# Adaptive Online Cache Capacity Optimization via Lightweight Working Set Size Estimation at Scale

Rong Gu, Simian Li, Haipeng Dai, Hancheng Wang, and Yili Luo, *State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210023, China;* Bin Fan, *Alluxio Inc;* Ran Ben Basat, *University College London;* Ke Wang, *Meta Inc;* Zhenyu Song, *Princeton University;* Shouwei Chen and Beinan Wang, *Alluxio Inc;* Yihua Huang and Guihai Chen, *State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210023, China*

https://www.usenix.org/conference/atc23/presentation/gu

This paper is included in the Proceedings of the 2023 USENIX Annual Technical Conference.

July 10–12, 2023 • Boston, MA, USA

978-1-939133-35-9

# Adaptive Online Cache Capacity Optimization via Lightweight Working Set Size Estimation at Scale

Rong Gu[1] Simian Li[1] Haipeng Dai[1] Hancheng Wang[1] Yili Luo[1] Bin Fan[2] Ran Ben Basat[3]
Ke Wang[4] Zhenyu Song[5] Shouwei Chen[2] Beinan Wang[2] Yihua Huang[1] Guihai Chen[1]
*State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210023, China*[1]
*Alluxio Inc*[2] *University College London*[3] *Meta Inc*[4] *Princeton University*[5]

## Abstract

Big data applications extensively use cache techniques to accelerate data access. A key challenge for improving cache utilization is provisioning a suitable cache size to fit the dynamic working set size (WSS) and understanding the related item repetition ratio (IRR) of the trace. We propose Cuki, an approximate data structure for efficiently estimating online WSS and IRR for variable-size item access with proven accuracy guarantee. Our solution is cache-friendly, thread-safe, and light-weighted in design. Based on that, we design an adaptive online cache capacity tuning mechanism. Moreover, Cuki can also be adapted to accurately estimate the cache miss ratio curve (MRC) online. We built Cuki as a lightweight plugin of the widely-used distributed file caching system Alluxio. Evaluation results show that Cuki has higher accuracy than four state-of-the-art algorithms by over an order of magnitude and with better stability in performance. The end-to-end data access experiments show that the adaptive cache tuning framework using Cuki reduces the table querying latency by 79% and improves the file reading throughput by 29% on average. Compared with the cutting-edge MRC approach, Cuki uses less memory and improves accuracy by around 73% on average. Cuki is deployed on one of the world's largest social platforms to run the Presto query workloads.

## 1 Introduction

Nowadays, distributed data-intensive frameworks like Flink [9], Spark [49], Presto [37], which frequently read data from tables and files, commonly use a caching layer as one key optimization to improve data accessing performance. However, allocating the right amount of cache storage can be non-trivial: excessive resource unnecessarily increases the cost, while insufficient capacity degrades the performance. Dynamic online workloads [24, 42] make this problem even more challenging. Particularly, when operating the Presto deployments, we introduced Alluxio [1, 25] as its caching layer and observed a high cache hit ratio. It was important, however, unclear to us based on existing cache metrics to tell if we could reduce the cache capacity of Presto servers while maintaining the high cache hit ratio.

Existing approaches about how to tune the cache capacity can be mainly summarized into four categories: (1) Rule-based approaches [21, 24, 34, 42] tune cache sizes based on cache metric related rules. However, it always adjusts the cache size to fit the working set size blindly and frequently. (2) ML-based approaches [4, 28, 30, 33, 35] train machine learning models with historical data offline and predict proper cache sizes in the future. Nevertheless, the model might be inaccurate under online dynamic workloads. (3) MRC-based approaches [15,19,22,36,40,41,45,51,52] explore the optimal cache size by exploring a miss ratio curve (MRC) as the function of the cache size. However, MRC is generated by assuming each item has the same size or cost, which is not always true in practice. (4) Window-based approaches [5, 11, 20] determine the cache capacity by estimating the cardinality of items in a sliding window. But, it can not estimate the working set size of items in variable size or organized in multiple scopes.

Understanding the online accurate working set size (WSS) and item repetition ratio (IRR) is important for tuning appropriate cache capacity [35, 51].

Accurate WSS estimation supports better cache capacity planning, leading to higher cache hit rate and significant end-to-end performance improvement. In the cluster, WSS and IRRs insight need to be captured in real-time since the data access load may vary dynamically. In addition, it is imperative to use low CPU and memory resource, as it is long-running and may scale to dozens and hundreds of nodes. What's more, to monitor WSS and IRR of various applications, it needs to track the online items that have different sizes and structure levels. To sum up, ideally, to effectively tune the cache size online, an accurate, time-efficient, dynamic, light-weighted approach for tracking the working set size (WSS) and variable-size item repetition ratio (IRR) in a sliding window is needed.

We propose Cuki, an approximate data structure for estimating the online WSS and IRR for variable-size item access with proven accuracy guarantee and little overhead in the sliding window. Generally, we face three challenges in the design and usage of Cuki.

The first challenge is how to estimate the WSS and IRR online with little resource and proven accuracy. The item size can span over 8 ~ 9 orders of magnitude [24, 42] in the production environment. Therefore, inaccurate tracking items such as sampling may lose critical items, which may cause a huge drop in WSS estimation. The amount of data accessed in a time window can be quite large. It would be very time-inefficient and space-costly to store and calculate the item information online. To address this challenge, we carefully design a compact data structure with the approximate and item-wise tracking mechanisms.

The second challenge is how to achieve good scalability in high concurrency scenarios like multi-threading. It is common for real-world applications to access data concurrently. As the number of threads increases, the consistency and efficiency of concurrent access issues become obvious. To address this challenge, we adopt and propose a series of fine-grained concurrency control methods (§ 4), such as opportunistic aging.

The third challenge is how to judge the cache status under various scenarios with Cuki. It is non-trivial to tell whether the cache is overloaded or underused at a moment due to the variety of WSS and the cache-friendliness of the applications online. To address this challenge, we get the cache status insights by comparing the real cache size and the cache hit ratio with the WSS and IRR estimated by Cuki online, respectively.

By working with both Presto and Alluxio open source communities, our contributions can be summarized as follows:

- **Lightweight and Accurate WSS/IRR Estimation:** We design an approximate data structure, called Cuki, to estimate WSS and IRR online over a sliding window with little resource overhead and proven accuracy. Cuki uses an item-wise fine-grained tracking mechanism to reduce WSS error caused by missing critical items (*e.g.,* large ones). In our experiment, with a 96KB memory space size, Cuki can provide 99.07% accuracy for a 511MB working set size over the MSR data trace (§ 6.3). In addition, Cuki supports multi-scope WSS estimation with an easy feature extension.

- **Fine-grained Concurrency Control Methods:** To improve the efficiency of the concurrent access in Cuki, we propose opportunistic aging which decreases the lock contention risk in high concurrency scenarios. In addition, we adopt the segmented lock and two-phase based insertion mechanisms to guarantee data consistency in concurrent access.

- **Adaptive Online Cache Capacity Tuning Framework Using Cuki:** Finally, we propose an adaptive online cache capacity tuning mechanism based on Cuki. It judges whether the current workload, such as table querying and file reading, is cache-friendly or not, and further tells whether the cache system is overloaded or underused. Accordingly, the proposed cache capacity tuning mechanism can adjust the cache storage size online to fit current workloads.

- **Extensive Evaluation and Application Practice:** Experimental results on extensive benchmarks show that Cuki achieves over 10× higher accuracy with more stable performance compared with state-of-the-art methods. The cache tuning mechanism using Cuki can reduce the table reading latency by 79% on average, and improve the file reading throughput by 29% on average, respectively. Compared with the cutting-edge MRC approach, Cuki uses less memory and improves the accuracy by 73% on average. In addition, end-to-end real-world query workload experiments show that the proposed approach is effective for large-scale cache systems.

## 2 Background

**Cuckoo Filter:** A Cuckoo filter [17] is a well-known approximate data structure for deciding whether a given item is in a set or not. It consists of several buckets, and each bucket has four slots by default. Each item has two candidate buckets in a Cuckoo filter. To save space, a Cuckoo filter stores the fingerprint of an item rather than the item itself.

To insert item $x$, the Cuckoo filter first gets the fingerprint of $x$ as $f$. Then, the Cuckoo filter hashes the $x$ to get the first candidate bucket position $b_1$. The other candidate bucket position can be obtained by computing $b_2 = b_1 \oplus hash(f)$. The item $x$ will be inserted into an empty slot of these two candidate buckets. If both candidate buckets are full, the Cuckoo filter relocates other items iteratively until it finds an empty slot. To check whether item $x$ is already in the set, the Cuckoo filter first computes the two candidate buckets of $x$ as described above. Then, it checks the items' fingerprints in these two buckets. If the Cuckoo filter finds one's fingerprint is the same as $x$'s fingerprint, it returns true. Otherwise, it returns false.

**Miss Ratio Curve:** A key challenge of cache resource allocation is understanding the relationship between the cache hit ratio and the cache size. The miss ratio curve (MRC) is a common approach to figure out this relationship. The basic idea of MRC is to generate a miss ratio curve as a function of the cache size. With the generated miss ratio curves, users can allocate the cache size properly by observing the trend of the cache miss ratio with the cache size.

A traditional way [29] to generate a miss ratio curve of the specific trace is to compute the reuse distance of each item. The reuse distance of a specific item $x$ represents how many items have been cached since the last access of the $x$. Since the reuse distance of each item has been recorded, this approach will give an ideal miss ratio curve. However, the online overhead of this approach is non-negligible.

To reduce the overhead of generating MRCs, recent research work use sampling techniques. Counter Stack [45] uses down-sampled and pruned probabilistic counters. SHARDS [41] samples the input trace. AET [22] uses average eviction time to construct MRC. Mini-sim [40] extends SHARDS by using miniature simulation.

However, these methods [22, 40, 41, 45] have three limitations. First, they need to store or process a separate I/O trace. Second, they use sampling techniques, which are likely to miss heavy hitters (large-sized items) and incur inaccuracy. Third, they focus on processing fixed-size item accesses and need some redesign to handle variable-size objects. RAR-CM [51] uses the hashmap to store the item access information and estimate the item repetition ratio (IRR) for generating an approximate MRC. However, RAR-CM is still primarily designed for fixed-size item access and needs 128 bits to store each item. Our Cuki only needs 52 bits for each item to support variable-size item access MRC generation and is around 73% more accurate than RAR-CM (§ 6.7).

Though the overhead for Cuki to generate MRC is low, MRC generation brings additional overhead for Cuki after all. Since WSS/IRR estimation is usually sufficient for cache size tuning in our environment, we finally choose the WSS/IRR estimation as the main approach.

