Identifying Hot and Cold Data in Main-Memory Databases

Justin J. Levandoski [1], Per-Åke Larson [2], Radu Stoica [3]

[1,2]*Microsoft Research*

[3]*École Polytechnique Fédérale de Lausanne*

[1]justin.levandoski@microsoft.com, [2]palarson@microsoft.com, [3]radu.stoica@epfl.ch

***Abstract*— Main memories are becoming sufficiently large that most OLTP databases can be stored entirely in main memory, but this may not be the best solution. OLTP workloads typically exhibit skewed access patterns where some records are hot (frequently accessed) but many records are cold (infrequently or never accessed). It is more economical to store the coldest records on secondary storage such as flash. As a first step towards managing cold data in databases optimized for main-memory we investigate how to efficiently identify hot and cold data. We propose to log record accesses – possibly only a sample to reduce overhead – and perform offline analysis to estimate record access frequencies. We present four estimation algorithms based on exponential smoothing and experimentally evaluate their efficiency and accuracy. We find that exponential smoothing produces very accurate estimates, leading to higher hit rates than the best caching techniques. Our most efficient algorithm is able to analyze a log of 1B accesses in sub-second time on a workstation-class machine.**

I. Introduction

Database systems have traditionally been designed under the assumption that data is disk resident and paged in and out of memory as needed. However, the drop in memory prices over the past 30 years is invalidating this assumption. Several database engines have emerged that optimize for the case when most data fits in memory [1], [2], [3], [4], [5], [6]. This architectural change necessitates a rethink of all layers of the system, from concurrency control [7] and access methods [8], [9] to query processing [10].

In OLTP workloads record accesses tend to be skewed. Some records are "hot" and accessed frequently (the working set), others are "cold" and accessed infrequently, while "lukewarm" records lie somewhere in between. Clearly, good performance depends on the hot records residing in memory. Cold records can be moved to cheaper external storage such as flash with little effect on overall system performance.

The work reported here arose from the need to manage cold data in an OLTP engine optimized for main-memory. The problem is to efficiently and accurately identify cold records that can be migrated to secondary storage. We propose to perform classification offline using logged record accesses, possibly using only a sample. Our best classification algorithm is very fast: it can accurately identify the hot records among 1M records in sub-second time from a log of 1B record accesses on a workstation-class machine. Because it is so fast, the algorithm can be run frequently, say, once per hour or more if the workload so warrants.

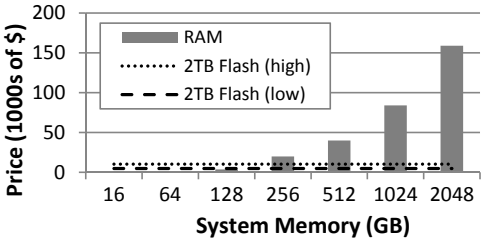


Fig. 1. Cost of DRAM for various memory sizes of Dell PowerEdge Blade vs. low-end Flash SSD (Intel DC S3700) and high-end SSD (Intel 910).

A. Motivation

Our work is motivated by the need to manage cold data in a main-memory optimized OLTP database engine, code named Hekaton, being developed at Microsoft (details in Section II-A). Three main considerations drive our research:

(1) *Skew in OLTP workloads.* Real-life transactional workloads typically exhibit considerable access skew. For example, package tracking workloads for companies such as UPS or FedEx exhibit *time-correlated skew*. Records for a new package are frequently updated until delivery, then used for analysis for some time, and after that accessed again only on rare occasions. Another example is the *natural skew* found on large e-commerce sites such as Amazon, where some items are much more popular than others. Such preferences may change over time but typically not very rapidly.

(2) *Economics.* It is significantly cheaper to store cold data on secondary storage rather than in DRAM. Figure 1 plots the current price of DRAM for various server memory configurations along with the price for a high and low-end 2TB flash SSD (plotted as constant). High-density server class DRAM comes at a large premium, making flash an attractive medium for cold data. In fact, using current prices Gray's n-minute rule [11] says that a 200 byte record should remain in memory if it is accessed at least every 60 minutes (Appendix E contains our derivation). "Cold" by any definition is longer than 60 minutes.

(3) *Overhead of caching.* Caching is a tried-and-true technique for identifying hot data [12], [13], [14], [15]. So why not simply use caching? The main reason is the high overhead of caching in a database system optimized for main memory. (a) *CPU overhead.* Main memory databases are designed for speed with very short critical paths and the overhead of maintaining the data structure needed for caching on every record access is high. We implemented a simple LRU queue

(doubly-linked list) in our prototype system and encountered a 25% overhead for updating the LRU queue on every record access (lookup in a hash index). The queue was not even thread-safe, so this is the *minimal* overhead possible. Better caching polices such as LRU-2 or ARC would impose an even higher overhead, possibly costing as much as the record access itself. (b) *Space overhead.* Hekaton, like several other main-memory systems [1], [5], does not use page-based storage structures for efficiency reasons; there are only records. Our goal is to identify cold data on a record basis, not a page basis. On a system storing many millions of records, reserving an extra 16 bytes per record for an LRU queue adds up to a significant memory overhead.

B. Our Contributions

We propose a technique for identifying hot and cold data, where hot records remain in memory while cold records are candidates for migration to secondary storage. We propose to sample record accesses during normal system runtime, and record the accesses on a consolidated log. A transaction copies its record access information into large (shared) buffers that are flushed asynchronously only when full; the transaction does not wait for log flushes. Sampling and logging accesses reduces overhead on the system's critical path. It also allows us to move classification to a separate machine (or CPU core) if necessary. Estimated record access frequencies are then computed from the logged accesses, and the records with the highest estimated frequency form the hot set.

The core of our technique is a set of novel algorithms for estimating access frequencies using exponential smoothing. We first explore a naive *forward* algorithm that scans the log from beginning to end (i.e., past to present) and calculates record access frequencies along the way. We then propose a *backward* algorithm that scans the log in reverse (i.e., present to past) and calculates upper and lower bounds for each record's access frequency estimate. Experiments show that the *backward* algorithm is both faster and more space efficient than the *forward* algorithm. We also show how to parallelize the *forward* and *backward* algorithms to speed up estimation dramatically. A recent paper described management of cold data in the HyPer main-memory database [16]; we provide a detailed comparison with this work in Section VI.

An experimental evaluation finds that our approach results in a higher hit rate than both LRU-k [14] and ARC [15] (two well-known caching techniques). The experiments also reveal that our algorithms are efficient, with the backward parallel algorithm reaching sub-second times to perform classification on a log of 1B record accesses. We also provide a mathematical analysis showing that exponential smoothing estimates access frequencies much more accurately than LRU-k.

The rest of this paper is organized as follows. Section II provides preliminary information and the overall architecture of our framework. Our classification algorithms are presented in Section III, while Section IV discusses how to parallelize them. Section V provides an experimental evaluation. Section VI provides a survey of related work. Finally, Section VII concludes this paper.

II. Preliminaries

Our problem is to efficiently identify the K hottest records, i.e., most frequently accessed, among a large set of records. The access frequency (heat) of each record is estimated from a sequence of record access observations. The K records with the highest estimated access frequency are classified as hot and stored in main memory, while the remaining records are kept on secondary "cold" storage. The value of K can be determined by a variety of metrics, e.g., working set size or available memory. Data moved to cold storage is still available to the database engine, albeit at a higher access cost. The more accurately access frequencies can be estimated, the higher the hit rate in main memory or, conversely, the fewer expensive trips to the cold store.

The rest of this section covers preliminary details. We begin by providing context for this work by discussing Hekaton, a memory-optimized OLTP engine being developed at Microsoft. We then outline Siberia, the cold data management framework we are prototyping for Hekaton. We then discuss logging as a technique for storing record access observations. Next we describe exponential smoothing, our technique for estimating record access frequencies. Finally, we cover sampling as a method to reduce system overhead for logging accesses.

A. The Hekaton Memory-Optimized OLTP Engine

Microsoft is developing a memory-optimized database engine, code named Hekaton, targeted for OLTP workloads. Hekaton will be described in more detail elsewhere so we provide only a brief summary of its characteristics here.

