Cheetah: An Efficient Flat Addressing Scheme for Fast Query Services in Cloud Computing

Yu Hua

Wuhan National Lab for Optoelectronics
School of Computer Science and Technology
Huazhong University of Science and Technology
Wuhan, China
E-mail: csyhua@hust.edu.cn

***Abstract*—Cloud computing generally needs to handle large amounts of data in a real-time manner. Typical metrics include fast write and query performance. However, existing cloud systems fail to efficiently offer fast query and write services due to two main reasons. One is the design separation between query and write operations. Most schemes mainly optimize one aspect. The other is the lack of cost-efficient data analytics, which leads to the identical resource consumption for each data item. In order to address the two problems and efficiently support fast query and write services, this paper proposes a novel flat-addressing scheme, called Cheetah. The idea behind Cheetah is to leverage efficient online data compression to reduce the amounts of data to be written, and meantime make use of a flat-addressing cuckoo hashing scheme to support fast query service. In practice, conventional cuckoo hashing schemes suffer from the endless loops, thus not only leading to insertion failure but also causing long operation latency. In order to alleviate the endless loops in item insertion, we use extra space to temporarily store the items that cause hash collisions, which are often shared by multiple loops. In order to further improve system performance, we prefetch the collision data in a batch to further reduce the probability of the occurrence of endless loops, thus offering fast query services to identify the searched items. Extensive experimental results demonstrate that the flat-addressing Cheetah can efficiently support query services in the cloud.**

I. Introduction

Cloud computing contains large amounts of heterogeneous resources, such as computation, storage and networks. In order to efficiently leverage these resources, fast query service is important, which can identify available and useful items. In the meantime, cloud systems often produce large amounts of data in burst, which needs to be handled in a real-time manner. The premise is to write these data into the permanent storage devices. In general, the speed of writing the large amounts of data is much larger than the limited I/O throughput, leading to potential data loss and long write latency. The bottleneck in terms of write and query operations exacerbates the overall performance of cloud computing.

One simple solution to improve the write performance is to reduce the amount of data to be written. Online data sanitization can be helpful to efficiently decrease the written data. Data sanitization is able to remove the identical data, e.g., data deduplication [1], and compress the similar data, e.g., delta compression [2], [3].

In general, the main function of data deduplication is able to divide files into multiple data chunks with the average size 4KB or 8KB. A chunk can be uniquely identified and detected in terms of redundancy via a secure hash signature (called a fingerprint). This fingerprint-based deduplication removes and deletes content redundancy at the chunk or file level, thus obtaining significant space savings and delivering high performance. Moreover, in order to compute the fingerprints of data chunks, Rabin fingerprint [4] has been widely used due to its ease of use and simplicity. The Rabin fingerprint exploits polynomial arithmetic to build and construct the unique representation. We need to carefully determine the length of the fingerprints, which is an important parameter to guarantee a low probability of hash collisions. In general, we use 64 bits to represent Rabin fingerprints to meet the needs of most real-world applications. Hence, while avoiding the high-complexity byte-by-byte comparisons, we can identify duplicate chunks via checking their fingerprints, and remove data redundancy. On the other hand, for data compression [2], [3], when two chunks are similar, data compression schemes need to compute their differences and maintain the mapping information [2], which has been widely used in networking transmission due to the significant decrease of the amounts of data to be transmitted [5]. In the context of network transmission, both sender and receiver possibly have the identical or similar files, and thus only transmitting their difference can obtain significant bandwidth savings and improve the transmission efficiency.

In order to offer real-time and fast query service in cloud computing and deliver high performance, we need to address the problem in conventional redundancy-oblivious design. Based on the observations from Los Alamos National Laboratory (LANL), Google and EMC datasets [6]–[8], we gain the following insights. In conventional cloud backup systems, source-end systems have little knowledge about the contents and layouts of remote backup servers. All data or their fingerprints have to be first transmitted to the backup servers, in which their redundancies are verified. If the redundancy exists, deduplication and compression are used to reduce the

978-1-4673-9953-1/16/$31.00 ©2016 IEEE

amounts of the transmitted data. This redundancy-oblivious methodology works in cloud computing due to some tolerance in backup delays, and however fails to meet the needs of completing hard-deadline recovery. The main reason is that this methodology overlooks the difference between backups and recovery, thus causing inefficiency and more costs in data recovery. A recovery needs to transmit data in backup servers to the source. Except the damaged data, the source and backup servers almost contain the same contents. The significant locality and similarity of data between source and backup servers allow the recovery to be executed in the larger granularity than conventional backups.

In practice, in order to carry out fast queries, hashing-based schemes have been widely used due to their simplicity, ease of use and fast access performance. Cuckoo hashing is a cost-efficient hashing scheme [9]. A cuckoo hashing offers open and flat addressing functions that can efficiently deliver high performance in indexing large-scale datasets. Moreover, in order to mitigate the hash collisions in hash functions, the cuckoo hashing makes use of a multi-hash dynamization of a static dictionary. Cuckoo hashing has salient features without the needs of dynamic memory allocation and can efficiently support query services among large-scale datasets in a flat-addressing manner. The efficiency of cuckoo hashing in queries exhibits a constant-scale lookup time. The main reason is that cuckoo hashing makes use of flat addressing among multiple choices of one stored item, thus being able to carry out independent and parallel query operations on multiple candidate positions.

Cuckoo hashing leverages two hash tables, i.e., T_1 and T_2 with the length n, as well as two hash functions h_1, h_2: $U \rightarrow \{0,\ldots,n-1\}$. We store and maintain each key $x \in S$ in cell $h_1(x)$ of T_1 or $h_2(x)$ of T_2. The hash functions h_i demonstrates the properties of independent and random distribution. In the cuckoo hashing, conventional insertion operation leverages random-walk approach, which unfortunately suffers from endless loops and causes high costs of potential reconstruction of entire structure. In order to address the problem of endless loops, we need to improve the efficiency of data insertion and meanwhile exploit the properties and requirements of real-world applications in cloud computing.

