Data Storage in Cloud: A View of Cuckoo Hashing

[1]Prof. Yathish Aradhya B C, [2]Ms. Ashwini Shekar, [3]Mrs. Chaithra S, [4]Ms. Sindhu Y Shirur

Assistant Professor CS&E, M.Tech 4th sem CS&E

KIT, Tiptur-572201

Abstract-Cloud computing as an emerging technology trend is expected to reshape the advances in information technology. As the data produced by individuals and enterprises that need to be stored and utilized are rapidly increasing. As sensitive cloud data may have to be encrypted before outsourcing, service based on plaintext keyword search. Search is a fundamental and powerful tool widely used in plaintext information retrieval the problem is particularly challenging, we address two fundamental issues in a cloud environment: privacy and efficiency. the privacy-preserving guarantee of the proposed mechanism under rigorous security treatment. Among various multi keyword semantics, we choose the efficient similarity measure of "coordinate matching", i.e., as many matches as possible, to capture the relevance of data documents to the search query. Among all these things data collision will happens during storage in hashing schemes. In hashing with chaining with a table of size m =αn, whereα >0 is a constant, the worst-case search time is equal to the length of the longest chain. Cuckoo hashing is a new hashing method with very interesting worst-case properties. it hashes n data points into two tables of size m in expected time O(n) as long as m/n >1 + ε1 for some ε1 > 0. Once the table is constructed, each search takes at most two probes. Due to the hash collisions, the cuckoo hashing suffers from endless loops and high insertion latency, even high risks of re-construction of entire hash table. In order to address these problems, we propose a cost-efficient cuckoo hashing scheme, called Min Counter. The idea behind this is to alleviate the occurrence of endless loops in the data insertion by selecting unbusy kicking-out routes. Counter selects the "cold" rather than random, buckets to handle hash collisions.

Keyword-cuckoo hashing, cloud storage, data insertion and query, key word search, lock free, kick out, mincounter.

I. INTRODUCTION

Cloud computing refers to both the applications delivered as services over the Internet and the hardware and systems software in the data centers that provide those services. The services themselves have long been referred to as Software as a Service (SaaS). Some vendors use terms such as IaaS (Infrastructure as a Service) and PaaS (Platform as a Service) to describe their products, but we eschew these because accepted definitions for them still vary widely. The line between "low-level"infrastructure and a higher-level "platform"is not crisp. We believe the two are more alike than different, and we consider them together. Similarly, the related term "grid computing," from the high-performance computing community, suggests protocols to offer shared computation and storage over long distances, but those protocols did not lead to a software environment that grew beyond its community. The data center hardware and software is what we will call a *cloud*. When a cloud is made available in a pay-as you- go manner to the general public, we call it a *public cloud*; the service being sold is *utility computing*. We use the term *private cloud* to refer to internal data centers of a business or other organization, not made available to the general public, when they are large enough to benefit from the advantages of cloud computing(1).

Considering the large number of on-demand data users and huge amount of outsourced data files in the cloud, this problem is particularly challenging as it is extremely difficult to meet also the requirements of practical performance and acceptable system usability. As textual information is ubiquitous in data management systems, many applications have an increasing need to support similarity keyword searches on data collections, i.e., the discovery of similar keywords with respect to a given distance measure(2).

A typical cloud application would have a data owner outsourcing data services to a cloud, where the data is stored in a keyword-value form, and users could retrieve the data with several keywords. Since a cloud is operated by a third party, there have been some concerns over the possible privacy leaks that may occur. Such concerns have led researchers to propose various techniques to protect user privacy. An improvement based on our previous work, where the cloud can return a certain percentage of matched files to the user. This is motivated by the fact that under certain cases, a user may only be interested in a certain percentage of matched files. By returning a smaller percentage of files, the communication cost can be reduced(3).

To enable ranked search for effective utilization of outsourced cloud data under the aforementioned model, our system design should simultaneously achieve security and system design should simultaneously achieve security and performance guarantees as follows.