The Hekaton engine is integrated into SQL Server; it is not a separate database system. A user can declare a table to be memory-optimized which means that it will be stored in main memory and managed by Hekaton. Hekaton has a "record-centric" data organization, it does not organize records by page (and is oblivious to OS memory pages). A Hekaton table can have several indexes and two index types are available: hash indexes and ordered indexes. Records are always accessed via an index lookup or range scan. Hekaton tables are fully durable and transactional, though non-durable tables are also supported.

Hekaton tables can be queried and updated in the same way as regular tables. A query can reference both Hekaton tables and regular tables and a single transaction can update both types of tables. Furthermore, a T-SQL stored procedure that references only Hekaton tables can be compiled into native machine code. Using compiled stored procedures is by far the fastest way to query and modify data in Hekaton tables.

Hekaton is designed for high levels of concurrency but it does not rely on partitioning to achieve this; any thread can access any row in a table without acquiring latches or locks. The engine uses latch-free (lock-free) data structures to avoid physical interference among threads and a new optimistic, multi-version concurrency control technique to avoid interference among transactions [7].

B. Siberia: A Cold Data Management Framework

The initial release of Hekaton will require its tables to fit entirely in memory, but this will not be sufficient going forward. The goal of our project, called Project Siberia, is to enable Hekaton to automatically migrate cold records to cheaper secondary storage while still providing access to such records completely transparently. Siberia consists of four main components, each of which address a unique challenge for managing cold data in a main-memory system:

- *Cold data classification:* efficiently and non-intrusively identify hot and cold data in a main-memory optimized database environment (the topic of this paper).
- *Cold data storage:* evaluation of cold storage device options and techniques for organizing data on cold storage.
- *Cold data access and migration mechanisms:* mechanisms for efficiently migrating, reading, and updating data on cold storage that dovetail with Hekaton's optimistic multi-version concurrency control scheme [7]
- *Cold storage access reduction:* reducing unnecessary accesses to cold storage for both point and range lookups by maintaining compact and accurate in-memory access filters.

This paper focuses solely on classifying cold and hot data. Solutions to the other challenges are outside the scope of this paper and will be presented in future work.

C. Logging and Offline Analysis

Access frequencies can be estimated *inline* or *offline*. By inline we mean that an estimate of each record's access frequency or rank order is maintained in memory and updated on every record access. Caching policies such as LRU-k, MRU, or ARC are forced to follow this approach because eviction decisions must be made online. In the offline approach record access data is written to a log (separate from the transactional log) for later offline analysis. We chose the offline approach for several reasons. First, as mentioned earlier, the overhead of even the simplest caching scheme is very high. Second, the offline approach is generic and requires minimum changes to the database engine. Third, logging imposes very little overhead during normal operation. Finally, it allows flexibility in when, where, and how to analyze the log and estimate access frequencies. For instance, the analysis can be done on a separate machine, thus reducing overhead on the system running the transactional workloads.

In our logging scheme, we associate each record access with a discrete time slice, denoted $[t_n,\ t_{n+1}]$ (the subsequent time slice begins at t_{n+1} and ends at t_{n+2}, and so on). Time is measured by record accesses, that is, the clock "ticks" on every record access. In the rest of this paper, we identify a time slice using its beginning timestamp (e.g., t_n represents slice $[t_n,\ t_{n+1}]$). A time slice represents a discrete period when a record access was observed, so conceptually our log stores (RecordID, TimeSlice) pairs. Physically, the log stores a list of record ids in access order delineated by time markers that represent time slice boundaries.

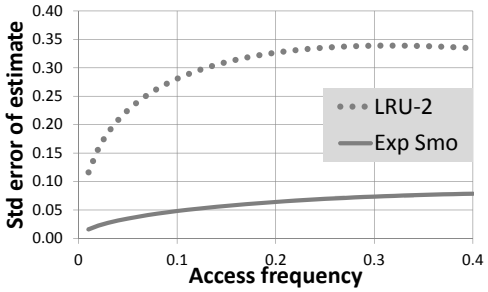


Fig. 2. Standard error of estimated access frequency for LRU-2 and exponential smoothing with $\alpha = 0.05$.

D. Exponential Smoothing

We use exponential smoothing to estimate record access frequencies. Exponential smoothing calculates an access frequency estimate for a record r as

$$est_r(t_n) = \alpha * x_{t_n} + (1 - \alpha) * est_r(t_{n-1}) \qquad (1)$$

where, t_n represents the current time slice, x_{t_n} represents the observation value at time t_n. In our framework x_{t_n} is 1 if an access for r was observed during t_n and 0 otherwise. $est_r(t_{n-1})$ is the estimate from the previous time slice t_{n-1}. The variable α is a decay factor that determines the weight given to new observations and how quickly to decay old estimates. α is typically set in the range 0.01 to 0.05 – higher values give more weight to newer observations.

We chose exponential smoothing because of its simplicity and high accuracy. The accuracy of an estimator is often measured by its standard error, that is, the standard deviation of the probability distribution of the estimated quantity. For a record with true access frequency p, it can be shown that the standard error for exponential smoothing is $\sqrt{\alpha p(1-p)/(2-\alpha)}$ [17]. Appendix D derives the complete distribution of estimates computed by LRU-k from which the standard error can be computed. Figure 2 plots the standard error for LRU-2 and exponentially smoothing with $\alpha = 0.05$. Exponential smoothing is significantly more accurate than LRU-2.

Poor accuracy causes records to be misclassified which reduces the in-memory hit rate and system performance. In Section V, we experimentally compare the hit rates obtained by exponential smoothing as well as the *LRU-2* [14] and *ARC* [15] caching techniques. Our experiments show that exponential smoothing is clearly more accurate that *LRU-2* and *ARC* and achieves a hit rate close to a fictional perfect classifier (a classifier that knows the true frequencies).

E. Sampling

Logging every record access produces the most accurate estimates but the overhead may degrade system performance. Therefore, we consider logging only a sample to reduce system overhead. To implement sampling, we have each worker thread flip a biased coin before starting a new query (where bias correlates with sample rate). The thread records its accesses in log buffers (or not) based on the outcome of the coin flip. In Section V, we report experimental results showing that sampling 10% of the accesses reduces the accuracy by *only*

2.5%, which we consider a tolerable trade-off for reducing log buffer writes by 90%.

III. Classification Algorithms

We now describe algorithms for classifying records as hot and cold. The input and output of the algorithms is exactly the same but they differ in how the result is computed. They all take as input (1) a log L that stores record access observation (as discussed in Section II-C), (2) a parameter K signifying the number of records to classify as "hot". They all employ exponential smoothing (Section II-D) to estimate record access frequency. The algorithms report the "hot" record set as the K record with the highest estimated access frequency. The rest of the records are "cold".

We first briefly present a naive *forward* algorithm that scans the log from a beginning time period t_b to an end time period t_e (where $t_b < t_e$). We then present a novel *backward* algorithm that reads the log in reverse, starting from t_e, and attempts to preemptively stop its scan before reaching t_b while still ensuring correct classification.

A. Forward Algorithm

The forward algorithm simply scans the log forward from a beginning time slice t_b (we assume $t_b = 0$). Upon encountering an access to record r at time slice t_n, it updates r's current access frequency estimate $est_r(t_n)$ using the exponential smoothing equation

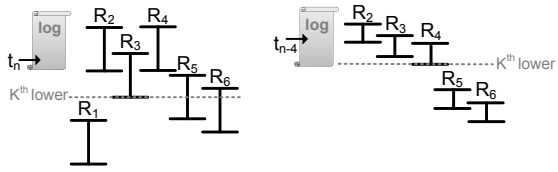
$$est_r(t_n) = \alpha + est_r(t_{prev}) * (1-\alpha)^{(t_n - t_{prev})} \quad (2)$$

where t_{prev} represents the time slice when r was last observed, while $est_r(t_{prev})$ represents the previous estimate for r at that time. To avoid updating the estimate for every record at every time slice (as implied by Equation 1), Equation 2 decays the previous estimate using the value $(1-\alpha)^{(t_n - t_{prev})}$. The exponent $(t_n - t_{prev})$ allows the estimate to "catch up" by decaying the previous estimate across time slices when r was not observed in the log (i.e., when value $x_{t_n} = 0$ in Equation 1). Once the forward algorithm finishes its scan, it ranks each record by its estimated frequency and returns the K records with highest estimates as the hot set.

