An Adaptive Eviction Framework for Anti-caching Based In-Memory Databases

Kaixin Huang, Shengan Zheng, Yanyan Shen(✉), Yanmin Zhu, and Linpeng Huang

Shanghai Jiao Tong University, Shanghai, China
{kaixinhuang,venero1209,shenyy,yzhu,lphuang}@sjtu.edu.cn

Abstract. Current in-memory DBMSs suffer from the performance bottleneck when data cannot fit in memory. To solve such a problem, anti-caching system is proposed and with proper configuration, it can achieve better performance than state-of-the-art counterpart. However, in current anti-caching eviction procedure, all the eviction parameters are fixed while real workloads keep changing from time to time. Therefore, the performance of anti-caching system can hardly stay in the best state. We propose an adaptive eviction framework for anti-caching system and implement four tuning techniques to automatically tune the eviction parameters. In particular, we design a novel tuning technique called window-size adaption specialized for anti-caching system and embed it into the adaptive eviction framework. The experimental results show that with adaptive eviction, anti-caching based database system can outperform the traditional prototype by 1.2x–1.8x and 1.7x–4.5x under TPC-C benchmark and YCSB benchmark, respectively.

Keywords: In-memory database · Anti-caching · Database tuning

1 Introduction

In-memory DBMSs remove heavy components such as data buffers and locks, thus providing higher OLTP throughput than traditional disk-oriented DBMSs [1,24]. However, a fundamental problem of in-memory DBMSs is that the improved performance is only achievable when database is smaller than the amount of available physical memory. If database grows larger than main memory while executing transactions, operating system will start to page virtual memory, and accesses to main memory will cause page faults; the performance of in-memory DBMSs may suffer a rapid decrease. One widely adopted method to enhance performance is to apply a main memory distributed cache, such as Memcached [23], in front of a disk-based DBMS. Such implementations with a two-tier model, however, come with a problem of double data buffering, causing a serious waste of memory resources.

As a better solution, anti-caching system [4] is proposed. In an anti-caching based in-memory database, when memory is exhausted, the DBMS gathers the

J. Pei et al. (Eds.): DASFAA 2018, LNCS 10828, pp. 247–263, 2018.
https://doi.org/10.1007/978-3-319-91458-9_15

"coldest" tuples and writes them to disk with minimal translation from their main memory format, thereby freeing up space for more recently accessed tuples. As such, the "hotter" data resides in main memory, while the colder data resides on disk in the anti-cache portion. Unlike a traditional DBMS architecture, each tuple is in either memory or a disk block, but never in both places at the same moment. Anti-caching system is not bound with any specified database systems, it is a design architecture which can be applied to any in-memory DBMSs aimed at dealing with OLTPs [3].

However, the configuration for anti-caching system prototype is fixed. Eviction parameters such as eviction size, eviction threshold and eviction check interval, are all preset by DBMS using a configuration file. For different types of workloads, an in-memory DBMS with anti-caching cannot run at its best performance with these fixed eviction parameters. Furthermore, anti-caching system with a fixed eviction configuration cannot always work at a high performance level when transaction workloads change from time to time.

One natural method is to manually modify the anti-caching configuration file each time a new workload comes. For example, DBMSs such as Oracle and MySQL are equipped with tuning manuals for DBAs and users. But for performance tuning, manual method comes with obstacles such as inflexibility, inefficiency and time consumption.

The drawbacks of both fixed anti-caching configuration and labored manual tuning motivate us to develop an adaptive tuning design, which is able to support evicting data dynamically with respect to temporary workload. Instead of tuning components like buffer pool in traditional disk-oriented DBMSs, we tune eviction-related parameters for anti-caching in a main memory DBMS. The whole procedure is automatic and we name it adaptive eviction framework in this paper.

We modify a few existing tuning techniques in order to embed them into the adaptive eviction framework. However, it is hard for these methods to exert their full potential because none of them is anti-caching oriented.

To overcome such a challenge, we propose a novel tuning technique called window-adaption specialized for anti-caching system. By shrinking and extending the window size related to anti-caching eviction parameters according to the workloads and system information, unsuitable eviction parameters will be adaptively tuned to a more proper set.

Our contributions can be concluded as follows.

- We make an observation into the relationship between different workloads and anti-caching parameter configurations; we find that fixed eviction parameter configuration limits the potential of anti-caching system.
- To the best of our knowledge, we are the first to propose an adaptive eviction framework for anti-caching based in-memory databases.
- We implement a variety of tuning methods aimed for in-memory anti-caching system based on previous work and propose a novel tuning technique called window-size adaption with high efficiency.