- Multi-keyword Ranked Search: To design search schemes which allow multi-keyword query and provide result similarity ranking for effective data retrieval, instead of returning undifferentiated results.
- Privacy-Preserving: To prevent the cloud server from learning additional information from the

dataset and the index, and to meet privacy requirements specified in section III-B.

- Efficiency: Above goals on functionality and privacy should be achieved with low communication and computation overhead.

To efficiently achieve multi-keyword ranked search, we propose to employ “inner product similarity” [4] to quantitatively evaluate the efficient similarity measure “coordinate matching”(4).

Similarity search has been widely studied in peer-to-peer environments. In this paper, we propose the Bounded Locality Sensitive Hashing (Bounded LSH) method for similarity search in P2P file systems. Compared to the basic Locality Sensitive Hashing (LSH), Bounded LSH makes improvement on the space saving and quick query response in the similarity search, especially for high-dimensional data objects that exhibit non-uniform distribution property. We present simple and space-efficient Bounded-LSH to map non-uniform data space into load-balanced hash buckets that contain approximate number of objects. Load-balanced hash buckets in Bounded-LSH, in turn, require less number of hash tables while maintaining a high probability of returning the closest objects to requests(5) .

Scatter storage (hash coding) techniques are used to minimize the time required to enter and retrieve information in tables. 'Rather similar techniques can be used for internal tables, such as the symbol tables of compilers and assemblers, and large files which are stored on random-access devices such as disks or drums. Aim is to describe the method for entering information so that subsequent retrievals are very efficient. Suppose that each item consists of an identifying name or key, which may be regarded as an integer, and an associated value. If m keys kx, -.- ,km are stored at addresses a(kl), . . . , a(km) in a table T of length n _> m (i.e. T(a(ki)) = kl for i = 1, . . . , m) and a key k is given, the problem is to determine efficiently whether k is in T, and if so, to find a(k). In order to compare the efficiency of different algorithms, we count the number of fetches of elements of T, i.e. probes, that they require(6).

Hashing is one of the fundamental techniques used to implement query processing operators such as grouping, aggregation and join. This paper studies the interaction between modern computer architecture and hash-based query processing techniques. First, we focus on extracting maximum hashing performance from super-scalar CPUs. In particular, we discuss fast hash functions, ways to efficiently handle multi-column keys and propose the use of a recently introduced hashing scheme called Cuckoo Hashing over the commonly used bucket-chained hashing(7).

An accurate de_nition of concurrent programming techniques in terms of hash tables requires an accurate de_nition of a hash table, along with a set of operations and the semantics of those operations. Consider a standard open hash table, using chaining within buckets. The hash table consists of an array of buckets, each containing a pointer to the head of the linked list for that bucket. Each bucket contains zero or more items in its linked list chain. An item present in the hash table will exist in the bucket corresponding to its hash value. [5, 6] A hash table can support many dierent operations, and any given application may need some subset of these. Common hash table operations include insertion, deletion, replacement, resizing, lookup, and moving an item to a new key. This work will focus on two of those operations: lookup and move. Lookup provides the only read-only operation, and thus a comparison of concurrent programming techniques that dierentiate readers and writers must use the lookup operation in the readers(8).

