HeuristicDB: A Hybrid Storage Database System Using a Non-Volatile Memory Block Device

Jinfeng Yang
University of Minnesota
yang3116@umn.edu

Bingzhe Li
Oklahoma State University
bingzhe.li@okstate.edu

David J. Lilja
University of Minnesota
lilja@umn.edu

Abstract

Hybrid storage systems are widely used in big data fields to balance system performance and cost. However, due to a poor understanding of the characteristics of database block requests, past studies in this area cannot fully utilize the performance gain from emerging storage devices. This study presents a hybrid storage database system, called HeuristicDB, which uses an emerging non-volatile memory (NVM) block device as an extension of the database buffer pool. To consider the unique performance behaviors of NVM block devices and the block-level characteristics of database requests, a set of heuristic rules that associate database (block) requests with the appropriate quality of service for the purpose of caching priority are proposed. Using online analytical processing (OLAP) and online transactional processing (OLTP) benchmarks, both trace-based examination and system implementation on MySQL are carried out to evaluate the effectiveness of the proposed design. The experimental results indicate that HeuristicDB provides up to 75% higher performance and migrates 18X fewer data between storage and the NVM block device than existing systems.

Keywords

Non-volatile memory block device, Hybrid storage database management system, relational database management system

ACM Reference Format:
Jinfeng Yang, Bingzhe Li, and David J. Lilja. 2021. HeuristicDB: A Hybrid Storage Database System Using a Non-Volatile Memory Block Device. In *The 14th ACM International Systems and Storage Conference (SYSTOR '21), June 14–16, 2021, Haifa, Israel.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3456727.3463774

SYSTOR '21, June 14–16, 2021, Haifa, Israel

ACM ISBN 978-1-4503-8398-1/21/06…$15.00
https://doi.org/10.1145/3456727.3463774

1 Introduction

As an I/O-intensive application, relational database management system (RDBMS) performance is usually limited by laggard storage devices [27–29]. Current researches [49, 50] show that if an NVM block device is used as a persistent storage for RDBMS, I/O time is no longer a bottleneck in many cases. However, this is an expensive solution. Because of the high cost of NVM block devices, it might be impossible to provide an adequate number to preserve the entire database in real time. In this circumstance, building a hybrid storage database system that uses an NVM block device as a block cache for conventional storage devices becomes a cost-efficient way to improve RDBMS performance.

Directly employing conventional replacement algorithms (e.g. LRU, LFU, ARC [37]) designed for the database buffer pool caching in a hybrid storage database system may not be suitable. In a database system, all required pages must be read into the buffer pool for processing, thus admitting the accessed pages does not incur an extra cost. On the contrary, in hybrid storage architecture, as a page is read from the storage into the database buffer pool, it is a candidate for admission into the block cache. Admitting a page that will not be re-accessed in the future pollutes the limited block cache space. This leads conventional replacement algorithms to deliver suboptimal solutions for hybrid storage systems.

In addition, several factors limit the performance of previous hybrid schemes that are based on using a NAND solid-state drive (SSD) as a block cache for hard disk drives (HDD) [18–20, 36] when trying to directly replace the SSD with NVM devices. First, these prior schemes do not take into account the different behaviors of the NVM devices, such as their much higher read-write bandwidth [23, 24, 49] compared to NAND SSD, and the NVM device's elimination of write-driven garbage collection. Second, by ignoring the unique characteristics of database block requests, previous systems [18–20, 36, 54] produce lower cache hit ratios and increase the migration of many non-reusable pages from the storage subsystem to the block cache. Since the NVM block devices have only limited endurance [21, 22, 26, 40], frequently admitting these types of non-reusable pages into the NVM uses up its lifespan without delivering any performance benefit. Finally, without an efficient demotion mechanism for cached pages, prior work suffered severe performance degradation when

the database workload access pattern changed. As a result of these differences between NVM and SSD devices, a new approach is needed for using NVM devices as a block cache in a hybrid system. We call our new approach HeuristicDB. It is an additional component for a database system that uses an NVM block device as an extension of the database buffer pool.

