Efficient Estimation of Read Density when Caching for Big Data Processing

Sacheendra Talluri
AtLarge Research and TU Delft,
Delft, the Netherlands
s.talluri@atlarge-research.com

Alexandru Iosup
Vrije Universiteit,
Amsterdam, the Netherlands
a.iosup@vu.nl

***Abstract*—Big data processing systems are becoming increasingly more present in cloud workloads. Consequently, they are starting to incorporate more sophisticated mechanisms from traditional database and distributed systems. We focus in this work on the use of caching policies, which for big data raise important new challenges. Not only they must respond to new variants of the trade-off between hit rate, response time, and the space consumed by the cache, but they must do so at possibly higher volume and velocity than web and database workloads. Previous caching policies have not been tested experimentally with big data workloads. We address these challenges in this work. We propose the Read Density family of policies, which is a principled approach to quantify the utility of cached objects through a family of utility functions that depend on the frequency of reads of an object. We further design the Approximate Histogram, which is a policy-based technique based on an array of counters. This technique promises to achieve runtime-space efficient computation of the metric required by the cache policy. We evaluate through trace-based simulation the caching policies from the Read Density family, and compare them with over ten state-of-the-art alternatives. We use two workload traces representative for big data processing, collected from commercial Spark and MapReduce deployments. While we achieve comparable performance to the state-of-art with less parameters, meaningful performance improvement for big data workloads remain elusive.**

I. Introduction

Big data workloads are increasingly more needed in our professional and personal lives. Their execution in datacenters, as cloud services, is increasingly required for economic growth and productivity [1]–[3]. The systems for big data processing must respond to increasing and more sophisticated demands, from unprecedented volume and velocity of data, to new applications and more complex data-storage solutions. Systems designers achieve this through a variety of approaches, e.g., by incorporating increasingly more sophisticated techniques for managing data, such as dynamic partitioning [4] and auto-tiering, and by tuning well-known mechanisms, such as indexing and caching. (Recursively, a cache is fast storage situated in-between the processor and slower storage. Requesting an object that is in cache—a *hit*—results in a significant performance improvement, relatively to when the object is not in cache.) We focus in this work on the challenges raised by caching for big data workloads, in particular, on the design of efficient caching policies, and on evaluating these and existing caching policies against representative big data workloads.

Caches are based on the hypothesis that some objects are more frequently accessed than others; in general, placing these objects on fast storage improves performance. Typically, the popularity of different objects (their frequency of access) changes throughout their lifetime, and often also throughout the lifetime of the application accessing the object(s). For a cache to remain useful, the population of objects in the cache also needs to change, which happens subject to a *cache policy*. Cache policies consist of at least two components: an *eviction policy*, which decides which object to remove from the cache to make room for a new object; and an *admission policy*, which decides whether a new object should be stored in the cache. We focus in this work on eviction policies. We conjecture the techniques proposed in this paper should also apply to admission policies, but leave testing this for future work.

The design of cache policies must respond to a complex trade-off, whose components include at least the *hit rate* (ratio, when considered instantaneously), that is, the ratio between the number of objects found in cache and the number of requested objects from the cache, the time it takes the policy to take a decision, and the storage size used by the cache including its data structures. Consequence of this complex trade-off, which can change also due to the needs of both datacenter clients and operators, cache policies are typically simple by design. Commonly used policies are typically simple and based on heuristics (e.g., LRU, LFU, LIRS [5]), or are based on complex mechanisms that manage however collections of simple policies (e.g., ARC [6], Hyperbolic [7]).

Focusing on the system model introduced in Section II, this work makes a three-fold contribution to the existing body of work (see Section VII):

1) We design the Read Density family of caching policies (Section III). Our approach to design is based on two core principles, which extend in particular 2018 work on database caching [8] to scenarios where classification using simple categories can leads to miscategorization and where application-specific information is unavailable. Our approach also generalizes the Hit Density metric [8] to an entire family of utility functions.
2) We use the Approximate Histogram in caching for big data (Section IV). We employ this data structure, which is used in distributed systems and in networking, to keep track of reads of an object over time. We use the result