In order to support real-time queries, hashing-based data structures have been widely used in constructing the index due to constant-scale addressing complexity and fast query response. Unfortunately, hashing-based data structures cause low space utilization, as well as high-latency risk of handling hashing collisions. Traditional techniques used in hash tables to deal with hashing collisions include open addressing, chaining and coalesced hashing. Unlike conventional hash tables, cuckoo hashing addresses hashing collisions via simple “kicking-out” operations (i.e., flat addressing), which moves items among hash tables during insertions, rather than searching the linked lists. Architecture-conscious hashing has demonstrated that cuckoo hashing is much faster than the chaining hashing with the increase of load factors. The cuckoo hashing makes use of $d \geq 2$ hash tables, and each item has d buckets for storage. Cuckoo hashing selects a suitable bucket for inserting a new item and alleviates hash collisions by dynamically moving items among their d candidate positions respectively in hash tables. Such scheme ensures a more even distribution of data items among hash tables than uses only one hash function in conventional hash tables. Due to the salient feature of flat addressing with constant-scale complexity, cuckoo hashing needs to probe all hashed buckets only once and obtains the query results. Even probing at most d buckets in the worst case, the cuckoo hashing guarantees constant-scale query time complexity and constant amortized time for insertion and deletion process, which is also considered by Rivest in. Cuckoo hashing thus improves space utilization without the increase of query latency. In order to implement data structures of hash tables to adapt to concurrent hardware, e.g., multiprocessor machines, efficient synchronization of concurrent access to data structures is essential and significant. More and more studies focus on proposing concurrent hash tables. Hash-based data structures and algorithms are currently a booming industry in the Internet, particularly for applications related to measurement, monitoring, and security. Hash tables and related structures, such as Bloom filters and their derivatives, are used billions of times a day, and new uses keep proliferating. Indeed, one of the most remarkable trends of the last five years has been the growing prevalence of hash-based algorithms and data structures in networking and other areas. At the same time, the field of hashing, which has enjoyed a long and rich history in computer science, has also enjoyed something of a theoretical renaissance. Arguably, this burst of activity began with the demonstration of the power of multiple choices: by giving each item multiple possible hash locations, and storing it in the least loaded, remarkably balanced loads can be obtained, yielding quite efficient lookup schemes. An extension of this idea, cuckoo hashing, further allows items to be moved among its multiple choices to better avoid collisions, improving memory utilization even further(9).

Hash table based structures can support real-time query efficiency, while unfortunately causing low space utilization. Unlike general hashing design, cuckoo hashing 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. 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(9).

II. RELATED WORK

In this section, we present the research background of the cuckoo hashing scheme. The cuckoo hashing was described in [26] as a dynamization of a static dictionary. Cuckoo hashing leverages two or more hash functions for handling hash collisions to mitigate the computing complexity of using the linked lists of conventional hash tables. An item x has two candidate positions to be placed, i.e, $h1(x)$ and $h2(x)$, instead of only one single position in cuckoo hashing scheme. A bucket stores only one item and hash collisions can be decreased. For a general lookup, we only probe whether the queried item is in one of its candidate buckets. A hash collision occurs when all candidate buckets of a newly inserted item have been occupied. Cuckoo hashing needs to execute “kicking-out” operations to dynamically move existing items in the hashed buckets and select a suitable bucket for the new item. The kicking-out operation is similar to the behavior of cuckoo birds in nature, which kicks other eggs or young birds out of the nest. In the similar manner, the cuckoo hashing recursively IEEE Transactions on Parallel and Distributed Systems (Volume: 28, Issue: 3, March 1 2017) kicks items out of their buckets and leverages multiple hash functions to offer multiple choices and alleviates hash collisions. However, cuckoo hashing fails to fully avoid hash collisions. An insertion of a new item causes a failure and an endless loop when there are collisions in all probed positions until reaching the timeout status. To break the endless loop, an intuitive way is to perform a full rehash if this rare incident occurs. In practice, the expensive overhead of performing a rehashing operation can be dramatically reduced by taking advantage of a very small additional constant-size space. Cuckoo hashing offers multiple, not one, hash positions for an item and allows the items to move within these hash positions. Additionally, the space overhead is approximately $2n$ space units, which is similar to the space overhead of binary search trees.

A. *Hash function basics*

A hash function with an n-bit output is expected to have three minimal security properties. (In practice, a number of other properties are expected, as well.)

1) Collision-resistance: An attacker should not be able to find a pair of messages M 6= $M0$ such that hash(M) = hash($M0$) with less than about $2n=2$ work.

2) Pre-image-resistance: An attacker given a possible output value for the hash Y should not be able to find an input X so that Y = hash(X) with less than about $2n$ work.

3) Second pre-image-resistance: An attacker given one message M should not be able to find a second message, $M0$ to satisfy hash(M) = hash($M0$) with less than about $2n$ work.

A collision attack on an n-bit hash function with less than $2n=2$ work, or a pre-image or second pre-image attack with less than $2n$ work, is formally a break of the hash function. Whether the break poses a practical threat to systems using the hash function depends on specifics of the attack (10).