By considering the performance behaviors of the NVM block device and the characteristics of database block requests, a set of heuristic rules that associate database (block) requests with the appropriate quality of service for the purpose of caching priority are proposed to guide HeuristicDB in managing database block requests more efficiently. To support the system implementation of the proposed rules, four programs, including *a query profile queue*, *an eviction and demotion (EV) program*, *a table access pattern detector*, and *a page placement controller* are developed. The table access pattern detector prioritizes each table or index required by a query by monitoring its access behavior. The page placement controller is used to redirect each database block request either to the NVM block device or to storage based on the reusability of the accessed page and the priority of the table or index to which the page belonged. Finally, to maintain high performance under changing database query access patterns, a low-overhead query profile queue and an EV program adaptively adjust the priorities of both the database tables and indices and the cached pages.

HeuristicDB is empirically assessed under different circumstances using an OLAP benchmark, TPC-H [1], and an OLTP benchmark, TPC-E [2]. Both trace-based examination and system implementation are conducted to test the cache hit ratio, data migration size between storage and the NVM buffer pool, execution time, and transaction rate of HeuristicDB and six other baseline algorithms. The experimental results show that HeuristicDB provides up to 75% higher performance and migrates 18X fewer data between storage and the NVM block device than existing systems.

2 Backgrounds and Related Work

This section presents background information on the NVM block device and the hybrid storage database system.

2.1 NVM block device

With DRAM-like performance and disk-like persistency and capacity, non-volatile memory such as phase-change memory (PCM) [17, 33, 41, 46], conductive bridging RAM (CBRAM) [25, 30, 45], and 3D Xpoint [23, 43, 49] is becoming critical importance for various data-intensive applications. The NVM serves as either main memory through a memory bus [7], known as an NVM DIMM [52], or persistent storage through a PCIe interface [53], known as an NVM block device [21]. The NVM block device has several advantages. First, compared with NAND SSDs, the NVM block device delivers orders of magnitude better read-write performance [49]. Second, because NVM supports in-place update, there is no need for garbage collection [9, 47, 49] which usually causes significant performance degradation for NAND SSDs. Third, the NVM block device inherits the PCIe interface developed for the NAND SSD and can be directly controlled by all existing systems without requiring any hardware updates. In contrast, the NVM DIMM works only when a specific processor is provided [10, 24, 48]. Considering all these advantages, this work is focused on using an NVM block device in a hybrid storage database system.

2.2 Related Work & Limitation

Many studies have investigated hybrid storage architectures in the past [3, 34, 38, 42, 44, 51]. Designing a hybrid storage system with an appropriate replacement algorithm to accelerate database applications is an important research topic. Canim et al. [19] developed a temperature-aware caching (TAC) algorithm that used a NAND SSD as an extension of the database buffer pool. During query processing, only randomly accessed pages from hot regions of HDD are admitted. However, its performance is affected by multiple tunable parameters, such as the region size, band size, and the number of reset accesses. In addition, because pages within the same region have different access frequencies, a rarely accessed page inside a hot region may be mistakenly cached into the NAND SSD, resulting in performance degradation. Do et al. [20] proposed a lazy cleaning (LC) method for dealing with pages exited from the database buffer pool. The LC method manages the block cache device using the LRU-2 [39] replacement algorithm. However, because the LRU-2 algorithm's performance is highly correlated with reuse distance [54], the LC method delivers even worse performance than TAC in some cases.

Developing an appropriate table placement algorithm is another approach to hybrid storage database system design. Through profiling database workloads or extracting information from the database query execution plan tree, the tables or indices that generate the most random I/Os are placed in the NAND SSD to improve database performance [18, 36]. The shortcomings of this type of approach are obvious. First, because replacement decisions are made based on the granularity of an entire table or index, once the size of the target is larger than the capacity of the NAND SSD, the hybrid storage system delivers zero performance gains. Second, because the table placement schemes cannot always accurately predict the access pattern of database tables or indices for the incoming query, they encounter misplacement issues.