**Prior Cache Size Tuning Approaches:** The critical difficulty in improving cache utilization is tuning the proper cache size online with limited resource. Prior approaches can be mainly summarized into four categories:

- **Rule-based:** Rule-based approaches [21, 24, 34, 42] observe cache usage metrics. If the observed metrics exceed or are less than the predefined threshold, it tunes the cache size. For example, Pocket [24] increases the cache size when cache usage exceeds 80%. However, due to lacking knowledge of the working set sizes of online workloads, it does not know what the best cache size should be tuned to each time.
- **ML-based:** ML-based approaches [4, 28, 30, 33, 35] train machine learning models with historical data offline for further predicting proper cache sizes according to the workloads online. However, the pre-trained models based on historical data can hardly be adapted to dynamic online workload scenarios which have quite different data access patterns.
- **MRC-based:** MRC-based methods [15, 19, 36, 52] explore the optimal cache size by generating a miss ratio curve (MRC) as a function of the cache size. To reduce the overhead of generating MRCs, several approaches [22, 40, 41, 45] use sampling techniques. However, they are likely to miss heavy hitters, which would incur inaccuracy. Moreover, most MRC-based approaches are designed for fixed-size item access, which might be inaccurate for variable-size item.
- **Window-based:** Window-based methods [5, 11, 20] estimate the cardinality of items in a sliding window. However, they can hardly compute the accurate total size due to being unaware of each item's size with limited memory space and little time cost.

## 3 Design of Cuki

To efficiently estimate the real-time working set size (WSS) and the item repetition ratio (IRR) of various-granularity data access over sliding time windows, we design a compact approximate data structure called Cuki. In addition, we have theoretically proved that Cuki outperforms the state-of-the-art comparing algorithms in space usage under the same false positive rate. The proof details are moved in Appendix B due to page limitation. In this section, we introduce the main data structure and supported operations of Cuki.

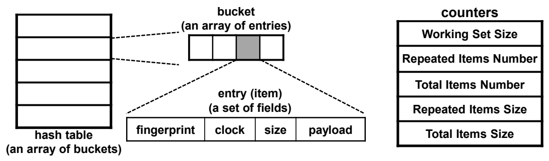


Figure 1: Data structure of Cuki.

### 3.1 Data Structure

Cuki is built on the Cuckoo filter [17]. The first reason we choose the Cuckoo filter is that it supports deletion so that we can remove stale items. Second, different from the Bloom filter, one item occupies one slot in the Cuckoo filter, so it is easy to extend cells for recording items' size.

In general, Cuki is an approximate membership query data structure with the following features: 1) similar to the Cuckoo filter [17], Cuki stores the items' fingerprints rather than the original data, which is memory-efficient. Different from the Cuckoo filter, Cuki has a more sophisticated design to support time window semantics, working set size estimation, and payload field extension. 2) Cuki supports insertion, lookup, deletion, and aging operations at the item level with efficient concurrency access control mechanisms. 3) Cuki provides built-in efficient and accurate working set size estimation in multi-scopes over the sliding time window based on the lightweight tracking of each item insertion and deletion.

Figure 1 shows the data structure of Cuki. It contains a *hash table* with multiple fixed-length *buckets*, each of which has several fixed-length *entries* to store items. Each entry has four fields to track an item's information: fingerprint, clock value, encoded size, and payload. In addition, there are five kinds of atomic *counters*, including *working set size*, *repeated items number*, *total items number*, *repeated items size*, and *total items size*, which are high-level global metrics. Particularly, *repeated items number* records the number of items that are repeatedly accessed in a sliding window, and the *total items number* is the total accessed items number in a sliding window. Cuki can calculate IRR by simply dividing *repeated items number* by *total items number*. All these five metrics are updated along with inserting or deleting items. Similar to the IRR, the bytes repetition ratio can also be easily calculated as *repeated items size* / *total items size*.

The **fingerprint** field is a succinct representation of an item. Usually, the fingerprint has few bits and is much smaller than the original item size. Moreover, similar to the Cuckoo filter, the fingerprint length of Cuki also offers a trade-off between accuracy and space, *i.e.,* Cuki can achieve more accurate estimation with longer fingerprints.

The **clock** value represents the freshness of an item. The higher, the fresher. Cuki sets the clock value of an item to a predefined value of MAX_AGE when the item is accessed (insertion or lookup) and periodically decreases it over time by the aging operation. Using $s$ bits for each clock, MAX_AGE is set to $2^s - 1$, where $s$ is an accuracy-to-space trade-off parameter. Suppose the sliding window size is $\mathcal{T}$, the aging operation will be executed every $\frac{\mathcal{T}}{2^s-1}$. The aging operation ensures that the stale items are cleared timely. Moreover, since the aging operation is executed more frequently using a longer clock bits length, there will be fewer errors in sliding windows. Aging operations can work in the background. In addition, we can further amortize the computation overhead by the opportunistic aging strategy in § 4.2.

The **size** field stores the encoded size of each item. There exist some naive several size encoding techniques, such as **Full-size Encoding** which directly stores each item's exact accurate size and **Truncation Encoding** that only stores the higher bits of the item size, since they are more important than the lower bits. To make a better tradeoff between accuracy and space overhead, we propose the **Grouped Size Encoding** technique. It saves the lower bits of each item into size groups. Every prefix has a corresponding size group to record the size of items with the same prefix. Each size group has two counters: $counts$ (total number of items) and $total\_bytes$ (total item size). For insertion, Cuki increases $counts$ by 1 and $total\_bytes$ by the item's size. When an item is removed, Cuki decreases $total\_bytes$ by the average size $total\_bytes/counts$, and $counts$ by 1. For a prefix length of $\gamma \cdot len$ bits, there are $2^{\gamma \cdot len}$ size groups in total. The space overhead of grouped size encoding is $\gamma \cdot N \cdot len + 2^{\gamma \cdot len} \cdot C$, where $C$ is the bits length of the above two counters for each size group.

Apart from the above size encoding methods, more sophisticated size encoding strategies [6, 7, 14] are also compatible with Cuki. However, these methods require additional computation. Thus, we choose not to use them as the main strategies.

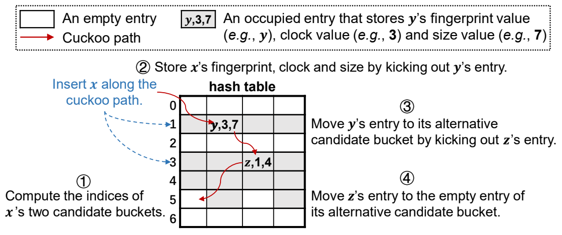
The **payload** field stores auxiliary information of an item. Although the former three fields are enough for the working set size estimation problem, we leave the payload as an auxiliary field for customized needs. In § 3.2, we introduce an example extension usage of the payload field, namely the multi-scope working set size estimation.

## 3.2 Operations in Cuki

**Item Insertion:** First, Cuki computes the fingerprint and two bucket indices $b_1$ and $b_2$ of a given item $x$ by Equations (1) ~ (3), respectively.

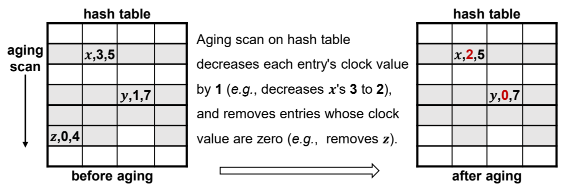
$$f = fingerprint(x), \tag{1}$$
$$b_1 = hash(x), \tag{2}$$
$$b_2 = b_1 \oplus hash(f). \tag{3}$$

Through Equations (1) ~ (3), Cuki can compute two candidate buckets by fingerprint without original item information. Next, it searches for an empty entry in the two candidate buckets. If successful, Cuki stores the item's fingerprint and encoded size in that entry and initializes the entry's clock value to MAX_AGE. Otherwise, it relocates other entries iteratively until it finds an empty entry. Specifically, Cuki finds a cuckoo path in the hash table [18]. The cuckoo path starts with a candidate bucket and ends with an empty entry. Cuki performs the insertion by moving items along this cuckoo path. For example, the red color line in Figure 2(a) is a cuckoo path, which starts with the bucket 1 and ends with the bucket 5. For inserting $x$ into bucket 1, the entry $y$ in bucket 1 will be kicked out to bucket 3. This leads the entry $z$ in bucket 3 will be kicked out to bucket 5. The Cuckoo path length will grow as item insertion, which might lead to long tail latency. We will discuss how to mitigate this in § 4.1.



An empty entry
Cuckoo path
y,3,7
An occupied entry that stores y's fingerprint value (e.g., y), clock value (e.g., 3) and size value (e.g., 7)
② Store x's fingerprint, clock and size by kicking out y's entry.
Insert x along the cuckoo path.
hash table
0
1
2
3
4
5
6
y,3,7
z,1,4
③ Move y's entry to its alternative candidate bucket by kicking out z's entry.
① Compute the indices of x's two candidate buckets.
④ Move z's entry to the empty entry of its alternative candidate bucket.

(a) Item insertion operation.



hash table
aging scan
x,3,5
y,1,7
z,0,4
before aging
Aging scan on hash table decreases each entry's clock value by 1 (e.g., decreases x's 3 to 2), and removes entries whose clock value are zero (e.g., removes z).
hash table
x,2,5
y,0,7
after aging

(b) Aging operation.

Figure 2: Illustration of insertion/aging operations in Cuki.

**Item Lookup:** Cuki's item lookup first computes the fingerprint and two bucket indices of a given item $x$ by Equations (1) ~ (3). Then, it checks if there exists an entry that matches the fingerprint within the two candidate buckets. If yes, it resets this entry's clock value to MAX_AGE and returns true. Otherwise, it returns false.

**Item Deletion and Aging:** Cuki supports removing an entry (item) by the item deletion operation or the item aging operation (§ 4.2) at an item's maximum age. Cuki's deletion method first looks up the candidate buckets which are described above. Then, it removes the entry which matches the fingerprint.

**Item updating:** If an item's attribute (*e.g.,* size) changes, Cuki needs only one-single table access to swiftly alter the hash table entry's fields without searching the cuckoo path. If an item's ID changes, Cuki considers it a new insertion. The old item can be deleted ad-hoc or via aging with performance-accuracy tradeoff. In addition, data updates are non-common in big data OLAP applications.

It is not trivial to automatically remove stale items from the sliding window. A straightforward accurate solution is to record all item IDs and timestamps (64 bits). This method requires too much memory because of the large number of timestamps. In recent years, some methods [3,5,20] try removing stale items without timestamps. However, as we analyze in § 6.3, these methods are either poorly memory utilized or inaccurate. Different from these methods, the clock algorithm [13] can remove stale items in time with little memory overhead (8~16 bits, as shown in § 6.2).

Therefore, we introduce *clock* into our data structure. Every entry in Cuki is associated with a clock value. Once an entry's clock value reaches zero, it should be removed because of the staleness. This can be done by periodical aging operations in the background. Specifically, suppose the length of the sliding window is $\mathcal{T}$, the bits length of the clock field is $s$, then the aging period of Cuki is $\frac{\mathcal{T}}{2^s-1}$. The length of window $\mathcal{T}$ can be either time-based or count-based [11,20]. The time-based sliding window contains items that arrive in the last $\mathcal{T}$ time units. The count-based sliding window contains the last $\mathcal{T}$ items. In each aging operation, it iterates the whole hash table in order and decreases each entry's clock value by 1. If an entry's clock value is already down to zero before aging, Cuki deletes this entry. Figure 2(b) shows an aging example. For items $x$ and $y$, the aging operation decreases their clock values by 1; while for item $z$, whose clock value was zero, it is removed. Though using clock to remove stale items can save much memory, it brings errors in results. We have put the theoretical analysis of the above statement in Appendix B.1 due to space limitation.

**Entire Working Set Size Enquiry:** Besides the above item-level operations, Cuki also natively supports working size-level operations, such as the entire working set size enquiry.