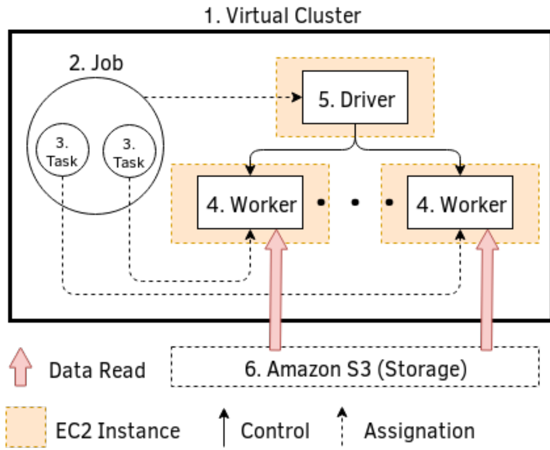

Fig. 1: System architecture of a virtual cluster for big data.

to make the Read Density caching policies usable in practice for long-running big data workloads.

3) We employ trace-based simulation to analyze the caching policies proposed in this work, and to compare them Performance evaluation of cache policies for big data storage workloads (Section VI). Ours are the first experiments to quantify the performance of state-of-the-art caching policies for big data workloads.

II. System Model

We use in this work the system model for the operation of (Spark-based) big data workloads in the cloud that we have introduced in our previous work [9]. This model is common in practice across many organizations, in particular, for commercial big data operations at Yahoo (MapReduce) and Databricks (Spark). Figure 1 depicts the system model.

Job stream: jobs are submitted by users or created by events, and arrive for execution in the system as a *stream*. Each *job* (component 2 in Figure 1) is structured as a directed acyclic graph of inter-dependent tasks. It *reads* input from the persistent storage, processes it, and produces output.

System Architecture: a big data system running on one or more *virtual clusters*, comprised of virtual machines (VMs) that are organized into a logical group and communicate using a virtual network. The big data system is composed of typical software that runs big data tasks and that coordinates the execution (components 4 and 5, respectively). The VMs are leased, from a cloud provider such as Amazon Web Services or from a private cloud.

The virtual cluster is connected at runtime to a source of incoming data, from which it reads. In our model, a typical source of data is a cloud-based, persistent, *object-store* (component 6), for example, Amazon S3.

Cache workload: We focus in this work on caching for the workload of requests issued to the storage layer, between the system workers (component 4 in the figure) and the system storage (6). The cache *workload* is comprised of all the data-requests across all jobs that try to reach the storage system.

III. Design of the Read Density Family

We propose in this section the principled design of the Read Density family of caching policies. Today's storage systems deployed in the cloud or on-premises commonly treat applications as black-boxes. Reasons relate to business (e.g., different vendors), operations (e.g., no access to application code or large variety of applications), and regulations (e.g., the EU-law GDPR and other privacy-related aspects). Thus, the first principle of our design is: (**P0**) Caching policies have no access to application-specific information.

We use two more principles: (**P1**) Objects with a high number of reads are more valuable, and (**P2**). Objects read earlier are more valuable. Principle P1 focuses on cache hits. Principle P2 focuses on resource-use. Each object in the cache uses a resource (a *slot*), accumulating resource-use over time. An object which maximizes the number of hits, while using a slot for the least amount of time, has the highest *read density*. Focusing on read density has an added benefit: the system does not try to predict the time to eviction, and thus avoids the second-order effect of evicting an object based on the prediction of when it will be evicted; we conjecture that such an effect can increase rapidly with the volume of objects in a big data system.

Family of Caching Policies: To design policies related to these principles, we start from the common assumption that past IO-behavior of an object is indicative of its future IO-behavior. Thus, we use quantities (metrics) that consider the *reuse time*, that is, the time elapsed between two consecutive reads of an object. Based on reuse time, we propose in this section a mechanism and two utility functions, that is, functions that quantify the value of keeping an object in the cache. For any caching policy using reuse-time to decide about object-eviction, and in particular for any heuristic that uses this information straightforwardly, the designer can use the mechanism and the proposed utility functions (and others similarly defined) to create *a family of caching policies*.