B. *Cuckoo Hashing*

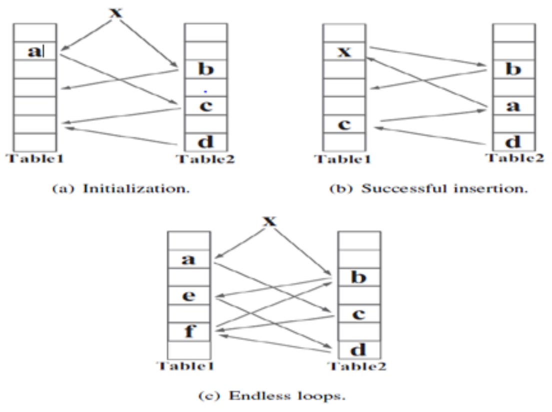

Figure : Example of item insertion to the cuckoo hashing

Cuckoo hashing is a dynamization of a static dictionary described in. The dictionary uses two hash tables, $T1$ and $T2$, of length r and two hash functions $h1$, $h2$: $U \rightarrow \{0,.....,r-1\}$. Every key x € S is stored in cell $h1(x)$ of $T1$ or $h2(x)$ of $T2$, but never in both. Our lookup function is

```
function lookup(x)
return T1[h1(x)] = x  T2 [h2(x)] = x.
end;
```

C. *Lock Free Concept*

A lock-free (also called non-blocking) implementation of a shared object guarantees that if there is an active thread trying to perform an operation on the object, some operation, by the same or another thread, will complete within a finite number of steps regardless of other threads' actions. Lock free objects are inherently immune to priority inversion and deadlock, and offer robust performance, even with indefinite thread delays and failures. Shared sets (also called dictionaries) are the building blocks of hash table buckets. Several algorithms for lock-free set implementations have been proposed. However, all suffer from serious drawbacks that prevent or limit their use in practice. Lock-free set algorithms fall into two main categories: array based and list based. Known array based lock-free set algorithms are generally impractical. In addition to restricting maximum set size inherently, they do not provide mechanisms for preventing duplicate keys from occupying multiple array elements, thus limiting the maximum set size even more, and requiring excessive over allocation in order to guarantee lower bounds on maximum set sizes (11).

III. DESIGN AND IMPLEMENTATION

Illustrates the practical operations, including item insertion, query and deletion. Also, we give the theoretical analysis of the size allocation of counters.

A. *The Min Counter Architecture*

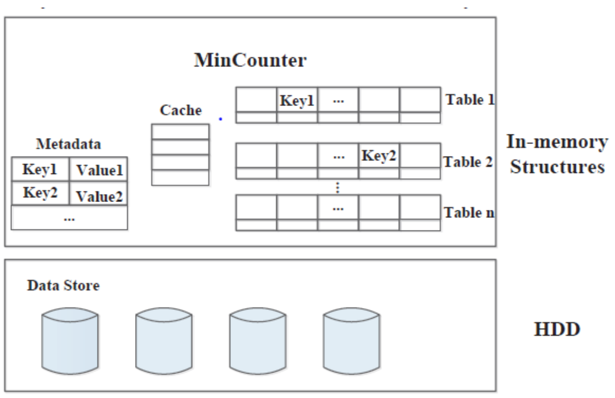

Fig.2 The architecture of min Counter storage system

Min Counter supports a fast and cost-effective cuckoo hashing scheme for data insertion. Due to the simplicity and ease of use, Min Counter has the salient features of high utilization, less data migration and less time overheads. The summarized structure fits into the main memory to improve overall performance. figure shows the storage architecture of Min Counter.

We implement the Min Counter component as in the DRAM. The metadata are in the form of key value pairs. A key is the hashed value of a file ID and the value is the correlated metadata. The hard disk at the bottom stores and maintains the correlated files. The proposed Min Counter scheme is compatible to existing systems, such as the Hadoop Distributed File System (HDFS) and General Parallel File System (GPFS).