In fact, computing the entire WSS by online scanning the entries in a full hash table and summing up their sizes is very time-inefficient and resource-costly for each query request. Instead, we maintain a counter inside Cuki. The counter tracks the WSS, updates it when inserting or deleting (*e.g.*, by aging) items, and can thus always answer entire WSS enquiry in constant time. The counter is implemented with an atomic class. Thus, it can be concurrently updated safely and efficiently.

**Multi-scope Working Set Size Enquiry:** In addition to entire WSS enquiry, Cuki also supports WSS enquiry at the scope level, which queries the sizes of specific scopes of the entire working set. Different scopes can be regarded as different parts of the entire working set (*e.g.*, different tables of a database, or different partitions of a table). The information of each scope size is useful for resource scheduling methods [39] and optimizing multi-tenant systems [23,46]. For example, in a large-scale query engine, we can use multi-scope WSS estimation to find the table with the biggest WSS, which is usually queried frequently. Replicating this table to more nodes of the cache system may help increase the throughput of the query engine.

To estimate multi-scope WSS, we can easily encode the scope information (*e.g.,* mapping scopes to an integer by a hash table) into several bits and store them in the payload field of Cuki. In addition, we need to maintain a set of independent counters (*e.g.,* WSS, repeated items size, and total items size) for each scope in Cuki. For example, when an item $x$ belonging to scope $Scope_k$ is inserted, Cuki will store the encoded scope $Scope_k$ along with the entry of $x$, and increase the independent WSS counter of $Scope_k$. When the deletion or aging operation removes $x$, Cuki can figure out the scope that $x$ belongs to, by checking the encoded scope information in its payload field, and decreases the relevant WSS counter of that accordingly.

In practice, for existing methods, it is non-trivial to allocate a suitable memory size for each scope without prior knowledge of each scope's cardinality. Instead, in Cuki, the items of different scopes can share the same total hash table space, by using the encoded scope information in their payload fields to distinguish from each other. Thus, it is not necessary to allocate static memory space for each scope in Cuki.

## 4 Concurrency Control in Cuki

### 4.1 Segmented Lock and Concurrent Insertion

We first introduce the basic concurrency control technique called **segmented lock** adopted in Cuki. Then, we describe how Cuki supports concurrent insertion.

**Segmented Lock:** Cuki divides the whole hash table into several segments, and each segment is guarded by one single lock. Users can configure the number of buckets per segmented lock to tradeoff lock overhead and contention. On the one hand, the item insertion, lookup, and deletion may access different segments at the same time. To avoid deadlock in operations, we always acquire and release the locks in order. On the other hand, each segmented lock guards a group of adjacent buckets. Therefore, for the aging operation, there is no need to repeatedly acquire a lock for scanning items in the same segment. Moreover, each lock manages a physically continuous space. Benefiting from this cache-friendly design, the aging operation can be executed faster. This is because the aging operation accesses Cuki sequentially.

**Concurrent Insertion:** It is non-trivial to handle the concurrent insertion operations in Cuki. As analyzed in [18], there will be a false negative error under concurrency when moving items along the cuckoo path. To eliminate the false negative error, similar to [18, 26], we separate the insertion process into two phases: the path discovery and item movement phases. In the path discovery phase, Cuki finds a cuckoo path [18] that starts from two alternative buckets and ends at an empty entry. Then, in the item movement phase, Cuki moves items backward along the cuckoo path. Cuki always acquires locks before each above phase, guaranteeing each operation's atomicity.

With more items inserted into Cuki, the Cuckoo path length increases, which might lead to long tail latency. The item movement may also fail as analyzed in [26]. The probability of insertion failure is less than $1.75 \times 10^{-5}$ in their environment. In our experiment and production environment, there is almost no insertion failure most of the time. Also, we find that 97% of Cuckoo paths have lengths below 2, and 99.99% of Cuckoo paths have lengths below 4 in MSR trace with 192KB memory. To totally avoid insertion failure and long tail latency, one can allocate appropriate memory size for Cuki by using space resizing techniques [10, 27, 43, 50]. Specifically, when the load of Cuki reaches the high watermark, according to the solution proposed in [50], we can resize the Cuki's capacity by adding an extra homogeneous Cuki data structure after the existing one. The new incoming items can be inserted into the expanded data structure [50]. Except for this solution, we can adopt "partial-key linear hashing" technique proposed in [43] to increase the capacity of Cuki in a fine-grained fashion. Furthermore, similar to [26], we adopt **breadth-first search** to find an empty entry. It can be theoretically proven that the Cuckoo path found by BFS is shorter than that found by DFS [26].

## 4.2 Opportunistic Aging

To update the data freshness over the sliding window, Cuki performs aging operations periodically in the background. At each background aging, Cuki scans the whole hash table. It first acquires the lock of each segment, then ages the items in that segment in turn. However, the background aging suffers from the following issues in high concurrency scenarios.

**Issue 1. Large fluctuation of estimation result:** In background aging, massive obsolete items will be cleared simultaneously. As a result, the estimated working set size varies a lot before and after the aging execution. Therefore, the aging can significantly affect the error in the estimated WSS, which decreases the estimation accuracy and stability. We conduct an experiment to verify this, and it is in Appendix C.1.

**Issue 2. Lock contention with user operations:** Most operations in Cuki (*e.g.*, insertion, lookup, and aging) require holding the lock first, which causes lock contention among these operations. It brings in two kinds of issues. First, when the aging operation is in execution, if there are too many obsolete items that need to be removed, the other data access operations will be blocked for a long time until the lock held by the aging process is released. It increases the delay of other data access operations. Second, when the aging operation is waiting for execution, if there exist so many insertions or lookup operations, the aging operation might wait a long period before getting the lock. Thus, the obsolete items in Cuki may not be removed in time by aging, which decreases the estimation accuracy of Cuki.

To address these issues, we propose a lightweight concurrency control strategy called **opportunistic aging**. It amortizes the aging operation into the insertion operations in Cuki. It brings two main advantages. First, the full aging task is split into multiple minor aging tasks, making the sliding window move smoother. Second, since fewer entries need to be checked in background aging, it reduces the lock contention risk with the background aging.

**Algorithm 1** Opportunistic Aging in Cuki

```
Input: $S_i$ is the segment to be aged, $P_i$ is the aging pointer of $S_i$.
1: $N_{oa} \leftarrow$ the number of items to be aged;
2: while $N_{oa} > 0$ && $P_i < S_i.length$ do
3:     /* aging the $P_i$th buckets of segment $S_i$ */
4:     AgingBucket(GetBucket($S_i$, $P_i$));
5:     $P_i \leftarrow P_i + 1$;
6:     $N_{oa} \leftarrow N_{oa} - 1$;
```

Specifically, each segment in Cuki has a pointer to track its aging progress. Both opportunistic aging and background aging start working from the pointer's position. $N_{oa}$ items are aged during each opportunistic aging. The pointer advances accordingly. Subsequently, background aging ages the remaining items in each segment from the position of the pointer left by opportunistic aging. If the aging pointer is at the end of the segment, background aging will skip this segment. Therefore, opportunistic aging reduces lock contention.

Algorithm 1 elaborates the procedure of opportunistic aging. First, it computes the number of items that need to be aged (noted as $N_{oa}$) by the elapsed time from the beginning of the aging period (Line 1). Suppose $S_i$ is the segment to be aged, $P_i$ is the aging pointer (index) of $S_i$, $N_i$ is the number of buckets in $S_i$, $T$ is the time interval of each aging period, $t_{cur}$ is the elapsed time from the beginning of the aging period, $N_{oa}$ can be computed by the equation:

$$N_{oa} = N_i \times \frac{t_{cur}}{T} - P_i. \tag{4}$$

It guarantees that the aging progress is consistent with the movement of the sliding window. Besides, to reduce the latency of each insertion operation, we limit the maximum number of items cleared during each opportunistic aging. Then, We conduct the aging operation on the $N_{oa}$ items in the segment $S_i$ (Lines 2-6). The remaining stale items, which have not been removed by opportunistic aging, will be cleared by the background aging.

Regarding accuracy, ClockSketch [11] reveals that some stale items are not cleaned timely by background aging (also analyzed in our Appendix B). Opportunistic aging can mitigate this error by preemptively removing certain stale items before background aging.

## 5 Cache Capacity Online Tuning Using Cuki

In this section, we show how Cuki can facilitate online cache capacity tuning for many data access applications. First, Cuki can be used in implementing the cache size adaptive tuning mechanism. Based on that, it can accelerate data access, including table querying and file reading. In addition, Cuki can also help generate miss ratio curves (MRCs), which provides an in-depth understanding of the relationship between the cache hit ratio and the cache size.

## 5.1 Data Access Application Acceleration

During setting the cache capacity for applications, we are mainly faced with two key cache-related questions: 1) What is the degree of the data access temporal locality for a given data access stream? 2) How to optimize the cache utilization online for a given data access stream?

### 5.1.1 Adaptive Cache Capacity Tuning Framework

In the following, we introduce the key metrics of the Cuki, which can be used for adaptive cache capacity tuning. To explain how to track and optimize the cache utilization with Cuki, we define the following key metrics.

- **CSS:** The cache space size, which can usually be obtained from configurations or metric monitoring of the cache system.
- **WSS:** The working set size over the time window, which is estimated by Cuki online.
- **CHR:** The realistic cache hit ratio of the cache system, which is often exposed by the cache metric monitor system.
- **IRR:** The item repetition ratio over the time window estimated by Cuki. IRR is computed by $\frac{\|R\|}{\|O\|}$, where $O$ and $R$ are the set of total accessed items and repeatedly accessed items in the time window, respectively.

The proposed adaptive cache capacity tuning mechanism can answer the above questions by tracking WSS and IRR in constant time with Cuki.

IRR measures the data access temporal locality of the application online. Specifically, since every repeatedly accessed item is counted by Cuki, IRR can be regarded as the upper bound of the cache hit ratio for the realistic cache system over the past time window. WSS is the total size of recently accessed items. It reflects the realistic cache demand of the application in the time window. In fact, as we observed in our real-world query service scenarios and other applications reported in existing work [48, 53], the working set size and data repetition ratio do not significantly change in a short period, following the law of temporal data locality. Thus, for a workload, its estimated WSS and IRR over adjacent time windows are likely similar, and we can use the current estimation to optimize the cache capacity for the near future.

**Cuki** has two main advantages in estimating WSS and IRR. First, it can track the WSS over sliding time windows accurately and stably. Second, it supports updating and querying WSS with constant time complexity, which makes real-time tracking and dynamic adjustment possible.

Figure 3 illustrates how Cuki and the above defined metrics can help to improve the cache system efficiency. Cuki is embedded into cache layer and cooperates with

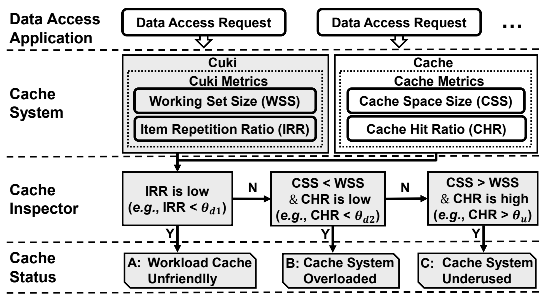

Data Access Application
Data Access Request
Data Access Request
...
Cache System
Cuki
Cuki Metrics
Working Set Size (WSS)
Item Repetition Ratio (IRR)
Cache
Cache Metrics
Cache Space Size (CSS)
Cache Hit Ratio (CHR)
Cache Inspector
IRR is low (e.g., IRR < $\theta_{d1}$)
N
CSS < WSS & CHR is low (e.g., CHR < $\theta_{d2}$)
N
CSS > WSS & CHR is high (e.g., CHR > $\theta_u$)
Y
Y
Y
Cache Status
A: Workload Cache Unfriendly
B: Cache System Overloaded
C: Cache System Underused

Figure 3: Workflow of adaptive cache capacity tuning mechanism based on Cuki (grey components are proposed by us).