This forward algorithm has two primary drawbacks: it requires a scan of the entire log and it requires storage proportional to the number of unique record ids in the access log. In the next section, we propose an algorithm that addresses these two drawbacks.

B. Backward Algorithm

We would like to avoid scanning the entire log from beginning to end in order to improve classification performance. In this vein, we propose a backward algorithm that attempts to preempt the computation early. The core idea is to scan the log in reverse and derive successively tighter upper and lower bounds on the estimates for the records encountered. Occasionally, the algorithm performs classification using these bounds to hopefully terminate the scan early. The algorithm only stores estimates for records still in contention for the hot set, resulting in a memory footprint proportional to the number of hot records instead of the total number of records.



(a) At time slice t_n (b) At time slice t_{n-4}

Fig. 3. Backward classification example

1) Bounding Access Frequency Estimates: While reading the log in reverse and encountering an access to a record r at time slice t_n, the backward algorithm incrementally updates a running backwards estimate $estb$ as

$$estb_r(t_n) = \alpha(1-\alpha)^{(t_e - t_n)} + estb_r(t_{last}) \quad (3)$$

where $estb_r(t_{last})$ represents the backward estimate calculated when record r was last encountered in the log at time slice t_{last} (where $t_{last} > t_n$ since we are scanning in reverse). This recursive equation can be derived easily from the non-recursive (unrolled) version of the formula for exponential smoothing. Using the backward estimate, we can compute an upper bound for a record r's actual estimate value at time slice t_n as

$$upEst_r(t_n) = estb_r(t_n) + (1-\alpha)^{t_e - t_n + 1} \quad (4)$$

In this equation, t_e represents the end time slice in the log. The value produced by this equation represents the *largest* access frequency estimate value r can have, assuming that we encounter it at *every* time slice moving backward in the log. Likewise, the lower bound on r's estimated value is

$$loEst_r(t_n) = estb_r(t_n) + (1-\alpha)^{t_e - t_b + 1} \quad (5)$$

This lower bound is the *lowest* estimate value r can have, assuming we will not encounter it again while scanning backward. As the backward classification approach continues processing more record accesses, the upper and lower bounds converge toward an exact estimate. The promise of this approach, however, is that a complete scan of the log may not be necessary to provide a correct classification. Rather, the algorithm can preempt its backward scan at some point and provide a classification using the (inexact) bound values.

2) Backward Classification Optimizations: In order to provide an intuition and outline for the backward classification approach, Figure 3(a) gives an example of upper and lower bounds for six records (R_1 through R_6) after scanning the log back to time slice t_n. Assuming $K = 3$, five records (R_2 through R_6) are in contention to be in the hot set, since their upper bounds lie *above* the kth lower bound defined by R_3.

We use the kth lower bound to provide two important optimizations that reduce processing costs. (1) We drop records with upper bound values that are less than the kth lower bound (e.g., record R_1 in the example). By definition, such records cannot be part of the hot set. (2) We translate the value of the kth lower bound to a time slice in the log named the *accept threshold*. The *accept threshold* represents the time slice in

Algorithm 1 Backward classification algorithm

```
1:  Function BackwardClassify(AccessLog L, HotDataSize K)
2:  Hash Table H ← initialize hash table
3:  Read back in L to fill H with K unique records with calculated bounds
4:  kthLower ← RecStats r ∈ H with smallest r.loEst value
5:  acceptThresh ← ⌊t_e − log_(1−α) kthLower⌋
6:  while not at beginning of L do
7:      rid ← read next record id from L in reverse
8:      RecStats r ← H.get(rid)
9:      if r is null then
10:         /* disregard new record ids read after acceptThresh time slice */
11:         if L.curTime ≤ acceptThresh then goto line 6
12:         else initialize new r
13:     end if
14:     update r.estb using Equation 3
15:     H.put(rid,r)
16:     /* begin filter step - inactivate all records that cannot be in hot set*/
17:     if end of time slice has been reached then
18:         ∀r ∈ H update r.upEst and r.loEst using Equations 4 and 5
19:         kthLower ← find value of k^th lower bound value in H
20:         ∀r ∈ H with r.upEst ≤ kthLower, remove r from H
21:         if num records ∈ H is K then goto line 25
22:         acceptThresh ← ⌊t_e − log_(1−α) kthLower⌋
23:     end if
24: end while
25: return record ids in H with r.active = true
```

the log where we can instantly discard any *new* record ids observed at or beyond the threshold, since we can guarantee that these records will have an upper bound *less* than the kth lower bound. The accept threshold is computed as

$$Threshold = t_e - \lfloor log_{(1-\alpha)} kthLowerBound \rfloor \quad (6)$$

where t_e is the log's end time slice. Since the *accept threshold* allows the algorithm to instantly disregard records with no chance of making the hot set, it greatly limits the memory requirements of the hash table used by the algorithm. As we will see in our experiments (Section V), this optimization allows the memory requirements of the algorithm to stay close to optimal (i.e., close to the hot set size).

Another primary advantage of the backward strategy is its ability to end scanning early while still providing a correct classification. As a concrete example, Figure 3(b) depicts our running example after reading back four time slices in the log to t_{n-4}. At this point, the bounds have tightened leading to less overlap between records. Here, the kth lower bound is defined by R_4, and only three records are in contention for the hot set (R_2, R_3, and R_4. At this point, we can stop scanning the log and report a correct hot set classification, since no other records have upper bounds that cross the kth threshold.

3) Algorithm Description: Algorithm 1 (*BackwardClassify*) provides the pseudo-code for the backward algorithm. First, *BackwardClassify* creates a hash table H to store running estimates for each record it processes (Line 1). Table H maps a record id rid to a structure *RecStats* containing three fields: (a) $backEst$, the running backward access frequency estimate (Equation 3), (b) $loEst$, a record's lower-bound access frequency estimate (Equation 5), and (c) $upEst$, a record's upper-bound access frequency estimate (Equation 4). The algorithm scans backward in the log to fill H with an initial set of K unique records and then finds $kthLower$, the value of the k^{th} lower bound (Line 4). The value of $kthLower$ is then used to define the accept threshold value $acceptThresh$ (Line 5), defined by Equation 6.

After the initialization phase completes, the algorithm scans the log in reverse reading the next rid. If rid does not exist in the hash table and the current time slice ($L.curTime$) is less (older) than $acceptThresh$, we discard rid and read the next record (Line 11). Otherwise, we initialize a new $RecStats$ object for the record (Line 12). Next, the algorithm updates the backward estimate and upper and lower bound values using Equations 3 through 5, respectively, and the $RecStats$ object is put back into the hash table (Line 14-15).

When the algorithm reaches the end of a time slice in log L, it commences a *filter step* that attempts to deactivate records that are out of contention for the hot set and terminate early. The filter step begins by adjusting the upper and lower bounds (Equations 3 through 5) of all active records in H as of the current time slice of the scan defined by $L.curTime$ (Line 18). This step adjusts the distance between upper and lower bounds to be uniform between all active records. Next, the algorithm finds the current kth lower bound value and removes from the hash table H all records with upper bounds *lower* than the new kth lower bound (Lines 19- 20). Removing records allows us to shrink the hash table size, reducing space overhead and allowing for more efficient filter operations in the future. (We discuss the correctness of removing records in Appendix A.) If the number of records in H equals K, the algorithm ends and reports the current set of active records as the hot set (Line 21). Otherwise, the filter step ends by calculating a new *accept threshold* based on the new kth threshold (Line 22). This adjustment moves the accept threshold closer to the current scan point in the log. That is, since the kth threshold is greater than or equal to the last kth threshold, the new accept threshold is guaranteed to be greater than or equal to the last accept threshold. In the worst case, the algorithm ends when a scan reaches the beginning of the log. At this point, all calculated access frequency estimates are exact (i.e., upper and lower bounds are the same), thus the algorithm is guaranteed to find K hot records.

