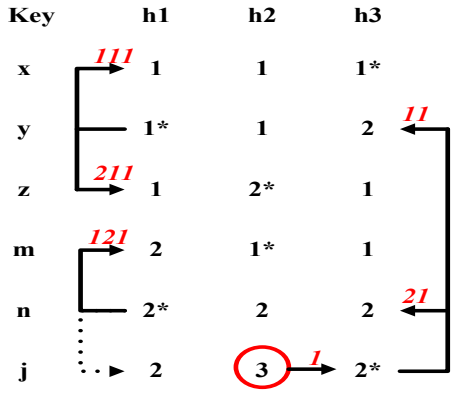


Fig. 6. The flag formation in the Necklace

IV. Performance Evaluation

In this section, we evaluate the performance of the proposed Necklace scheme by implementing a prototype in a large-scale cloud computing environment. The evaluation metrics mainly include accuracy and latency of query and I/O costs.

A. Experiments Setup

We implement Necklace in a large-scale cloud computing environment. Each server is equipped with Intel 2.0GHz dual-core CPU, 16GB DRAM, 500GB 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 Necklace are implemented in the user space. We make use of the Microsoft [8] to examine the system performance.

From 2000 to 2004, metadata traces [8] 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 [9]. 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 [8] further show the data locality properties in terms of read, write and query operations.

To investigate the real performance when using this trace, we randomly allocate the available data segments/snapshots among all servers in a round-robin way. Moreover, a client captures the system operations from these datasets and then delivers the query requests to servers. Both clients and servers use multiple threads to exchange messages and data via TCP/IP.

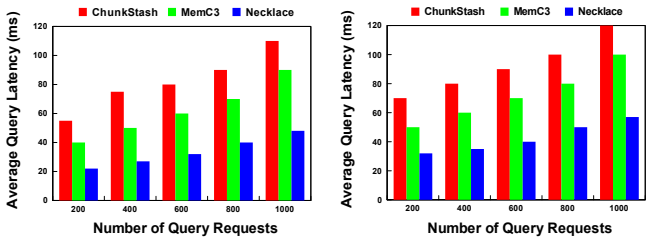
Query requests are generated from the attribute space of the above traces and are randomly selected by considering 1000 uniform or 1000 Zipfian distributions. We set the zipfian parameter H to be 0.8. 2000 query requests constitute the query set and we examine the query accuracy and latency. Moreover, 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 3 or more hash functions and with a load factor up to 91%, insertion operations can be done in an expected constant time [10]. We hence set a maximum load factor of 90% for the cuckoo hashing implementation.

We compare the Necklace performance with the current cuckoo hashing based schemes, including *ChunkStash* [5] and MemC3 [3]. These schemes for performance comparisons are not only state-of-the-art but also their correlation with our work. Specifically, ChunkStash uses an in-memory hash table to index metadata, and the hash collisions can be resolved by a variant of cuckoo hashing. The ChunkStash scheme attempts a small number of key relocations after which it makes room by picking a key to move to an auxiliary linked list (or, hash table). Moreover, MemC3 uses concurrent cuckoo hashing to improve memory efficiency. In MemC3, a short summary of each key, i.e., a tag, is used to improve the cache locality of hash table operations. Since there are no open-sourced codes of MemC3 and ChunkStash, we choose to re-implement their components, algorithms and structures, and a compact CLOCK-based eviction algorithm guarantees strong consistency. 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 Necklace is a better scheme for alleviating the infinite loops in large-scale cloud computing applications.

B. Performance Results

We show advantages of Necklace over ChunkStash and MemC3 approaches by comparing their experimental results in terms of query latency, accuracy and I/O costs.

1) Query Latency: Figure 7 shows the query latency when using Microsoft metadata trace. We observe that compared with ChunkStash, MemC3 obtains on average 9.79% latency reduction. ChunkStash needs to maintain an extra stash, i.e., a linked list, which stores the items with hash collisions. For a query, the ChunkStash scheme checks if the stash contains the queried results, thus incurring longer query latency. Unlike it, MemC3 can carry out the query operations in a constant-scale complexity. Moreover, Necklace obtains on average 45.3% shorter running time than ChunkStash respectively in uniform and zipfian distributions, which is much better than those of MemC3. The main reason is that MemC3 heavily relies on a compact CLOCK-based eviction algorithm to guarantee strong consistency, which exacerbates the query efficiency.