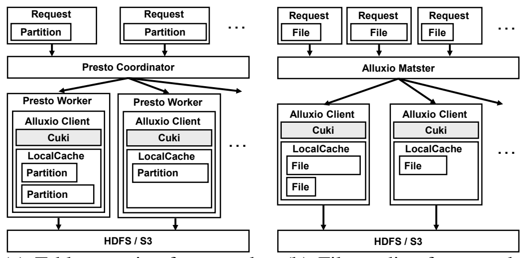
the cache system seamlessly. The cache system has online CSS and CHR metrics, while Cuki contains WSS and IRR statistics during the corresponding time windows. When a data access request arrives, the metrics of the cache system and Cuki are simultaneously updated.

The cache inspector figures out the cache system status by comparing these metrics according to the logic in Figure 3. Based on the cache status and the metrics in Cuki, the cache space size can be appropriately tuned up and down online.

Specifically, the cache inspector will measure the data temporal locality of the workload by checking the item repetition ratio IRR estimated by Cuki. For case A in Figure 3, if IRR is low (smaller than a predefined threshold $\theta_{d1}$, *e.g.*, 50%), the workload itself is not cache-friendly, which means that there exists little repeated data access during the time window. In this case, even if adding huge cache space, we can barely get a higher cache hit ratio CHR.

In other cases, when IRR is high, the cache inspector will check CHR, cache space size CSS, and the working set size WSS over the time window. For case B in Figure 3, when CHR is low (smaller than a predefined threshold $\theta_{d2}$, *e.g.*, 50%) and CSS<WSS, it means that the cache system is overloaded, and there indeed exists some room to improve the cache performance further. This is because the CHR is low, but CSS is still less than the realistic cache demand measured by the estimated WSS. In this case, we can improve the cache efficiency by increasing CSS. For example, the size of an application's data table usually increases as the number of application users grows. The cache system will be under-provisioned if CSS is not carefully configured accordingly. However, with the estimated WSS as the indicator, we can allocate an appropriate amount of cache resource easily online.

Besides, if both IRR and CHR are high (larger than a predefined threshold $\theta_u$, *e.g.*, 90%), it means that the cache system has sufficient cache resource. However, the resource might be wasted when we allocate superfluous cache space over the realistic cache demand measured by the estimated WSS in Cuki (case C in Figure 3). In this case, the cache system is underused. In real-world practice, we can tune down the cache space or leverage this information to optimize the query task scheduling algorithms or the cluster resource routing strategies. For example, we can facilitate the load balance of the cache system by prioritizing scheduling the query tasks



(a) Table querying framework (b) File reading framework

Figure 4: Data access applications using Cuki.

to the compute nodes where the cache is still underused.

In the last case, where both the cache hit ratio CHR and the item repetition ratio IRR are high, and the cache space size CSS matches the estimated realistic cache demand WSS, the cache system is working in healthy status.

### 5.1.2 Table Querying Acceleration Framework

Figure 4(a) shows how the proposed adaptive cache capacity tuning mechanism is integrated with Presto and Alluxio. Presto is designed for performing SQL query computation in memory. The Presto coordinator distributes the execution plan fragments to Presto workers according to the scheduling strategy. Presto workers execute query plan fragments on the data read from the remote HDFS/S3. Since Presto workers do not store the data, they tend to use Alluxio clients as their cache tier. We implement the Cuki in Alluxio client to track the LocalCache access. In order to use Alluxio LocalCache as Presto worker's cache, Alluxio client Jar files are distributed to each Presto worker.

Each Presto worker queries Alluxio LocalCache inside the same JVM through a standard HDFS interface. First, Presto transforms the queried partitions into several splits. Then, Presto coordinator makes the best attempt to assign the same split to the same worker, which is cache-friendly. If the queried splits are in the Alluxio LocalCache, splits are directly read from local RAM and returned to Presto. Otherwise, it retrieves data from HDFS/S3 and caches the data to local RAM of Presto worker. Cuki monitors the whole process of the split access in each Presto worker and updates WSS/IRR.

### 5.1.3 File Reading Acceleration Framework

File reading is common in distributed applications, such as online video websites and cloud downloading services. It's common that there exist some hot files which are more likely to be accessed by applications in a nearby time period. Thus, we can use a cache system to accelerate file reading by storing hot files. However, the size of hot files changes as time flies, which makes it hard to determine the proper cache size. As shown in Figure 4(b), the proposed adaptive cache capacity tuning mechanism based on Cuki can be used to solve this problem. The implementation of the proposed adaptive cache capacity tuning mechanism in file reading is similar to the above section. First, the requests for files are sent to the Alluxio master. Then, the Alluxio master checks whether the requested files are stored in one of the Alluxio clients. If so, the requested files are directly read from the local RAM of the Alluxio client. Otherwise, Alluxio reads files from HDFS/S3 and caches the data to the local RAM of the Alluxio client. Cuki monitors the whole file access process in each Alluxio client and updates WSS and IRR accordingly online.

## 5.2 Miss Ratio Curves Generation Using Cuki

Although WSS and IRR are useful enough for the adaptive cache capacity tuning mechanism, they still can not show in-depth insights into the relationship between the cache hit ratio and the cache size. Generating a miss ratio curve (MRC) as a function of the cache size is a common method to understand the relationship between the cache hit ratio and the cache size thoroughly. It only needs a little change in Cuki to generate MRCs for variable-size item access.

Similar to most MRC generation approaches [40,41,51], Cuki needs to store the reuse distance distribution as $RD(x)$. $RD(x)$ represents how many items are re-accessed at $x$ LRU stack size ($x$ is also called as the reuse distance). With $RD(x)$, Cuki can compute $MRC(x)$ simply by $\frac{\sum_{i=1}^{x} RD(i)}{TC}$, where $TC$ is the total items number. To compute $RD(x)$, Cuki tracks each item's clock value and stores the clock distribution as $CD(y)$. $CD(y)$ represents the total size of items whose clock value is $y$. With $CD$, Cuki computes the reuse distance of the accessed item by $distance = \sum_{i=y}^{max} CD(i)$, where $y$ is the clock value of the accessed item. Then, Cuki increases the $RD[distance]$ by one. The length of the array $CD$ is the MAX_AGE, which is decided by the clock bits length. Because the clock bits length is a small constant number (never exceeding 16 in our evaluation), the space cost of $CD$ is negligible.

In the following, we introduce how Cuki maintains $CD$ when the item's clock value changes. We use $oc$ and $nc$ to represent the old clock value and the new clock value, respectively. The item's clock value changes when the item is accessed or aged. Then, Cuki decreases the $CD[oc]$ by the item size and increases the $CD[nc]$ by the item size.

## 6 Evaluation

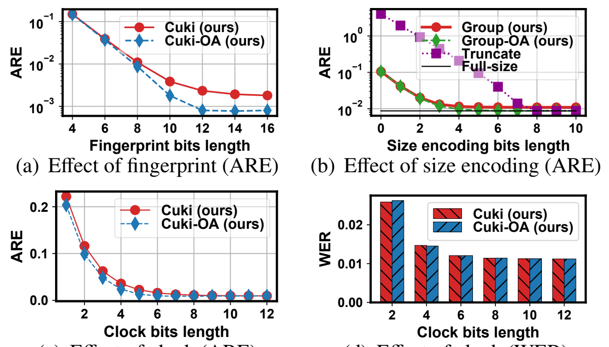
## 6.1 Experimental Setup

To be consistent with Alluxio and Presto, we implement Cuki and comparison methods in Java. If not explicitly mentioned, all approaches run on a server with Intel Xeon(R) Gold 6248 CPU with ten 2.5GHz cores. The version of Alluxio and Presto in the experiment is 2.7.0 and 0.266, respectively.

**Datasets and Workloads.** Experiments are run on both existing benchmarks and real-world datasets with workloads:

(1) **MSR I/O trace dataset [31].** We choose the first 12,518,968 records of MSR web proxy workload

as a typical dataset. Each record consists of three disk access information: timestamp, offset, and size. In our experiment, we use the offset to represent the item ID.
(2) **Twitter dataset [48]**. We choose the representative Twemcache-cluster37 first-hour data which has 10,169,267 records, and the similar Twemcache-cluster35 first-day data as our datasets. The record's key is regarded as the item ID, and the item's size is the sum of key_size and value_size.
(3) **YCSB dataset [12]**. We generate a concatenated trace that contains 10 million records by the YCSB generator [47]. Each base trace follows a zipfian distribution [32] with a skewness factor of 0.99. The item size of each base trace ranges from 512B to 1MB, but follows different zipfian distributions.
(4) **TPC-DS [38]**. Typical I/O bound queries in TPC-DS are used for the end-to-end performance evaluation.
(5) **Real-world query workloads**. We also adopt the real-world query workloads from one of our large scale Presto clusters with 200 servers in § 6.8. The total data access size of the workload is PB-level, and the cache space is TB-level.
**Comparison Approaches.** Following methods are evaluated:
(1) **ClockSketch [11]**. We add a 32-bit size counter for each cell of ClockSketch for WSS estimation. When a cell is first inserted, its size counter will be set to the item's size.
(2) **SlidingSketch [20]**. We apply SlidingSketch to the Bloom filter [8] for WSS estimation. Each domain of its bucket is used to record the item size.
(3) **SWAMP [5]**. SWAMP stores each item's frequency in a data structure called TinyTable [16]. We extend the TinyTable in SWAMP to record each item's size.
(4) **MBF [3]**. We use the Multiple Bloom Filter (MBF) implementation in the latest Alluxio version [3]. Each Bloom filter [8] is implemented with Google's Guava library [2].
(5) **RAR-CM [51]**. In RAR-CM, each block has a counter to record the last access number. To support variable-size item, we use the RAR-CM's counter to record its last access bytes.
(6) **Cuki and Cuki-OA**. Cuki is the basic approximate data structure proposed in this paper. Cuki-OA further uses the opportunistic aging strategy in § 4.2.
**Metrics.** We measure accuracy and speed performance by following metrics:
(1) **Weighted Error Rate (WER)**. Let *error_bytes* be the total size of items that are evicted faster or slower than the ideal sliding window. The WER can be calculated by $\frac{error\_bytes}{total\_bytes}$.
(2) **Relative Error (RE)**. $\frac{w-\hat{w}}{w}$, where $w$ and $\hat{w}$ are exact and estimated working set size (WSS), respectively.
(3) **Average Relative Error (ARE)**. $\frac{1}{|T|}\sum_{t\in T}\frac{|w_t-\hat{w}_t|}{w_t}$, where $w_t$ and $\hat{w}_t$ are the exact and estimated WSS at moment $t$.
(4) **Mean Absolute Error (MAE)**. $\frac{1}{N}\sum|MRC(x)-MRC'(x)|$, where $N$ is the length of reuse distance array, $MRC(x)$ is the hit ratio at the cache size $x$.
(5) **Throughput**. In file reading experiment (§ 6.7), it is the number of MB/s. In other experiments, it is # of operations/s.
(6) **Query Latency**. The end-to-end SQL query latency.



