Dynamic and Transparent Data Tiering for In-Memory Databases in Mixed Workload Environments

Carsten Meyer
Hasso Plattner Institute
Potsdam, Germany
carsten.meyer@hpi.de

Martin Boissier
Hasso Plattner Institute
Potsdam, Germany
martin.boissier@hpi.de

Adrian Michaud
EMC^2 Corporation
Hopkinton, USA
adrian.michaud@emc.com

Jan Ole Vollmer
Hasso Plattner Institute
Potsdam, Germany
jan.vollmer@student.hpi.de

Ken Taylor
EMC^2 Corporation
Hopkinton, USA
ken.taylor@emc.com

David Schwalb
Hasso Plattner Institute
Potsdam, Germany
david.schwalb@hpi.de

Matthias Uflacker
Hasso Plattner Institute
Potsdam, Germany
matthias.uflacker@hpi.de

Kurt Roedszus
EMC^2 Corporation
Hopkinton, USA
kurt.roedszus@emc.com

ABSTRACT

Current in-memory databases clearly outperform their disk-based counterparts. In parallel, recent PCIe-connected NAND flash devices provide significantly lower access latencies than traditional disks allowing to re-introduce classical memory paging as a cost-efficient alternative to storing all data in main memory. This is further eased by new, dedicated APIs which bypass the operating system, optimizing the way data is managed and transferred between a DRAM caching layer and NAND flash. In this paper, we will present a new approach for in-memory databases that leverages such an API to improve data management without jeopardizing the original performance superiority of in-memory databases. The approach exploits data relevance and places less relevant data onto a NAND flash device. For real-world data access skews, the approach is able to efficiently evict a substantial share of the data stored in memory while suffering a performance loss of less than 30%.

1. INTRODUCTION

Storage Class Memory (SCM) is a class of solid state memory whose performance characteristics set it apart from main memory as well as classical disk drives. The latest generation of PCIe-connected NAND flash cards has considerably lowered the performance gap between main memory as the fastest storage layer and disks, promising improved I/O latency and bandwidth [21]. These characteristics make SCM especially attractive to be used as a memory extension for main memory intensive systems such as main memory-resident databases.

Main memory-resident databases are databases whose primary persistence is main memory, therefore also called in-memory databases (IMDBs). In-memory databases have recently been in the focus of database research [8, 11, 12] as well as in the focus of commercial database vendors as Microsoft [5], SAP [7], Oracle [15], or IBM [17]. While the first in-memory databases were optimized for transactional enterprise workloads (so-called Online Transaction Processing, OLTP), more recent approaches focus on mixed workloads. A mixed workload combines a transactional workload with a more complex and computation intensive analytical workload (so-called Online Analytic Processing, OLAP).

Observations of production enterprise systems have shown that data is kept over a period of five to ten years for regulatory or 'just-in-case' purposes. However, looking at the actual workload reveals that accesses are highly skewed towards small portions of the data, while the larger part remains rarely accessed or even untouched. While storing all data in *dynamic random-access memory (DRAM)* is viable when bandwidth requirements are high [2], storing irrelevant data on DRAM can be considered a waste of resources, as DRAM is expensive and limited in capacity. Consequently, one of the major research questions of this paper is "How to place less relevant data on an SCM tier in a mixed workload scenario with minimal performance impact?". This research topic of how to allocate data on different tiers is well known in the context of transactional workloads by tracking recently accessed tuples or blocks [4, 6]. But mixed workloads pose totally new challenges as analytical queries often access data that is of low relevance for the daily transactional business, but of high relevance for analytical tasks.

If the database is able to evict substantial parts of the database to secondary storage without sacrificing the performance advantages of in-memory databases, the total cost of ownership (TCO) can be reduced. Not only are large main memory-based server systems more expensive to acquire than their disk-based counterparts, they are also more

International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures - ADMS 2015, Kohala Coast, Hawaii, USA, August 31, 2015. http://www.adms-conf.org/

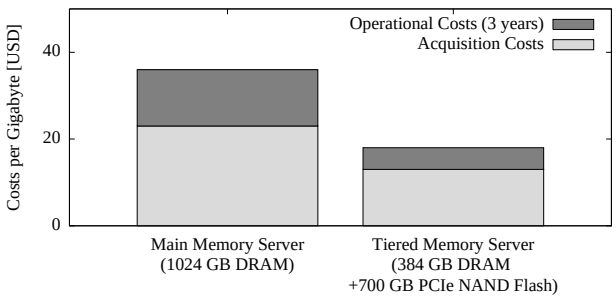


Figure 1: Cost calculations for an in-memory-only system and a tiered memory system (costs for a GB database storage).

expensive in terms of operational costs as large DRAM installations contribute a substantial part to the energy consumption [18] of a server.

Figure 1 shows an exemplary TCO calculation for two server systems (calculated using a configuration tool from a large server vendor). The main memory system has 1024 GB of main memory. The tiered memory system has 384 GB of main memory and a 700 GB PCIe-connected NAND flash SSD. Comparing the price per GB of database storage shows that the three-year TCO for the full main memory system is almost a factor of two more expensive.

In this paper, we will present a new approach called *Relevance-Based Partitioning* (RBP) using tiered memory. The approach improves memory utilization by placing less relevant data on SCM using the *EMC Memory Tiering API*[1] and is aware of mixed workload access patterns. Instead of a plain horizontal partitioning of a whole table, RBP is able to partition each column of a table individually, tuning the allocation as well as the data eviction policy for each column exactly to the workload. This way the partitioning of a table can reflect both the OLTP and the OLAP access patterns. Our concept is integrated into HYRISE [8], an open source research database.

In this paper, we will make the following contributions:

- Introduction of the SCM-optimized *EMC Memory Tiering API* (henceforth referred to as *EMT*) in Section 2 and evaluation of its performance compared with *malloc* and *mmap* under memory pressure for typical database access patterns in Section 3.
- In Section 4, we present a data tiering concept for HYRISE – called Relevance-Based Partitioning – that uses so-called *hot data views* to classify data as well as to optimize data accesses. The result is a transparent data tiering approach that places less frequently accessed data onto PCIe-connected NAND flash devices while hot data is pinned in main memory.
- Performance of the tiered memory approach is evaluated for a mixed enterprise workload in Section 5.

The remainder of the paper presents related work in Section 6, a discussion and future work in Section 7, and concludes with Section 8.

2. BACKGROUND

The EMT API provides an optimized software interface to access an SCM device as secondary storage by means of virtual memory. Bypassing the OS layer, the EMT's design goal is to provide a more predictable alternative to the standard Linux *mmap* and paging implementation in terms of access latency. This section discusses SCM in general, the EMC Memory Tiering API, and the system model of an in-memory database for mixed workloads.