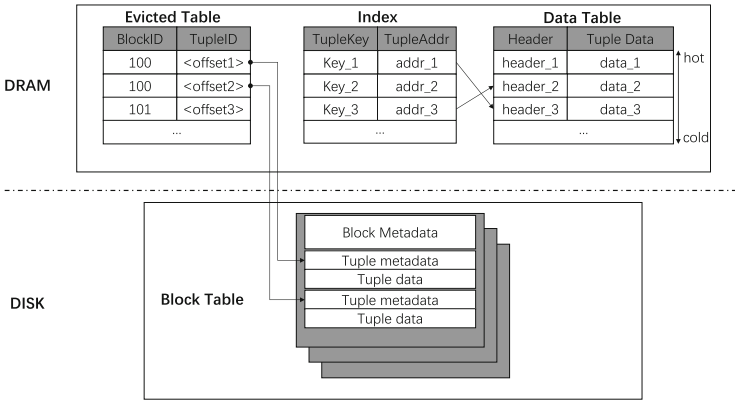


Fig. 1. Anti-caching architecture

- We conduct extensive experiments with standard TPC-C and YCSB benchmarks. The experimental results show that with adaptive eviction framework, anti-caching based database system can outperform the previous one by 1.2x–1.8x and 1.7x–4.5x under TPC-C benchmark and YCSB benchmark, respectively.

The remainder of this paper is organized as follows. Section 2 introduces the background of anti-caching system and the motivation of our proposed adaptive eviction framework. Section 3 provides the design of adaptive eviction framework and introduces four tuning techniques for adaptive eviction. The experimental results are presented in Sect. 4 and we review the related work in Sect. 5. The paper is concluded in Sect. 6.

2 Background and Motivation

2.1 Anti-caching Background

Anti-caching system aims at alleviating the pressure of in-memory DBMS while the data grows larger than memory space with transaction processing. The basic idea is to manage data in tuple granularity instead of page granularity. Since a data tuple is always much smaller than a page size (usually 4 KB), tuple-grained eviction can avoid evicting hot tuples to disk when a page includes both hot and cold tuples but considered to be a cold page by OS.

Figure 1 shows the storage architecture of anti-caching system, which includes four main components: Data Table, Index Table, Evicted Table and Block Table. Among them, Data Table, Index Table and Evicted Table reside in DRAM while Block Table is in disk.

Similar to other in-memory DBMSs, in an anti-caching based database system, the whole database stays in the memory when transactions begin. All the data tuples reside in Data Table. The Index Table is built by sampling the tuple

accesses. Notice that Data Table stores data tuples in a LRU linked list for each database table, with the top being hotter and the bottom being colder. Along with transactions processing, both Index Table and Tuple Data may grow larger and the DRAM space can be very limited. Therefore, an eviction decision should be made by anti-caching system to free up memory for hot data. Once the eviction check interval is reached, anti-caching system scans all the Data Tables and decides how much data should be evicted for each table.

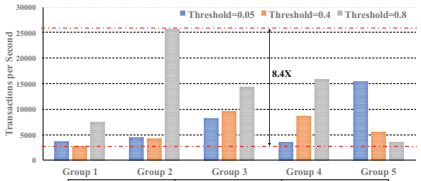
Anti-caching system globally tracks hot and cold data. However, the cost of maintaining a single chain across partitions is prohibitively expensive due to the added overhead of inter-partition communication. Instead, anti-caching maintains a separate LRU Chain per table that is local to a partition. Therefore, in order to evict data, anti-caching system must determine (1) from which tables to evict data and (2) the amount of data that should be evicted from a given table. In its current implementation, the DBMS answers these questions by the relative skew of accesses to tables. The amount of data accessed at each table is monitored, and the amount of data evicted from each table is inversely proportional to the amount of data accessed in the table since last eviction. Therefore, the hotter a table is, the less data will be evicted from it.

Block Table is a disk-resident hash table that stores evicted tuples in block format. Since Block Table stores relatively colder data, it's less accessed than the memory-resident Data Table. Data tuples stay in either DRAM or disk, thus the memory space occupation and coherence maintaining overhead is efficiently reduced. To track the tuples in Block Table, Evicted Table is designed to map evicted tuples to block ids. When a transaction requires data not in Data Table, the database system then looks up the Evicted Table, obtains its corresponding block id, and finally locates the binding block address for data access.

Main memory DBMSs, like H-Store [2], owe their performance advantage to processing algorithms which assume that data is in main memory. But any system will slow down if a disk read must be processed in the middle of a transaction. Anti-caching avoids stalling transaction execution at a partition whenever a transaction accesses an evicted tuple by applying a pre-pass process [4].

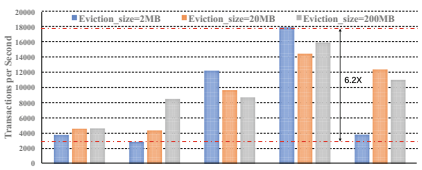
2.2 Motivation

With proper configuration, anti-caching system can behave much better than traditional disk-oriented DBMSs with large-than-memory database, due to its fine-grained eviction data control. However, the eviction parameters of anti-caching system prototype is fixed, thus unsuitable for multiple workloads or changing workload. The key parameters of anti-caching system fall on three: eviction threshold, eviction size and eviction check interval. Their functions are: deciding when to evict, how much to evict and how often should the system check if eviction is needed, respectively. In this section we give a few observations of the fixed configuration performance for anti-caching system and then analyze some typical cases to give insight about the drawbacks of fixed eviction parameter configuration.