(a) Effect of fingerprint (ARE) (b) Effect of size encoding (ARE)
(c) Effect of clock (ARE) (d) Effect of clock (WER)

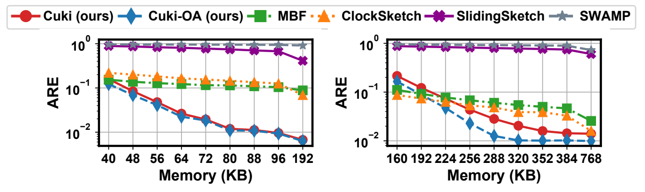
Figure 5: Effects of parameters in Cuki.

**Parameter Settings of Approaches.** All methods use the same memory size in each experiment. For the count-based sliding window, we set the window size to 262,144 ($2^{18}$) and measure RE every 64 time units. For the time-based sliding window, we set the window size to one-hour and one-day for the MSR and Twitter traces as different traffics, respectively. The default size encoding approach for Cuki is grouped size encoding. The bits length of the fingerprint, clock and size fields in Cuki are set to 8 if not explicitly mentioned. The settings of the comparing methods are fully tuned to nearly achieve their best performance for a fair comparison.

## 6.2 Effect of WSS Estimation Parameters

To understand the impact of the Cuki's parameters, we conduct experiments on the YCSB trace with a count-based sliding window. The number of entries in Cuki is fixed to 262,144 ($2^{18}$), just enough to track all the items within a sliding window. To reduce other parameter interference, we use the full-size encoding method by default in this section.
(1) **Effect of fingerprint bits length.** As shown in Figure 5(a), as the fingerprint bits length grows, the ARE of both Cuki and Cuki-OA is dramatically decreased. In fact, an item's key is represented by its fingerprint. Thus, a small fingerprint bits length leads to different items being hashed to the same fingerprint, resulting in high ARE. Moreover, compared with Cuki, Cuki-OA decreases ARE by 37% on average, which verifies the effectiveness of the opportunistic aging strategy.
(2) **Effect of size encoding methods.** Figure 5(b) illustrates the influence of different size encoding methods. The performance of the truncation encoding method using and not using opportunistic aging is the same. Thus we only show the truncation encoding in the figure. The black line represents the ARE of the most accurate baseline (full-size encoding).

As shown in Figure 5(b), on the one hand, full-size encoding achieves the best accuracy but it stores the entire accurate size. Compared with the truncation encoding strategy, the grouped size encoding strategy decreases ARE by 92% on average when the group bits length is small (< 6 bits). Thus, we can conclude that grouped size encoding achieves the best trade-off between memory space and estimation accuracy.



(a) ARE on MSR trace (b) ARE on Twitter trace

Figure 6: Performance comparison of accuracy.

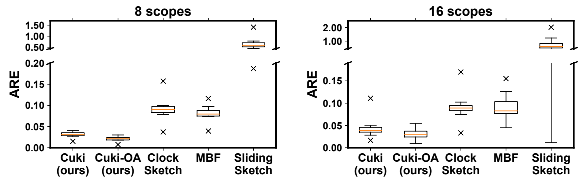


Figure 7: Accuracy of multi-scope estimation.

(3) **Effect of clock bits length.** As shown in Figures 5(c) and 5(d), the ARE and WER can be reduced by using more clock bits. As shown in Figure 5(c), opportunistic aging (Cuki-OA) decreases the ARE of Cuki by 26% on average when the length of the clock bits is small ($< 8$ bits). Also, we can observe from Figure 5(d) that Cuki-OA barely increases WER.

Besides the above three parameters, the parameter sliding window size can be set as the user demands. In the above experiments, Cuki only needs a few extra bits to track each item's key, access freshness, and size. Therefore, we can conclude that Cuki can track each item only using several bits by sacrificing negligible accuracy.

## 6.3 Accuracy of WSS Estimation

In this subsection, we evaluate the accuracy of Cuki by comparing it with cutting-edge WSS estimation methods over the sliding window mechanism. Figure 6 exhibits the ARE of different methods measured in the same run on two traces. We double the memory size at the last point of each experiment to meet the memory requirement of each approach. As shown in Figure 6, while the performance of all comparison approaches gets improved with more space, Cuki and Cuki-OA exhibit better memory-accuracy efficiency. For example, Cuki-OA decreases ARE from 12.26% to 0.93% as the memory space increases to 96KB on the MSR trace. In addition, Cuki-OA decreases the ARE of Cuki by an average of 11% and 37% on the MSR and Twitter traces, respectively. As the memory space gradually becomes larger, the ARE of Cuki decreases to 1% and lower. However, the ARE of coarse-grained tracking methods, such as MBF, SlidingSketch, and ClockSketch, can hardly further decrease even with sufficient memory.

Finally, we compare the accuracy of various methods on multi-scope WSS estimation. We use the MSR dataset as a typical benchmark and replay it with a 144× speedup. The time-based sliding window size is set to one hour. All methods in the experiments use the same 24MB memory size because of the large multi-scope combined workload. As shown in Figure 7, Cuki-OA reduces the ARE of Cuki by 33% and 22% on average for the 8 and 16 scopes WSS estimation, respectively. Secondly, the ARE of Cuki-OA is 11 and 8 times lower than the comparison algorithms on average for the 8 and 16 scopes WSS estimation, respectively. It mainly benefits from Cuki's extensibility which allows items of different scopes to make better usage of memory together.

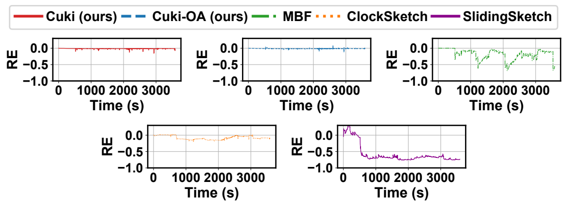


Figure 8: Performance comparison of stability on MSR trace (comparison methods use 2× larger memory than Cuki).

To conclude, Cuki and Cuki-OA achieve the best accuracy with the same memory consumption among all the methods. More experiments on the YCSB trace or using the WER metric are in Appendix C.2. They have similar conclusions.

## 6.4 Stability Performance of WSS Estimation

We evaluate the stability performance of different methods under the time-based sliding window. More experiments on the Twitter trace and the count-based sliding window are available in Appendix C.3. They have similar conclusions.

We replay the MSR trace with 168× speedup and use 192KB memory for the Cuki. In order to meet the comparison methods' memory requirements, they use double amount of memory than Cuki. Figure 8 illustrates the stability performance of different methods over the time-based sliding window. SWAMP is omitted due to it only supports the count-based sliding window. There are jagged fluctuations in estimation for all methods because of the movement of the sliding window. Specifically, MBF switches a Bloom filter out periodically and drops the corresponding items. ClockSkech's fluctuations are mainly due to hash collision with limited memory. SlidingSketch can hardly track all items within a sliding window due to limited memory space.

For Cuki, despite its performance being affected by aging operations, its estimation is stable. The stability is mainly attributed to its per-item size tracking. Notably, opportunistic aging can make the movement of sliding windows smoother.

To conclude, Cuki and Cuki-OA use less memory and achieve the most stable estimation results.

## 6.5 Scalability of WSS Estimation

We evaluate the thread scalability of the comparison methods. Specifically, we use the MSR trace with the count-based sliding window, and the memory size is set to 40KB. SWAMP is omitted due to not supporting multi-thread concurrency.

Figure 9 shows that increasing concurrency can not improve ClockSketch's throughput significantly. SlidingSketch

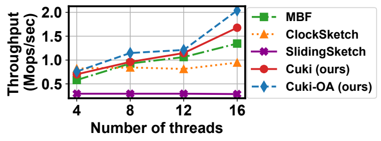
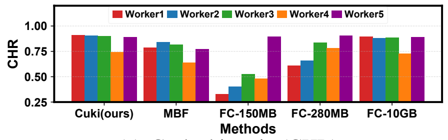
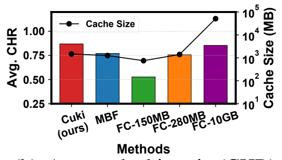


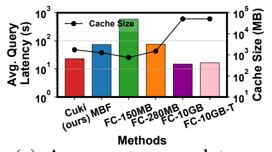
Figure 9: Throughput with concurrent threads.



(a) Cache hit ratio (CHR)



(b) Avg. cache hit ratio (CHR)



(c) Avg. presto query latency

Figure 10: Performance of adaptive cache capacity tuning mechanism in table querying (**FC-100MB** represents the fixed cache size 100MB, **FC-10GB-T** represents running with WSS estimation in the fixed cache size 10GB, others are similar).

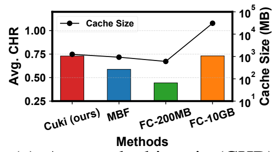
has heavy aging tasks after each operation. Thus, its scalability is limited. MBF can improve throughput by increasing concurrency, but it needs to manage Bloom filters for insertion and lookup, resulting in lower throughput than Cuki.

Cuki and Cuki-OA have near-linear multi-threading scalability due to their fine-grained concurrency control optimization strategies. To conclude, both Cuki and Cuki-OA achieve near-linear multi-threading scalability.

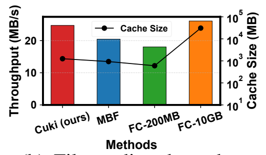
## 6.6 Cache Tuning Performance with Cuki

(1) **End-to-End Performance in Table Querying :** The experiments run on a Presto cluster with one coordinator and five workers using the I/O-bound TPC-DS dataset. FC-150MB, FC-280MB, and FC-10GB represent the cache system is in overloaded, healthy, and underused statuses, respectively. The 280MB cache size is manually chosen because it is the most competitive cache size that makes a good trade-off between the cache hit ratio and the cache capacity.

The adaptive cache capacity tuning mechanism use 125MB memory, which is the default value in MBF [3]. As shown in Figure 10(a), the cache hit ratio of FC-150MB is the lowest one. And, the cache hit ratio of FC-10GB can be regarded as the upper bound. By using Cuki, the cache system nearly achieves the upper bound of the cache hit ratio. Figure 10(b) shows the average cache hit ratio and the maximum total cache space allocated by the proposed adaptive cache capacity tuning mechanism and others. Compared with FC-280MB, our method improves the average cache hit ratio by around 11% while using a similar total cache size. This is because that our method allocates the cache space to each Presto worker according to their different demand. Overall, by using Cuki, the cache system can not only reach the upper limit of the cache hit ratio, but also improve the cache utilization.

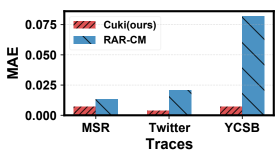


(a) Avg. cache hit ratio (CHR)

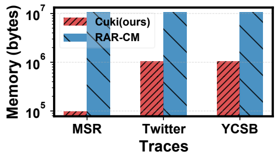


(b) File reading throughput

Figure 11: Performance of adaptive cache capacity tuning mechanism in file reading (**FC-200MB** and **FC-10GB** represent the fixed cache size are 200MB and 10GB).