IV. Parallel Classification

We now turn to a discussion of how to parallelize the classification task. In this section, we first discuss a parallel version of the forward classification algorithm. We then discuss how to parallelize the backward classification approach.

A. Parallel Forward Algorithm

Record id partitioning. One way to parallelize the forward algorithm presented in Section III-A is to split the log into n pieces (hash) partitioned by record id. To perform classification, we assign a worker thread to each partition. Each thread uses the serial forward algorithm (Section III-A) to calculate access frequencies for its set of records. After each thread finishes, a final step finds the hot set by retrieving the K records with highest access frequency estimates across *all* partitions.

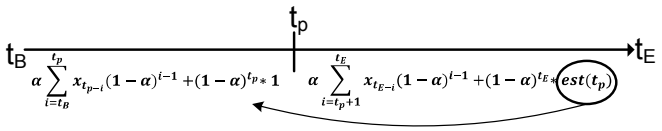


Fig. 4. Exponential smoothing broken up on time boundary

Time slice partitioning. Another parallelization approach is to use a single log but partition the workload by time slices. To illustrate the intuition behind this approach, we unroll the recursive exponential smoothing estimate function from Equation 2 obtaining

$$est_r(t_n) = \alpha \sum_{i=t_b}^{t_n} x_{t_{n-i}}(1-\alpha)^{i-1} + (1-\alpha)^{t_n} \qquad (7)$$

Here, t_b represents the beginning time slice in the log, t_n is the current time slice, while $x_{t_{n-i}}$ represents an observed value at time slice t_{n-i} (0 or 1 in our case). Figure 4 depicts a hypothetical log broken into two segment delineated by time slices t_p. Under each segment is the summation from Equation 7 representing the segment's "contribution" to the access frequency estimate value. The circled term and arrow in the figure highlight that only a single term in the summation of the last segment relies on the estimate value calculated from the previous segment. This property is true for n segments as well: only a *single* term in the summation for each of the n segments relies on the estimate calculated in the previous segment so the bulk of the calculation for each segment can be done in parallel.

The forward parallel algorithm splits the log into n partitions on time slice boundaries, where each partition contains consecutive time slices and the number of time slices in each partition is roughly equal. We assign a worker thread to each partition, whose job is to scan its log partition forward and calculate partial access frequency estimates for the records it encounters (using the serial forward algorithm from Section III-A). Each thread stores its estimate values in a separate hash table.[1] At the end of this process, we have n hash tables populated with the partial estimate values from each log segment. For ease of presentation, we assume hash table $H[n]$ covers the tail of the log, $H[n-1]$ covers the partition directly before partition n, and so forth. We then apply an aggregation step that computes final estimates for each record using the partial estimates in the n hash tables. The K records with highest complete estimate values are then returned as the hot set.

B. Parallel Backward Algorithm

We now discuss a parallelization strategy for the backward approach. Unlike the forward approach, we want to avoid partitioning the workload by time, since the point of backward classification is not to scan back to the beginning time slice in the log. Therefore, our parallel backward classification approach assumes the log is partitioned into n pieces by record id. Our partitioning strategy creates n separate log streams.

[1]For presentation clarity we use n separate hash tables to store estimates. We can also use a shared hash table.

Algorithm 2 Backward parallel classification

```
1:  Function BackParController(HotDataSize K, NumParts n)
2:  /* Phase I: Initialization */
3:  Request from each worker (1) knlb: lower bound of K/n th record, (2) up: num
    records with upper bounds above knlb, (3) low: num records with lower bound
    above knlb.
4:  /* Phase II: Threshold search */
5:  Q ← median knlb reported from Phase I
6:  tlow ← total low count from all workers
7:  tup ← total up count from all workers
8:  if tlow < K then decrease Q
9:  else increase Q
10: issue ReportCounts(Q) command to workers, get new tlow and tup values
11: if |tup − tlow| > 0 then issue TightenBounds command to workers
12: repeat steps 8- 11 until tlow = K and |tup − tlow| = 0
13: /* Phase III: Finalization */
14: List S ← record ids from all workers with upper bound estimates above Q
15: return S
16:
17: Function BackParWorker(LogPartition L, ControllerCommand C)
18: if C = Initialization then
19:     read back in L far enough to find knlb, low, and up
20:     return knlb, lower, and upper to controller
21: else if C = ReportCounts(Q) then
22:     perform new counts for low and up given Q
23:     return new low and up values to controller
24: else if C = TightenBounds then
25:     read back in L to tighten upper and lower bounds for all records
26: else if C = Finalize(Q) then
27:     return record ids with upper bounds above Q to controller
28: end if
```

When logging an access during runtime, the system uses a hash function to direct the write to the appropriate log stream.

In this approach, a single controller thread manages a set of worker threads each assigned to a single log partition. The controller uses the workers to perform a distributed search for a hot set. The worker threads are responsible for reading back in their logs and maintaining backward estimates, upper bounds, and lower bounds using Equations 3 through 5 in the same way as the serial backward algorithm. The controller thread, meanwhile, issues commands to the workers asking for upper and lower bound counts around a given threshold and also instructs the workers how far to read back in their logs. Algorithm 2 provides the psuedo-code for both controller and workers outlining the backward-parallel approach. This algorithm works in three main phases, namely *initialization*, *threshold search*, and *finalization*.

1) Phase I: Initialization: The goal of the initialization phase (Line 3) is to have each worker thread report to the controller an initial set of statistics to determine the "quality" of the records contained in each worker's log. In a perfect world, each worker will hold $\frac{K}{n}$ records that contribute to the hot set. This is rarely, if ever, the case. Since the controller has no a priori information about the records in the log, it requests that each worker read back in its log partitions far enough to find (a) $knth$, the lower-bound estimate of the partition's $\frac{K}{n}^{th}$ hottest record, (b) low a count of the number of records that have lower bounds above or equal to $knth$, and (c) up, a count of the number of records with upper bounds above $knth$. To report accurate counts, each worker must read back far enough to ensure it has read all records that can possibly have upper bound estimates greater than the $knth$ threshold. We ensure that this happens by translating the value $knth$ to a time slice

t in the log using Equation 6 (the equation used to defined accept threshold in serial backward classification algorithm). All records read before reaching t will have upper bounds above $knth$.

Figure 5(a) provides a graphical example of this initialization phase for three worker threads and K value of 9. In this example, worker w_1 reports a $knth$ estimate of 0.7, a low count of 3 and an up count of 6. For worker w_2, $knth$ is 0.6, low is 3, and up is 7, meanwhile for w_4, $knth$ is 0.8, low is 3 and up is 8. This data serves as a running example for the rest of this section to describe the algorithm.

2) Phase II: Threshold Search: The goal of the threshold search phase (Lines 5- 12) is to search for a common threshold across all log partitions guaranteed to yield a final hot set size of K. The basic idea of this phase is to use the $knth$ threshold values, up, and low counts reported in the initialization phase as a search space for finding a threshold that will yield the correct hot set. We know that such a threshold value must exist between the highest and lowest $knth$ threshold values reported from the workers in the initialization phase (Appendix B provides an intuitive explanation).

During this phase, the controller communicates with the workers using two commands:

- *TightenBounds*: this command requests that each worker read back in its log partition further in order to tighten the upper and lower bound estimates for its records. Scanning further back in the log guarantees that the upper and lower bounds for all records will converge and reduce overlap between records. This means the gap between up and low counts will converge, giving the controller a better resolution of the number of records in contention for the hot set.
- *ReportCounts(Q)*: this command asks each worker to report their up and low counts for a given threshold Q. The controller uses this information to test how many records are in contention for the hot set at a given threshold value.