2.1 Storage Class Memory Characteristics

In a traditional, disk-based database architecture, the engine performs its own read/write IO to the underlying storage layer that contains the database files. In order to avoid performance penalties due to that direct IO access, the database engine will typically employ one or more database buffer pools to hold recently used database blocks/pages. These buffer pools are typically resident in DRAM and the database engine will manage when pages need to be written to the underlying media to make room for new pages.

For in-memory databases, traditional buffer pools – which are responsible for a substantial part of the execution time of disk-based databases [9] – are no longer necessary. Even though there are direct IO to persistence layer to handle data changes and inserts it is expected that all data resides in volatile memory.

Today, applications do not address the physical memory directly but instead use the operating system to translate between the application's *virtual address space* and the system's *physical address space*. In this approach, every program has its own private address space and thus can run independently from other programs on the system. The memory is organized in *pages* of typically 4 kB and the translation between virtual and physical address space is done using a *page table*. This mechanism would theoretically allow an in-memory database system to extend its storage beyond the installed memory by using a swapfile on disk. In practice, however, the system suffers from unpredictable slowdowns due to the transparent execution of the page fault handler and swap subsystem.

In an SCM scenario, the desired functionality is to have multiple page caches of various sizes, page caches backed by multiple types of physical media, functional design improvement to the memory management that take next generation flash media into account, and a way for the application to interact with these improvements.

2.2 Storage Class Memory Tiering API

The *EMC Memory Tiering (EMT)* API allows the database software to have granular control over OS memory tiering, yet there is a fundamental re-write of the Linux virtual memory management sub-system beneath the API level that allows for this functionality. It provides an alternative to the Linux `mmap`, `msync`, runtime `malloc` and page fault handler implementations [19]. The EMT `mmap` promises better performance, more predictable access latencies, and additional control over the paging process. It is not intended as a general purpose page fault handler, but as an efficient interface to create SCM tiers to extend memory and for accessing and caching SCM tiers by means of virtual memory.

Rather than allocating physical pages from the entire memory when needed, EMT provides a facility to pre-allocate one or more system-wide fixed-size page caches. Appli-

cations control which page cache to use. This results in a more predictable execution time per process, because Linux no longer is managing a single system-wide page cache between competing processes. The EMT supports pluggable `mmap` and extensible page cache management policies. Two different policies for deciding which pages to evict from a cache are currently supported: a simple first-in, first-out (FIFO) principle and a more complex least recently used (LRU) mechanism. In addition, the application can tune the caching behavior by setting a low water level and an eviction size.

Each page cache maintains the availability of free physical pages via two settings: The *low water level* specifies a threshold for the free memory in a page cache below which an eviction is triggered and the *eviction size* determines the number of pages evicted in such an event. This eviction strategy attempts to ensure page slot availability upon page fault. The page fault latency is further reduced by bypassing the Linux virtual file system and directly accessing the storage device driver when combined with a compatible storage device.

The EMT supports coloring of individual pages to maximize page cache residency times and minimize the number of page faults. A page color (or temperature) is represented as a 16-bit integer, where higher values mean the page is accessed more frequently and should be kept in the page cache when possible. Individual pages may also be pinned which maintains residency. It is the applications responsibility to set the colors appropriately according to its access pattern. In addition to the explicit specification, the EMT tracks access to pages and dynamically adjusts page colors based on those statistics.

Furthermore, EMT employs a technique called *dynamic read ahead*, where it reads a number of subsequent pages starting from the faulting page, comparable to modern OS's read ahead mechanisms. In contrast to the OS's read ahead, EMT automatically adapts the number of read ahead pages to the applications access patterns. The algorithm starts reading ahead based on a given minimum value. Every successive read ahead will double the amount until the defined maximum value is reached, or it will get reset back to the minimum for any out-of-order major page faults.

These features promise better performance and control for accessing secondary storage in an in-memory database. This may form the basis of an effective memory tier containing colder data, where the classification of data (e.g. hot and cold) by the database is mapped onto page colors. The underlying EMT library can use this information as a hint for which data should be kept in memory and thus reduce the number of page faults.

2.3 In-Memory Databases for Mixed Workloads

The data tiering concepts presented in this paper are based on the HYRISE[2] database system [8]. HYRISE is an open source research database – sharing many concepts with SAP HANA – and is a column-oriented, in-memory database system designed for mixed enterprise workloads [7].

As opposed to row-oriented databases, columnar databases store all values of a column in a contiguous block. This layout exploits the locality principle of CPU caches in analytical queries. It eliminates gaps between the values of a column. Full column scans and aggregations on arbitrary columns can be performed very fast especially when the operation is split across multiple CPU cores.

A table in HYRISE is separated in two main data structures: The immutable *main store* contains the majority of the data, while the *delta store* (also differential buffer) contains recently changed data. The delta store is periodically merged with the main store, thereby creating a new main store that replaces the current. This separation allows to optimize the main store towards reading accesses and allows higher compression, while the delta store is optimized for modifying accesses. Both stores are dictionary encoded with the addition that the main store has an order-preserving dictionary for improved read performance. HYRISE as well as SAP HANA is an insert-only database using multiversion concurrency control (MVCC). Tuple updates implicitly result in the invalidation of a tuple and its reinsertion into the delta partition.

2.4 Characteristics of Mixed Workloads

Mixed workloads combine both the workloads of typical transactional OLTP systems as well as the workloads of analytical OLAP systems (e.g., data warehouses). Both workloads have very diverging characteristics. OLTP queries typically request rather small numbers of tuples and project many attributes. A typical system would be a sales system with an OLTP query receiving all items of a given sales order. Most OLTP queries select on primary key columns (at least partially) and rarely trigger scans on a table.

In contrast, OLAP queries require accessing many rows but few attributes. A typical analytical query would request the sum of sold items of the past six months. Such analytical queries project very few columns but trigger large scans on (multiple) tables.

Combining both workloads not only poses challenges to the database in general, but especially to the question of how to evict less relevant data, because the definition of relevant data differs for both workloads. While hot data for transactional workloads is usually a list of most recently accessed tuples, hot data for analytical queries are columns that are typically scanned or aggregated.

3. EMT PERFORMANCE EVALUATION

In order to obtain an overview of the performance impact of using the EMT API over OS functionalities such as `malloc` and `mmap`, this section presents two evaluations. First, different allocators are benchmarked with an increasing memory pressure. Second, the allocators are benchmarked with a varying access skewness.