The cuckoo hashing can efficiently support fast write and query services via flat addressing scheme and significantly improve the overall performance of large-scale cloud systems. For example, cuckoo hashing can be used in data deduplication for cloud computing. The main performance bottleneck of data deduplication is the indexing efficiency, which can be significantly improved by flat-addressing cuckoo hashing. Moreover, in order to alleviate the endless loops and improve the performance of data sanitization in cloud computing, we leverage an active prefetching scheme in the flat-addressing cuckoo hashing. Specifically, we have the following contributions.

Efficient Data Sanitization. In order to efficiently support data query service in cloud computing, Cheetah proposes to leverage coarse-grained (i.e., feature-based) data sanitization that consists of deduplication and delta compression components. Due to the significant locality and similarity of data between source and backup servers, the coarse granularity allows the recovery to be faster and more efficient. Cheetah hence removes duplicate data and compress similar data to offer efficient data sanitization. *Cheetah goes far beyond the simple combination of deduplication and delta compression.* In general, the two data reduction approaches work independently. In Cheetah, to support coarse-grained data sanitization, we exploit and explore the duplicate-relevance patterns, as well as data locality, to identify similar data to be compressed and deliver high performance. Moreover, in order to efficiently handle the massive data, we also leverage coarse-grained grouping via locality sensitive hashing (LSH) [10] to reduce the amounts of the analyzed data.

Active Prefetching to Mitigate Endless Loops. Cuckoo hashing can efficiently improve the write and query performance in cloud computing. However, the cuckoo hashing scheme potentially introduces the endless loops when inserting new data items. The endless loop possibly leads to the expensive reconstruction of the entire cloud system. In order to mitigate the occurrence of endless loops, we propose to use an extra space to temporarily store and maintain the data items causing hash collisions. Unlike existing schemes that mainly place items into the space, we actively prefetch the items that are shared by multiple loops, thus significantly reducing the probability of endless loops. In order to further improve the efficiency of identifying similar data for compression, Cheetah exploits and leverages the relevance from deduplication. Due to the significant locality of similar data, the non-duplicate chunks that are close to the duplicate ones are considered to be potential candidates for delta compression. The relevance-aware detection can judiciously identify the chunks to be delta compressed, which significantly narrows the scope of retrieval and reduces the computation and space overheads.

Prototype Implementation and Real-world Evaluation. We have implemented all components of Cheetah between two data centers, each of which contain multiple servers. The implemented prototype can compute fingerprints for deduplication and features for delta compression, which are stored together in cache units, called storage containers. We also implement the active prefetching scheme in the flat-addressing cuckoo hashing, which can efficiently improve the data insertion performance and meantime offer the fast write and query services in cloud computing. We examine the real performance of Cheetah by using multiple real-world datasets, including Los Alamos National Laboratory, HP, MSN, Google and EMC.

The rest of the paper is organized as follows. Section II presents Cheetah design and implementation details. Section III shows the metadata index management in DRAM. Section IV shows the locality-enabled data storage in disks. Section V presents the performance evaluation. Section VI presents the related work. We conclude our paper in Section VII.

II. The Cheetah Design

In this section, we present the details of Cheetah design. The cost-effective Cheetah is to synergize cuckoo hash indexing and deduplication schemes to meet the needs of fast query and write services in cloud computing. Data deduplication is a dictionary based data reduction and compression approach that has been widely used in cloud computing in terms of network transmission and backups. Besides duplicate data to be removed, some data are non-duplicate but very similar. Delta compression is used to further compress similar data and remove redundancy among non-duplicate but very similar files, which data deduplication often fails to identify and eliminate. Existing schemes generally combine the two techniques that are executed independently, thus failing to deliver high performance. In fact, duplicate data exhibits more similarity that can be well exploited and explored to handle the large amounts of redundant data in a cost-efficient manner.

A. The Flat-addressing Design Methodology

In order to improve query performance, we propose to use loop-oblivious cuckoo hashing that supports constant-scale query complexity, as well as an application-aware deduplication scheme. Specifically, we label the nodes that cause endless loops to be "labeled", and we only remove the redundancy of labeled nodes. The occurrence of endless loops in cuckoo hashing stems from the existence of labeled nodes with hash collisions. Cheetah can efficiently mitigate the endless loops in item insertion. In the meantime, Cheetah optimizes the locality of data accesses, since not all redundant data are removed. The rationale is that the large amounts of massive data in cloud computing systems are often stored in low-speed and large-capacity storage devices, e.g., hard disks.

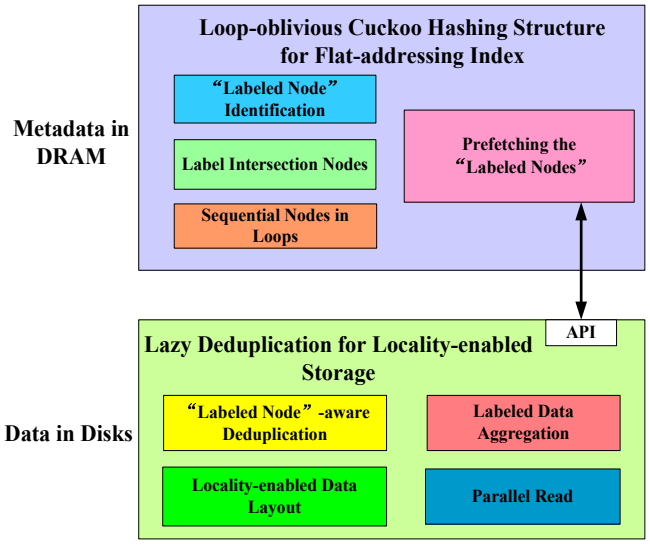


Fig. 1. The Cheetah architecture.