(a) Mean absolute error



(b) Memory

Figure 12: Performance of Cuki in MRCs generation.

The average query latency of different approaches is shown in Figure 10(c), the average query latency of FC-10GB-T (the fixed 10 GB cache size with WSS estimation) is close to FC-10GB. The average query latency of Cuki is close to the FC-10GB which is the lower bound of latency. Compared with MBF, FC-150MB, and FC-280MB, Cuki can reduce the query latency by around 69%, 97%, and 71%, respectively.
(2) **End-to-End Performance in File Reading:** This experiment uses the first 9000 data access requests in YCSB [47] trace as the workload. For each unique trace item, we generate a file whose size is the item value and store the file in remote storage S3. We run the experiments on an Alluxio cluster with three EC2 servers and deploy an EC2 client which runs in three threads to access data. Each thread sends 3000 file reading requests and repeats three times.

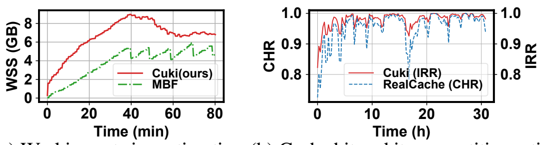
As shown in Figure 11(a) the cache hit ratio of FC-200MB is the lowest. The cache hit ratio of FC-10GB can be seen as the upper bound because the 10GB cache size is enough to cover all workloads. The cache hit ratio of Cuki is close to the FC-10GB, which means Cuki helps the cache system to reach almost the upper bound of the cache hit ratio.

As shown in Figure 11(b), we compare the end-to-end file reading throughput of the above comparison methods. The throughput of Cuki is close to FC-10GB which is the upper bound of the throughput. Overall, Cuki can improve the cache utilization of file reading to reach higher throughput.

## 6.7 Accuracy of Miss Ratio Curves Generation

We compare the accuracy of miss ratio curves (MRCs) generation among Cuki and RAR-CM. Considering the poor support for the sliding window mechanism in RAR-CM [51], the window length is the same as the trace length.

As shown in Figure 12(b), Cuki uses 96KB memory for MSR trace and 1MB memory for other traces. Each item in RAR-CM needs 128 bits to be stored, which is larger than Cuki's 56 bits. In order to make RAR-CM more accurate, we



(a) Working set size estimation (b) Cache hit and item repetition ratios

Figure 13: Large-scale real-world practice.

allocate RAR-CM 10 MB memory, which is 10× larger than Cuki. Figure 12(a) shows the accuracy of Cuki in MRCs generation. Compared with RAR-CM, Cuki reduces the MAE by around 48%, 82%, and 91% in the MSR, Twitter, and YCSB traces, respectively. This is because that Cuki can better support variable-size item. Moreover, RAR-CM estimates the re-access ratio to compute the reuse distance, which is inaccurate. In addition, Cuki achieves comparable throughput with RAR-CM in experiments. Overall, Cuki costs less memory and generates more accurate MRC than RAR-CM.

### 6.8 Real-world Practice

We elaborate on how Cuki is used in our real-world large-scale query platforms with the cache system called ShadowCache. ShadowCache is being leveraged to understand the system bottleneck and help with query system routing design decisions. Specifically, with ShadowCache, the overall system can efficiently decide how to size the cache for each tenant, and what the potential cache hit ratio improvement is. In the following, we evaluate the usability of the working set size estimation methods on a middle-scale Presto cluster (GB-level cache space). We implement the proposed Cuki-based cache capacity tuning mechanism. MBF is also used for comparison.

Figure 13(a) shows the estimated WSS of Cuki and MBF on a middle-scale cluster. There exist fluctuation for MBF in its estimation due to periodically removing a part of its statistics as analyzed in § 6.3 and § 6.4. In fact, the cache system can hardly distinguish the normal workload changes from the MBF fluctuations. In contrast, Cuki provides stable working set size estimation with little fluctuation. Thus, Cuki is more credible and effective in real-world scenarios.

Next, we deploy the proposed approach on a large-scale real-world Presto cluster (TB-level cache space with 200 servers). Figure 13(b) shows the performance of query workloads over one day on the Presto cluster, showing the realistic cache hit ratio (CHR) performance of the cache system and the item repetition ratio (IRR) estimated by Cuki. It can be seen that IRR is much higher than the CHR of cache between the 16th hour to the 19th hour. We can find that there is an opportunity to increase the cache capacity based on the estimated WSS during that period to improve the cache hit ratio.

Another interesting discovery during our deployment is that the WSS of each Presto worker is quite unbalanced. This is because that the data hotness of each table or partition is different in real-world scenarios. The extent of the imbalance is related to the access patterns. Cuki is very helpful for global cache space allocation with multiple-scope optimization.

## 7 Related Work

A key challenge for improving cache utilization is provisioning the suitable cache size to fit dynamic workloads online. As analyzed in § 2, we summarize the prior works in four categories: Rule-based approaches [21, 24, 34, 42], ML-based approaches [4, 28, 30, 33, 35], MRC-based approaches [15, 19, 22, 36, 40, 41, 45, 51, 52], and window-based approaches [5, 11, 20].

The most recent works related to ours are ClockSketch [11], RAR-CM [51], and MBF [3]. ClockSketch [11] maintains a clock value for each item to support the sliding window mechanism. However, ClockSketch uses the bitmap [44] or the Bloom filter [8] to estimate cardinality. It brings WSS estimation error as not being aware of items' various sizes but using maximum likelihood estimation with inferior ARE. RAR-CM [51] uses a hashmap to record item access information and estimate the item repetition ratio. However, RAR-CM is designed for fixed-size item tracking and might be inaccurate for variable-size item tracking. Moreover, RAR-CM has non-negligible memory consumption when handling a large number of unique items. MBF [1, 25] uses a series of Bloom filters to record different statistics in segments of the sliding window. However, the switching of Bloom filters makes the estimation result accuracy unstable.

## 8 Conclusion and Future Work

In this paper, we propose Cuki, an approximate data structure for estimating the online WSS and IRR for variable-size item access with proven accuracy guarantee. Cuki can also be extended to solve the multi-scope WSS tracking problem. Experimental results show that Cuki outperforms the cutting-edge algorithms by 10× in accuracy. Moreover, the proposed adaptive cache capacity tuning method based on Cuki can significantly improve the cache performance online.

In the future, we plan to explore more application scenarios of Cuki in the cloud-native data processing environment.

## Acknowledgements

We thank reviewers and shepherd for their valuable comments and help. This work is funded in part by the National Natural Science Foundation of China (No.62072230, 62272223), Jiangsu Province Science and Technology Key Program (No.BE2021729), the Postgraduate Research & Practice Innovation Program of Jiangsu Province (No.KYCX22_0152), the Fundamental Research Funds for the Central Universities (No.020214380089, 020214380098, 020214912216), and the Collaborative Innovation Center of Novel Software Technology and Industrialization. Bin Fan, Haipeng Dai, Rong Gu, and Guihai Chen are the corresponding authors of this paper.

## References

[1] Alluxio. https://www.alluxio.io/, 2016.

[2] Guava: Google Core Libraries for Java. https://github.com/google/guava, 2020.

[3] The Implementation of Multiple Bloom Filter. https://github.com/Alluxio/alluxio/blob/v2.7.0/core/client/fs/src/main/java/alluxio/client/file/cache/CacheManagerWithShadowCache.java, 2021.

[4] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. CherryPick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI'17)*, pages 469–482, 2017.

[5] Eran Assaf, Ran Ben Basat, Gil Einziger, and Roy Friedman. Pay for a Sliding Bloom Filter and Get Counting, Distinct Elements, and Entropy for Free. In *37th IEEE International Conference on Computer Communications (INFOCOM'18)*, pages 2204–2212. IEEE, 2018.

[6] Ran Ben Basat, Michael Mitzenmacher, and Shay Vargaftik. How to Send a Real Number Using a Single Bit (And Some Shared Randomness). In *48th International Colloquium on Automata, Languages, and Programming, (ICALP'21)*, pages 439–458, 2021.

[7] Ran Ben-Basat, Gil Einziger, and Roy Friedman. Give me some slack: Efficient network measurements. In *43rd International Symposium on Mathematical Foundations of Computer Science, (MFCS'18)*, pages 543–559, 2018.

[8] Burton H Bloom. Space/Time Trade-Offs in Hash Coding with Allowable Errors. *Communications of the ACM*, 13(7):422–426, 1970.

[9] Paris Carbone, Stephan Ewen, Gyula Fóra, Seif Haridi, Stefan Richter, and Kostas Tzoumas. State Management in Apache Flink®: Consistent Stateful Distributed Stream Processing. *Proceedings of the VLDB Endowment (VLDB'17)*, 10(12):1718–1729, 2017.

[10] Hanhua Chen, Liangyi Liao, Hai Jin, and Jie Wu. The Dynamic Cuckoo Filter. In *25th IEEE International Conference on Network Protocols (ICNP'17)*, pages 1–10, 2017.

[11] Peiqing Chen, Dong Chen, Lingxiao Zheng, Jizhou Li, and Tong Yang. Out of Many We are One: Measuring Item Batch with Clock-Sketch. In *48th ACM Conference on Management of Data (SIGMOD'21)*, pages 261–273, 2021.

[12] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *1st ACM symposium on Cloud computing (SoCC'10)*, pages 143–154, 2010.

[13] Fernando J Corbato. A paging experiment with the multics system. Technical report, MASSACHUSETTS INST OF TECH CAMBRIDGE PROJECT MAC, 1968.

[14] L De La Peña-Auerbach. A simple derivation of the schroedinger equation from the theory of markoff processes. *Physics Letters A*, 24(11):603–604, 1967.

[15] Chen Ding and Yutao Zhong. Predicting whole-program locality through reuse distance analysis. In *24th ACM conference on Programming language design and implementation (PLDI'03)*, pages 245–257, 2003.

[16] Gil Einziger and Roy Friedman. Counting with Tinytable: Every Bit Counts! In *17th International Conference on Distributed Computing and Networking (ICDCN'16)*, pages 1–10, 2016.

[17] Bin Fan, Dave G Andersen, Michael Kaminsky, and Michael D Mitzenmacher. Cuckoo Filter: Practically Better than Bloom. In *10th ACM Conference on Emerging Networking Experiments and Technologies (CoNext'14)*, pages 75–88, 2014.

[18] Bin Fan, David G Andersen, and Michael Kaminsky. Memc3: Compact and Concurrent Memcache with Dumber Caching and Smarter Hashing. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI'13)*, pages 371–384, 2013.

[19] Alexander Fuerst and Prateek Sharma. FaasCache: keeping serverless computing alive with greedy-dual caching. In *26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event (ASPLOS'21)*, pages 386–400, 2021.

[20] Xiangyang Gou, Long He, Yinda Zhang, Ke Wang, Xilai Liu, Tong Yang, Yi Wang, and Bin Cui. Sliding Sketches: a Framework Using Time Zones for Data Stream Processing in Sliding Windows. In *26th ACM Conference on Knowledge Discovery and Data Mining (KDD'20)*, pages 1015–1025, 2020.

[21] Rong Gu, Kai Zhang, Zhihao Xu, Yang Che, Bin Fan, Haojun Hou, Haipeng Dai, Li Yi, Yu Ding, Guihai Chen, and Yihua Huang. Fluid: Dataset Abstraction and Elastic Acceleration for Cloud-native Deep Learning Training Jobs. In *38th IEEE International Conference on Data Engineering (ICDE'22)*, pages 2182–2195, 2022.

