ABSTRACT

Database platform support for efficient ranking can have positive performance implications for a number of rank-aware applications, including data exploration, social network analysis, and keyword search. This talk highlights two pieces of work where ranking appears in the research area of database architectures for new hardware. First, we highlight the Bw-tree, a new high-performance B+-tree supporting sorted key-sequential access. The Bw-tree is re-architected to run efficiently on new hardware: its in-memory operations are completely latch-free, removing blocking behavior while also improving multi-core cache behavior, while its storage layer implements a novel log-structured flash storage layer for that exploits fast sequential writes and mitigates adverse performance impact of random writes. Second, we highlight a new classification technique for identifying cold (infrequently accessed) data in main-memory database systems. Using a log of sampled record accesses, our technique estimates record access frequencies using exponential smoothing. This classification approach is very efficient: it is able to accurately identify hot and cold records among 1M records in sub-second time from a log of 1B record accesses on a workstation class machine.

1. THE Bw­Tree: A RANGE INDEX FOR MODERN HARDWARE

A necessity in many rank-aware applications is to store objects in sorted order according to some computed value (e.g., an aggregate “score” combining price and distance). The ubiquitous B-tree index is a very attractive option for this purpose. A B-tree supports high performance key-sequential access to both individual keys and designated subranges of keys. It is the combination of random and range access that has made B-trees the indexing method of choice within database systems and in stand-alone atomic record stores. While the B-tree has been around for a while, the hardware infrastructure for which it was designed has changed dramatically. We recently created the Bw-tree [10, 11], a new B-tree whose design enables very high performance in the new hardware environment that has recently emerged. The Bw-tree addresses two important trends:

Design for multi-core. We now live in a multi-core world where uni-core speed will at best increase modestly. We need to better exploit a large number of cores by addressing at least two important aspects:

1. Multi-core CPUs mandate high concurrency. But, as the level of concurrency increases, latches are more likely to block, limiting scalability [1].

2. Good multi-core processor performance depends on high CPU cache hit ratios. Updating memory in place results in cache invalidations, so how and when updates are done needs great care.

Addressing these issues, the Bw-tree is latch-free, ensuring a thread never yields or even re-directs its activity in the face of conflicts. The Bw-tree also performs “delta” updates that avoid updating a page in place, hence preserving previously cached lines of pages.

Design for Modern Storage Devices. Hard disk latency is a major problem. But even more crippling is their low I/O rate. Flash storage offers higher I/O operations per second at lower cost. The Bw-tree targets flash storage. Flash has some performance idiosyncrasies, however. While flash has fast random and sequential reads, it needs an erase cycle prior to write; making random writes slower than sequential writes [5]. While flash SSDs typically have a mapping layer (the FTL) to hide this discrepancy from users, a noticeable slowdown still exists. As of 2011, even high-end FusionIO drives exhibit a 3x faster sequential write performance than random writes [2]. The Bw-tree performs log structuring itself at its storage layer. This approach avoids dependence on the FTL and ensures that the Bw-tree write performance is as high as possible for both high-end and low-end flash devices.

1.1 The Bw-Tree Design

1.1.1 Bw-tree Architecture

The Bw-tree is a classic B+-tree in many respects. It provides logarithmic access to keyed records from a one-dimensional key range, while providing linear time access to sub-ranges. The Bw-tree architecture consists of three
layers: (1) the top B-tree layer provides the access method API, and is responsible for the search, record update, and structure modification logic common to a B-tree. (2) The middle cache layer serves the B-tree layer with pages that may be in-memory or on flash; it is also responsible for installing “delta” updates on a page in a latch-free manner. (3) The bottom flash layer implements our log-structured store (LSS).

1.1.2 The Mapping Table

Our cache layer maintains a mapping table, that maps logical pages to physical pages; logical pages are identified by a logical “page identifier” or PID. The mapping table translates a PID into either (1) a flash offset, the address of a page on stable storage, or (2) a memory pointer, the address of the page in main memory. The mapping table is the central location for managing our “paginated” tree. All links between Bw-tree nodes are PIDs, not physical pointers. The mapping table enables the physical location of a Bw-tree node (page) to change on every update and every time a page is written to stable storage, without requiring that the location change propagate to the root of the tree, because inter-node links are PIDs that do not change. This “relocation” tolerance enables both delta updating of the node in main memory and log structuring of our stable storage.