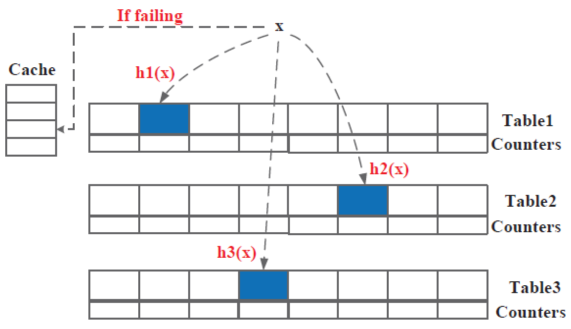

Fig.3 The Data structure of mincounter.

B. *The Min Counter Working Scheme*

In order to alleviate the occurrence of endless loops in cuckoo hashing, we improve the conventional cuckoo hashing by allocating a counter for each bucket of hash tables. We utilize the counters to record kicking-out times occurring at buckets in history. When a hash collision occurs in a bucket, the corresponding counter increases by 1. If an item x is inserted into the hash tables without the availability of empty candidate buckets, we choose the bucket with the minimum counter to execute the replacement. Particularly, if more than one counter has the minimum, we choose the bucket with the minimum number of hash tables by default. As shown, we take d =3 to give an example. When the item x is inserted into hash tables, we first check the buckets of $h1(x)$, $h2(x)$ and $h3(x)$ in each hash table respectively to find an empty bucket. Each candidate bucket of x is occupied by a, b, c respectively (as shown in Figure 5(a)). Moreover, we compare the counters of candidate buckets and choose the minimum one, and further replace item c with x. In the meantime, the counter of the bucket of $h3(x)$ increases by 1 up to 19. The kicked-out item c becomes the one needed to be inserted, and the insertion procedure goes on, until an empty slot is found in hash tables. Min Counter allows items to be inserted into hash tables to improve the storage space efficiency, but fails to fully address endless loops. we leverage an extra space to temporarily store the insertion-failure items rather than rehash the structure immediately. The viability of this approach has also been established in, where the authors show, through massiv analysis and simulations, that a very small constant size extra space yields significant improvements, dramatically reducing theinsertion failure probabilities associated with cuckoo hashing and enhancing cuckoo hashing's practical viability in both hardware and software. Some schemes also use the similar mechanism with a stash. Due to the negligible extra space, we construct a small hash table, in memory. The insertion-failure items store in the table through a random hash function for supporting operations in expected constant time.

Find Collision(α , hin)

Finding a collision pair with lengths 1 and α, starting from hin (20).

Variables:

1. α=desired length of second message.

2. A , B = lists of intermediate hash values.

3. q = a fixed “dummy" message used for getting the desired length.

4. hin = the input hash value for the collision.

5. htmp = intermediate hash value used in the attack.

6. M(i) = the ith distinct message block used in the attack.

7. n = width of hash function output in bits.

Steps:

1. Compute the starting hash for the α-block message by processing α- 1 dummy message blocks:

```
    htmp = hin.
    For i = 0 to α- 2:
    htmp = F(htmp, q)
```

2. Build lists A and B as follows:

```
    for i = 0 to $2^{n/2}$ - 1:
    A[i] = F(hin, M(i))
    B[i] = F(htmp, M(i))
```

3. Find i , j such that A[i] = B[j]

4. Return colliding messages (M(i), q||q||...||q||M(j)), and the resulting intermediate hash F(hin , M(i)).

Work: $\alpha - 1 + 2^{n/2+1}$ compression function calls.

Insert (Item x, kickcount, exclude)

```
if DirectInsert(Item x) then
        Return
end if
if kickcount ≤ MaxLoop then
        FindMinCounter(x, exclude) → k
        lock()
        D[hk(x)] → y
        x → D[hk(x)]
unlock()
C[hk(x)] ++
Insert(y, kickcount+1, k)
else
        x→S[x]
end if
```

Delete(Item *x*)

```
i = 1
while i ≤d do
        if (D[hi(x)] == x) then
                Delete x
                Return
        end if
        i ++
end while
        j = 1
while j ≤M do
        if (S[ j] == x) then
                Delete x
                Return
        end if
        j ++
end while
Return Result
```