In order to improve entire system performance, we leverage the "divide and conquer" scheme respectively handling metadata index in DRAM and data storage in disks to efficiently support the flat-addressing scheme. As shown in Figure 1, the flat-addressing Cheetah consists of two main components, i.e., loop-oblivious cuckoo hashing structure for metadata in DRAM, and lazy deduplication for locality-enabled data storage in disks. The two components communicate with the specified API that indicates the labeled data. We mainly carry out the deduplication upon the labeled data. The benefits are to remove redundant data and meantime mitigate the endless loops.

B. The Index Structure for Metadata

Cheetah makes use of a flat-addressing index structure to store and maintain the metadata of large-scale datasets. This structure consists of the constant-scale cuckoo hashing scheme that unfortunately leads to endless loops in data insertion. Existing schemes fail to efficiently address this problem due to the use of passive-avoidance methodology. For example, in order to avoid or break a loop, one frequently-kicked-out node, i.e., the newly inserted data, is removed or moved to a new position. This operation can temporarily alleviate the endless loop (e.g., one time). A new incoming data item may still result in the endless loop, thus leading to potential performance decrease or even data loss.

In order to improve the entire system performance and significantly reduce the probability of occurrence of endless loops, Cheetah leverages an active prefetching scheme. Specifically, the node causing an endless loop is first labeled. In practice, it is difficult to accurately identify a loop since the data insertion generally consumes long time for frequent kick-out operations among the data with hash collisions. One solution is to pre-define a threshold. If the latency of kick-out operations is larger than the threshold, an endless loop is assumed to occur, which may introduce some false positives. The false positive means that some data without collisions are assumed to cause loops. More importantly, it is more challenging to define a suitable and accurate threshold in advance. The threshold values can determine the actual performance of data insertion.

In order to alleviate the difficulty of accurately defining the initial threshold, we exploit the property of cloud computing applications. In Cheetah, we define that a loop occurs if a node is kicked-out by the same prefix node. This definition may slightly increase the latency of data insertion. In order to address this problem and accelerate the processing speed, we use the data deduplication and parallel operations to improve the write performance.

C. Locality-enabled Storage for Data

The large-scale datasets are often stored in disks for the long-term storage. Moreover, due to the physical properties, the low-speed hardware devices, like disks, can obtain better query performance with the aid of high data locality. Cheetah tries to maintain the data locality that is important to improve the query or prefetching performance. Correlated data are aggregated together and placed into the same or adjacent positions in disks. We further present the implementation

details of Cheetah in terms of indexing structure in DRAM and locality-enabled data storage in disks.

III. The Metadata Index in DRAM

A. The Labeled Nodes in a Loop

The index structure in Cheetah leverages the cuckoo hashing based scheme due to its fast query performance and efficient data storage. However, the conventional cuckoo hashing has to handle the potential endless loop due to hash collisions. In Cheetah, we identify the nodes, which cause endless loops, by checking their prefix kicking-out node. In general, a loop appears if a node is kicked out by its identical prefix node.

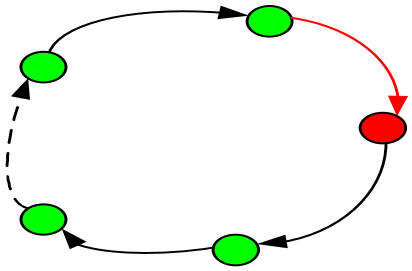


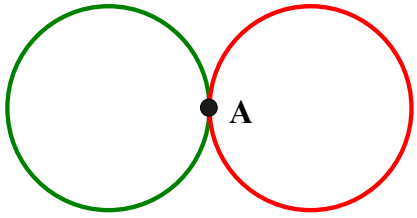
Fig. 2. A labeled node causing a loop.

As shown in Figure 2, a loop appears if the red node is kicked out by the same prefix node again. We hence need to record the information of the prefix node, thus identifying the occurrence of an endless loop. In general, during the frequent kicking-out operations, due to the concurrent operations of inserting multiple data items, it is difficult to accurately determine if a loop occurs, since parallel insertion may also lead to the repeated kicking-out operations upon a node. In the context of cloud storage, as we previously discussed, this application needs to carry out fast write operation and requires efficient data layout. Hence, in data insertion, we carry out the kicking-out operations within a limited area, in which only one insertion operation occurs. Thus, we can accurately determine the occurrence of a loop. We further label the red node in a loop as shown in Figure 2.

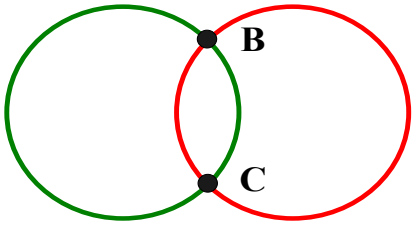
B. Label Intersection Nodes

1) Two Loops: In order to alleviate or avoid the endless loops, we need to break or disconnect the loops via removing one or more nodes. There exist multiple cases of building the loops based on the relationships of multiple loops. First, if one loop has no shared nodes with others, we can randomly select the nodes within the loop, which are moved to the extra storage space for temporarily storing the data causing hash collisions. Moreover, the endless loops often share the nodes with other loops due to hash collisions. For example, as shown in Figure 3, two adjacent and intersected loops may share one or two nodes, thus respectively becoming tangent and intersected.