Mechanism: For each object in the cache workload (see Section II), the caching policy considers the histogram with reuse time on the horizontal axis and the read count (the popularity) on the vertical axis. The histogram is updated every time a read operation occurs. Dividing the popularity of an object at a particular reuse time by the total number of reads gives the empirical probability that the object is read after that (reuse) time, as in Equation 1:

$$P(R = o|I = t) = \frac{N(R = o|I = t)}{\mathrm{T}} \tag{1}$$

where P is the probability computed for object o and reuse time t, T is the total number of reads of that object, N is the empirical popularity, and R is the random variable of reads of objects and I is the random variable of reuse times.

The policy can now compute the expectation of a read for a given object, taking into account the time since the object was last accessed—the *age* of the object (a), in Equation 2:

$$E(R = o) = \sum_{i=a}^{n} 1 \times P(R = o|I = i) \tag{2}$$

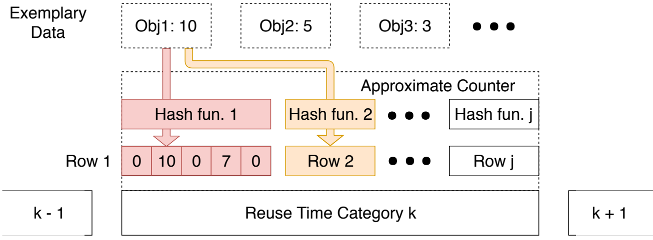

Fig. 2: Example of an approximate histogram.

Utility Functions: Equation 2 does not penalize reads with long reuse time and do not reward reads with short reuse time. We need to include these to support **P2**. A penalty can be applied in two main ways. One is to divide the expectation of a read by the expected reuse time [8]. The other is to divide the probability of a read at a particular reuse time, by that reuse time. We give two utility functions of the latter kind.

(1) Using Expected Reuse Time replacing in Equation 2 the value one with the expected reuse time (i), the expected reuse time of a particular object is given by:

$$EI(R = o) = \sum_{i=a}^{n} i \times P(R = o|I = i) \tag{3}$$

Using Equations 2 and 3, Equation 4 gives the value of keeping an object in the cache, $D1$. Intuitively, $D1$ uses the expected reuse time to quantify the reward for letting an item stay in the cache.

$$D1(R = o) = \frac{\sum_{i=a}^{n} 1 \times P(R = o|I = i)}{\sum_{i=a}^{n} i \times P(R = o|I = i)} \tag{4}$$

(2) Using Division by Reuse Time

Instead of calculating the expected reuse time, we can penalize the probability of a read directly during the expected probability calculation, by dividing it by the reuse time. Equation 5 quantifies this penalty:

$$D2(R = o) = \sum_{i=a}^{n} 1 \times \frac{P(R = o|I = i)}{i} \tag{5}$$

IV. Approximate Histogram

We propose in this section to use Approximate Histograms for keeping track of object reuse-time. In Section III, ranking objects for eviction is based on histograms used to calculate utility functions, one per object. At big-data scale, keeping complete histograms for each object is not feasible, due to space constraints. A mitigating approach is to categorize items by fine-grained features, such as the application type and the reuse time of the last read [8], and to keep histograms per category instead of per object. However, this forces the policy designer to decide on (optimal) categories, based on information that may not be available at design time or even at all; for example, a file-system cache may not know the application type of a given IO-request. Instead, we propose to use Approximate Histograms to keep track of every object, but without the space requirements of storing whole histograms. Approximate Histograms operate on coarse-grained features, so they also requires less parameter-tuning than using hand-picked categories.

Overview: An Approximate Histogram is an array of approximate counters coupled with a decay function. Figure 2 depicts an example. Each approximate counter in the array keeps the counts of objects accessed in a reuse-time category. The whole range of the reuse-time needs to be split into *coarse-grained* categories, to limit memory usage and to make the histogram immune to noise. The split, linear or logarithmic, designer-defined or auto-tuned, and results in a number of adjacent ranges (categories). For example, a time range of 0 to 1000 milliseconds can be divided into adjacent categories of 100 milliseconds each: 0-99 for category 1, 100-199 for category 2, and so on.

Approximate Counters: Each category has an approximate counter (sketch). Examples of approximate counting data structures include count-min sketch [10], cuckoo filter [11], and counting quotient filter [12]. They have different properties related to performance, storage size, and other maintenance aspects such as duplication resistance. When an object is read, the counter in the corresponding reuse time category is incremented for that object.