The experiments presented in this section simulate a single table column (5 GB) that can be split into various partitions. For each partition, possible configuration options include different memory allocation strategies, workloads, access skewness, as well as the possibility to limit the physical memory available for the benchmark. The performance metrics used for the experiments include execution time and major page faults.

We run sequential and random access tests to reflect the access patterns of mixed workloads on a columnar database. The sequential access tests reflect typical patterns for analytical workloads. The random access tests reflect patterns of transactional workloads.

All tests have been executed on a *Hewlett-Packard ProLiant DL580 G7* machine with four *Intel Xeon X7560* CPUs

and 512 GB DDR3 memory at 1066 MHz. A *Micron P320* PCIe card with 350 GB of SLC-flash was used as SCM for cold data, either via the EMT API or configured as Linux swap space. The system ran on *Ubuntu Server* 12.04 with Kernel 3.2. For all tests, the default read-ahead settings of 128 kB have been used.

3.1 Increasing Memory Pressure

This first benchmark of the EMT API evaluates sequential and random reads with different main memory limitations. It provides a baseline representing the cached SCM performance without partitioning as well as without access skewness. The benchmark provides a side-by-side performance comparison of the EMT API, `malloc`, and `mmap`. The data vector consists of a single partition without coloring. During the benchmark, 50 sequential scans of the entire data set are performed, as well as 10,000,000 random accesses reading 4 kB each.

Figure 2a shows the total execution time of the sequential scans (y-axis) for all configurations. As expected, `malloc` and `mmap` perform equally due to the implementation of `malloc` as an anonymous `mmap` call in most Kernels. Both exhibit drastic performance losses by factors of 28-81× in the case where insufficient main memory is left (0% evicted data corresponds to unlimited memory), where the effect worsens with increasing memory pressure (x-axis).

The loss in performance is less distinctive for EMT, ranging around a 6.5× slowdown for all three evicted data ratios (20%, 33% and 50%) compared to unlimited memory.

The major page fault counts in Figure 2b confirm this observation: For `malloc` and `mmap`, lower limits lead to more page faults, causing the longer execution time. The results for EMT demonstrate the effects of the dynamic read ahead mechanism, which adjusts to read-access patterns.

Due to the strictly sequential access, EMT's LRU-cache acts like a window moving across the data, effectively degrading to a FIFO-queue (first in, first out). This explains the constant performance of EMT under memory pressure.

In case of unlimited memory, all three allocators perform similarly with major page fault counts tending towards zero.

Similar to sequential reads, `mmap` and `malloc` perform equally for random reads as well (Figure 2c). The execution times for `mmap` and `malloc` grow exponentially with increasing memory pressure in the tested range resulting in a slow down of up to a 120× at 50% evicted data compared to unlimited memory.

EMT outperforms the Linux page fault handler again, although the difference is less distinctive.

Since no access prediction is possible for fully randomized accesses, the number of page faults is expected to grow linearly with decreasing memory limit. Figure 2d confirms this expectation for all three allocators in the case of major faults. Furthermore, all three allocators exhibit similar amounts of major page faults, suggesting EMT's dynamic read-ahead mechanism was automatically disabled. The reason that EMT outperforms `mmap` and `malloc` while having the same number of page faults is the direct I/O bypassing the OS's virtual file system.

The results obtained from this test show a significantly improved performance of EMT compared to `malloc` and `mmap` under increasing memory pressure.

3.2 Performance for Skewed Accesses

The proposed approach for Relevance-Based Partitioning using EMT (see Section 4) builds on the assumption that accesses are skewed towards hot/relevant data. Typical examples are financial systems where data of the current fiscal year is of much higher relevance for transactional processing than previous fiscal years. In contrast, analytical accesses usually cover a large range of data (e.g., comparing the current fiscal year's performance with previous years). Our approach strives to statistically identify relevant data and partition a table accordingly (not part of this paper) in order to eventually prune the query processing to hot or warm partitions. Random reads and especially sequential scans are supposed to gain performance benefits while correct query results must be ensured. To measure the effect of such an access skewness and subsequent query pruning under increasing memory pressure we have benchmarked different configurations.

3.2.1 EMT and Partition Pruning

In order to better reflect real-world access patterns, the benchmark in Section 3.1 was extended to simulate a database workload that exposes skewness towards a small partition of hot (10%) and warm (30%) data of the 5 GB sized vector. The benchmark assumes that 80% of all random and sequential reads (partial scans) only access the hot data partitions. The remaining 20% of random reads access the warm partition. The remaining 20% of sequential reads have to scan the entire vector including the 60% cold data (i.e., all three partitions). This execution path is based on the assumption that the database is able to pin 80% of all column scan operations to the hot partition (see hot-only pruning in Section 4.1.1) and prune the other partitions.

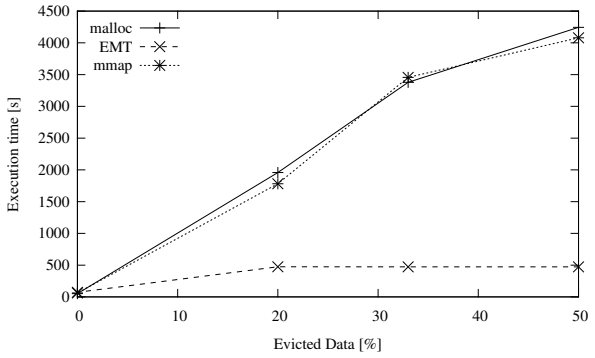
We compare three scenarios:

- All partitions allocated with *malloc*
- Warm and cold partitions allocated using EMT using coloring
- Warm and cold partitions allocated using EMT without coloring

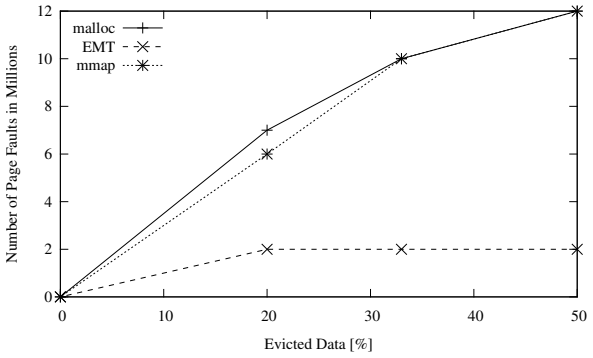
For the latter two configurations, the hot partition is allocated using *malloc*.

The configurations are again tested with different memory limits. The hot partition (10% of the data) is always allocated with malloc and pinned in main memory. The remaining main memory can be used as a cache. E.g., a memory limit of 50% with 10% hot data means that the cache size is 40% of the overall data.