We present the details of two cases. Specifically, as shown in Figure 3(a), two loops are tangent and share only one node. We observe that the tangent node is important to construct the two loops. If this node is removed, we can break two loops at the same time, thus significantly reducing the risks of building loops in the incoming insertion. We hence need to accurately identify the tangent node that is repeatedly kicked out by two different nodes. If a node is kicked out by its two different prefix (i.e., one-hop) nodes, this node is defined as a tangent node. In general, we need to remove this tangent node. In the meantime, Cheetah randomly selects and removes some other nodes from different loops by moving these nodes to the extra storage space, thus further reducing the risks of loop construction.



(a) One shared node in two loops.



(b) Two shared nodes in two loops.

Fig. 3. The general cases in the interaction of two correlated loops.

Moreover, as shown in Figure 3(b), two intersected loops share two different nodes. The intersected nodes have larger probability of causing the endless loops, compared with other nodes in the loops. We hence need to remove the intersected nodes from the loops, and move and store them into the extra storage space, thus reducing the risks of loop construction. Another choice of alleviating the risks of building the loop is to explore and exploit the adjacent and neighboring positions of the intersected nodes to support the approximate storage, like NEST [11].

2) Multiple Loops: As shown in Figure 4, three loops produce six shared nodes. It is worth noting that if two loops becoming tangent and sharing one node can be considered as a special case, which can also be guaranteed by our analysis. For handling multiple loops, we can identify the labeled node via the same definition like the "two-loop" case. We label a node if it is repeatedly kicked out by its different prefix nodes. Unlike the "two-loop" case, the multi-loop case introduces more labeled nodes. In fact, it is difficult to identify all labeled nodes with a limited interval due to the different lengths of kicking-out paths.

Extra waiting time may exacerbate the overall write performance in data insertion. We hence propose to use time or event-driven approaches based on the requirements of real-world applications. Specifically, the time-driven scheme means

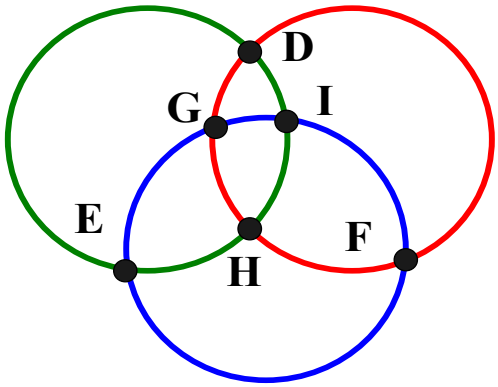


Fig. 4. Multiple loops share the labeled nodes.

that we deal with the labeled nodes in a batch after a pre-defined time. The benefits of using the time-driven scheme are to support hard-deadline scheduling and guarantee the quality of write operation. However, the time-driven scheme fails to exhibit the access patterns of inserting new items and offer flexible and dynamic management upon arriving data. On the other hand, the event-driven scheme means that Cheetah needs to prefetch the labeled data to the extra storage space when the capacity is full based on the predefined number. The event-driven scheme possibly increases the complexity of scheduling the insertion operations within different loops. Therefore, the in-batch schemes can be determined via the requirements of real-world applications.

IV. Locality-enabled Data Storage in Disks

In this section, we present the implementation details of storing data in disks. In order to efficiently support the query performance, we need to improve the data locality in disks, which can facilitate the prefetching efficiency. In the meantime, in order to mitigating the side effect of deduplication in terms of producing segments, we propose to carry out the deduplication operations upon the labeled, not all, redundant data.

A. "Labeled Node"-aware Deduplication

A deduplication can result in the segmented data that exacerbate the query performance. Originally continuous data chunks are placed into different positions. A read operation has to frequently visit different disks or even remote servers. In order to improve read performance and reduce the access costs, we propose to mainly remove the redundant and labeled nodes. Hence, the storage devices need to obtain the knowledge of the labeled nodes via the specified interface. We can reduce the redundancy of the labeled nodes and in the meantime, set a pointer to the actual data.

The deduplication upon labeled nodes can efficiently improve read performance via exploiting the data locality, while decreasing the data redundancy and spatial overhead. Cheetah hence obtains a suitable tradeoff between the data deduplication and query performance.

B. Labeled Data Aggregation for Improving Data Locality

We leverage the locality sensitive hashing (LSH) [10] to aggregate and group the similar data together to improve the data locality and optimize the data layout in hard disks. Conventional content-based approaches fail to efficiently identify duplicate and similar data due to handling large amounts of data with high-complexity computation overhead. The bottleneck comes from probing and analyzing the practical contents of the large amounts of data. In order to alleviate the computation complexity and narrow the processing scope, we make use of light-weight coarse-grained grouping scheme to pre-process the massive data. Unlike existing methods to analyze the contents, our scheme first groups similar data via checking their multi-dimensional metadata with the aid of locality sensitive hashing.

Cheetah uses LSH to identify potential duplicate and similar data. In general, two data items a and b with d-dimensional attributes are represented as semantic vectors $\vec{a_d}$ and $\vec{b_d}$. If their distance in terms of vectors $\vec{a_d}$ and $\vec{b_d}$ is smaller than a pre-defined threshold, we consider that the two items are close-by and similar [12]. We further leverage LSH to map similar data items into the same or adjacent hash buckets with a high probability. In this way, the metadata-based grouping and aggregation schemes can significantly reduce the amounts of data to be probed and processed.

In the LSH design, for a given query request, Cheetah needs to first hash the query point q into the hashed buckets that exist in multiple hash tables. In order to improve query accuracy, Cheetah further combines all hashed items together and then rank them based on the evaluation metric, in which we compute their distances to the query point q. We hence obtain the similar items for the queried point.

LSH function family is able to identify data similarity and carry out fast classification due to its salient property that the close-by data items can be hashed into one hash bucket with a higher probability than the far-apart items. Specifically, S is defined as the item domain, and distance functions $||*||$ exhibit different LSH families of l_s norms within the s-stable distribution. In this way, Cheetah allows each hash function $h_{a,b} : R^d \rightarrow Z$ to map a d-dimensional vector v onto a set of integers, which can efficiently support fast query service.