	Check interval (ms)	Eviction Size (MB)
Group 1	100	2
Group 2	100	20
Group 3	1000	20
Group 4	1000	200
Group 5	10000	200

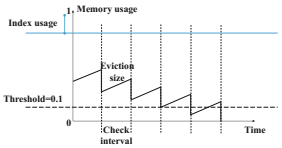
(a) Observation 1



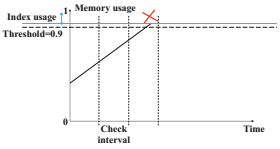
	Check interval (ms)	Threshold
Group 1	100	0.05
Group 2	100	0.4
Group 3	1000	0.4
Group 4	1000	0.8
Group 5	10000	0.8

(b) Observation 2

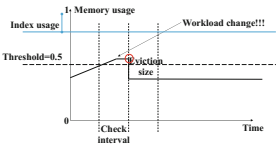
Fig. 2. Performance range with different eviction parameter groups



(a) low threshold setup



(b) high threshold setup



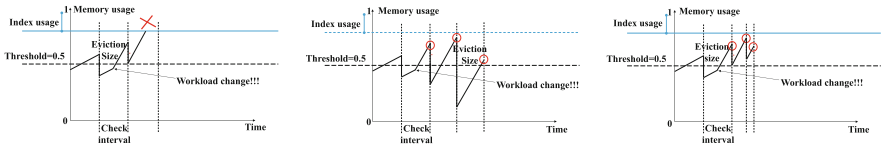
(c) fixed threshold setup

Fig. 3. Anti-caching with different types of threshold setup

Observations: We conduct a variety of experiments using the same type of YCSB benchmark to observe the limitations of fixed eviction configuration. Figure 2 shows that under different anti-caching configuration for three eviction parameters listed above, the performance of anti-caching system distinguishes from each other a lot. We can see that the best versus the worst ratio on transaction performance is 8.4x, 6.2x for each. We notice that the best performance group in Fig. 2(a) is <Check_interval = 100 ms, Eviction_size = 20 MB, Threshold = 0.8>. However, for other groups with Check_interval = 100 ms, Eviction_size = 20 MB, their performance seems to be very poor. It might be considered that the threshold makes sense. The <Check_interval = 10000 ms, Eviction_size = 200 MB, Threshold = 0.8> group, however, just gives the extreme reverse result. Therefore, it's hard to find the optimized eviction parameter group pattern for a certain workload. The deeper reason is that the performance of anti-caching system is not bound with a certain parameter group, but tightly related to workload state (e.g., skew, read/write ratio) and system state (e.g., memory usage). **In conclusion, we argue that fixed eviction parameter configuration is improper for anti-caching system.**

Case Analysis: In this part we offer some cases to further explain why fixed eviction parameter configuration is improper for anti-caching system. To make the explanation simpler, we assume that the index size is stable (even though it can grow larger and larger under real workloads).

Figure 3 shows the case how different types of eviction threshold configuration can affect the anti-caching processing. Figure 3(a) illustrates that when low



(a) fixed eviction size and check interval (b) dynamic eviction size (c) dynamic check interval

Fig. 4. Anti-caching with different types of eviction size and interval setup

threshold is set up by a DBA or system maintainer, the memory utilization rate would be very low. Even if the user workload is slow and steady without too much data expansion in memory, eviction process will still be invoked frequently, naturally deriving the conclusion that such a fixed low-threshold configuration is not optimized. Figure 3(b) shows the case when an improper high threshold is set up. It might be considered that high threshold can better utilize the available memory. However, once the total database memory is not enough, page swap will happen automatically. Since page swap is transparent to user transactions, when the anti-caching engine checks whether all the data tuples needed for a specific transaction, it may consider those tuples in pages which are swapped before still reside in memory. However, such accesses need disk I/O indeed, thus resulting in much longer latency. If a high threshold is set up, then fewer evictions will occur with the price of page swap during a transaction execution. We argue that such a fixed high-threshold configuration is also not proper. Figure 3(c) shows how a fixed threshold fails to utilize available memory when meeting with workload changes, say, from write-heavy to read-heavy. The main difference between these two workloads in anti-caching environment is that the latter workload results in much fewer data appended into memory, thus only a little more memory is used in addition. In such cases, the available memory can be abundant. However, the anti-caching system just performs conservatively with regular evictions. We argue that the fixed eviction threshold configuration is not optimized.

Figure 4 shows how eviction size and check interval can affect the transaction performance. Figure 4 shows the case when fixed eviction size and check interval are set up. For example, if a normal read-heavy workload turns into a write-heavy workload, then the data in memory will increase very quickly. Page swap might happen during a transaction processing and the database performance can suffer a rapid descending. To better illustrate the inefficiency of fixed configuration for eviction size and check interval, we compare it with the cases that either eviction size or check interval changes along with the workload change. These two cases are shown in Fig. 4(b) and (c), respectively. With workload-aware adjustment for eviction size and check interval, page swap can be avoided to some extent, thus maintaining a high transaction performance. Therefore, we argue that both fixed eviction size and check interval are not optimized.