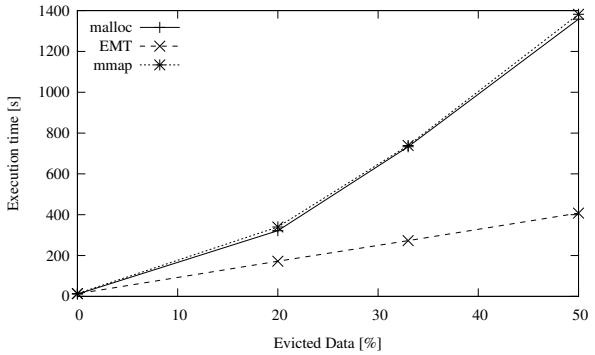
The execution times of the *sequential read* operation in Figure 3a show a significant performance benefit of EMT over *malloc* under memory pressure. The latter suffers from a 21-44× slowdown when less memory than data is available compared to unlimited memory. As for the test in Section 3.1, EMT's performance without coloring for all three actual memory limits is rather constant with a 3.4× slowdown compared to unlimited memory. This confirms the expected effect of query pruning for sequential reads, as the slowdown is considerably lower than the 6.5× slowdown from the test without query pruning in Figure 2a. When EMT is used with explicit coloring, performance improves, resulting in a 2.9× speedup for higher memory limits.



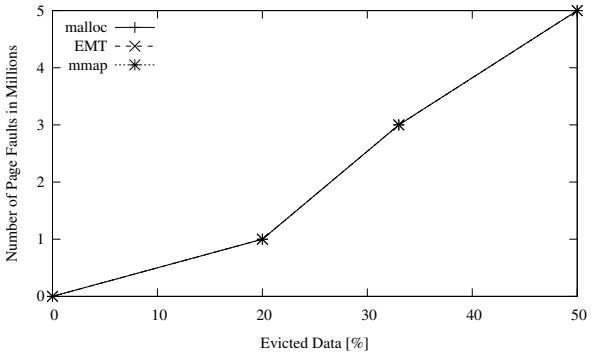
(a) Sequential Read: Execution Time



(b) Sequential Read: Major Page Faults



(c) Random Read: Execution Time



(d) Random Read: Major Page Faults

Figure 2: Results in terms of execution time ((a), (c)), and major ((b), (d)) page faults for the benchmark described in Section 3.1.

3.2.2 Impact of Access Skewness

In order to evaluate the impact of an access skewness towards the hot data partition, a setup with changing access distributions was performed. Again, execution times for sequential reads (Figure 4a) and random reads (Figure 4b) are compared with different shares of queries that can be pinned to the hot partition only (50%, 65%, 80%, and 95%), and different memory limits (30%, 60%, and 90%). All benchmarks assume a hot data share of 10% that is always allocated with malloc and pinned in main memory.

Figures 4a and 4b reveal that access distribution has a considerable impact on overall runtime for random as well as for sequential reads. The impact of the memory limit is less significant if the skewness towards hot data is high. The cache size can be reduced without impacting performance severely. The additional cache size that comes with a higher memory limit does not improve performance in Figure 4b. However, for sequential reads the cache size as well as the memory limit has a high impact when the query skewness towards hot data is low as the whole column needs to be processed. As shown in Figure 4a performance losses by low query skewness can be compensated by bigger cache sizes. If an application is able to prune accesses in 95% of the cases, memory limits can be set more aggressively.

3.3 Results

The presented results reveal that memory tiering using the standard Linux paging mechanism is clearly outperformed by EMT for scenarios where the amount of data grows beyond the available memory. In addition, the application has little control over the process which is a major drawback for applications that are aware of relevant and less relevant data.

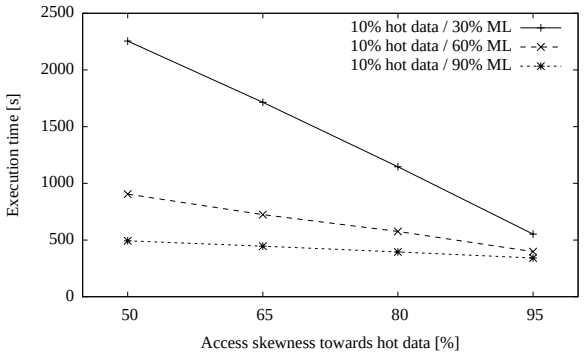
Besides EMT's performance gains through the dynamic read-ahead and virtual file system bypassing, the additional support for coloring can improve performance further, given that data accesses are skewed.

The results in Section 3.2.2 show that an in-memory database system using the EMT library can potentially handle data sets larger than the available main memory when skewness is high. Consequently, if the application – in our case HYRISE – pins 10% of hot data in memory and is able to limit ∼90% of the transactional query accesses to the hot partition, the resulting performance impact is within a factor of 2-3× of a full in-memory database. In Section 4, we will present an approach that provides such pruning rates.

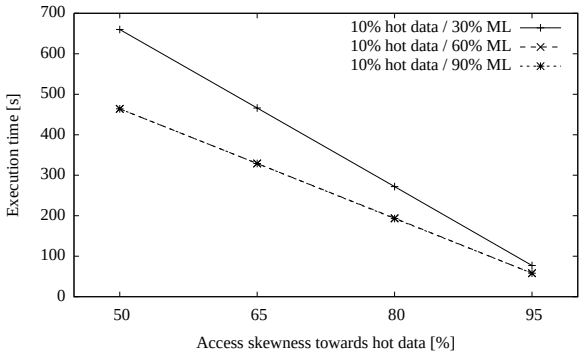
4. DATA TIERING IN HYRISE

This section describes the concept of data tiering for in-memory databases focussing on mixed workloads. First, we introduce the basic concept of data tiering for HYRISE in Section 4.1. Afterwards, we briefly outline how we implemented the concept in Section 4.2.

The data tiering concept is built on the idea of reorganizing and splitting tables into hot and cold data partitions based on workload relevance. Hot data comprises relevant data that is required to process the major portion of the workload. For optimal performance, hot data is allocated



(a) Sequential Reads



(b) Random Reads

Figure 4: Execution run times for sequential reads (a) and random reads (b) for changing memory limits (ML), hot-only access ratios, and hot/cold ratios.

using `malloc` while cold data is allocated on secondary storage using the EMT API. Our concept of data tiering is the periodical optimization of storage allocation and data structures that help HYRISE to optimize memory utilization and performance.