Recently, multiple approaches for DRAM-NVMDIMM-SSD hybrid systems have been proposed [11, 31]. In these systems, the NVM DIMM serves as a main memory, and the

database application can directly access the data inside it without copying it into the DRAM. Those schemes were evaluated using NVM emulators. However, a recent study [48] has shown that the direct access performance of a real NVM DIMM device drops significantly as the thread count increases. As a result, the prior DRAM-NVMDIMM-SSD hybrid system could encounter a performance degradation for multi-threads applications such as RDBMS. In contrast, our work employs an NVM block device as an extension of the database buffer pool, and the data within it must be copied to the DRAM for producing query results to deliver scalable performance. Thus, our work is more suitable for multi-threaded database applications.

3 Design Principles & Heuristic Rules

This section discusses the characteristics of database block requests, the design principles behind the HeuristicDB, and the proposed Heuristic rules.

3.1 Characteristics of Database Block Requests

This subsection explains the characteristics of database block requests from the following two aspects.

Data Type: Three major data types are considered, including regular table, index, and page. A *table* is a collection of data elements. An *index* is a data structure [32, 35] that is developed to retrieve data quickly without accessing all data elements inside a table [13]. A *page* is the smallest unit that is accessible inside a table or index.

Access Behavior: This refers to the access pattern of a table or index during querying. After a query has been optimized [14, 15, 32] and sent into the execution engine, the access pattern of each required table or index is determined. The possibilities include (1) *sequential access*: when no index is used for a query, the database engine scans the target table sequentially; (2) *random access*: when an index is provided, the database engine retrieves the matching data from the table using the address inside the corresponding index. Both index and table are accessed randomly; and (3) *repeated access*: when multiple operators access a table or index during a query, some pages inside that table or index are accessed repeatedly. Detailed discussions of the access behaviors of tables and indexes during querying, along with trace-based characterizations, are provided in [49, 50].

3.2 System Design Principles

This study aims to leverage the NVM block device as an extension of the database buffer pool to boost database application performance. By considering the characteristics of database block requests and the performance behaviors of the NVM block device, the goal is approached from the following three directions.

Guarantee reusability: For the hybrid storage system, a performance gain can be delivered only when a hit occurs on the NVM buffer pool. Hence, the reusability of an accessed page is the main factor to consider for page admission. In this work, the tables and indices required by a query or queries are prioritized based on their reusability. Only pages from a high-priority table or index are admitted into the NVM buffer pool. A table or index is defined as reused either when some pages inside that table or index are re-accessed during query execution or when that table or index is also required by subsequent queries.

Adaptive demotion or eviction is required: How long a page stays in the NVM buffer pool is another key factor affecting the performance of a hybrid storage system. For a database application, the pages inside a table or index have a chance to be accessed only when that table or index is required by a query. Because the tables or indices required by the previous query may no longer be required by the subsequent query, and the set of hot pages inside the same table or index varies as the queries change. Keeping (hot) pages required by the previous query but are rarely accessed by the new incoming query in the NVM buffer pool reduces the performance of a hybrid storage database system. Hence, it is reasonable to adaptively adjust the priorities of database tables, indices, and cached pages before running a new query.

Performance benefit for caching a page in the NVM buffer pool: The NVM block device provides unprecedented read-write performance compared with conventional storage devices (e.g. NAND SSD). Hence, it is preferable to cache all block requests, including sequential read, random read, sequential write, and random write, with high reuse probability in the NVM buffer pool to deliver performance gains.

3.3 Heuristic Rules for a Query

Based on the characteristics of database block requests and the design principles discussed above, a set of heuristic rules is presented that associate database (block) requests with the appropriate quality of service for the purpose of caching priority. This subsection considers priority assignments for a single query. Priority assignments for concurrent queries will be discussed in the next subsection.

3.3.1 Sequential Requests

When a table is accessed sequentially, because all pages inside that table are accessed once, caching that mass of accessed pages delivers no performance benefit and pollutes limited space in the NVM buffer pool.