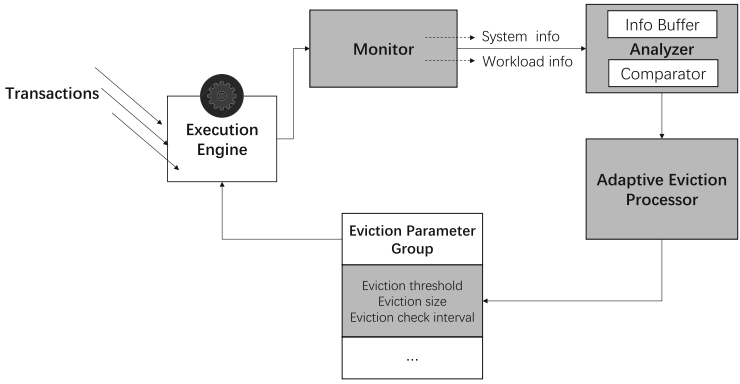


Fig. 5. Adaptive eviction framework

3 Adaptive Eviction

3.1 Overview

Figure 5 presents the framework of adaptive eviction. The three important components are Monitor, Analyzer and Adaptive Eviction Processor (AEP). The Monitor is responsible for gathering information about transaction workload and system state. Workload information includes read/write times, query numbers in a certain time period, accessed tuple numbers; system information includes real-time transaction performance and memory usage. The raw information collected by Monitor is conveyed to Analyzer for further computation. For example, the read/write ratio of the workload is computed by $rw\_ratio = read\_number/total\_access\_number$; the access skew is computed by $access\_skew = total\_access\_number/access\_tuple\_number$. The Info Buffer of Analyzer is used to store the historic workload and system information while the Comparator is used to compare its current results with the history, which implies whether there is a change of workload characteristics or system state. The result computed by Analyzer is then transferred to AEP. AEP is the core tuning model of adaptive eviction framework. We can implement different tuning techniques in AEP to modify the eviction parameter group for anti-caching based database systems.

Algorithm 1 further introduces the work flow of adaptive eviction framework. First, necessary components go through either startup or initiation. During transactions processing, the Monitor keep tracking the information of both system and workload, and then transfer them to Analyzer. Once the eviction check interval is reached, Analyzer deals with the workload/system information data received during the check interval period. It counts the access skew, read/write ratio and transaction rate. Then it makes a comparison of these data with corresponding ones computed last time to tell whether eviction parameters should be reconfigured or not. In our current implemented version of adaptive eviction framework, when either one of the following two conditions are satisfied, a

reconfiguration action should be performed: (1) available memory space change is over 10%; (2) workload state change is over 10% (e.g., 10% write-heavier, 10% more skew access). The choice of our threshold 10% is based on a few experimental observations, which show such a configuration outperforms than most other ones. We expect to explore more critical knowledge about the choice of threshold representing changes in our future work. AEP is the core component which performs actual tuning procedure. It decides the concrete method of how eviction parameters change. We implement four tuning techniques as AEP to tune the eviction parameters, which are presented in next part of this section. Notice that even if the eviction parameters have been changed, the data eviction do not need to happen if only the memory occupation of data is smaller than the eviction threshold.

Algorithm 1. *AdaptiveAntiCaching*

```
1  P ← default_eviction_parameters
2  while transaction_state is ON do
3      Analyzer ← Monitor.getInfo()
4      if Monitor.timer.last() == P.check_interval then
5          Analyzer.compute()
6          Analyzer.compare()
7          AEP ← Analyzer.changeInfo()
8          if AEP.shouldChangeConfig() is True then
9              AEP.tune(P)
10         if data_memory > P.threshold then
11             Evict(database, P);
12     Monitor.timer.continue()
```

3.2 Tuning Techniques

We design our adaptive eviction framework as a pluggable platform for equipping various tuning techniques. Four tuning methods are implemented in our study. They are simple-rule based tuning (SRB), experiment-reflected tuning (ER), candidate block replacement tuning (CBR) and window-size adaption tuning (WSA). Among them, the first three refer to previous research [12–14] and WSA is an efficient tuning technique we propose for better adapting to anti-caching system. Next we will introduce each of the tuning technique with more details.

SRB: Simple Rule-Based. Pavlo et al. [14] introduce machine learning into in-memory database to tune the indexes, views, storage layout and etc. We adopt the simpler off-line machine learning version to obtain the patterns with certain <workload info, system info, eviction parameter set> format. It is fast to make a tuning decision and can often work better than the default configuration of anti-caching. However, it is not accurate indeed and can suffer serious performance degrading in a few cases.

Table 1. Design issues for different tuning techniques

	SRB	ER	CBR	WSA
In-memory DBMS oriented	Yes	No	No	Yes
Anti-caching oriented	No	No	No	Yes
Tuning time	Short	Long	Middle	Short
Memory overhead	Small	Neutral	Large	Small
Performance improvement	Low	Middle	Middle	High

ER: Experiment-Reflected. Duan et al. [13] proposes this feedback-driven tuning method based on adaptive sampling. Adaptive sampling analyzes the samples collected so far to understand how the surface looks like (approximately), and where the good settings are likely to be. Based on this analysis, more experiments are done to collect new samples that add maximum utility to the current samples.Although this technique is able to be quite adjacent to the optimized eviction parameter values for single-type workload, it is time-consuming and suffer from serious performance bottleneck under changing workload.