Definition 1: LSH functions, i.e., $\mathbb{H} = \{h : S \rightarrow U\}$ can represent (R, cR, P_1, P_2)-sensitive property based on the distance function $||*||$ for any $p, q \in S$

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

In order to support aggregation based fast queries and offer efficient similarity identification, Cheetah allows $c > 1$ and $P_1 > P_2$. In Cheetah, we leverage multiple hash functions to increase the gap between P_1 and P_2. Cheetah also allows the hash function in $\mathbb{H}$ to be $h_{a,b}(v) = \lfloor \frac{a \cdot v + b}{\omega} \rfloor$, in which a is a d-dimensional random vector with the s-stable distribution. In the meantime, we define b to be a real number selected from the range $[0, \omega)$, in which ω is a constant. After carrying out the coarse-grained grouping, we identify the data that may

be duplicate or similar. Cheetah further executes fine-grained identification for similar data in the groups.

C. The Cheetah Components for Cost-efficient Cloud Recovery

Figure 5 shows the Cheetah components to efficiently support data sanitization. Cheetah contains three-level design, i.e., storage, index and function units. For an incoming recovery stream, Cheetah first executes the duplicate detection. The data stream is first chunked via the CDC approach [13]. Cheetah further computes the fingerprints via SHA-1, as well as features, and groups the correlated data into the container of sequential chunks to maintain and improve the data locality. In order to mitigate the variable sizes of chunks, a large and fixed-size container is used in Cheetah. We hence consider the container serving as the basic unit of read and write operations to improve the I/O throughput. Cheetah is able to improve compression performance during cloud recovery via exploring and exploiting the similarity matching across the containers. Moreover, Cheetah makes use of the similarity detection module in sanitization to identify correlated and close-by chunks in the containers.

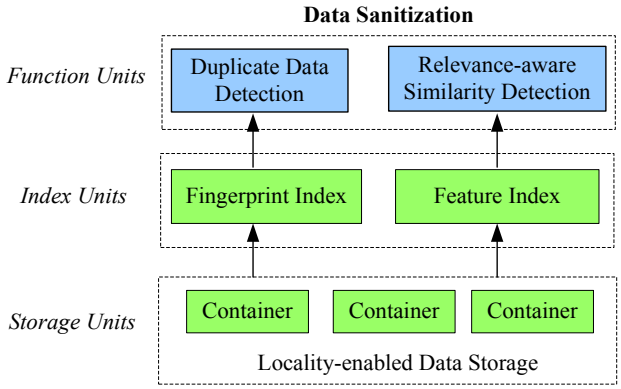


Fig. 5. The components of data sanitization.

For each of the similar chunks identified by the data sanitization, Cheetah reads its base-chunk, and then delta encodes their differences via the Xdelta algorithm [2]. Furthermore, for data recovery, Cheetah needs to read the referenced files from the stored containers on disks based on the mapping information, which can facilitate the fast recovery in cloud computing. Cheetah also aggregates the chunks into the containers to improve and maintain the data locality. A recovery in Cheetah carries out the container-size operation to support the query and prefetching.

Data sanitization in Cheetah consists of chunk-level deduplication and delta compression. First, the deduplication eliminates duplicate data, thus effectively improving the space efficiency and enhance the network transmission. The deduplication divides a data stream into variable-size chunks. The identified redundant chunks are placed and queried via the pointers to the previously stored copies in a cost-efficient manner. In order to accelerate the query performance, Cheetah uses a fingerprint index to map fingerprints of the stored chunks to their physical addresses, which can be updated with the insertion and deletion of data items. Second, the feature-based delta compression in fact is an efficient scheme to improve network transmission, which exploits and leverages the similarities between chunks via feature-based comparisons. Thus, the significantly small compressed files, i.e., the difference (or delta) between two files, are transmitted across the long-distance network to improve the performance of real-time cloud recovery.

V. Performance Evaluation

In this section, we present the experimental results of Cheetah in terms of multiple performance metrics for cloud computing. Cheetah leverages the flat-addressing cuckoo hashing scheme to improve query performance, as well as using data sanitization to reduce the amounts of data to be written. Cheetah also proposes the active prefetching approach to alleviate the probability of occurrence of endless loops. Unlike conventional system-level deduplication for exact-matching chunks, Cheetah offers near-duplicate sanitization based on the semantic correlation that is derived from both metadata and contents.

A. Experiments Setup

We have implemented the mentioned components of Cheetah between two data centers with more than 1500km distance. One center contains 128 servers, each of which is equipped with an 8-core CPU, a 32GB RAM, a 500GB 7200RPM hard disk and Gigabit network interface card.

To the best of our knowledge, due to the lack of real-world workloads of large-scale data centers with comprehensive information for public use, we choose to emulate typical data center traces via generating the access patterns along the lines of traffic distributions. To examine the system performance, we leverage five real-world datasets, i.e., Los Alamos National Laboratory (LANL) dataset [6], HP file system trace [14], MSN trace [15], Google clusters [7] and EMC recovery systems [8]. We further describe the characteristics of these datasets as follows.

- Los Alamos National Laboratory (LANL) offers multiple datasets [6]. These datasets demonstrate the properties of files' attributes in multiple dimensions. The entire LANL dataset is about 19GB and consists of 112 million lines of archive data and roughly 9 million lines of home/project space data. In the LANL dataset, the main attributes contain unique ID, file sizes (in bytes), creation time, modification time, block sizes (in bytes) and the paths to files.
- HP file system contains a 10-day 500GB trace [14], which is visited by 236 users. The HP dataset presents the practical operations, including READ, WRITE, LOOKUP, OPEN, and CLOSE, on the accessed files that have file names and device numbers.
- MSN trace [15] exhibits the multi-dimensional metadata information in a 6-hour period. The entire MSN dataset

has been divided into multiple 10-minute intervals. MSN trace records 3.3 million "READ" and 1.17 million "WRITE" operations on 1.25 million files. Given an examined interval, MSN trace allows to query the files with multi-dimensional attributes, including access time, the amounts of READ, the amounts of WRITE, operational sequence IDs and file size.