To perform the search, the controller first picks the median threshold value Q reported from the initialization phase and issues a *ReportCounts(Q)* command to each worker. The workers then return their low and up counts. The total low count from all workers $tlow$ represents the lower bound count for records in contention to be in the hot set at threshold Q. Likewise, the total up count tup represents the upper bound count for records in contention for the hot set. If $tlow$ is below K (i.e., too few records are in contention), the controller reduces Q in order to yield more records. On the other hand, if $tlow$ is above K, it increases Q. Initially, choosing a new value for Q involves taking the next step (greater or less) in the list of threshold values generated in the initialization phase. After such a move causes the $tlow$ count to become too low (or too high), the search makes incremental half-steps (like binary search) between the current Q and previous Q value. After finding a new Q, the controller issues another *ReportCounts(Q)* command and receives new $tlow$ and tup counts. If at any point, the absolute difference between $tlow$ and the total tup is greater than zero, it issues a *TightenBounds* command in order to converge the total count resolution. This search process continues until the $tlow$ count is K, and the absolute difference between tup and $tlow$ is equal to zero.

As an example of how this process works, we can return to the example data in Figure 5(a), assuming $K = 9$. The search phase begins with the controller picking an initial threshold value of 0.7 (the median $knth$ value from the initialization phase). After issuing the command *ReportCounts(0.7)*, assume that $tup = 21$ and $tlow = 11$. At this point, the difference between tup and $tlow$ is above zero, so the controller issues a *TightenBounds* command. The controller next sets Q to 0.8 (the next highest $knth$ value reported during initialization), since $tlow$ is currently greater than K. After issuing the command *ReportCounts(0.8)*. Assume $tlow = 6$ and $tup = 7$. Since $tlow$ value is now less than K, the controller sets Q to 0.75 (the average of the previous and current Q values). Figure 5(b) provides an example of the data after the controller issues the command *ReportCounts(0.75)*, where worker w_1 returns up and low counts of 3, w_2 returns up and low counts of 2, and w_3 returns up and low counts of 4. At this point, $tlow = 9$ and the absolute difference between $tlow$ and tup is zero, and the search process ends.

3) Phase III: Finalization: In this phase (Line 14), the controller threads sends the worker a final threshold value Q. Each worker then reports to the controller all record ids in its log partition with upper bound values above Q. The controller then returns the union of these record ids as the hot set.

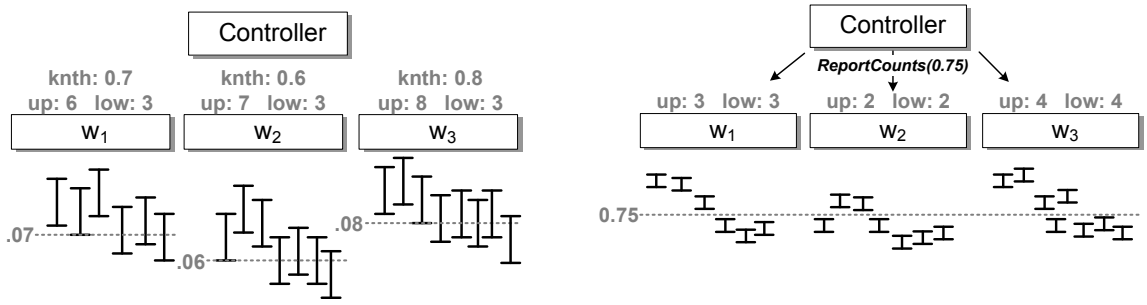
V. Experiments

In this section, we experimentally evaluate the hit rate obtained by the exponential smoothing estimation method as well as the performance of our algorithms. All experiments were implemented in C/C++ and run on an Intel Core2 8400 at 3Ghz with 16GB of RAM running Windows 7. The input in our experiments consists of 1B accesses for 1M records generated using two distributions: (1) *Zipf* with parameter $s = 1$ and (2) *TPC-E* using the TPC-E non-uniform distribution. For the exponential smoothing method, we set the time slice size to 10,000 accesses, while the default hot data set size is 10% of the number of records.

Our experimental evaluation is organized as follows. In Section V-A, we compare the hit rate obtained using the exponential smoothing approach to that of a fictional perfect estimator, as well as two well-known caching techniques, *LRU-2* and *ARC*. In Section V-B, we evaluate the effect of sampling on the hit rate. Finally, Section V-C studies the performance and space overhead of our classification algorithms.

A. Hit Rate Comparison

In this section, we experimentally evaluate the hit rate obtained by exponential smoothing (abbr. *ES*). This experiment feeds the 1B record accesses to the classifier, whose job is to rank order records according to their estimated access frequency and identify the top $K\%$ of the records as the hot set. We then count the number of accesses to "hot" records (out of the 1B) that the classifier would produce for varying



(a) Initialization (b) Controller issuing command *ReportCounts(0.75)*

Fig. 5. Backward parallel classification example

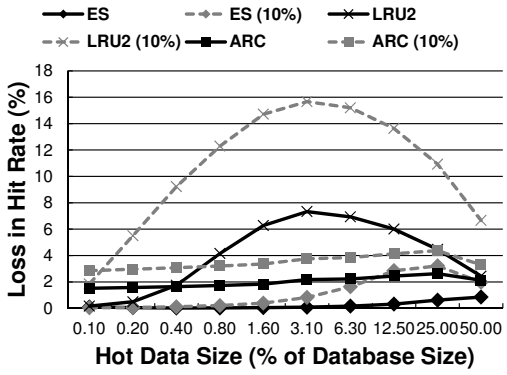


Fig. 6. Loss in hit rate (Zipf)

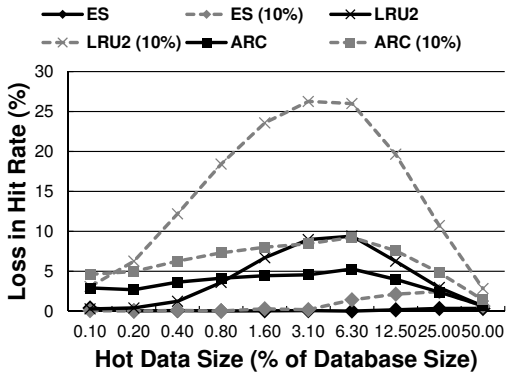


Fig. 7. Loss in hit rate (TPC)

hot data sizes. We compare *ES* to a fictional perfect classifier (abbr. *Perfect*) that knows exact access frequencies and thus identifies the hot set with exact precision.

We also compare *ES* with two well-known cache replacement techniques: (1) *LRU-2* [14], which ranks records based on the distance between the last two accesses to the record. (2) *ARC* [15], a method that manages its cache using two queues, one for "recency" (i.e., recently requested single-access items) and the other for "frequency" (i.e., recently requested items with multiple accesses). ARC adaptively adjusts the size of each queue to react to workload characteristics.

The dark solid lines in Figure 6 plot the loss in hit rate caused by each method for varying hot data sizes using the Zipf distribution. Loss in hit rate is the difference between the hit rate for each tested method and the *Perfect* classifier. *ES* is consistently the most accurate, maintaining a loss in hit rate below 1% for all hot data sizes. *LRU-2* remains fairly accurate for small hot set sizes, but exhibits a hit rate loss as high as 7.8% for larger hot set sizes. *ARC* produces a consistent 2% loss in hit rate, outperforming LRU-2 for most hot data sizes, a result that is consistent with the original ARC experiments [15].

Figure 7 provides the loss in hit rate when using accesses following the TPC-E distribution. Again, the *ES* approach consistently tracks the *Perfect* classifier, maintaining a loss in hit rate well below 1%. *ARC* and *LRU-2* are considerably less accurate, reaching a loss in hit rate of 6.5% and 8.5%, respectively. These experiments show that *ES* is consistently accurate and motivate our use of exponential smoothing.

B. Effect of Sampling

As mentioned in Section II-E, sampling is very attractive to our framework as it reduces the overhead of logging record accesses. Sampling can also be applied to *LRU-2* and *ARC* to reduce their overhead. We implement sampling as follows. On a hit in the cache – presumably the most common case – we randomly choose, with a probability equal to the sampling rate, whether to update the *LRU-2* or *ARC* queues. On a miss we don't have a choice: the record has to be brought in and the queues updated.