CBR: Candidate Block Replacement. Storm et al. [12] describes such a self-tuning approach based on cost-benefit analysis. This method previously attempts to solve the problem of developing a database-wide, memory-tuning algorithm by considering each of the memory consumers such as compiled cache-pool, buffer-pool, sort-buffer, etc., wherein each has a different usage. It accumulates the cost savings in processing time for each component with which a database process is interacting with the memory subcomponent. This technique is also called shared memory management technique (SMMT) and has been incorporated in IBM's DB2. It behaves in a block-grained memory replacement, thus losing the flexibility of tuple-grained data control.

WSA: Window-Size Adaption. The idea of window-size adaption comes from the TCP flow control mechanism in network communication. Figure 6 shows the implementation of WSA. The aim of WSA is to balance the trade-off between utilizing memory space and avoiding page swap. If more data can reside in the memory, access to disk will become less, thus promoting the overall transaction performance and decreasing the average delay. However, if the system greedily holds too much data tuples in DRAM, the sum of space used by indexes, data tuples, buffers and evicted tables may exceed the available memory space, thus causing heavy OS page swap. To explore the full potential of anti-caching, we design two phases for WSA. One is Relaxation Phase, during which transaction burden is not heavy and memory space is sufficient, anti-caching system tries to hold more data tuples in the memory and reduce the tuning overhead (i.e., decrease eviction size, increase eviction threshold and check interval); the increasing/decreasing ratio is chosen to be 0.1 in our current implementation version. The other is Shrinking Phase, where transaction burden is detected to be heavy or memory resource is inadequate, anti-caching system makes a radical change

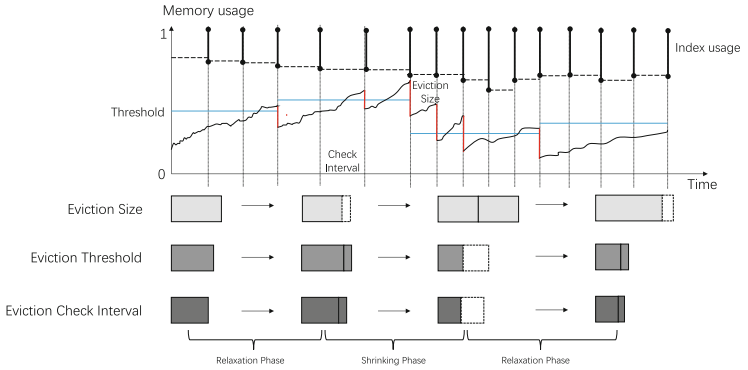


Fig. 6. Window-size adaption

for its eviction parameters (i.e., double eviction size, halve eviction threshold and check interval). Notice that for most workloads the Relaxation Phase can improve transaction behavior with optimizing the eviction parameters gradually and this process is slow and steady. As for Shrinking Phase, it changes rapidly to avoid the danger of OS page swap by sacrificing the memory usage and adding more frequent checking overhead. However, compared to access tuples in the method of OS page swap, it is more reasonable to evict data in advance to save memory space. This is because anti-caching itself allow asynchronously fetching disk-resident tuples while executing next transactions which only concern data in memory. By taking advantage of this intrinsic feature of anti-caching system, WSA becomes a well-suited tuning method in our study.

Table 1 makes a conclusion of four tuning techniques used in our adaptive eviction framework. Compared the other three tuning techniques, WSA is anti-caching oriented, thus it can obtain considerable performance improvement with small overhead and short tuning time. SRB is also a fast-tuning technique with small memory occupation. However, it is unaware of the dynamic workload/system changing, which limits its ability. As for ER and CBR, they are previously designed for disk-oriented DBMSs to optimize buffer resources, thus cannot fully exert their potential for anti-caching based database systems.

4 Experiments

We implement our adaptive eviction framework in H-Store and compare its performance against traditional anti-caching system. Four kinds of adaptive eviction techniques introduced in Sect. 3.2 are tested in our experiments. We first describe the two benchmarks and the DBMS configurations used in our analysis.

4.1 Benchmarks

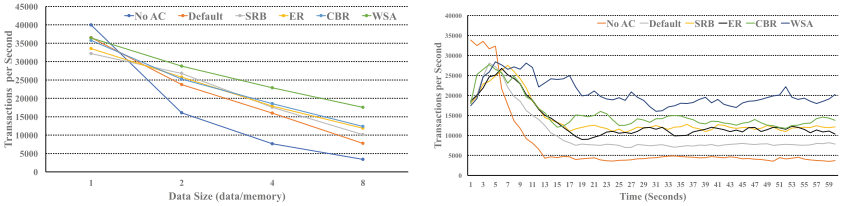
TPC-C: This benchmark is the current industry standard for evaluating the performance of OLTP systems. It consists of nine tables and five procedures

that simulate a warehouse-centric order processing application. Only two of these procedures modify or insert tuples in the database, but they make up 88% of the benchmark's workload. For our experiments, we use a 10 GB TPC-C database containing 100 warehouses and 100,000 items. For this benchmark, we set the available memory to the system to 12 GB. As the benchmark progresses and more orders accumulate, the data size will continue to grow, eventually exhausting available memory, at which point the anti-caching system will begin evicting cold data from the data tables to disk.