[22] Xiameng Hu, Xiaolin Wang, Lan Zhou, Yingwei Luo, Chen Ding, and Zhenlin Wang. Kinetic Modeling of Data Eviction in Cache. In *27th USENIX Annual Technical Conference (USENIX ATC'16)*, pages 351–364, 2016.

[23] Faria Kalim, Le Xu, Sharanya Bathey, Richa Meherwal, and Indranil Gupta. Henge: Intent-Driven Multi-Tenant Stream Processing. In *9th ACM Symposium on Cloud Computing (SoCC'18)*, pages 249–262, 2018.

[24] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. Pocket: Elastic Ephemeral Storage for Serverless Analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*, pages 427–444, 2018.

[25] Haoyuan Li, Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In *5th ACM Symposium on Cloud Computing (SoCC'14)*, page 1–15, 2014.

[26] Xiaozhou Li, David G Andersen, Michael Kaminsky, and Michael J Freedman. Algorithmic Improvements for Fast Concurrent Cuckoo Hashing. In *9th European Conference on Computer Systems (EuroSys'14)*, pages 1–14, 2014.

[27] Lailong Luo, Deke Guo, Ori Rottenstreich, Richard T. B. Ma, Xueshan Luo, and Bangbang Ren. The Consistent Cuckoo Filter. In *38th IEEE International Conference on Computer Communications (INFOCOM'19)*, pages 712–720, 2019.

[28] Ashraf Mahgoub, Alexander Michaelson Medoff, Rakesh Kumar, Subrata Mitra, Ana Klimovic, Somali Chaterji, and Saurabh Bagchi. OPTIMUSCLOUD: Heterogeneous configuration optimization for distributed databases in the cloud. In *31st USENIX Annual Technical Conference (USENIX ATC'20)*, pages 189–203, 2020.

[29] Richard L. Mattson, Jan Gecsei, Donald R. Slutz, and Irving L. Traiger. Evaluation Techniques for Storage Hierarchies. *IBM Systems journal*, 9(2):78–117, 1970.

[30] Djob Mvondo, Mathieu Bacou, Kevin Nguetchouang, Lucien Ngale, Stéphane Pouget, Josiane Kouam, Renaud Lachaize, Jinho Hwang, Tim Wood, Daniel Hagimont, Noël De Palma, Bernabé Batchakui, and Alain Tchana. OFC: An Opportunistic Caching System for FaaS Platforms. In *16th European Conference on Computer Systems (EuroSys'21)*, page 228–244, 2021.

[31] Dushyanth Narayanan, Austin Donnelly, and Antony Rowstron. Write Off-Loading: Practical Power Management for Enterprise Storage. *ACM Transactions on Storage (TOS'08)*, 4(3):1–23, 2008.

[32] David MW Powers. Applications and explanations of zipf's law. In *New methods in language processing and computational natural language learning*, 1998.

[33] Gourav Rattihalli, Madhusudhan Govindaraju, Hui Lu, and Devesh Tiwari. Exploring potential for non-disruptive vertical auto scaling and resource estimation in kubernetes. In *12nd International Conference on Cloud Computing (CLOUD'19)*, pages 33–40, 2019.

[34] Francisco Romero, Gohar Irfan Chaudhry, Íñigo Goiri, Pragna Gopa, Paul Batum, Neeraja J Yadwadkar, Rodrigo Fonseca, Christos Kozyrakis, and Ricardo Bianchini. Faa$T: A transparent auto-scaling cache for serverless applications. In *12nd ACM Symposium on Cloud Computing (SoCC'21)*, pages 122–137, 2021.

[35] Krzysztof Rzadca, Pawel Findeisen, Jacek Swiderski, Przemyslaw Zych, Przemyslaw Broniek, Jarek Kusmierek, Pawel Nowak, Beata Strack, Piotr Witusowski, Steven Hand, and John Wilkes. Autopilot: Workload Autoscaling at Google. In *15th European Conference on Computer Systems (EuroSys'20)*, pages 1–16, 2020.

[36] Trausti Saemundsson, Hjortur Bjornsson, Gregory Chockler, and Ymir Vigfusson. Dynamic Performance Profiling of Cloud Caches. In *5th ACM Symposium on Cloud Computing (SoCC'14)*, pages 1–14, 2014.

[37] Raghav Sethi, Martin Traverso, Dain Sundstrom, David Phillips, Wenlei Xie, Yutian Sun, Nezih Yegitbasi, Haozhun Jin, Eric Hwang, Nileema Shingte, and Christopher Berner. Presto: SQL on Everything. In *35th IEEE International Conference on Data Engineering (ICDE'19)*, pages 1802–1813. IEEE, 2019.

[38] TPC-DS Benchmark. http://www.tpc.org/tpcds/, 2006.

[39] Vojislav Ðukić, Sangeetha Abdu Jyothi, Bojan Karlaš, Muhsen Owaida, Ce Zhang, and Ankit Singla. Is Advance Knowledge of Flow Sizes a Plausible Assumption? In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI'19)*, pages 565–580, 2019.

[40] Carl Waldspurger, Trausti Saemundsson, Irfan Ahmad, and Nohhyun Park. Cache modeling and Optimization using Miniature Simulations. In *28th USENIX Annual Technical Conference (USENIX ATC'17)*, pages 487–498, 2017.

[41] Carl A Waldspurger, Nohhyun Park, Alexander Garthwaite, and Irfan Ahmad. Efficient MRC Construction with SHARDS. In *13rd USENIX Conference on File and Storage Technologies (FAST'15)*, pages 95–110, 2015.

[42] Ao Wang, Jingyuan Zhang, Xiaolong Ma, Ali Anwar, Lukas Rupprecht, Dimitrios Skourtis, Vasily Tarasov, Feng Yan, and Yue Cheng. InfiniCache: Exploiting Ephemeral Serverless Functions to Build a Cost-Effective Memory Cache. In *18th USENIX Conference on File and Storage Technologies (FAST'20)*, pages 267–281, 2020.

[43] Hancheng Wang, Haipeng Dai, Meng Li, Jun Yu, Rong Gu, Jiaqi Zheng, and Guihai Chen. Bamboo Filters: Make Resizing Smooth. In *38th IEEE International Conference on Data Engineering (ICDE'22)*, pages 979–991, 2022.

[44] Kyu-Young Whang, Brad T Vander-Zanden, and Howard M Taylor. A Linear-Time Probabilistic Counting Algorithm for Database Applications. *ACM Transactions on Database Systems (TODS'90)*, 15(2):208–229, 1990.

[45] Jake Wires, Stephen Ingram, Zachary Drudi, Nicholas JA Harvey, and Andrew Warfield. Characterizing Storage Workloads with Counter Stacks. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*, pages 335–349, 2014.

[46] Le Xu, Shivaram Venkataraman, Indranil Gupta, Luo Mai, and Rahul Potharaju. Move Fast and Meet Deadlines: Fine-Grained Real-Time Stream Processing with Cameo. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI'21)*, pages 389–405, 2021.

[47] Yahoo! Cloud Serving Benchmark (YCSB). https://github.com/brianfrankcooper/YCSB, 2020.

[48] Juncheng Yang, Yao Yue, and KV Rashmi. A large Scale Analysis of Hundreds of In-Memory Cache Clusters at Twitter. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*, pages 191–208, 2020.

[49] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI'12), San Jose, CA, USA, April 25-27, 2012*, pages 15–28, 2012.

[50] Fan Zhang, Hanhua Chen, Hai Jin, and Pedro Reviriego. The Logarithmic Dynamic Cuckoo Filter. In *37th IEEE International Conference on Data Engineering (ICDE'21)*, pages 948–959, 2021.

[51] Yu Zhang, Ping Huang, Ke Zhou, Hua Wang, Jianying Hu, Yongguang Ji, and Bin Cheng. OSCA: An Online-Model based cache allocation scheme in cloud block storage systems. In *31st USENIX Annual Technical Conference (USENIX ATC'20)*, pages 785–798, 2020.

[52] Weiming Zhao, Xinxin Jin, Zhenlin Wang, Xiaolin Wang, Yingwei Luo, and Xiaoming Li. Low Cost Working Set Size Tracking. In *22nd USENIX Annual Technical Conference (USENIX ATC'11)*, pages 223–229, 2011.

[53] Yang Zhou, Tong Yang, Jie Jiang, Bin Cui, Minlan Yu, Xiaoming Li, and Steve Uhlig. A Community Cache with Complete Information. In *19th USENIX Conference on File and Storage Technologies (FAST'21)*, pages 323–340, 2021.

## Technical Appendix

## A Artifact Appendix

### Abstract

Cuki is implemented on Alluxio. It also relies on Presto, Hive, and HDFS to function properly. We prepare the programs, assemble a workflow of Cuki and package the artifact into the Git repository.

### Scope

The artifact estimates the WSS of different traces. It verifies the basic function of Cuki and validates the accuracy improvement brought by item-wise fine-grained tracking. In addition, this artifact also validates the MRC generation accuracy of Cuki is higher than the SOTA algorithm.

### Contents

The artifact includes the source code of Cuki and experiments scripts. A "README.md" file can be also found in the artifact. It contains detailed description of the artifact and a step-by-step instruction for evaluation.

### Hosting

The artifact is available at GitHub[1]. All branches are needed to be cloned or downloaded for evaluation. The commit version is the latest one.

### Requirements

The environment of the artifact includes Hive 3.1.3, Maven 3.5.4, Hadoop 3.3.1, Java 8, Prometheus 2.37.0, Mysql 8.0.3, and S3.

## B Theoretical Proof of Cuki

We first analyze the false positive rate of Cuki. Then, we theoretically demonstrate that Cuki outperforms the competitive state-of-the-art algorithms in space usage under the same false positive rate. We summarize the notations in Table 1.

**Theorem B.1.** *For Cuki with $f$-bits fingerprint and $s$-bits clock, the false positive rate is given by*

$$\varepsilon = 1 - \left(1 - \frac{1}{2^f}\right)^{2b \cdot \frac{2^s}{2^s-1} \cdot \frac{\mathcal{D}}{n \cdot b}} \approx \frac{2^s}{2^s - 1} \cdot \frac{2\mathcal{D}}{n \cdot 2^f}, \tag{5}$$

[1] Our artifact: https://github.com/shadowcache/Cuki-Artifact-WSS-Estimation.

Table 1: Notations

| Notations | Definition |
|---|---|
| $f$ | Bits length of the fingerprint |
| $s$ | Bits length of the clock |
| $\varepsilon$ | False positive rate |
| $b$ | Number of entries in a bucket |
| $\mathcal{D}$ | Number of distinct items in a sliding window |
| $n$ | Number of buckets in a Cuki |
| $\alpha$ | Load factor of a Cuki |
| $N$ | Number of items in a Cuki |
| $\mathcal{T}$ | Size of a sliding window |

*where $b$ represents the number of entries in each bucket, $\mathcal{D}$ represents the number of distinct items in each sliding window, and $n$ represents the number of buckets in Cuki.*

*Proof:* The false positive rate of Cuki comes from two aspects: (i) Cuki stores fingerprints instead of original item keys. (ii) The outdated items in Cuki might not be cleaned up timely. For Cuki with $n$ buckets, we define the load factor as

$$\alpha = \frac{N}{n \cdot b}, \tag{6}$$

where $N$ represents the number of fingerprints stored in Cuki, and $b$ represents the number of entries in each bucket.