The gray dashed lines in Figures 6 and 7 plot the results for each method when sampling 10% of the total accesses (i.e., dropping 90% of the accesses) when compared to the *Perfect* classifier. *LRU-2* exhibits a roughly 3x accuracy degradation for both data sets when sampling is a applied. *ARC* is more oblivious to sampling with a consistent 1.75x drop in accuracy. However, *ES* is still more accurate when accesses are sampled, showing at most a 3.2 percentage point loss in hit rate.

Focusing on *ES*, for smaller hot data sizes (less than 1% of the the database size), sampling does not have a noticeable effect on accuracy. However, for larger hot data sizes, sampling decreases the accuracy of *ES*. A likely explanation is that for both the Zipf and TPC-E distributions, most of the accesses are skewed toward a small number of records. Sampling does not noticeably affect frequency estimations for these records. However, for records with fewer accesses, sampling reduces the accuracy of the estimates, thus causing errors in the rank ordering used for classification. While sampling clearly reduces the hit rate, we believe the loss is still manageable. For instance, a sample rate of 10% introduces *only* a roughly

2.5% drop in hit rate compared with logging and analyzing all accesses. We see this as a tolerable trade-off for eliminating 90% of logging activity.

C. Performance

In this section, we study the performance of the classification algorithms presented in Sections III and IV. All experiments report the time necessary for each algorithm to perform classification on a log of 1B accesses. We use eight threads for the parallel algorithms.

1) Varying Hot Data Set Sizes: Figures 8(a) and 8(b) plot the performance of the forward (abbr. *Fwd*), forward parallel (abbr. *Fwd-P*), backward (abbr. *Back*), and backward parallel (abbr. *Back-P*) classification algorithms using both the Zipf and TPC-E data sets. The run time of each algorithm remains relatively stable as the hot set size increases, with *Back-P* demonstrating the best performance. For the Zipf experiments, the run time for *Back-P* increases from 0.03 seconds to 4.7 seconds as the hot set size increases from 0.1% to 80% of the data size. The run time for the serial *Back* increase from 0.03 seconds to 14 seconds. The run time for the *Fwd* algorithm remains relatively stable around 205 seconds for most hot set sizes, while the *Fwd-P* algorithm finishes in around 46 seconds. The results for the TPC-E data reveal similar results but the run times for all algorithms are slightly higher across the board. This is caused by less skew in the TPC-E data.

2) Varying Accesses per Time Slice: Figures 8(c) and 8(d) plot the performance of each algorithm as the number of accesses per time slice increase from ten to 1M. In the extreme case of 1M accesses per slice (i.e., the total number of records in our experiments), it is likely that all records will have nearly the same access frequency estimates, since there is a greater chance that an access for each record will be present in each time slice. While we do not expect such a scenario in real life (i.e., time slice sizes should be set to a fraction of the total data size in practice), we test these extreme cases to explore the boundaries of our classification algorithms.

Figure 8(c) depicts the results for the Zipf data set. The classification time for *Fwd* slowly decreases as the number of time slices decrease. We believe this trend is mostly due to increased redundancy within a time slice: since a record's estimate is only updated once per time slice, subsequent accesses for the same record in the same slice will not trigger an update. *Back* and *Back-P* demonstrate superior performance for smaller time slices, but show a steep performance decline for the larger time slices. For time slices that contain between ten and 10K accesses, the classification times for the *Back-P* algorithm remain in the sub-second range going from 0.039 seconds to 0.96 seconds, while times for the *Back* algorithm increase from 0.031 seconds to 8.03 seconds in this range.

For small time slice sizes, both algorithms are able to terminate early but, for larger time slices, the performance of both backward algorithms degrades. In this case, many records have overlapping upper and lower bounds due to the greater chance that most records will have an access within each time slice. At 100K (i.e., the size of the hot set size), the backward algorithms start to see an overwhelming amount of overlap between upper and lower bounds and thus are unable to quickly delineate a threshold to determine the hot set. For time slice sizes of 1M accesses, early termination is even more difficult for these algorithms.

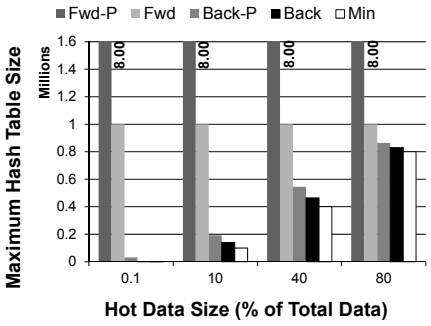


Fig. 9. Space overhead (Zipf)

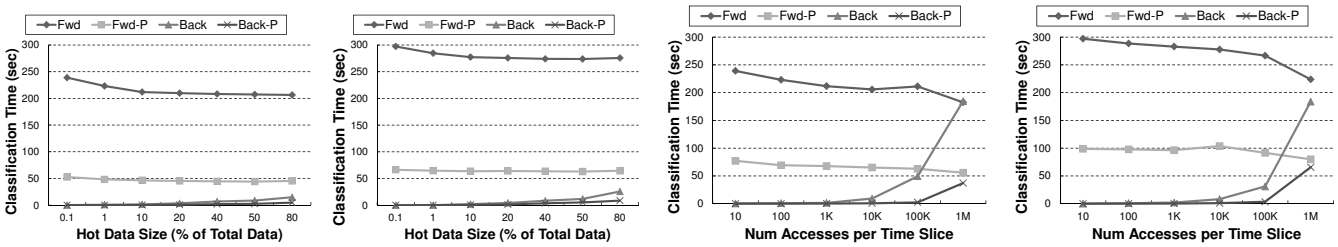
Figure 8(d) reports the classification times for the TPC-E dataset. The results are similar in nature to the Zipf experiments. However, classification times across the board are slightly higher for reasons similar to those discussed in Section V-C.1.

3) Space Overhead: Figures 9 and 10 depict the space used by each algorithm for varying hot data sizes. We measure space overhead as the maximum number of entries that each algorithm stores in its hash table. The *Min* bar in the graph represents the minimum number of entries possible for each hot set size (i.e., the number of records in the hot set).

The *Fwd* algorithm has space overhead equivalent to the number of unique record in the access log (in our case 1M). The *Fwd-P* requires the most space of all algorithms, requiring roughly 6.7M entries for the TPC workload and 8M entries for the Zipf workload. The reason is that a record id may be present in several log partitions, meaning each worker thread will store its own statistics for that record. The *Back* algorithm uses the *accept threshold* to only keep records in its hash table that are in contention for the hot set. As we can see, this optimization allows the space overhead of *Back* to stay close to the optimal *Min* overhead. Meanwhile, the *Back-P* algorithm also tracks the *Min* overhead fairly closely. It requires space overhead slightly more than that used by *Back*, since some worker threads will maintain local statistics for records that end up not making it into the hot set. Overall, the backward variants of the algorithms are clearly superior in terms of space requirements.

D. Discussion

Our experiments show the relative strengths of our classification framework. We now briefly summarize two scenarios where this approach is not applicable. (1) *Detecting rapid access fluctuations*. Our framework is not designed to detect rapid, short-lived changes in access patterns. We aim to observe access patterns over a longer period (on our log) to ensure that infrequently accessed records are sufficiently cold and are likely to remain cold. (2) *Use with very low sample rates*. If sampling rates are very low (e.g., possibly due to a



(a) Zipf (b) TPC (c) Zipf (d) TPC

Fig. 8. Performance of classification algorithms. Figures (a) and (b) plot performance for varying hot set sizes. Figures (c) and (d) plot performance for varying number of accesses per time slice.

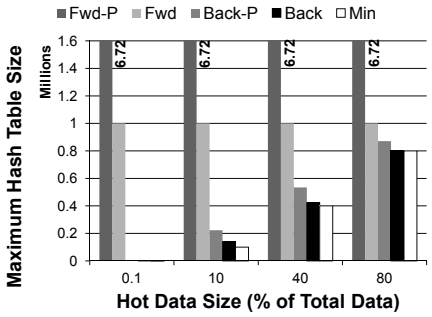


Fig. 10. Space overhead (TPC)

high load on the system), the accuracy of our estimates may drop outside of a reliable threshold. This is true of *any* estimation technique including well-known caching approaches. However, we find that a 10% sampling rate provides estimates that are accurate enough while sufficiently reducing access logging on the system's critical path.