YCSB: The Yahoo! Cloud Serving Benchmark is a collection of workloads that are representative of large-scale services created by Internet-based companies. For all of the YCSB experiments in this paper, we use a 20 GB YCSB database containing a single table with 20 million records. Each YCSB tuple has 10 columns, each with 100 bytes of randomly generated string data. The workload consists of two types of transactions; one that reads a single record and one that updates a single record. We use the write-heavy transaction workload mixtures (i.e., 50% reads/50% updates). We also vary the amount of skew in workloads to control how often a tuple is accessed by transactions. In our experiments, we use a Zipfian skew with values of s between 0.5 and 1.5.

4.2 System Setup

We deploy latest H-Store with our adaptive eviction framework on a single node with a dual socket Intel Xeon E5-2620 CPU (12 cores per socket, 15M Cache, 2.00 GHz) processor running 64-bit Ubuntu Linux 14.04. All transactions are executed with a serializable isolation level. The benchmark clients in each experiment are deployed on a separate node in the same cluster. In each trial except one that tests the connection between performance change and running time, H-Store is allowed to "warm-up" for two minutes. During the warm-up phase, transactions are executed as normal but throughput is not recorded in the final benchmark results. For H-Store, cold data is evicted to the anti-cache and hot data is brought into memory. After the warm-up, each benchmark is run for a duration of ten minutes, during which average throughput is recorded. The final throughput is the number of transactions completed in a trial run divided by the total time (excluding the warm-up period). Each benchmark is run five times and the throughputs from these runs are averaged for a final result. To properly control the data size for experimental presentation, we write extra codes for benchmark testing. Once the goal (e.g., 2x memory size) is achieved for one experiment, new data will not be generated throughout this experiment. For each trial we test the performance of six database configurations: one is pure DBMS without anti-caching system (i.e., No AC), another is anti-caching system without eviction parameter configuration as baseline (i.e., Default), the other four are anti-caching systems equipped with our proposed adaptive eviction framework, which includes four different tuning techniques (i.e., SRB, ER, CBR, WSA). We don't perform manually-tuned experiments because such a method fails to access the optimal performance, due to the random changing patterns and slow human reactions to the OLTP workloads.



(a) Performance for different data sizes (b) Real-time performance change

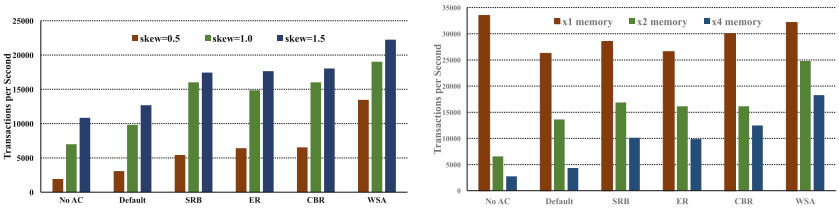
Fig. 7. Adaptive eviction performance under TPC-C benchmark

4.3 Results and Analysis

We now discuss the results of executing two benchmarks with our adaptive eviction framework across different size configurations and workload skews.

TPC-C: The results for running the TPC-C benchmark are shown in Fig. 7. We first examine the performance of different tuning methods with the change of data size, whose result is depicted in Fig. 7(a). It can be simply observed that with data size being larger and larger, the performance of DBMS degrades again and again. This is because more data tuples are evicted to disk thus more disk I/Os should be taken. When the data size can just fit into memory (data/memory = 1), DBMS with no anti-caching behaves the best because the overhead of eviction process is removed. However, when data size become larger, the performance of DBMS without anti-caching decrease rapidly. It can be concluded that using tuning methods with our adaptive eviction framework can obtain higher average throughputs than anti-caching system with default eviction parameter configuration. Among the four tuning techniques, window-size adaption is the best and it beats default anti-caching system in transaction throughput with 21%, 52% and 118% under 2x, 4x and 8x memory data size, respectively. Compared with other tuning techniques, WSA wins 43% to 75% when the data size is 8x memory. WSA can efficiently balance the trade-off between utilizing memory space and avoiding page swap. Therefore, it is more proper than the other implemented tuning methods for anti-caching system.

Figure 7(b) shows the performance change of each tuning technique along with time. Notice that we just choose the first 60 s to give explanations for better visual effect. The time-sequential throughput change can tell more detailed information about what happens for each tuning technique while transactions are being executed. When transactions just begin, the memory space is sufficient and DBMS without anti-caching works at its best state with fetching all the target tuples from memory. Since no eviction threshold is limited for it, the performance will not be worse until OS page swap happens. However, once the data size start to exceed the available memory, page swap can occur frequently for DBMS without anti-caching and the performance goes down rapidly and finally reaches a stable state. Other anti-caching choices, start with relatively lower transaction throughputs, grow steadily until the same bottleneck occur



(a) Performance for different skews (b) Performance for different data sizes

Fig. 8. Adaptive eviction performance under YCSB benchmark

for them: the data size become too large and only a small fraction of them can reside in memory. The interesting thing is that compared to the baseline, all tuning techniques have a jump-after-fall phenomenon. For example, WSA experiences a performance fall from 10 s to 12 s and just after this, it has an obvious performance jump to 17 s. Another instance is for the CBR tuning method. Its performance falls from 8 s to 15 s and jumps from 15 s to 22 s. The reason for the fall-and-jump style is simple: when performance degrading is captured by Analyzer in our adaptive eviction framework, a tuning decision should be made by AEP, configuring the eviction parameters to a relatively proper set. After such a procedure, anti-caching system fits more to the current workload than before and obtains a performance growth, which we regard as a jump action. Among the four tuning techniques, WSA's stable state is the highest and it gains 1.25x better performance than the baseline.