In this work, we use the count-min sketch as approximate counter. A count-min sketch consists of several arrays (*rows*), each associated with its own hash function. The hash function takes the identifier (*key*) of an object and returns a position in the array (the *column*). Whenever an object is read, the positions with the minimum value across all rows for this object are incremented by one; this means that the count-min sketch can only return the *minimum* number of times an object has been read. The size of the count-min sketch, that is, the number of rows and columns, is decided at design time, and gives known time-space properties and a bounded probability of error.

Approximate Histogram for Big Data Processing: Big data processing involves many repeated operations, where the iterations commonly reuse old data. Thus, caches can serve the same application for a long period of time. During this period, the counters of the Approximate Histogram can get saturated. To prevent saturation, periodically, the counters can be reset to zero or they can be *decayed* (e.g., divided by a constant). Because there are relatively few approximate counters, and they are placed in memory contiguously, decaying them is relatively inexpensive. We use for this the technique presented in TinyLFU [13]: all counters are reset periodically, with the period determined from a scalar time that is incremented by one for every increment of the approximate counter; for example, the period can be set such that the counters are decayed every 10000 reads.

V. Experiment Setup

We design in this section a set of experiments, with two main goals. First, we aim to conduct the first analysis of the Approximate Read Density (ARD) policy, which combines the family of caching policies based on Read Density (Section III) and using the Approximate Histogram (Section IV). Second,

TABLE I: List of experiments and their configurations.

Subsection	Goal	Workload	Policy	Parameters Varied	Cache Size [K]
VI-B	Compare Read Density based eviction metrics	Spark (Databricks)	ARD	eviction metric	5–625
VI-B1	Analyze sensitivity of approximate histogram	Spark (Databricks)	ARD	max count, num. time categories	5–625
VI-C1	Compare eviction policies for Databricks workload	Spark (Databricks)	All	None	5–625
VI-C2	Compare eviction policies for Yahoo workload	MapReduce (Yahoo)	All	None	5–625

we aim to conduct the first comprehensive comparison of caching policies for big data workloads.

A. Overview of the Setup

Table I summarizes the experiment design. The experiments first focus on the ARD policy, for which they focus on the Read Density and the Approximate Histogram aspects, and then proceed to compare eviction policies for two big data workloads.

Workloads: We use real traces collected from commercial big data deployments. The Spark workload trace is a subset of trace W2 analyzed in [9]. The MapReduce trace is a subset of the publicly available Yahoo Webscope 3 dataset[1].

Cache Size: We experiment with the cache size set between 5,000 and 625,000 objects, which are in line with real-world usage of caches for big data processing. For these values, the ratio of the working set size to the cache size is in the range of around 10,000 to under 10, respectively.

Key performance metric: The performance metric measured in all experiments is the *hit rate*, which refers to the total fraction of reads that were hits. All experiments were run ten times and the mean hit rate for each configuration was used. The standard deviation of the results was less that 1%.

Simulator: We use for experiments the simulation framework of Caffeine[2], a popular in-memory cache library for Java. This simulator comes equipped with the policies described in Section V-B and has been previously used in at least one other high-quality peer-reviewed study [13]. We implement our ARD policy in this simulator.

Parameters: The Approximate Histogram has a preconfigured number of approximate counters—all are count-min sketches. The number and length of the arrays ("max count" and "num time categories" in Table I, respectively) is decided from the values configured for epsilon and confidence, which are the same as in the original count-min sketch paper [10].

B. Tested Policies

We compare ARD with over 10 state-of-the-art caching policies, and with the *Random* policy (evicting a random object). Next to the commonly used FIFO, LFU, and LRU, we consider all the policies described and contrasted conceptually with ARD in Section VII-A.

CLOCK is similar to FIFO, but an object is not evicted the first time it reaches the highest priority for eviction; instead, it gets evicted the second time it is considered in such a position.

Segmented LRU (**SLRU**) divides the cache into two partitions. Both partitions use the LRU eviction policy. An object is evicted from the first into the second partition. We set the second partition to take 90% of the cache space.