VI. Related Work

Main memory OLTP engines. There has been much work recently exploring OLTP engine architectures optimized for main-memory access [1], [3], [4], [5], [6]. Research prototypes in this space include H-Store [18], [5], HYRISE [1], and HyPer [2]. Commercial main-memory systems are currently available such as IBM's solidDB [3], Oracle's TimesTen [4], and VoltDB [6]. VoltDB assumes that the entire database fits in memory, completely dropping the concept of secondary storage. We diverge from this "memory-only" philosophy by considering a scenario where a main-memory optimized engine may migrate cold data to secondary storage and investigate how to identify hot and cold data.

Caching. One straightforward solution is to apply caching where hot records are those inside the cache while other records are cold. Caching has been an active area of research for over 40 years and many caching algorithms have been proposed, for example, LRU [12], LRU-k [14], ARC [15], 2Q [13], and others [19], [20], [21], [22], [23]. However, caching has significant drawbacks for our particular problem. The overhead is high both in space and time: a minimum of 25% in CPU time and 16 bytes per record. We also found that exponential smoothing estimates access frequencies accurately which results in a higher hit rate than the best caching techniques.

"Top-k" processing. The backward algorithms we propose create upper and lower bounds thresholds on record access frequency estimates in order to potentially terminate early. This gives the techniques a flavor of "top-k" query processing (see [24] for an extensive survey). The problem we study in this paper is not equivalent to top-k processing, since our environment differs in two fundamental ways: (1) Top-k processing ranks objects by scoring tuples using a monotonic function applied to one or more of the tuple's attributes. The problem we study deals with efficiently estimating access frequencies based on logged record accesses. (2) To run efficiently, all top-k techniques assume sorted access to *at least* a single attribute used to score the tuple [24]. Our proposed algorithms scan accesses as they were logged, and do not assume any preprocessing or sort order.

Cold data in main-memory databases. The HyPer system is a main-memory hybrid OLTP and OLAP system [2]. HyPer achieves latch-freedom for OLTP workloads by partitioning tables. Partitions are further broken into "chunks" that are stored in a decomposed storage model in "attribute vectors" with each attribute vector stored on a different virtual memory (VM) page. This approach enables VM page snapshoting for OLAP functionality. HyPer has a cold-data management scheme [16] capable of identifying cold transactional data, separating it from the hot data, and compressing it in a read-optimized format for OLAP queries.

HyPer's cold data classification scheme differs from ours in several dimensions. (1) *Classification granularity*. HyPer performs cold/hot data classification at the VM page level, which is the granularity of its data organization. Our method classifies data at the record level due to Hekaton's record-centric organization. (2) *Classification method*. HyPer's classification technique piggybacks on the CPUs memory management unit setting of dirty page flags used for VM page frame relocation. Since HyPer pins pages in memory, it is able to read and reset dirty flags to help classify cold and hot pages. We propose logging a sample of record accesses and classifying cold data offline based on the algorithms proposed in this paper. (3) *Purpose*. HyPer's technique aims primarily at reducing copy-on-write overhead (caused by VM page snapshots) and reducing the memory footprint for data still accessed by OLAP queries. Our technique aims at maximizing main-memory hit-rate and assumes that cold data on secondary storage is infrequently (or never) accessed.

VII. Conclusion

This paper takes a first step toward realizing cold-data management in main-memory databases by studying how to efficiently identify hot and cold data. We proposed a framework that logs a sample of record accesses, a strategy that introduces minimal system overhead. We use exponential smoothing as a method to estimate record access frequencies, and show both theoretically and experimentally that it is more accurate than LRU-2 and ARC, two well-known caching techniques. We proposed a suite of four classification algorithms that efficiently identify hot and cold records based on logged record accesses. Through experimental evaluation, we found that the *backward* algorithms are very time and space efficient. Specifically, these algorithms are capable of sub-second classification times on a log of 1B accesses and provide close to optimal space efficiency.

References

[1] M. Grund, J. Krüger, H. Plattner, A. Zeier, P. Cudre-Mauroux, and S. Madden, "HYRISE - A Main Memory Hybrid Storage Engine," in *VLDB*, 2012.

[2] A. Kemper and T. Neumann, "HyPer: A Hybrid OLTP & OLAP Main Memory Database System Based on Virtual Memory Snapshots," in *ICDE*, 2011, pp. 195–206.

[3] "IBM solidDB, information available at http://www.ibm.com/."

[4] "Oracle TimesTen In-Memory Database, information available at http://www.oracle.com."

[5] M. Stonebraker et al, "The End of an Architectural Era (Its Time for a Complete Rewrite)," in *VLDB*, 2007.

[6] "VoltDB, information available at http://www.voltdb.com/."

[7] P.-Å. Larson, S. Blanas, C. Diaconu, C. Freedman, J. M. Patel, and M. Zwilling, "High-Performance Concurrency Control Mechanisms for Main-Memory Databases," in *VLDB*, 2012.

[8] I. Pandis, P. Tozun, R. Johnson, and A. Ailamaki, "PLP: Page Latch-free Shared-everything OLTP," in *VLDB*, 2011.

[9] J. Sewall, J. Chhugani, C. Kim, N. Satish, and P. Dubey, "PALM: Parallel Architecture-Friendly Latch-Free Modifications to B+ Trees on Many-Core Processors," in *VLDB*, 2011.

[10] S. Blanas, Y. Li, and J. M. Patel, "Design and Evaluation of Main Memory Hash Join Algorithms for Multi-Core CPUs," in *SIGMOD*, 2011, pp. 37–48.

[11] J. Gray and F. Putzolu, "The 5 Minute Rule for Trading Memory for Disk Accesses and the 10 Byte Rule for Trading Memory for CPU Time," in *SIGMOD*, 1987.

[12] P. J. Denning, "The Working Set Model for Program Behaviour," *Commun. ACM*, vol. 11, no. 5, pp. 323–333, 1968.

[13] T. Johnson and D. Shasha, "2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm," in *VLDB*, 1994, pp. 439–450.

[14] E. J. O'Neil, P. E. O'Neil, and G. Weikum, "The LRU-K Page Replacement Algorithm For Database Disk Buffering," in *SIGMOD*, 1993, pp. 297–306.

[15] N. Megiddo and D. S. Modha, "ARC: A Self-Tuning, Low Overhead Replacement Cache," in *FAST*, 2003.

[16] F. Funke, A. Kemper, and T. Neumann, "Compacting Transactional Data in Hybrid OLTP & OLAP Databases," *PVLDB*, vol. 5, no. 11, pp. 1424–1435, 2012.

[17] J. R. Movellan, "A Quickie on Exponential Smoothing, available at http://mplab.ucsd.edu/tutorials/ExpSmoothing.pdf."

[18] R. Kallman et al, "H-store: A HighPerformance, Distributed Main Memory Transaction Processing System," in *VLDB*, 2011.

[19] S. Bansal and D. S. Modha, "CAR: Clock with Adaptive Replacement," in *FAST*, 2004, pp. 187–200.

[20] S. Jiang and X. Zhang, "LIRS: An Efficient Low Inter-reference Recency Set Replacement Policy to Improve Buffer Cache Performance," in *SIGMETRICS*, 2002, pp. 31–42.

[21] D. Lee et al, "LRFU: a spectrum of policies that subsumes the least recently used and least frequently used policies," *Computers, IEEE Transactions on*, vol. 50, no. 12, pp. 1352–1361, dec 2001.

[22] J. T. Robinson and M. V. Devarakonda, "Data Cache Management Using Frequency-Based Replacement," in *SIGMETRICS*, 1990, pp. 134–142.

[23] Y. Zhou and J. F. Philbin, "The Multi-Queue Replacement Algorithm for Second Level Buffer Caches," in *USENIX*, 2001, pp. 91–104.

[24] I. F. Ilyas, G. Beskales, and M. A. Soliman, "A survey of top-*k* query processing techniques in relational database systems," *ACM Comput. Surv.*, vol. 40, no. 4, 2008.