- Google shows the anonymized log data from the clusters [7]. This dataset is a collected trace in a 7-hour period. The trace presents a set of tasks, each of which runs on a single machine with memory and one or more cores. The Google trace describes 3,535,029 observations, 9218 unique jobs and 176,580 unique tasks.
- More than 650 file-system snapshots are collected in MacOS production servers at the FSL lab in Stony Brook University, in collaboration with Harvey Mudd College and EMC [8]. The dataset presents daily file system scans, which have been anonymized and compressed, from 2011 to 2014 with 1.1TB capacity in size. The snapshots having multi-dimensional metadata information contain the hashes of all chunks in the scanned files. During collecting the snapshots, the dataset exhibits 2KB, 4KB, 8KB, 16KB, 32KB, 64KB, and 128KB chunking sizes.

We randomly allocate the available data segments or snapshots among 128 servers. We also allow a client to capture the system operations from these traces and then send the requests for various operations to servers. Cheetah allows both clients and servers to execute multiple parallel threads and leverage TCP/IP to exchange messages and data in an efficient manner.

In order to obtain a suitable tradeoff between the deduplication efficiency and query performance, we performed a large number of experiments to determine a suitable size of the container, i.e., the k value. The k value in Cheetah represents the data locality and I/O efficiency. To select suitable k values, the used metric is the normalized rate that is the value of the saved bandwidth divided by the indexing latency. We count the bandwidth against different k values and further compute their normalized rates between the minimum and maximum values. Figure 6 shows the normalized values when using different k values. We observe that the selected k value is 4MB in Google, MSN, EMC, LANL and HP traces.

B. Results and Analysis

We present the evaluation results and analysis in terms of multiple performance metrics.

Effective Network Throughput. In order to comprehensively and accurately evaluate the query and transmission performance using the flat-addressing Cheetah, we carry out multiple remote communication experiments. The main metric is the effective network throughput between two remote data centers, which can communicate the queried results in a near real-time manner.

Figure 7 shows the results of the average values of using multiple datasets. The network throughput is measured every 50 minutes. Compared with the baseline approach, the average

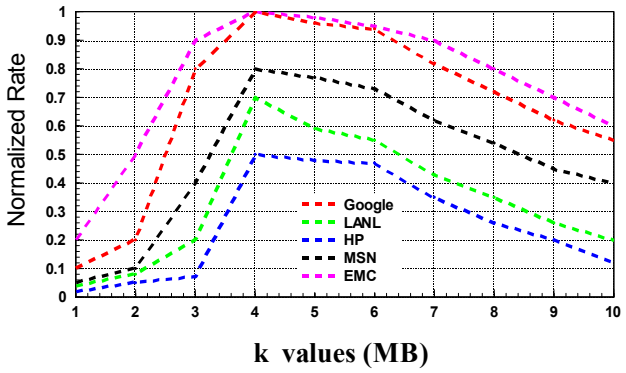


Fig. 6. The suitable k value.

effective throughput in Cheetah is 526.3Mb/s, much larger than the Baseline, 25.2Mb/s. The Baseline approach uses the exact-matching fingerprints to determine the similarity, which consumes substantial system resources. Cheetah obtains much better network throughput. The main reason is the use of Cheetah that can offer fast queried results and significantly reduce the query latency. In the meantime, the data sanitization also helps reduce the amounts of the transmitted data, thus improving the transmission efficiency. Cheetah makes use of the relevance-aware similarity detection, which is helpful to identify more similar chunks to be compressed and reduce the processing latency.

Recovery Overhead. For a cloud recovery system, it is important to examine the data recovery performance and the overheads in terms of write and query operations. Cheetah makes use of the flat-addressing cuckoo hashing to improve the query performance, as well as the data sanitization to reduce the amounts of data to be written. Figure 8 shows the recovery time by examining the related operations. We observe that the recovery time is approximately linear to the number of files to be recovered.

Computation Overhead. Data sanitization is used in Cheetah to improve the effective throughput, as well as query and write performance, which however causes extra computation and disk I/O overhead. In order to comprehensively evaluate this metric, we examine the overheads respectively in the local and remote servers. Specifically, we examine the storage capacity overhead that is consumed to store the fingerprints. Cheetah maintains the data chunk and the fingerprints in the containers. Due to small sizes, the disk I/O overhead is small (around 3.6%). Furthermore, in order to examine the overhead in carrying out the data sanitization in terms of computation cost, we examine the real CPU utilization in both source and destination servers as shown in Figure 9.