1.1.3 Latch-Free In-Memory Operations

In our Bw-tree design, threads never block on Bw-tree pages in memory because we use no latches. Being latch-free permits us to drive processors to close to 100% utilization. Instead of latches, we install state changes using compare and swap (CAS) instructions. The Bw-tree only blocks when it needs to fetch a page from stable storage (the LSS), which is rare with a large main memory cache.

Delta Updating. The Bw-tree updates pages by creating a delta record (describing the change) and prepending it to an existing page state (the delta record contains a pointer to the existing page state). It installs the (new) memory address of the delta record into the page’s slot in the mapping table using a CAS instruction. If the CAS succeeds, the delta record address becomes the new physical “root” address of the page, thus updating the page. This strategy is used both for data changes (e.g., inserting a record) and management changes (e.g., splitting a page or flushing it to stable storage). Delta updating simultaneously enables latch-free access in the Bw-tree and preserves processor data caches by avoiding update-in-place. Figure 1(a) depicts a delta update record \( D \) prepended to page \( P \); the dashed line represents \( P \)'s original address, while the solid line to \( D \) represents \( P \)'s new address.

We occasionally consolidate pages by creating a new page that applies all delta changes to a search optimized base page. This reduces memory footprint and improves search performance. A consolidated form of the page is also installed with a CAS, as depicted in Figure 1(b) showing the consolidation of page \( P \) with its deltas into a new “Consolidated Page \( P' \)”. We garbage collect the prior page (with deltas) by placing it on a pending list to be reclaimed when safe (i.e., when no other threads may access it). Garbage collection and the CAS failure protocol are described in our full paper [10].

Structure Modifications. Index structure modifications operations (SMOs) such as node splits and merges introduce changes to more than one page. This presents a problem in a latch-free environment since (a) we cannot change multiple pages with a single CAS and (b) we cannot employ latches to protect parts of our index during the SMO. All Bw-tree SMOs are performed in a latch-free manner; to our knowledge this has never been done before. The main idea is to break an SMO into a sequence of atomic actions, each on a single page and installable via a CAS. The details of Bw-tree SMOs are described in [10].

1.1.4 Log Structured Store

Caching. The cache layer of the LSS is responsible for reading, flushing, and swapping pages between memory and flash. It maintains the mapping table and provides the abstraction of logical pages to the Bw-tree layer. Pages in main memory are occasionally written (flushed) to stable storage for a number of reasons. For instance, the Bw-tree may assist in transaction log checkpointing if it is part of a transactional system such as Deuteronomy [9, 13], or to reduce memory usage. Flushing and “swap out” of a page installs a flash offset in the mapping table and permits reclaiming page memory.

Storage Management. Our LSS has the usual advantages of log structuring [14]. Pages are written sequentially in a large batch, eliminating any write bottle neck by reducing the number of separate write I/Os required. The cache manager marshals the bytes from the pointer representation of the page in main memory into a linear representation that can be written to the flash buffer. To keep track of which part of the page is on stable storage and where it is, we use a flush delta record, which is installed by updating the mapping table entry for the page using a CAS. Flush delta records also record which changes to a page have been flushed so that subsequent flushes send only incremental page changes to stable storage. This can dramatically reduce how much data is written during a page flush, increasing the number of page updates that fit in the flush buffer, and hence reducing the number of I/Os per page. There is a penalty on reads, however, as the discontinuous parts of pages must all be in main memory to make a page accessible in the main memory cache. This penalty is mitigated by the very high random read performance of flash. The LSS cleans prior parts of flash representing the old parts of its log storage. Delta flushing reduces pressure on the LSS cleaner by reducing the amount of storage used per page. This reduces the “write amplification” that is a characteristic of log structuring. During cleaning, LSS makes pages and their deltas contiguous on flash for improved access performance.