Adaptive Replacement Cache (**ARC**) [6] partitions the cache into two parts, which compete for space based on hit rate. They use the LRU and LFU for eviction, respectively.

CART [14] combines ARC two-partition dynamics with CLOCK second-hit eviction.

LIRS [5] evicts object based on their last reuse time.

Hyperbolic [7] evicts object based on utility, which is hits divided by time in cache.

W-TinyLFU [13] divides the cache into two partitions and uses the TinyLFU policy for controlling admission to the second partition. It uses LRU and SLRU as eviction policies for the first and second partition respectively.

VI. Experiment Results

We summarize in this section the main experimental results, obtained using the experiment design from Section V.

A. Parameter Sweep of Approximate Read Density Policy

The Approximate Read Density (ARD) policy has few parameters, in particular, fewer than LHD. We analyze and show evidence about the insensitivity of the hit rate to the ARD parameters, when the parameters are set to reasonable ranges. A policy is insensitive to its parameters means that its performance doesn't change when the parameters change. Reasonable ranges means that the parameters should not be set to absurd values such as zero or extremely large values.

B. Eviction Metric Comparison

We proposed two metrics belonging to the Read Density family in Section III: 1. Using expected reuse time (D1(R)), and 2. Using division by reuse time (D2(R)). We compare the hit rate performance the these two metrics along with just a metric based on expectation of read (E(R)). We compare their performance on the Databricks trace using a cache with entry and admission metrics set to allow all. The results are depicted

[1]https://webscope.sandbox.yahoo.com/catalog.php?datatype=s&did=57

[2]https://github.com/ben-manes/caffeine

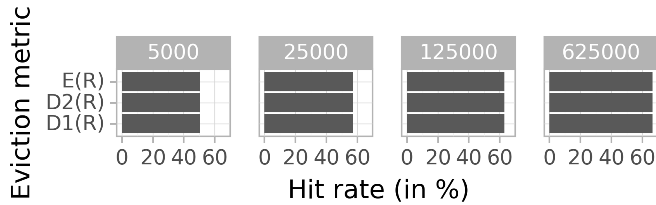

Fig. 3: Comparison of eviction metrics. Each panel corresponds to a different cache size (number of objects).

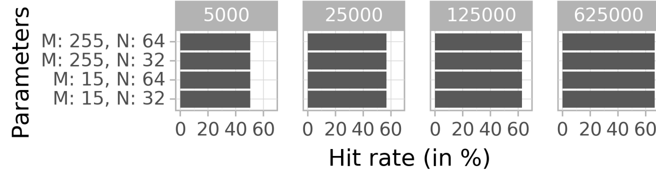

Fig. 4: Comparison of different parameter sets. Each panel corresponds to a different cache size (number of objects). M is the maximum count per category of the histogram. N is the number of time categories in a histogram.

in Figure 3. We observe that all three eviction policies perform equally well at all measured cache sizes.

1) Insensitivity to Parameters of Eviction Policy: Our eviction policy takes two parameters: maximum count and number of categories. Maximum count is the maximum count of the number of reads of an object stored per time category in the approximate histogram. Any reads higher than that are not counted till the histogram is reset. The number of categories refers to the number of categories of reuse time.

The results are depicted in Figure 4. We observe that the parameters don't have any effect on the performance of the cache policy as long as they are reasonable. Too few categories, 1 for example, and the algorithm degenerates to LFU. Too many, 1,000 for example, and the policy starves for data. The reset interval is the number of reads after which the counters in the histogram are reset. This generally corresponds to the size of the working set. We use a value of 10,000. We find that a reasonable interval functions well. Too low, and the policy doesn't have enough time to gather information. Too high, and all the counters are saturated leading to loss of information.

C. Cache Policy Comparison

We compare eviction policies using two traces: Databricks and Yahoo. The policies with randomization elements were run ten times and the results were averaged. The standard deviation was less than 1%. The policies are first compared at large cache sizes, 5,000 to 625,000.

1) Databricks Trace: The results of simulating the Databricks trace are depicted in Figure 5. The only policy that performs significantly worse at all cache sizes is LFU. All policies which focus on temporal locality or have a component focused on it perform well. We conjecture that this is due to the peculiar usage pattern of big data workloads. A file is read many times in a short interval because it is being processed.