In the source servers, the CPU utilization is around 4.36% to compute the identical and similar chunks via the data sanitization, thus removing redundant data and compressing the similar data. We observe that in the destination server, the CPU utilization shows an increasing trend, from 5.5% to 12.7%. This acceptable increase stems from the increasing CPU overhead, which is almost linear with the number of the

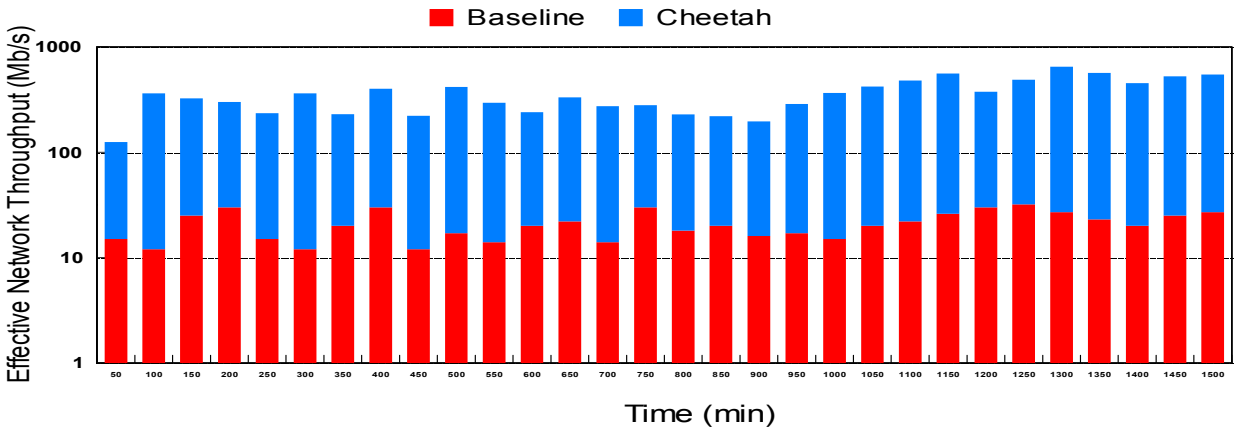


Fig. 7. The effective throughputs in Baseline and Cheetah schemes.

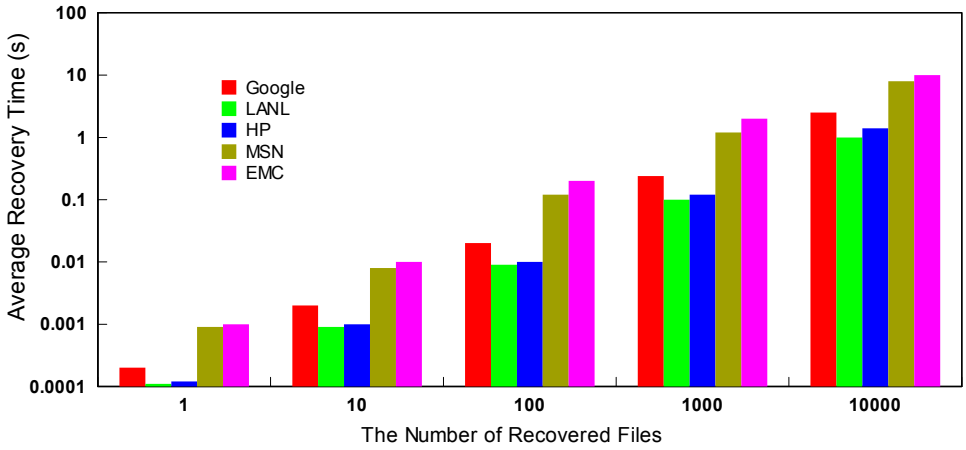


Fig. 8. The average recovery time.

transmitted data.

Time Overhead. The time overhead in Cheetah is used to execute the data sanitization, including both deduplication and delta compression, as shown in Figure 10. Since Cheetah needs to carry out both deduplication and delta compression, a slight extra time overhead is introduced in the relevance-aware similarity detection. Due to the significant bandwidth savings, the extra time overhead is acceptable.

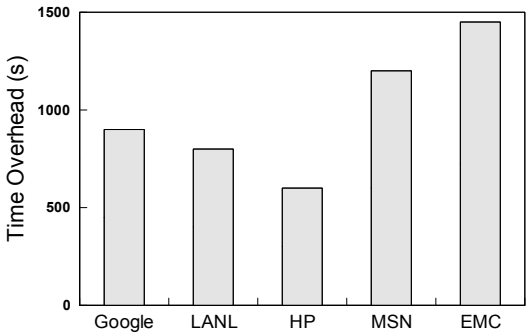


Fig. 10. Time overhead in the data sanitization.

VI. Related Work

Cloud computing requires cost-effective query services. Multiple hashing-based approaches have been proposed to improve system performance. Cuckoo Filter [16] supports approximate, not exact, set membership query by storing the fingerprints of items based on cuckoo hashing. NEST [11] leverages cuckoo-driven locality-sensitive hashing to address the imbalanced loads in the traditional locality-sensitive hashing and support approximate queries, which achieves performance improvements in space utilization and supports fast query response. Based on the shortest-queue selection, caching policy is used to minimize the query latency in variable network loads [17]. Moreover, a novel approach is proposed to leverage compressive sensing to efficiently reconstruct missing data [18]. In order to alleviate the endless loops in cuckoo hashing, a constant-size stash is used to reduce failure rates [19].

As an important research topic, network-wide redundancy elimination has received many attentions from both academia and industry [20], [21]. EndRE [22] proposes an adaptive SampleByte algorithm to compute fingerprints and leverages an optimized data structure to significantly reduce cache memory

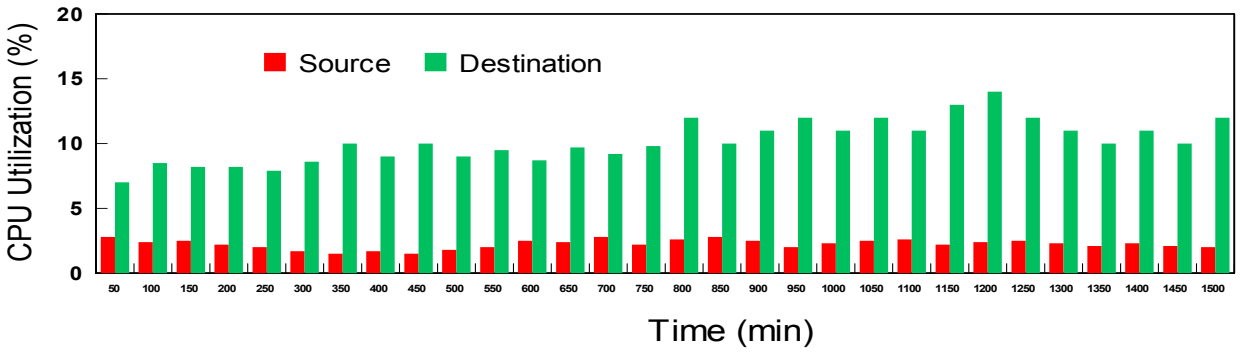


Fig. 9. The computation overhead.