The most differentiating aspects of our approach is the individual, horizontal split of each column instead of applying a horizontal partitioning across all columns of a whole table. This partitioning scheme is explained in more detail in Section 4.1.2.

4.1 Data Tiering Concept

Tracing the relevance of single data blocks, tuples or even per data item is a considerable overhead during query processing, especially when many pages or tuples need to be processed during query execution. Instead of classifying fine-granular data packages, our tiering approach defines hot data for a table using a set of conjunct views, so called *hot data views (HDVs)*. Before a query is executed, it is matched against the corresponding HDVs. This operation called *tiering check* concludes if a query can be executed solely on the hot partition while still guaranteeing correct query results.

The foundation of our data tiering concept, are workload statistics on the usage of query templates and attribute bindings that are extracted asynchronously from the workload. These statistics are used to create the HDVs. In this paper we do not go into details about the workload tracing as well as parsing and assume, that these characteristics are already available to us.

4.1.1 Hot Data Views

As the name suggests, *HDVs* define hot data within a table. These views are conceptually similar to SQL views as they define a subset of a given table. Hot data views leverage patterns in the application workload, such as time-related, recurring query templates and binding attributes. Logically an HDV consists of a *view definition* and a range of *binding attributes* extracted from the workload. Once defined, HDVs are used for two purposes: First, the separation of hot and cold data in order to partition the data. Secondly the subsequent check of whether a query can be partition-pruned and only needs to be executed on the hot partition.

Listing 1 shows an exemplary HDV for the columns *ol_amount*, *ol_w_id*, *ol_delivery_d* which declares all values of these attributes with a delivery date starting from '2014-05-25' as hot. To create such an HDV, the binding attributes on predicate *ol_delivery_d* are constantly sampled and analyzed right before the *tiering run* that is explained in Section 4.2.

```
CREATE VIEW hot_data_view_001 AS
    SELECT ol_amount, ol_w_id, ol_delivery_d
    FROM orderline
    WHERE ol_delivery_d >= '2014-05-25'
```

Listing 1: Hot Data View OLAP.

```
CREATE VIEW hot_data_view_002 AS
    SELECT *
    FROM orderline
    WHERE ol_o_id in (88573, 87736, ...)
```

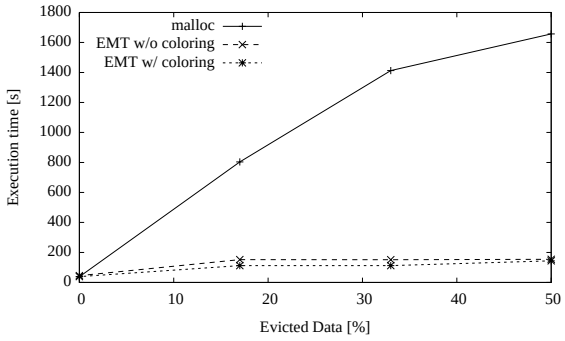
Listing 2: Hot Data View OLTP.

The number of HDVs that are necessary to classify the hot data of a table depends on the diversity (different selection and projection paths) of the workload.

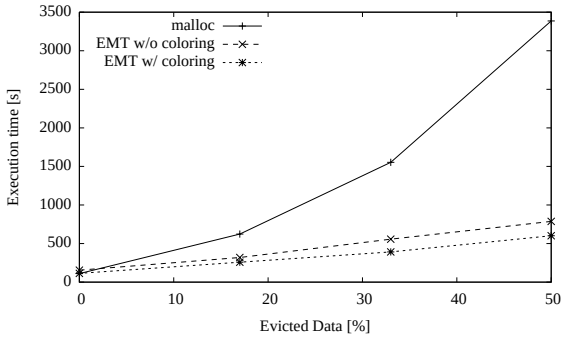
A hot data view is like a promise that all queries that match this view can be answered from hot data only. Scan and search operations on cold data can be pruned during query execution. The overall system load is supposed to be reduced and single query runtime is supposed to be reduced. Because of the insert-only, main / delta architecture of HYRISE (see: Section 2.3) update operations on cold data cannot break this promise because the delta partition is always part of query execution.

4.1.2 Hybrid Table Layout

Mixed workloads may include data access characteristics of typical OLTP as well as of OLAP workloads. OLTP queries select single or few tuples and have wide projections on many attributes. While they rely on the existence of corresponding indices in order to avoid full column scans and provide predictable response times, OLAP queries process large data ranges and require scanning of entire columns. However, OLAP queries only select and project on a low number of attributes.



(a) Sequential Reads



(b) Random Reads

Figure 3: Execution times for sequential (a) and random (b) read operations comparing *malloc* to EMT with and without coloring (80% queries skewed towards 10% hot data).

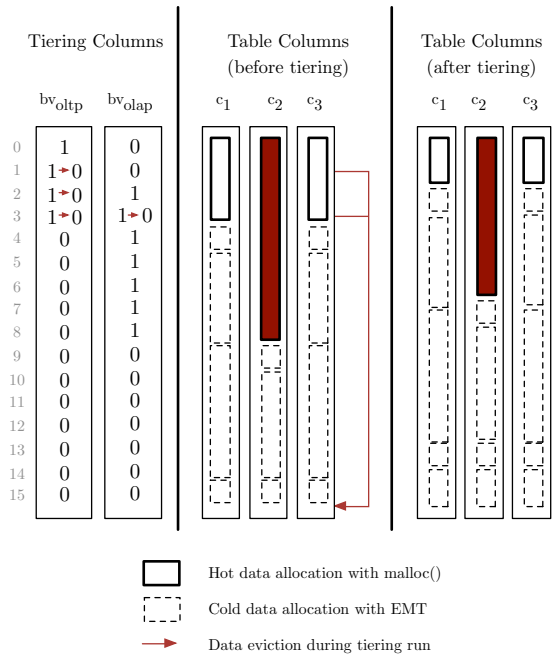


Figure 5: Hybrid table layout, showing one OLTP tiering column, one OLAP tiering column as well as hot and cold data partitions.

For this mixed set of access patterns, a strict horizontal or vertical partitioning is not optimal. Plain horizontal partitioning results either in a very low eviction rate when tuples required for analytical queries are considered entirely hot or in a high number of cold accesses when many tuples are evicted.

Listing 1 and 2 show two exemplarily HDVs. While Listing 1 is used to classify few columns over many tuples as hot, Listing 2 is used to classify few distinct tuples as hot. This way each column can be split individually into a single hot and multiple cold partitions as shown in Figure 5.