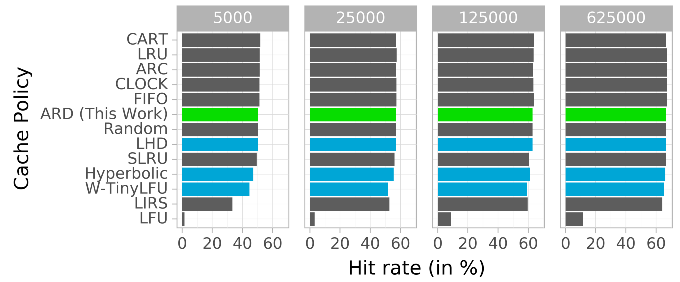

Fig. 5: Comparison using the Databricks trace. Each panel corresponds to a different cache size (number of objects). Policies published after 2010 are in blue.

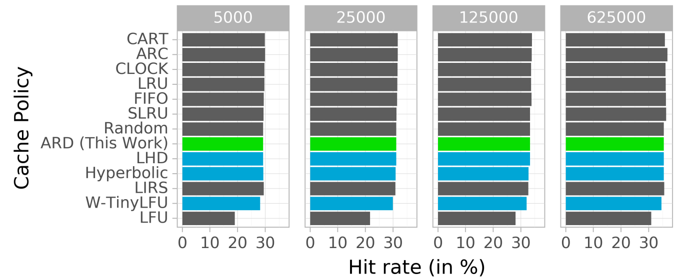

Fig. 6: Comparison using the Yahoo trace. Each panel corresponds to a different cache size (number of objects). Policies published after 2010 are in blue.

Then, it is not read again for long period of time till it is required again. So, unless an object remains in the cache for days, it is unlikely that algorithms based solely on popularity perform well.

2) Yahoo Trace: The results of simulating the Yahoo trace are depicted in Figure 6. The results are similar the Databricks trace. All policies which take temporal locality into account perform well, but the difference isn't significant. It seems that there is more opportunity for algorithms which take popularity into account to perform well here. This is evident from the success of ARC and CART. It is also evident by LFU perform better that it did with the Databricks trace.

We observe that the performance of ARD is similar to state-of-the-art work on caching policies, when applied to big data workloads. But, we do not see the gains we expect. Almost all cache policies perform similarly with recent policies (those published after 2010) performing slightly worse. We conjecture that this is due to the fact that new policies search for patterns to optimize themselves and are unable to find them. This might be due the extremely long length of the patterns. For example, some big data workloads repeat every day. In that period, the metadata collected about the pattern is already forgotten.

VII. Related Work

We report here the results of a systematic survey [15] of 18 high quality venues focusing on large-scale systems (HPDC, SC, etc.), systems and operating systems (NSDI, OSDI, etc.), and performance (SIGMETRICS, ICPE, etc.), starting in 2010.

TABLE II: Workloads used in cache-policy evaluations.

	Workloads
Huang 2013 [19]	Social Media
Berger 2017 [20]	Web
Blankstein 2017 [7]	Web, Database, Workstation
Einziger 2017 [13]	Web, Database, Workstation
Beckmann 2018 [8]	Web, Database, Workstation
This Work	**Big Data**

We also compare with prior work which has been highly cited by publications studied in our systematic survey.

A. *Cache Policy Design*

Caching is a well studied problem with a long history. We trace the lineage of the concepts used in this work. We also compare our work to recent studies on cache policy design.

LIRS [5] is one of the first algorithms to use reuse-distance as a metric. Hyperbolic caching [7] also divides popularity by time. LRU-K [16] remembers the K most recent reads of each object. The concept of using multiple approximate data structures in storage related scenarios has been introduced in counter stacks [17]. Databases have recently started to use multiple approximate data structures to keep track of histograms [18].

W-TinyLFU [13] and LHD [8] are the most closely related works to this one. W-TinyLFU uses approximate counters. Our eviction policy, also based on approximate counters, takes into account reuse time and is thus novel. LHD takes into account reads at different reuse times. For each reuse time category, object are further categorized based on features such as application name and database column. In many caching applications, such features are not available. Our work uses approximate counters instead of categories, which sidesteps the decision of choosing good categories and thus has less parameters to tune.