(a) Uniform. (b) Zipfian.

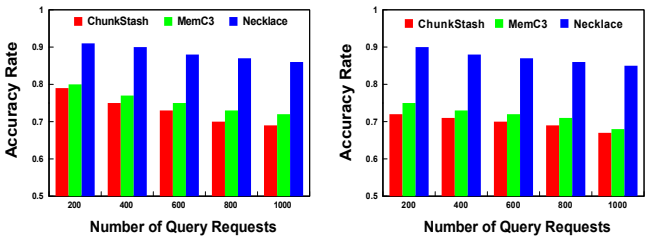
Fig. 7. Average query latency using Microsoft metadata trace.

2) Query Accuracy: We examine query accuracy of Necklace and other two approaches by using the metric of accuracy rate in the Microsoft traces by using uniform and zipfian query requests as shown Figure 8.

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 Necklace is 0.92 in the Microsoft trace, which are higher than the percentages of 0.83 in MemC3, and 0.79 in ChunkStash. Such improvement comes from the fast and accurate prediction in Necklace to efficiently identify the queried results. Moreover, ChunkStash consumes relatively smaller accuracy than MemC3 since the stash in the former is not locality-aware for the queries. We also observe that the uniform distribution receives higher query accuracy than the zipfian because items in the latter naturally incurs the burst requests, the longer latency and the lower accuracy.

3) I/O Costs: The I/O costs is examined by the access times that include the visits on high-speed memory and low-speed disk. Figure 9 illustrates the total I/O costs for queries when using the typical trace. ChunkStash needs to frequently examine the auxiliary space in the hard disks and hence incurs more costs than MemC3 and Necklace.

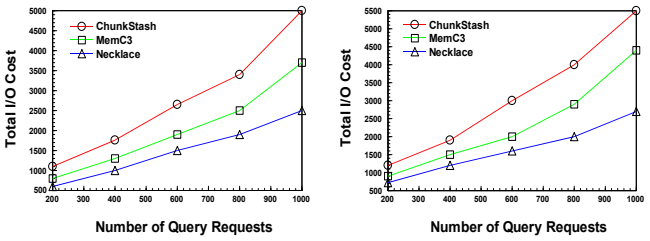
Furthermore, MemC3 produces 1.82 times more visits (on the average) than Necklace in the Microsoft trace. To



(a) Uniform. (b) Zipfian.

Fig. 8. Query accuracy using Microsoft trace.

alleviate the potential infinite loops, MemC3 has to frequently move the keys along the cuckoo path to the free slots in the sequential manner. In the meantime, these keys possibly lead to hash collision and are often stored in the hard disk. Instead, Necklace 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.



(a) Uniform. (b) Zipfian.

Fig. 9. Total I/O costs for queries using Microsoft trace.

V. Related Work

Cuckoo hashing [1] is a further variation on static dictionary with multiple choices. As a flat-addressing hashing scheme, cuckoo hashing provides multiple selective positions to each item and allows inserting items to kick out existing items to their backup positions. Methods of cuckoo hashing based on the graph theory provide a clear understanding of implementation process [7], [11]. Simple properties of branching process are analyzed in bipartite graph [11]. Cuckoo Graph is proposed to describe the cuckoo hashing and this paper obtains asymptotic results covering tree sizes, the number of cycles and the probability [7]. Studies also regard the case of multiple selectable choices $d > 2$ as hypergraphs [12]–[14]. Unlike them, we investigate the characteristics of the stereoscopic model and leverage the specific triangles instead of hyperedges (which represent the existing items). Our Necklace Model exploits the relation among the multiple selectable positions and identifies the shortest cuckoo path.