4.2 Implementation

In order to implement memory tiering in HYRISE several components had to be adapted or newly implemented. First of all, the data store needs to be aware of the secondary storage in order to allocate memory selectively for cold data column partitions. A new process called *Tiering Run* is introduced for managing the classification as well as physical sorting and partitioning of the data using HDVs. Before a query is executed the *Tiering Check* is used to evaluate, if pruning to hot data only is possible. Finally, a "hot-data-only" processing mode was implemented for operations such as the *Tiering Scan*, a modified full column scan to leverage query pruning.

Tiering Store. The Tiering Store exploits HYRISE's hybrid layout capabilities [8] to partition each column individually. That means the Tiering Store provides the capabilities to partition a column horizontally into a single hot partition and multiple cold partitions with each partition having its own memory allocator and colorization. While the hot partition still supports reallocation during a merge from delta to main, cold partitions have a fixed size, and only support invalidation of single tuples during update and delete operations. These invalidated tuples need to be collected and cleaned from time to time. However, this is not part of the current implementation as we optimize for a workload, with few updates and deletes [14].

Tiering Columns. Tiering Columns are bit-vectors (as shown in Figure 5) that are added to each table that is subject of tiering. They are used during the tiering run to mark tuples as being part of a hot data view. Each HDV requires its own Tiering Column. The example in Figure 5 illustrates a table that has one Tiering Column for its OLTP hot data view and one for its OLAP hot data view. Depending on the complexity of the workload there could be more than two Tiering Columns. Tiering columns are not subject of tiering, i.e., they are always kept completely in main memory.

Tiering Run. The Tiering Run is the core operation added to the existing system model. It is responsible for partitioning and allocating a given table. The data of a table is divided into one hot and several cold partitions using HDVs (one new cold partition during each Tiering Run). The Tiering Run decides which allocator to use for each partition and given that the allocator is EMT, the colorization of the allocation is adjusted.

The overall process consists of the following steps: First, evaluate recent workload statistics and extract hot data views.

Second, scan table using the hot data views updating Tiering Columns accordingly, and third, Sort table using Tiering Columns and allocate new cold partitions.

Because physically re-sorting a table and creating new cold partitions is a potentially expensive operation, we integrated the tiering run into the merge process [14] of HYRISE. Step one and two can be done offline without affecting the table's data structures. Then step three is being executed in conjunction with the delta merge. New tuples from the delta are considered hot. Tuples in a hot partition that are not marked as relevant by a Tiering Column, are moved into a new cold partition of the column allocated on secondary storage. Existing cold partitions do not need to be sorted. As a result, a continuous hot data partition arises for each column with as few cold data elements as possible.

With the ability to separate the tiering run into steps that are partially not time-critical and combine the final allocation with the already highly optimized merge process, the overall overhead is kept low.

Tiering Check. Before a plan operation of a query can be limited to hot-only data, it must be matched against a corresponding HDV. This is the most time-critical and defining step of the tiering concept.

In order to evaluate if the binding variables of the query embody a hot-only data selection, the HDVs are checked. In case the query can be matched to an HDV and the query's selection criteria matches the HDV's selection criteria, the query is partition-pruned to the hot partition only.

Tiering Scan. Fast full column scans are essential for analytical queries. Basic plan operations and especially the column scan need be adapted in order to utilize the Tiering Check. The Tiering Scan extends the full column scan plan operation. It is able to limit the scan and compare expressions to the hot part of a column only. Depending on the Tiering Check, only the hot partition or the entire column is processed.

5. DATA TIERING PERFORMANCE

In this Section, the implemented data tiering approach for HYRISE is evaluated using the well-known mixed workload *CH-benCHmark*.

5.1 Benchmark

The dataset and queries for the benchmark are derived from the *CH-benCHmark*, which is a combination of the TPC-C benchmark for transactional workloads and the analytical TPC-H benchmark [3]. We focus on the `ORDER_LINE` table containing the elements of each order. It is a table well-suited for tiering, since it is comparatively large and stores transactional data, most of which is rarely accessed in transactional workloads (e.g., closed orders from past years).

To evaluate our current implementation for mixed workloads, we have chosen four queries: three analytical queries of the CH-benCHmark and one transactional query accessing tuples via the primary key (see Listing 3 and Listing 4). The benchmark execution is divided into rounds, whereby in each round the three analytical queries are executed once, followed by 100 executions of the transactional query. As with the benchmarks in Section 3, we adapt hot data ratio, hot query ratio, and memory allocations.

```
-- CH-BenCHmark Query 1
SELECT    ol_number,
          SUM(ol_quantity) AS sum_qty,
          SUM(ol_amount) AS sum_amount,
          AVG(ol_quantity) AS avg_qty,
          AVG(ol_amount) AS avg_amount,
          COUNT(*) AS count_order
FROM      orderline
WHERE     ol_delivery_d > ?
GROUP BY ol_number
ORDER BY ol_number
-- Custom Query 1
SELECT SUM(ol_amount) AS sum_amount
FROM    orderline
WHERE   ol_w_id = ?
  AND   ol_delivery_d >= ?
-- Custom Query 2
SELECT SUM(ol_amount) AS sum_amount
FROM    orderline
WHERE   ol_i_id = ?
```

Listing 3: Analytical benchmark queries.

```
SELECT *
FROM    orderline
WHERE   OL_O_ID = ?
  AND   OL_D_ID = ?
  AND   OL_W_ID = ?
```

Listing 4: Transactional benchmark query.

The benchmarks are executed on a dataset with a TPC-C scale factor of 1000, resulting in a table size of about 19 GB. The database was configured to use 15 query execution threads and an EMT-managed secondary storage region of 110 GB accessed via a page cache with a size of 2.5 GB, a low water level of 260 MB, and an eviction size of 13 MB.

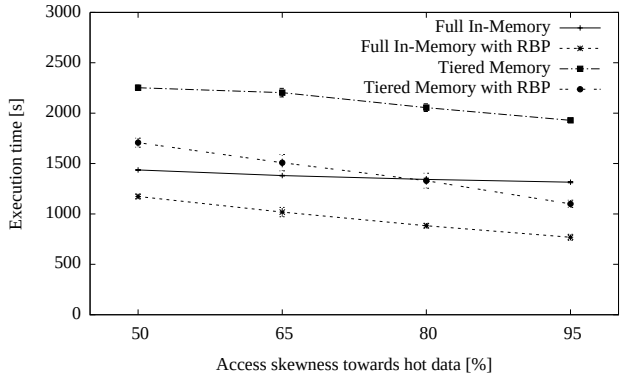
The benchmark was executed for 200 rounds using seven concurrent requests and 4×-parallelization within the plan operations. For each scenario, the runtime was measured using the `time`-utility shipped with typical Linux distributions for four different access skews (hot query ratio): 50%, 65%, 80%, and 95% (i.e., the share of queries that need to access the hot partition only). The remaining queries require data from the cold partition as well and thus trigger page faults for both tiered configurations. The benchmarks were executed on the same hardware as the benchmarks in Section 3.