overhead. Stream-informed delta compression [3] is proposed to improve the performance in existing deduplication systems. In order to avoid imbalance, Cluster-Based Deduplication [23] obtains the tradeoffs between stateless data routing approaches with low overhead and stateful approaches with high overhead. In order to improve data deduplication efficiency, Silo [24] serves as a similarity-locality deduplication system.

VII. Conclusion

In order to offer fast query and write services, we propose a novel flat-addressing hashing scheme, called Cheetah. Cheetah leverages active prefetching scheme to reduce the probability of the occurrence of the endless loops in the cuckoo hashing. Data sanitization is used to further reduce the amounts of data to be transmitted. We use multiple real-world traces to evaluate the performance of Cheetah. Extensive experimental results demonstrate the efficiency of Cheetah in fast query service for cloud computing. We plan to release the source codes for public use in the near future.

Acknowledgements

This work was supported in part by National Natural Science Foundation of China (NSFC) under Grant 61173043 and State Key Laboratory of Computer Architecture under Grant CARCH201505.

References

[1] B. Zhu, K. Li, and H. Patterson, "Avoiding the disk bottleneck in the data domain deduplication file system," *Proc. FAST*, 2008.
[2] J. MacDonald, "File system support for delta compression," *Masters thesis, EECS, University of California at Berkeley*, 2000.
[3] P. Shilane, M. Huang, G. Wallace, and W. Hsu, "WAN Optimized Replication of Backup Datasets Using Stream-Informed Delta Compression," *Proc. USENIX FAST*, 2012.
[4] M. Rabin, "Fingerprinting by random polynomials," *Technical Report TR-81-15, Aiken Laboratory, Harvard University*, 1981.
[5] Y. Hua, X. Liu, and H. Jiang, "ANTELOPE: A Semantic-aware Data Cube Scheme for Cloud Data Center Networks," *IEEE Transactions on Computers (TC)*, vol. 63, no. 9, pp. 2146–2159, 2014.
[6] Los Alamos National Lab (LANL) File System Data, *http://institute.lanl.gov/data/archive-data/*.
[7] J. L. Hellerstein, "Google Cluster Data," *http://googleresearch.blogspot.com/2010/01/google-cluster-data.html*, Jan.2010.
[8] EMC Backup File-system Snapshots *http://tracer.filesystems.org/*, 2014.
[9] R. Pagh and F. Rodler, "Cuckoo hashing," *Proc. ESA*, pp. 121–133, 2001.
[10] P. Indyk and R. Motwani, "Approximate nearest neighbors: towards removing the curse of dimensionality," *Proc. STOC*, pp. 604–613, 1998.
[11] Y. Hua, B. Xiao, and X. Liu, "NEST: Locality-aware Approximate Query Service for Cloud Computing," *Proc. INFOCOM*, 2013.
[12] S. Lee, S. Chun, D. Kim, J. Lee, and C. Chung, "Similarity search for multidimensional data sequences," *Proc. ICDE*, 2000.
[13] A. Muthitacharoen, B. Chen, and D. Mazieres, "A low-bandwidth network file system," *Proc. SOSP*, 2001.
[14] E. Riedel, M. Kallahalla, and R. Swaminathan, "A framework for evaluating storage system security," *Proc. FAST*, pp. 15–30, 2002.
[15] S. Kavalanekar, B. Worthington, Q. Zhang, and V. Sharda, "Characterization of storage workload traces from production Windows servers," *Proc. IEEE International Symposium on Workload Characterization (IISWC)*, 2008.
[16] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher, "Cuckoo filter: Practically better than bloom," *Proc. CoNEXT*, 2014.
[17] M. Bjorkqvist, L. Y. Chen, M. Vukolic, and X. Zhang, "Minimizing retrieval latency for content cloud," *Proc. INFOCOM*, 2011.
[18] L. Kong, M. Xia, X. Liu, G. Chen, Y. Gu, and M. Wu, "Data loss and reconstruction in wireless sensor networks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 11, pp. 2818–2828, 2014.
[19] 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.
[20] A. Anand, C. Muthukrishnan, A. Akella, and R. Ramjee, "Redundancy in network traffic: findings and implications," *Proc. SIGMETRICS*, 2009.
[21] A. Anand, A. Gupta, A. Akella, S. Seshan, and S. Shenker, "Packet caches on routers: the implications of universal redundant traffic elimination," *Proc. ACM SIGCOMM*, 2008.
[22] B. Aggarwal, A. Akella, A. Anand, A. Balachandran, P. Chitnis, C. Muthukrishnan, R. Ramjee, and G. Varghese, "EndRE: an end-system redundancy elimination service for enterprises," *Proc. NSDI*, 2010.
[23] W. Dong, F. Douglis, K. Li, H. Patterson, S. Reddy, and P. Shilane, "Tradeoffs in scalable data routing for deduplication clusters," *Proc. USENIX FAST*, 2011.
[24] W. Xia, H. Jiang, D. Feng, and Y. Hua, "SiLo: A Similarity-Locality based Near-Exact Deduplication Scheme with Low RAM Overhead and High Throughput," *Proc. USENIX Annual Technical Conference*, 2011.