FindMinCounter(Item x, exclude)

```
k = 1, m = 1
min = INTMAX
while m≤ d do
        if m! = exclude and C[hm(x)] ≤ min then
                C[hm(x)] → min
                m→k
        end if
        m++
end while
Return k /* find the minimum counter*/
```

A. Summary

Experimental results demonstrate Min Counter has the advantages in terms of the utilization ratio of hash tables, the total kicking out times and the time overheads. Min Counter can efficiently improve the utilization of cuckoo hash tables and decrease the rehash probability to optimize the cloud computing system performance. Meanwhile, it enhances experience of cloud users through decreasing the total kicking-out times and operation time.

IV. CONCLUSION

We predict cloud computing will grow, so developers should take it into account. Regardless of whether a cloud provider sells services at a low level of abstraction like EC2 or a higher level like App Engine, we believe computing, storage, and networking must all focus on horizontal scalability of virtualized resources rather than on single node performance. In order to alleviate the occurrence of endless loops, this paper proposed a novel concurrent cuckoo hashing scheme, named Min Counter, for large-scale cloud computing systems. The Min Counter has the contributions to three main challenges in hash based data structures, i.e., intensive data migration, low space utilization and high insertion latency. Min Counter takes advantage of "cold" buckets to alleviate hash collisions and decrease insertion latency. Min Counter optimizes the performance for cloud servers, and enhances the quality of experience for cloud users. Compared with state-of-the-art work, we leverage extensive experiments and real-world traces to demonstrate the benefits of Min Counter. We have released the source code of Min Counter for public use in Github at https://github.com/syy804123097/MinCounter.

REFERENCES

[1] M. Armbrust, A. Fox, R. Griffith, A. D. 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.

[2] C. Wang, K. Ren, S. Yu, and K. M. R. Urs, "Achieving usable and privacy-assured similarity search over outsourced cloud data,"

[3] Q. Liu, C. C. Tan, J.Wu, and G.Wang, "Efficient information retrieval for ranked queries in cost-effective cloud environments,"

[4] [N. Cao, C. Wang, M. Li, K. Ren, and W. Lou, "Privacy-preserving multi keyword ranked search over encrypted cloud data,"

[5] Y. Hua, B. Xiao, D. Feng, and B. Yu, "Bounded lsh for similarity search in peer-to-peer file systems,"

[6] R. P. Brent, "Reducing the retrieval time of scatter storage techniques,"*Communications of the ACM*,

[7] M. Zukowski, S. H´eman, and P. Boncz, "Architecture-conscious hashing,"*workshop on Data management on new hardware*, 2006.

[8] J. Triplett, P. E. McKenney, and J. Walpole, "Scalable concurrent hash tables via relativistic programming," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 3, pp. 102–109, 2010

[9] M. Mitzenmacher, "Some open questions related to cuckoo hashing,"

[10] J. Kelsey and B. Schneier, "Second preimages on n-bit hash functions for much less than 2 n work," *Proc.*

[11] M. M. Michael, "High performance dynamic lock-free hash tables and list-based sets," *Parallel algorithms and architectures*,

[12] B. Marsh, F. Douglis, and P. Krishnan. Flash memory file caching for mobile computers. In *Proceedings of the 27th Hawaii Conference on Systems Science*, Wailea, HI, Jan 1994.

[13] M. P. Mesnier and J. B. Akers. Differentiated storag services. *SIGOPS Oper. Syst. Rev.*, 45:45–53, February 2011.

[14] LBA Scrambler: A NAND Flash Aware Data Management Scheme for High-Performance Solid-State Drives Chao Sun, *Member, IEEE*, Ayumi Soga, Chihiro Matsui, Asuka Arakawa, and Ken Takeuchi, *Member, IEEE*

[15] A Re-configurable FTL (Flash Translation Layer) Architecture for NAND Flash based Applications Chanik Park, Wonmoon Cheon, Yangsup Lee, Myoung-Soo Jung, Wonhee Cho and Hanbin Yoon SAMSUNG Electronics. Co., Ltd., KOREA