5.2 Benchmark Configurations

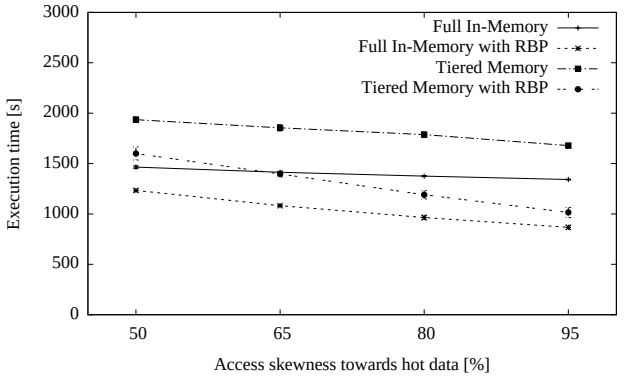
In order to compare the performance impact of first, the usage of the EMT API compared to a fully memory-resident database and second, the impact of hot and cold partitioning, we decided to evaluate the following four configurations:

Full In-Memory: represents the default HYRISE scenario in which all data is kept in main memory and is allocated via `malloc`.

Full In-Memory with RBP: similar to the *In-Memory* configuration, but with applied partitioning and query



(a) Size of Hot Partition: 10%



(b) Size of Hot Partition: 20%

Figure 6: Run time comparison for the four configurations presented in Section 5.2 with hot data sizes of 10% and 20% of the overall data.

pruning of the data tiering concept.

Tiered Memory: simulates the worst-case scenario where the system is forced to swap out parts of the data. For fairness, data is stored in EMT-managed memory due to its more efficient paging mechanism rather than default OS-managed memory.

Tiered Memory with RBP: the full implementation of data tiering concept combining `malloc` to allocate the hot data partition and EMT for the cold data allocation.

5.3 Results

Figure 6 shows the performance results for the HYRISE benchmarks. The status quo of HYRISE, i.e. the *Full In-Memory* configuration, sees only minor performance improvements for a growing access skew, while all *Tiered Memory* configurations see clear improvements as accessed data is more likely to be cached by EMT.

Most interestingly, given a certain skew, the *Tiered Memory with RBP* configuration outperforms the *Full In-Memory* configuration. Also, performance decreases only by a factor of 1.3-1.8× for 10% hot data and by a factor of 1.2-1.7× for 20% hot data compared to the *Full In-Memory with RBP* configuration.

6. RELATED WORK

Recently, several publications presented concepts for extending in-memory databases with a secondary storage. In contrast to the approach presented in this paper with a focus on mixed workloads, most approaches are aimed towards OLTP workloads. The following paragraph discusses the most recent work in that direction.

OLTP-optimized in-memory databases. Stoica et al. propose an extension to the in-memory database *VoltDB* (based on *H-Store* [11]) which rearranges data to move frequently accessed records to the beginning of the *mmap*'ed allocation [22]. The separation of hot and cold data is done using access statistics on tuple level and an offline analysis of these statistics. While the first part of the allocation – storing the frequently accessed data – is pinned in memory, the remainder is handled by the OS's paging mechanism. The approach uses a memory layout comparable to our approach but does not leverage different allocators and relies on *mmap*, which has been shown to be outperformed by the EMT API (see Section 3.1).

DeBrabant et al. describe a strategy called *Anti-Caching* implemented in the *H-Store* database in which the database system manages tuple movements and accesses to the secondary storage itself [4]. In contrast to [22], *Anti-Caching* directly handles data movements for a more predictable behavior over the OS's *mmap*. Tuple eviction is done based on a sampled LRU chain, comparable to our tracking of tuple accesses for the OLTP bit vector (see Section 4.1.2). Their approach outperforms a combination of *MySQL* and *Memcached* by a factor of ~7× for a modified and H-Store-optimized TPC-C benchmark.

Eldawy et al. present *Siberia* [6], which is an extension to the in-memory OLTP Engine *Hekaton* of the MS SQL Server [5]. Eviction is done tuple-wise based on workload traces [16]. In contrast to *Anti-Caching*, *Siberia* stores indices for hot data exclusively and uses adapted Bloom filters and adaptive range filters (ARFs [1]) to avoid accesses to the cold storage for OLTP workloads. In cases where no HDV is able to prune a query, additional indices as space-efficient bloom filters to further avoid cold accesses might further improve the performance of our implementation.

Columnar in-memory databases. Höppner et al. describe a hybrid-memory approach for the in-memory columnar database SAP HANA separating the columns of a columnar in-memory database into hot, warm, and cold columns [10]. While hot columns reside in main memory, warm columns are stored on a PCIe-connected solid state disk and are partially buffered in main memory. Cold columns are moved to hard disk. While this approach is theoretically suited for mixed workloads, it is not well suited for OLTP-dominated workloads as updates or full-width selects might require access to cold columns. Furthermore, tables which show a strong age-access correlation (i.e., only recent data is accessed) will not yield significant storage savings as several columns are kept in main memory entirely without any horizontal partitioning.

7. DISCUSSION & FUTURE WORK

Taking a look at recent enterprise systems [20] and general trends in data science, we believe that the focus on mixed workloads is very promising and not sufficiently covered in current research. However, projecting how upcoming workloads might look is challenging. At this point, more studies examining upcoming applications and their workloads are required.

As future work, several aspects are of high interest for us. The strong correlation of major page faults and execution time in all tests suggest that new emerging SCM devices will reduce the penalty of accessing cold data further. Therefore, we want to evaluate next generation SCM devices as discussed by Kim et al. [13]. Such new technologies continue to close the gap between non-volatile storage and disk, potentially making the presented approach in this paper increasingly viable and feasible.

Besides further research on surrounding topics of our approach as the automated generation of hot data views or improved merging of cold data, we also see the following issues especially worth looking into:

- Boot and recovery time: tiering large portions of data to SCM allows for much faster boot and recovery times for in-memory databases as only hot data has to be loaded.
- Partitioning and profiling of cold data: create multiple cold partitions and keep basic statistics for each partition to not only allow query pruning for hot/cold queries but also for cold-only queries.
- Further evaluation of the *coloring*-feature offered by the EMT API to introduce different levels of cold data based on their access probability.
- Explore the use of *EMT malloc* for finer-grained allocations and the use of *EMT persistent malloc* for persisting cold data structures.
- Tiered memory interactions in hyper-virtualized and containerized environments.