Appendix

A. Hash Record Removal in Backward Algorithm

We provide an intuitive argument for why it is safe to remove a record r from the hash table during the filter step in the backward algorithm (Section III-B). First, r cannot possibly be in the final hot set, since its upper bound access frequency estimate is *less than* the value representing the kth lower bound (abbr. q). Second, we can safely discard the statistics for r since it will never be added back to the hash table. If we encounter r while reading further back in the log at time slice t_n, the largest upper bound it can possible have at t_n is a value that falls below q. That is, it cannot be higher than its upper bound value that caused it to be removed from the hash table in the first place. Thus, by definition of the *accept threshold* (Equation 6), t_n must be less than than the accept threshold defined by q, otherwise r's upper bound would be above q, which contradicts our reason for removing it from the hash table in the first place. Since t_n is less than the accept threshold, r will never be added back to the hash table in the future. For the two reasons just described, removing r from the hash table is safe, i.e., we are not removing a record that could possibly be in the hot set.

B. Threshold Search Convergence

Using Figure 5(a) as an example, we now show that a threshold between 0.6 (the lowest $knth$ value reported in initialization) and 0.8 (the highest $knth$ value reported during initialization) will yield a correct hot set, assuming $K = 9$. With a threshold of 0.6, worker w_2 will retrieve at least three records (since its low count is three). Since the other workers have $knth$ values greater than 0.6, we are guaranteed to retrieve at least nine records in total (our target). However, at a threshold value of 0.6, workers w_1 and w_3 will likely have lower bound counts greater than three, meaning a threshold of 0.6 may be too conservative. Meanwhile, at a threshold of 0.8, there may be too few records, since w_3 is the only worker guaranteed to return three records at this threshold (i.e., has a low count of three). Therefore, a "sweet spot" must exist in between thresholds of 0.6 and 0.8 that produces a hot set size of exactly K records.

C. TightenBounds Details

In this section, we discuss how to define the distance each worker must read back after receiving a *TightenBounds* command from the controller. While executing a *ReportCounts(Q)* command, we require that each worker keep track of a *minimum overlap* value. This value is the minimum difference

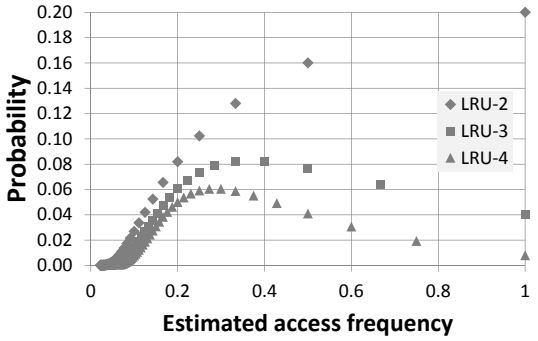


Fig. 11. Probability distribution of access frequency estimate ($p = 0.2$).

between the given threshold Q and the upper bound estimate value closest to but greater than Q. During the *TightenBounds* process, we require that each worker read back far enough to tighten its bounds within this *minimum overlap* value. The intuition behind this requirement is that if bounds are shrunk to the minimum overlap range, the overall overlap between records across partitions should be sufficiently small and shrink the $tlow$ and tup counts to a sufficient level.

Given a minimum overlap value of M, how far must a worker read back in its log partition to tighten estimate bounds within M? The goal is to translate M to a time slice in the log. Recall from Equations 3 through 5 that we calculate the upper bound by adding a value $U = (1-\alpha)^{t_e - t_n + 1}$ to the $estb_r$ value (from Equation 3). Likewise, we calculate the lower bound by adding the value $U = (1-\alpha)^{t_e - t_b + 1}$ to $estb_r$. Since U is the variable term (due to its reliance in t_n), we want to shrink U such that $U \leq M$. In other words, we want $(1-\alpha)^{t_e - t_n + 1} \leq M$. Solving for t_n gives us the time slice defined as: $t_n = t_e + 1 - log_{(1-\alpha)} M$.

D. Distribution of LRU-k Estimates

The LRU-k caching policy ranks items in the cache according to the observed distance between the last k accesses an item and evicts the item with the longest distance. This is equivalent to estimating the access frequency based on the last k accesses and rank ordering the items based on the estimate. In this section we derive the probability distribution of the access frequency estimated by LRU-k under the independent reference model.

Suppose we observe an access to a record r and the distance to the kth previous access is n. Represent an access to a record r by a red ball and an access to any other record by a blue ball. If so we have a sequence of $n+1$ balls where the first and last balls are both red and the remaining $n-1$ balls is a mix of $k-2$ red balls and $n-1-(k-2) = n-k+1$ blue balls. There are $\binom{n-1}{k-2}$ distinct ways to arrange a set of $n-1$ red and blue balls where $k-2$ balls are red. Suppose the probability of drawing a red balls is p (p is the probability of accessing record r). When drawing a red ball, then the probability of having a prior sequence of n balls that ends with a red ball and contains $k-2$ additional red balls is

$$Pr(n, k-1) = \binom{n-1}{k-2} p^{k-1} (1-p)^{n-k+1} \quad n \geq k-1 \quad (8)$$

The LRU-k policy amounts to estimating the probability p by $\hat{p} = (k-1)/n$, that is, the fraction of red balls in the sequence of n balls. The estimate takes only the discrete values $1, (k-1)/k, (k-1)/(k+1), (k-1)/(k+1), ...$ The probability distribution of the estimate computed by LRU-k is then

$$P_k(\hat{p} = \frac{k-1}{n}) = \binom{n-1}{k-2} p^{k-1} (1-p)^{n-k+1} \quad n \geq k-1 \quad (9)$$

The formulas for the special cases $k = 2$, $k = 3$, and $k = 4$ are much simpler as shown below. For example, the LRU-2 estimate follows a geometric distribution.

$$P_2(\hat{p} = 1/n) = p(1-p)^{n-1} \qquad n = 1, 2, 3, ...$$
$$P_3(\hat{p} = 2/n) = (n-1)p^2(1-p)^{n-2} \qquad n = 2, 3, 4, ...$$
$$P_4(\hat{p} = 3/n) = \frac{(n-1)(n-2)}{2} p^3 (1-p)^{n-3} \quad n = 3, 4, 5, ...$$

Figure 11 plots the distribution of the access frequency estimated by LRU-2, LRU-3, and LRU-4. The true access frequency is $p = 0.2$. As expected the variance of the estimate is reduced as k increases but, in general, it remains quite high.

E. N-Minute Rule Calculation

SSD and DRAM hardware trends can be translated into an updated version of the n-minute rule introduced by Gray and Putzolu [11]. The n-minute rule is defined as:

$$BEInterval = \frac{SSDPrice}{IO/s} \times \frac{1}{ItemSize \times RAMPrice} \quad (10)$$

The equation shows that data should be cached if it is accessed more often than once per break-even interval (BEInterval), otherwise it is more economical to store it on an SSD. The first term of the equation represents the price paid for one SSD IO per second, while the second term represents the cost of buffering an item (page, tuple. etc) in memory. The SSD cost per IO/s is surprisingly uniform across hardware vendors in the range of \$0.05-0.10 per read IO/s, while server class DRAM costs around \$15/GB. Therefore, the break-even point for a 200 byte record is about 60 minutes, while for a traditional 4kB page is about 3 minutes. The break-even interval for a 4kB page is around 175 sec, remarkably close to the 400 sec threshold derived by Gray and Putzolo 25 years ago despite orders of magnitude improvements in hardware. The break-even interval grew slowly over time as improvements in the cost of a HDD IOs (\$1000 per IO/s in 1986, \$2 per IO/s in 2012; approx. 500X improvement) lagged behind DRAM cost improvements (\$5/kB in 1986, \$15/GB in 2012; approx. 350,000X improvement). Switching from a magnetic disk to an SSD improves IO cost by about 100X. In essence, SSDs have made the n-minute rule once more relevant.

The prices used in the calculation were obtained from the Crucial and Dell web sites and from recent price quotes and web sites of suppliers of SSDs. I/O rates were obtained from SSD spec sheets.