YCSB: The results in Fig. 8 are for running the YCSB benchmark with write-heavy workload across a range of workload skews and data sizes. Figure 8(a) shows the performance difference of each tuning technique with different workload skews. We can observe that under the workload of same skew, anti-caching system can obtain higher average transaction throughput by taking advantage of our proposed adaptive eviction framework. With the skew becoming higher, more transaction accesses reach the same group of hot data in memory, DBMS can obtain a natural throughput improvement. However, in the case of low skew, anti-caching system with default eviction parameter configuration behaves poorly and is only 52% better than no anti-caching choice. While using adaptive eviction, the performance can obtain considerable improvement, up to 1.7x–4.5x compared with the baseline. In particular, under low-skew workload, WSA also beats other tuning techniques by 2.6x, 2.2x and 2.3x compared with SRB, ER and CBR, respectively. The reason is that when transaction accesses randomly fall into data tuples, WSA gently changes its window size to fit more cold data in memory while promising that the memory resource is not over-used. It keeps more access in memory rather than evicts a large amount to disk. In this way WSA fits anti-caching system better than other tuning techniques.

Figure 8(b) presents the performance behaviors of different tuning techniques for three data size setups. The performance difference among all the anti-caching choices in the 1x memory data size scenario is slight, but it can still be clearly

observed that WSA obtains an average throughput nearly to pure DBMS without anti-caching system. We can infer that with the transactions proceeding, different windows of WSA behave in the following style: the eviction size becomes smaller and smaller, while the eviction threshold and check interval becomes larger and larger. In larger-than-memory data scenarios, adaptive eviction obtains more obvious improvement than the baseline, up to 1.2x–1.8x for 2x memory and 2.3x–4.6x for 4x memory, respectively. The interesting thing is that with much larger data size, the improvement of using adaptive eviction is also larger. This is reasonable because with adaptive eviction framework, anti-caching system is able to leave more hot data in memory with more proper eviction parameters.

5 Related Work

Anti-caching Data Management. [3] concludes different kinds of "anti-caching" data management mechanisms and divides them into three categories: user-space, kernel space, hybrid of user- and kernel-space. H-Store anti-caching [4] falls into the user-space approach. Project Siberia [6–8] also adopts a user-space "anti-caching" approach for Hekaton [5]. Instead of maintaining an LRU like H-Store anti-caching, Siberia performs offline classification of hot and cold data by logging tuple accesses first, and data in Hekaton is evicted to or fetched from a cold store in disk. Kernel-space approaches mainly refers to OS paging, which is an important part of virtual memory management in most contemporary general purpose operating systems. OS paging tracks data access in the granularity of pages and it allows a program to process more data than the physically available memory [9]. As for hybrid of user- and kernel-space approach, efficient OS paging [10] and access observer method [11] are proposed to better assist the hot/cold data classification. Our work is based on H-Store Anti-caching as it is the state-of-the-art in-memory DBMS data management architecture and deals with workload online to serve multiple application scenarios.

Performance Tuning. Performance tuning of database systems has been an interesting and active area of research in the last three decades and recent trend has been in developing self-tuning database systems with little or no human intervention. Several methods have been proposed in the literature [12,15,16] to implement self-tuning techniques ranging from use of histograms, gradient descent technique, creation of index, use of materialized views, etc. There have been several attempts to self-manage the DBMS memory [13,17–19] for improved performance. Oracle 10 g uses automated shared memory management (ASMM) to resize the subcomponents of the shared memory pool based on current workload. When switched on, the ASMM controls the sizes of certain components in the SGA by making sure that the workload gets the memory it needs. It does that by shrinking the components which are not using all of the memory allocated to them, and growing the ones which need more than the allocated memory. Microsoft SQL Server also has an automatic memory tuning manager.

However, none of these techniques can fully exert their potential for anti-caching architecture because of either disk-oriented design or heavy and redundant buffer components.

Workload Characterization. The key to successful implementation of a self-tuning database system is the knowledge base [21,22] on two aspects. One is helping the system to identify important performance bottlenecks. The other is collecting information about tuning impact of each tuning parameter on the system performance under different workloads and user load conditions. There have been attempts to identify key tuning parameters that have significant tuning impact on performance. [20] has presented the impact of various tuning parameters on the performance and the parameters are ranked using statistical approach. In our study we consider that the workload characteristics are bound with anti-caching memory environment, since the workload itself does not really affect the system performance.