8. CONCLUSION

The presented results of data tiering for an in-memory database combined with the SCM-optimized tier management provided by the EMT API show that it is viable to place substantial parts of a data set without significantly sacrificing performance, even for mixed workloads. Furthermore, using memory tiering with an in-memory database can improve memory utilization and thus reduce costs.

In fact, the usage of an SCM-optimized API to efficiently access tiered storage not only improves performance compared to OS functionality such as *mmap* but also introduces a clean separation of concerns. This way the database is responsible for classifying data relevance and efficient query pruning, while the tiered memory API is responsible for handling storage accesses and efficient memory tier management.

We think that modern in-memory databases can greatly benefit from the approach presented here, both from a workload-aware partitioning based on data relevance as well as an optimized handling of an SCM-based memory tier. Looking at recent developments in the area of storage technology as presented by Kim et al. [13] leads us to the conclusion that tiered memory will become increasingly important for main memory-resident applications as the performance gap between main memory and non-volatile storage increasingly shrinks.

9. REFERENCES

[1] K. Alexiou, D. Kossmann, and P. Larson. Adaptive range filters for cold data: Avoiding trips to siberia. *PVLDB*, 6(14):1714–1725, 2013.

[2] M. Boissier, C. Meyer, M. Uflacker, and C. Tinnefeld. And all of a sudden: Main memory is less expensive than disk. In T. Rabl, K. Sachs, M. Poess, C. K. Baru, and H. Jacobsen, editors, *Big Data Benchmarking - 5th International Workshop, WBDB 2014, Potsdam, Germany, August 5-6, 2014, Revised Selected Papers*, volume 8991 of *Lecture Notes in Computer Science*, pages 132–144. Springer, 2014.

[3] R. Cole, F. Funke, L. Giakoumakis, W. Guy, A. Kemper, S. Krompass, H. Kuno, R. Nambiar, T. Neumann, M. Poess, K.-U. Sattler, M. Seibold, E. Simon, and F. Waas. The mixed workload CH-benCHmark. In *Proceedings of the Fourth International Workshop on Testing Database Systems*, DBTest '11, pages 8:1–8:6, New York, NY, USA, 2011. ACM.

[4] J. DeBrabant, A. Pavlo, S. Tu, M. Stonebraker, and S. B. Zdonik. Anti-caching: A new approach to database management system architecture. *PVLDB*, 6(14):1942–1953, 2013.

[5] C. Diaconu, C. Freedman, E. Ismert, P. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: SQL server's memory-optimized OLTP engine. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, pages 1243–1254, 2013.

[6] A. Eldawy, J. J. Levandoski, and P. Larson. Trekking through siberia: Managing cold data in a memory-optimized database. *PVLDB*, 7(11):931–942, 2014.

[7] F. Färber, N. May, W. Lehner, P. Große, I. Müller, H. Rauhe, and J. Dees. The SAP HANA database – an architecture overview. *IEEE Data Eng. Bull.*, 35(1):28–33, 2012.

[8] M. Grund, J. Krüger, H. Plattner, A. Zeier, P. Cudré-Mauroux, and S. Madden. HYRISE - A main memory hybrid storage engine. *PVLDB*, 4(2):105–116, 2010.

[9] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker. OLTP through the looking glass, and what we found there. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, pages 981–992, 2008.

[10] B. Höppner, A. Waizy, and H. Rauhe. An approach for hybrid-memory scaling columnar in-memory databases. In *International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures - ADMS 2014, Hangzhou, China, September 01, 2014.*, 2014.

[11] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. B. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi.

H-store: a high-performance, distributed main memory transaction processing system. *PVLDB*, 1(2):1496–1499, 2008.

[12] A. Kemper and T. Neumann. Hyper: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In S. Abiteboul, K. Böhm, C. Koch, and K. Tan, editors, *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany*, pages 195–206. IEEE Computer Society, 2011.

[13] H. Kim, S. Seshadri, C. L. Dickey, and L. Chiu. Evaluating phase change memory for enterprise storage systems: a study of caching and tiering approaches. In B. Schroeder and E. Thereska, editors, *Proceedings of the 12th USENIX conference on File and Storage Technologies, FAST 2014, Santa Clara, CA, USA, February 17-20, 2014*, pages 33–45. USENIX, 2014.

[14] J. Krüger, C. Kim, M. Grund, N. Satish, D. Schwalb, J. Chhugani, H. Plattner, P. Dubey, and A. Zeier. Fast updates on read-optimized databases using multi-core CPUs. *PVLDB*, 5(1):61–72, 2011.

[15] T. Lahiri, M. Neimat, and S. Folkman. Oracle timesten: An in-memory database for enterprise applications. *IEEE Data Eng. Bull.*, 36(2):6–13, 2013.

[16] J. J. Levandoski, P.-A. Larson, and R. Stoica. Identifying hot and cold data in main-memory databases. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, pages 26–37. IEEE, 2013.

[17] J. Lindström, V. Raatikka, J. Ruuth, P. Soini, and K. Vakkila. IBM solidDB: In-memory database optimized for extreme speed and availability. *IEEE Data Eng. Bull.*, 36(2):14–20, 2013.

[18] K. T. Malladi, B. C. Lee, F. A. Nothaft, C. Kozyrakis, K. Periyathambi, and M. Horowitz. Towards energy-proportional datacenter memory with mobile DRAM. *SIGARCH Comput. Archit. News*, 40(3):37–48, June 2012.

[19] A. Michaud. *EMC Memory Tiering API Software Functional Specification (SFS)*. EMC, Inc., 2013.

[20] H. Plattner. The impact of columnar in-memory databases on enterprise systems. *PVLDB*, 7(13):1722–1729, 2014.

[21] M. K. Qureshi, V. Srinivasan, and J. A. Rivers. Scalable high performance main memory system using phase-change memory technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 24–33, New York, NY, USA, 2009. ACM.

[22] R. Stoica and A. Ailamaki. Enabling efficient os paging for main-memory OLTP databases. In *Proceedings of the Ninth International Workshop on Data Management on New Hardware*, DaMoN '13, pages 7:1–7:7, New York, NY, USA, 2013. ACM.

Notes

[1]This publication makes reference to EMC proprietary technology and any use, copying, and distribution of any EMC software or technology that is described in this publication requires an applicable license from EMC.

[2]HYRISE project: `https://github.com/hyrise/hyrise`