Figure 1: Delta updates and consolidation.
2. COLD DATA IDENTIFICATION IN MAIN-MEMORY DATABASES

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 [4, 6, 12, 7, 15]. This section highlights a lightweight and efficient method for classifying cold and hot data in main-memory optimized database engines [8]. This work is part of project siberia, aimed at managing “cold” (infrequently accessed) data in Hekaton [3], Microsoft SQL Server’s main-memory optimized OLTP database engine.

Motivation. 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. 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.

Three main considerations drive the need for cold data management in main-memory databases: (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 accessed frequently updated until delivery, then used for analysis of workloads. (2) Economies. It is significantly cheaper to store cold data on external storage such as flash with little effect on overall system performance. (3) Overhead of caching. Caching is a tried-and-true technique for identifying hot data, but is unattractive for main-memory systems due to two main overheads. (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 (e.g., LRU-k or ARC queues). (b) Space overhead. Hekaton, like several other main-memory systems, does not use page-based storage structures for efficiency reasons; there are only records. 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.

2.1 Our Approach

The problem solved by this work 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.

2.1.1 Logging and Offline Analysis

Our cold data classification techniques samples 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 offline (i.e., off the system’s critical path) from the logged accesses, and the records with the highest estimated frequency form the hot set. 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).

2.1.2 Exponential Smoothing

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

$$est_r(t_n) = \alpha \cdot x_{t_n} + (1 - \alpha) \cdot est_r(t_{n-1})$$  \hspace{1cm} (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 $\alpha$ is a decay factor that determines the weight given to new observations and how quickly to decay old estimates.

2.2 Classification Algorithms

The novel contribution of this work is a set of novel algorithms for estimating access frequencies using exponential smoothing. We explore two general algorithmic approaches: (1) a forward algorithm that scans the log from beginning to end (i.e., past to present) and calculates record access frequencies along the way. (2) 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. In the full paper [8], we show how to parallelize both the forward and backward algorithms to speed up estimation times dramatically.

2.2.1 The Forward Approach

The forward algorithm simply scans the log forward from a beginning time slice $t_0$ (we assume $t_0 = 0$). 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

$$est_r(t_n) = \alpha + est_r(t_{prev}) \ast (1 - \alpha)^{t_n - t_{prev}}$$  \hspace{1cm} (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.

2.2.2 The Backward Approach

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
records with upper bound values that are less than the k-th lower bound (e.g., record R such records cannot be part of the hot set. (2) We translate t (gives an example of upper and lower bounds for six records line for the backward classification approach, Figure 2(a) provide a classification using the (inexact) bound values.

The value produced by this equation represents the memory footprint proportional to the number of hot records for records still in contention for the hot set, resulting in a

Estimate Bounds. While reading the log in reverse and encountering an access to a record r at time slice t, the backward algorithm incrementally updates a running backward estimate estb

\[ estb_r(t) = \alpha(1-\alpha)(t_{estb-1}) + estb_r(t_{estb}) \]

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 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 as

\[ upEst_r(t) = estb_r(t) + (1-\alpha)^{t_{estb-1}+1} \]

In this equation, t_est 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

\[ loEst_r(t) = estb_r(t) + (1-\alpha)^{t_{estb-1}+1} \]

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.

Backward Algorithm. To provide an intuition and outline for the backward classification approach, Figure 2(a) gives an example of upper and lower bounds for six records (R1 through R6) after scanning the log back to time slice t_{estb}. Assuming K = 3, five records (R2 through R6) are in contention to be in the hot set, since their upper bounds lie above the k-th lower bound defined by R3.

We use the k-th lower bound to provide two important optimizations that reduce processing costs. (1) We drop records with upper bound values that are less than the k-th lower bound (e.g., record R1 in the example). By definition, such records cannot be part of the hot set. (2) We translate the value of the k-th lower bound to a time slice in the log named the accept threshold. The accept threshold represents the time slice in 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 k-th lower bound. The accept threshold is computed as

\[ \text{Threshold} = t_e - [\log(1-\alpha)k-thLowerBound] \]

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.

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 2(b) depicts our running example after reading back four time slices in the log to t_{estb-4}. At this point, the bounds have tightened leading to less overlap between records. Here, the k-th lower bound is defined by R4, and only three records are in contention for the hot set (R2, R3, and R4). 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 k-th threshold.

3. REFERENCES