When querying an element that does not exist in Cuki, $2 \cdot b \cdot \alpha$ fingerprints need to be checked. For Cuki with $f$-bits per fingerprint, each check may match a wrong fingerprint and return a false positive with a probability of $1/2^f$. Therefore, the false positive rate caused by storing fingerprints is

$$\varepsilon = 1 - (1 - 1/2^f)^{2b\alpha}. \tag{7}$$

For any item in Cuki, it will be cleaned up after performing $2^s$ rounds of the aging operation. For a sliding window of size $\mathcal{T}$, to prevent an item from being mistakenly deleted before its time window ends, the frequency of the aging operation is $\frac{\mathcal{T}}{2^s-1}$. Thus, for an item in the data stream, the time interval between insertion and clean-up is $\frac{2^s}{2^s-1}\mathcal{T}$. In other words, Cuki actually stores all the items inserted within the time interval $\frac{2^s}{2^s-1}\mathcal{T}$, which is $\frac{2^s}{2^s-1}$ times of the sliding window size. Suppose the number of distinct items within each sliding window is $\mathcal{D}$, the number of items stored in Cuki is given by

$$N = \frac{2^s}{2^s - 1}\mathcal{D}. \tag{8}$$

Combining Equations (6), (7), and (8), we have

$$\varepsilon = 1 - \left(1 - \frac{1}{2^f}\right)^{2b \cdot \frac{2^s}{2^s-1} \cdot \frac{\mathcal{D}}{nb}} \approx \frac{2^s}{2^s - 1} \cdot \frac{2\mathcal{D}}{n \cdot 2^f}. \qquad \square$$

**Experimental verification:** We conduct experiments to validate Theorem B.1. We vary $f$ from 4 to 11, and set $s$ as 12 - $f$. Other parameters follow the settings in § 6. As shown in Figure 14, the experimental results show that the theoretical false positive rate well matches the experimental results.

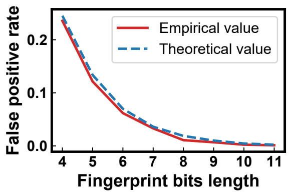


Figure 14: Verification of Theorem B.1.

**Corollary B.1.1.** *For a fixed memory consumption M, when $s = 1$, the minimum false positive rate is given by*

$$\frac{8\mathcal{D}}{n \cdot 2^{M/n \cdot b}},$$

*where $M = n \cdot b(f + s)$ represents the memory consumption of fingerprints and clocks in Cuki, n represents the number of buckets in Cuki, b represents the number of entries in each bucket, f represents the bits length of the fingerprint, s represents the bits length of the clock, and $\mathcal{D}$ represents the number of distinct items in each sliding window.*

*Proof:* As per Theorem B.1, the false positive rate is mainly affected by $f$ and $s$. Thus we only analyze the memory consumption of fingerprints and clocks. Plug $f = \frac{M}{nb} - s$ into Equation (5), and we get

$$\varepsilon(s) = \frac{2\mathcal{D}}{n \cdot 2^{M/nb}} \cdot \frac{4^s}{2^s - 1},$$

where $s = 1, 2, \ldots, \frac{M}{nb} - 1$. Obviously, the false positive rate increases as $s$ increases, and $\varepsilon(s)$ is the minimum when $s = 1$. This completes the proof. □

**Corollary B.1.2.** *For the same false positive rate ε, Cuki requires less space than ClockSketch [11] and SWAMP [5].*

*Proof:* According to Corollary B.1.1, let $\mathcal{T} = n \cdot b$, $n > 8$, the memory consumption of Cuki can be computed as

$$M(\varepsilon) = \mathcal{T}\log_2 \frac{8\mathcal{D}}{n\varepsilon} < \mathcal{T}\log_2 \frac{\mathcal{D}}{\varepsilon} \leq \mathcal{T}\log_2 \frac{\mathcal{T}}{\varepsilon}. \tag{9}$$

According to [11], by ignoring the memory consumption caused by storing the size field and the payload field, the memory consumption of SWAMP is

$$M_1(\varepsilon) > \mathcal{T}\log_2 \frac{\mathcal{T}}{\varepsilon}. \tag{10}$$

Therefore, to achieve the same false positive rate ε, the memory consumption of Cuki is always lower than that of SWAMP.

As per [11], the memory consumption of ClockSketch is

$$M_2(\varepsilon) \approx \frac{8}{3\ln 2}\mathcal{T}\log_2 \frac{1}{\varepsilon} \approx 3.8472\mathcal{T}\log_2 \frac{1}{\varepsilon}. \tag{11}$$

Let $\mathcal{T} = 2\mathcal{D}$, the memory consumption of Cuki is given as

$$M(\varepsilon) = 4\mathcal{T} + \mathcal{T}\log_2 \frac{1}{\varepsilon}. \tag{12}$$

When $\varepsilon < 37.76\%$, which is often satisfied in real-world applications [17], $M(\varepsilon) < M_2(\varepsilon)$. This completes the proof. □

## C Evaluation

### C.1 Motivated Example of Opportunistic Aging: Estimation Fluctuation

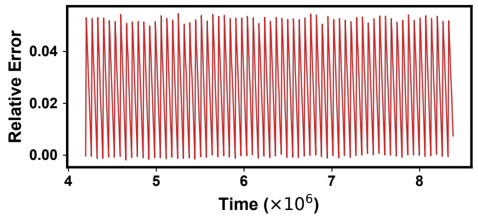
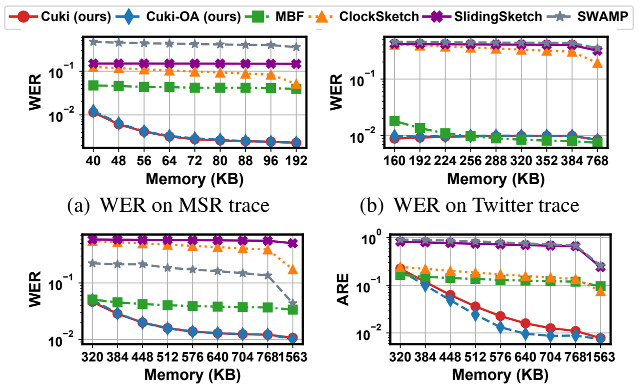


Figure 15: An example of estimation result fluctuating on the YCSB dataset (The size of a count-based sliding window is 65,536, and the clock bits is set to 4).

A large number of items will be cleared at the same time in the background aging process. As shown in Figure 15, the working set size is overestimated before aging. After the execution of aging, a tremendous amount of items are instantly cleared. Therefore the estimation result are fluctuating, and may affect the error of the estimated WSS. We propose an optimization method named opportunistic aging to alleviate this problem in aging operation.

### C.2 Accuracy Evaluation of Cuki

In this experiment, we evaluate the accuracy of different WSS estimation methods. This experiment observes an additional metric WER on three traces (including the YCSB trace not shown in § 6.3), which can be seen as a supplement to § 6.3.



(a) WER on MSR trace (b) WER on Twitter trace

(c) WER on YCSB trace (d) ARE on YCSB trace

Figure 16: Performance comparison of accuracy.

Figure 16 shows the ARE and WER of different methods measured in the same run on three traces. The ARE or WER of All methods is high without sufficient memory. The ARE or WER of Cuki decreases to 1% and lower as the memory space gradually becomes larger. However, even with sufficient memory, the ARE or WER of other methods can hardly further decrease. Take the ClockSketch as an example, The WER of

ClockSketch is decreased from 12.52% to 5.13% on MSR trace as the memory increases to 1563KB. In contrast, Cuki decreases the WER from 1.14% to 0.24% as the memory increases to 768KB. This is due to the fine-grained per-item tracking method in Cuki. Although the WER of MBF is close to Cuki on the Twitter trace, Cuki performs much better in other traces. This is because MBF switches a Bloom filter out periodically and causes errors for the estimated result.

To conclude, similar to the experiment results in § 6.3, Cuki and Cuki-OA still achieve the best accuracy with the same memory consumption regarding the WER metric and YCSB trace.

## C.3 Stability Evaluation of Cuki

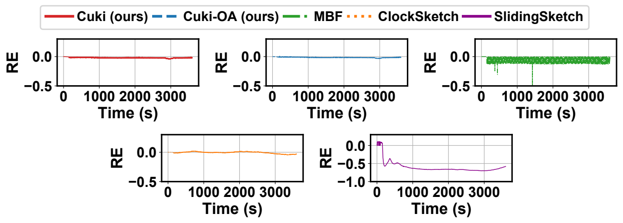


Figure 17: Performance comparison of stability in Twitter trace (time-based)

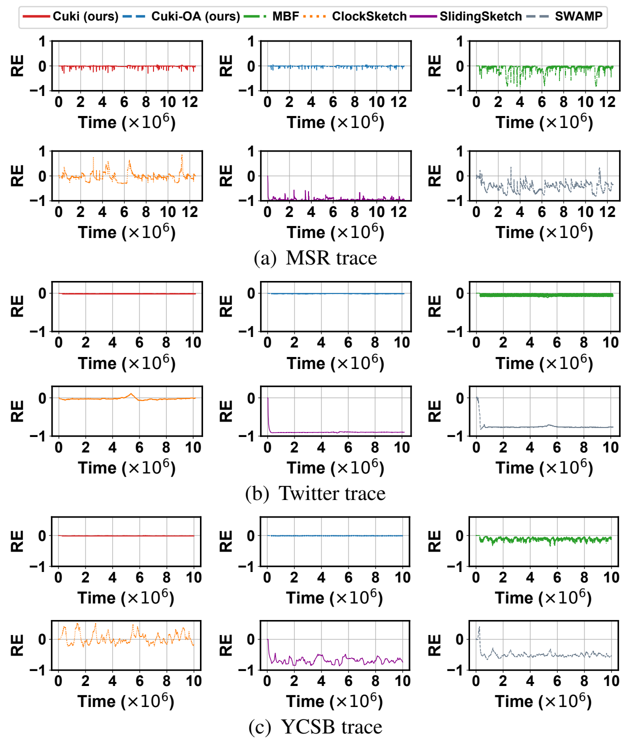


Figure 18: Performance comparison of stability (count-based)

In this experiment, we evaluate the stability of different methods under the time-based sliding window and the count-based sliding window. For the count-based sliding window, We use the default configuration described in § 6.1. For the time-based sliding window on Twitter trace, we allocate 1408KB memory for Cuki and double memory for other methods to meet their memory requirements. We replay the Twitter trace with 24× speedup according to the data request traffic. Figures 17 and 18 illustrate the stability performance of different methods over the time-based sliding window and the count-based sliding window, respectively. The estimation results of a count-based sliding window are more stable than that of a time-based sliding window. This is because the number of items in a count-based window is fixed. However, there are still some jagged fluctuations in all methods. The reasons for these fluctuations are the same as we show in § 6.4. Benefiting from the per-item size tracking, the RE of Cuki and Cuki-OA is the most stable of the other four methods. Cuki-OA has a more stable estimation result than Cuki because of the opportunistic aging. To sum up, we conclude that Cuki and Cuki-OA also achieve the most stable and accurate estimates on a count-based window.