B. *Cache Policy Evaluation*

No independent benchmarking activity for caching policies exists for big data workloads. Instead, recent evaluations of cache policies are paired with either a new policy design or the new characterization of a workload. We summarize the type of workloads used in such recent studies in Table II. Most evaluations use workloads related to caching for: (1) servers like nginx or application caches like memcached ("Web" in the table), (2) SQL databases used in web applications ("Database"), and (3) regular files of users ("Workstation"). This is the first study to evaluate the efficacy of cache policies for big data workloads.

VIII. Conclusion

The use of caching for big data workloads is both of great importance and understudied. Designs of caching policies are either very general, or still assume the availability of fine-grained, application-specific information. We propose in this work to use read density for the principled design of an entire family of caching policies. We further add to this design a data structure, the Approximate Histogram, to keep track of the behavior of objects over time, efficiently.

We further design and conduct comprehensive experiments with the family of caching policies we have proposed, which we contrast to over 10 caching policies. Our experiments use simulation based on two real traces collected from commercial big data environments that use the widely popular big data processing systems Spark and MapReduce, respectively. We find that our policy is competitive while needed fewer parameters than the state-of-the-art, but also that most policies perform approximately the same when using hit rate as the metric. The latter indicates obtaining meaningful performance improvements when caching for big data workloads remains elusive.

Acknowledgments

Work supported by the project Vidi MagnaData. We thank the AtLarge Research team, Databricks Amsterdam, and Yahoo; Laurens Versluis for proofreading.

References

[1] European Commission, "Big Data and data analytics," EU Parliament, Sep 2016.

[2] Gartner Inc., "Infrastructure and Operations (I&O) Leadership Vision for 2017, section CIO Technology Priorities," Tech.Rep., 2017.

[3] D. Reinsel *et al.*, "Data age 2025. the digitization of the world from edge to core," IDC White Paper US44413318., Nov 2018.

[4] Y. Guo *et al.*, "Modeling, analysis, and experimental comparison of streaming graph-partitioning policies," *JPDC*, vol. 108, 2017.

[5] S. Jiang *et al.*, "LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance," in *SIGMETRICS 2002*.

[6] N. Megiddo *et al.*, "ARC: A self-tuning, low overhead replacement cache," in *FAST 2003*.

[7] A. Blankstein *et al.*, "Hyperbolic caching: Flexible caching for web applications," in *USENIX AT 2017*.

[8] N. Beckmann *et al.*, "LHD: improving cache hit rate by maximizing hit density," in *NSDI 2018*.

[9] S. Talluri *et al.*, "Characterization of a big data storage workload in the cloud," in *ICPE*, 2019, (in print).

[10] G. Cormode *et al.*, "An improved data stream summary: the count-min sketch and its applications," *J. Algorithms*, vol. 55, no. 1, 2005.

[11] B. Fan *et al.*, "Cuckoo filter: Practically better than bloom," in *CoNEXT 2014*.

[12] P. Pandey *et al.*, "A general-purpose counting filter: Making every bit count," in *SIGMOD 2017*.

[13] G. Einziger *et al.*, "Tinylfu: A highly efficient cache admission policy," *TOS*, vol. 13, no. 4, 2017.

[14] S. Bansal *et al.*, "CAR: clock with adaptive replacement," in *FAST 2004*.

[15] B. Kitchenham *et al.*, "Guidelines for performing systematic literature reviews in software engineering," EBSE Technical Report EBSE-2007-01, 2007, updated, version 2.3.

[16] E. J. O'Neil *et al.*, "The LRU-K page replacement algorithm for database disk buffering," in *SIGMOD 1993*.

[17] J. Wires *et al.*, "Characterizing storage workloads with counter stacks," in *OSDI 2014*.

[18] E. Gan *et al.*, "Moment-based quantile sketches for efficient high cardinality aggregation queries," *PVLDB*, vol. 11, no. 11, 2018.

[19] Q. Huang *et al.*, "An analysis of facebook photo caching," in *SOSP 2013*.

[20] D. S. Berger *et al.*, "Adaptsize: Orchestrating the hot object memory cache in a content delivery network," in *NSDI 2017*.