6 Conclusion

In this paper, we propose an adaptive eviction framework for anti-caching based in-memory databases to figure out the problems caused by fixed eviction parameter configuration. With adaptive eviction for anti-caching system, in-memory DBMS is able to collect workload and system information to adaptively adjust the eviction parameters, taking more advantage of anti-caching system. We propose a novel window-size adaption strategy based on our designed general adaptive eviction framework; by extending and shrinking the eviction parameters along with the workload characteristics and system information change, an in-memory DBMS can smartly avoid weak memory utilization or slow OS page swap. The experimental results show that with adaptive eviction, an anti-caching based database system can obtain higher transaction performance under both TPC-C and YCSB benchmarks. In particular, window-size adaption tuning technique can outperform the base line up to 2.2x and 4.5x under TPC-C and YCSB benchmark, respectively. We conclude that for OLTP workloads, the results of this study demonstrate that adaptive eviction can efficiently improve the transaction performance of in-memory DBMS with anti-caching system.

Acknowledgment. This research is supported in part by 863 Program (no. 2015AA015303), NSFC (no. 61772341, 61472254, 61170238, 61602297 and 61472241), Singapore NRF (CREATE E2S2), and 973 Program (no. 2014CB340303). This work is also supported by the Program for Changjiang Young Scholars in University of China, and the Program for Shanghai Top Young Talents.

References

1. Harizopoulos, S., et al.: OLTP through the looking glass, and what we found there. In: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data. ACM (2008)
2. Kallman, R., et al.: H-store: a high-performance, distributed main memory transaction processing system. Proc. VLDB Endow. **1**(2), 1496–1499 (2008)
3. Zhang, H., et al.: Anti-caching based elastic memory management for big data. In: 2015 IEEE 31st International Conference on Data Engineering (ICDE). IEEE (2015)
4. DeBrabant, J., et al.: Anti-caching: a new approach to database management system architecture. Proc. VLDB Endow. **6**(14), 1942–1953 (2013)
5. Diaconu, C., et al.: Hekaton: SQL server's memory-optimized OLTP engine. In: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data. ACM (2013)
6. Eldawy, A., Levandoski, J., Larson, P.-Å.: Trekking through Siberia: managing cold data in a memory-optimized database. Proc. VLDB Endow. **7**(11), 931–942 (2014)
7. Levandoski, J.J., Larson, P.-Å., Stoica, R.: Identifying hot and cold data in main-memory databases. In: 2013 IEEE 29th International Conference on Data Engineering (ICDE). IEEE (2013)
8. Alexiou, K., Kossmann, D., Larson, P.-Å.: Adaptive range filters for cold data: avoiding trips to Siberia. Proc. VLDB Endow. **6**(14), 1714–1725 (2013)
9. Tanenbaum, A.S.: Modern Operating System. Pearson Education Inc., Upper Saddle River (2009)
10. Stoica, R., Ailamaki, A.: Enabling efficient OS paging for main-memory OLTP databases. In: Proceedings of the Ninth International Workshop on Data Management on New Hardware. ACM (2013)
11. Funke, F., Kemper, A., Neumann, T.: Compacting transactional data in hybrid OLTP&OLAP databases. Proc. VLDB Endow. **5**(11), 1424–1435 (2012)
12. Storm, A.J., et al.: Adaptive self-tuning memory in DB2. In: Proceedings of the 32nd International Conference on Very Large Data Bases. VLDB Endowment (2006)
13. Duan, S., Thummala, V., Babu, S.: Tuning database configuration parameters with iTuned. Proc. VLDB Endow. **2**(1), 1246–1257 (2009)
14. Pavlo, A., et al.: Self-driving database management systems. In: CIDR (2017)
15. Benoit, D.G.: Automatic diagnosis of performance problems in database management systems. In: Proceedings of the Second International Conference on Autonomic Computing, ICAC 2005. IEEE (2005)
16. Tran, D.N., et al.: A new approach to dynamic self-tuning of database buffers. ACM Trans. Storage (TOS) **4**(1), 3 (2008)
17. Chen, A.N.K.: Robust optimization for performance tuning of modern database systems. Eur. J. Oper. Res. **171**(2), 412–429 (2006)
18. Xu, J.: Rule-based automatic software performance diagnosis and improvement. Perform. Eval. **69**(11), 525–550 (2012)
19. Jeong, J., Dubois, M.: Cache replacement algorithms with nonuniform miss costs. IEEE Trans. Comput. **55**(4), 353–365 (2006)
20. Debnath, B.K., Lilja, D.J., Mokbel, M.F.: SARD: a statistical approach for ranking database tuning parameters. In: IEEE 24th International Conference on Data Engineering Workshop, ICDEW 2008. IEEE (2008)

21. Melcher, B., Mitchell, B.: Towards an autonomic framework: self-configuring network services and developing autonomic applications. Intel Technol. J. **8**(4), 279–290 (2004)
22. Wiese, D., Rabinovitch, G.: Knowledge management in autonomic database performance tuning. In: Fifth International Conference on Autonomic and Autonomous Systems, ICAS 2009. IEEE (2009)
23. Fitzpatrick, B.: Distributed caching with memcached. Linux J. **2004**(124), 5 (2004)
24. DeWitt, D.J., et al.: Implementation techniques for main memory database systems. **14**(2) (1984)