An accession of secondary structure can provide high efficiency to cuckoo hashing implementations. A constant-sized stash is used in the cuckoo hashing and the failure rate can be reduced by using the stash [10]. A parallel MapReduce cuckoo-hashing algorithm presents a generalization of stashes, which are larger than constant sizes [15]. A practical history-independent dynamic dictionary is used to maintain a record under insertion and deletion of items, with additional membership queries [16]. It presents a secondary structure in a bipartite graph, and labels specific nodes to identify an endless loop. In our paper, instead of using extra structures to maintain items themselves, we keep the selectable choice records of items to accurately identify the shortest path to a vacancy. Necklace can significantly reduce operation latency by reduce the data migration occurs in the cloud service applications.

VI. Conclusion

Efficient queries in cloud computing applications are important to offer cost-effective services. In order to support real-time query performance, we propose a novel cuckoo-hash based scheme, called Necklace. Necklace addresses the potential problem of infinite loops when inserting and querying items. The idea behind Necklace is to minimize data migration in cloud storage services via analyzing the relationship between candidate storage locations to identify the shortest augmenting path. Compared with state-of-the-art work, extensive experiments, using a real-world trace, demonstrate the benefits of using Necklace.

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 NSFC under Grant 61025008.

References

[1] R. Pagh and F. Rodler, "Cuckoo hashing," in *Algorithms - ESA 2001* (F. Heide, ed.), vol. 2161 of *Lecture Notes in Computer Science*, pp. 121–133, Springer Berlin Heidelberg, 2001.

[2] Y. Hua, B. Xiao, and X. Liu, "NEST: Locality-aware Approximate Query Service for Cloud Computing," *Proc. INFOCOM*, 2013.

[3] B. Fan, D. G. Andersen, and M. Kaminsky, "MemC3: Compact and concurrent memcache with dumber caching and smarter hashing," *Proc. USENIX NSDI*, 2013.

[4] A. Frieze, P. Melsted, and M. Mitzenmacher, "An analysis of random-walk cuckoo hashing," *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*, pp. 490–503, 2009.

[5] B. Debnath, S. Sengupta, and J. Li, "ChunkStash: speeding up inline storage deduplication using flash memory," *Proc. USENIX ATC*, 2010.

[6] R. Pagh and F. F. Rodler, "Cuckoo hashing," *Journal of Algorithms*, vol. 51, no. 2, pp. 122–144, 2004.

[7] R. Kutzelnigg, "Bipartite random graphs and cuckoo hashing," *Proc. DMTCS*, 2006.

[8] N. Agrawal, W. Bolosky, J. Douceur, and J. Lorch, "A five-year study of file-system metadata," *Proc. FAST*, 2007.

[9] Storage Networking Industry Association (SNIA), "http://www.snia.org/,"

[10] A. Kirsch, M. Mitzenmacher, and U. Wieder, "More robust hashing: Cuckoo hashing with a stash," *SIAM Journal on Computing*, vol. 39, no. 4, pp. 1543–1561, 2009.

[11] L. Devroye and P. Morin, "Cuckoo hashing: Further analysis," *Information Processing Letters*, vol. 86, no. 4, pp. 215 – 219, 2003.

[12] N. Fountoulakis, K. Panagiotou, and A. Steger, "On the insertion time of cuckoo hashing," *CoRR*, vol. abs/1006.1231, 2010.

[13] M. Mitzenmacher, "Some open questions related to cuckoo hashing," *Proc. ESA*, pp. 1–10, 2009.

[14] N. Fountoulakis, M. Khosla, and K. Panagiotou, "The multiple-orientability thresholds for random hypergraphs," *Proc. SODA*, 2011.

[15] M. T. Goodrich and M. Mitzenmacher, "Privacy-preserving access of outsourced data via oblivious ram simulation," in *Automata, Languages and Programming*, pp. 576–587, Springer, 2011.

[16] M. Naor, G. Segev, and U. Wieder, "History-independent cuckoo hashing," in *Automata, Languages and Programming*, pp. 631–642, Springer, 2008.