RULE 1: *A page belonging to a sequentially accessed table is not qualified for admission into the NVM buffer pool if it is read from storage.*

3.3.2 Random Requests

When a table or index is randomly accessed, only matching pages belonging to that table or index are fetched from storage. Because the number of these matched pages usually is extremely small, the cache pollution issues caused by admitting those matched pages are limited. On the contrary, those pages can deliver performance benefits once they are reused in subsequent queries.

RULE 2: *Pages from a randomly accessed table or index are qualified for admission into the NVM buffer pool.*

3.3.3 Repeated Requests

When multiple operators access a table or index, some pages inside that table or index will be accessed more than once [49, 50]. Admitting those pages that are accessed more than once guarantees performance gains.

RULE 3: *If a table or index is repeatedly accessed, it is preferable to cache the pages inside that table or index.*

RULE 4: *The page replacement happens unless the temperature of the accessed page is higher than the coldest one inside the NVM buffer pool.*

This design guarantees that the pages in the NVM buffer pool are the most profitable ones during query. Moreover, it avoids unnecessary and frequent page replacements that consume substantial power and shorten the lifespan of the NVM block device.

3.3.4 Write Requests

A block write request(s) is generated when a temporary table is created or an UPDATE/INSERT operation is required.

A temporary table is a special type of table for storing intermediate data sets generated during query execution. The pages within a temporary table will be read later to produce the final query results. Considering its reusability, the proposed design place a temporary table in the NVM buffer when it is created.

UPDATE and INSERT are the two most frequently used operations in OLTP workloads. They are used to renew or add new data from or into a table or index. Considering the unprecedented write performance of the NVM block device, in the proposed design, both updated and inserted pages are buffered into the NVM buffer pool.

In summary, the rule for database write requests is:

RULE 5: *All write requests are placed in the NVM buffer pool and flushed into storage asynchronously later if needed.*

3.3.5 New Query Requests

The tables or indices required and the access behavior of each table or index vary as queries change. To admit incoming hot pages to deliver performance acceleration, demotion or eviction of cached pages is required when a new query is submitted. The specific action depends on two factors: (1) the performance gains that have already been delivered from cached pages in the past; and (2) the existence of hot tables or indices is required by both the previous and the newly submitted queries.

RULE 6:

1. When a new query is submitted, an eviction of cached pages is required if no performance benefit has been provided in the past from cached pages and if one or more hot tables or indices are required by both the previous and the newly submitted queries. Otherwise, the cached pages are demoted.

2. If an eviction is preferred, pages inside each hot table or index are allowed to be admitted into the NVM buffer pool during the execution of the new query, regardless of their access behavior.

The goal of eviction is to admit pages from one or more high-reusability tables or indices by evicting pages that do not provide a performance gain inside the NVM buffer pool. Hence, in the proposed system, eviction can be triggered only when two conditions are satisfied: (1) no performance benefit has been provided by cached pages in the past; and (2) there exist one or more hot tables or indices that provide clues about tables or indices with high reusability.

3.4 Heuristic Rules for Concurrent Queries

The scenario of concurrent queries happens while a new query is submitted when other query(s) is running. From the storage layer's perspective, the block level characteristics of concurrent queries are similar to the single query's. Both of them are accessing the required tables or indices by sending block requests to storage. Hence, the priority assignment rule for concurrent queries is much like those used in single query execution.

RULE 7:

1. When a query is submitted while one or more other queries are still running, a priority adjustment (i.e. demotion or eviction) of the cached pages is required. The detailed criteria are the same as in Rule 6.

2. The access behavior of a table or index may change after a new query is submitted and is running concurrently with others. Hence, an access behavior re-verification is required for all tables and indices required by concurrent queries.

3. When multiple queries run concurrently, the quality of service for database block requests follows Rules 1 to 5.

4 HeuristicDB Overview

This section provides an overview of HeuristicDB. To support implementation of the heuristic rules in a real system, four programs, including *a query profile queue*, *an eviction/demotion (ED) program*, *an access pattern detector*, and *a page placement controller*, are developed. The query profile queue and the ED program are used to adjust the priorities of cached pages and detect hot tables or indices when a query is submitted to a database server according to Rules 6 and 7. The access pattern detector probes the access behavior of a

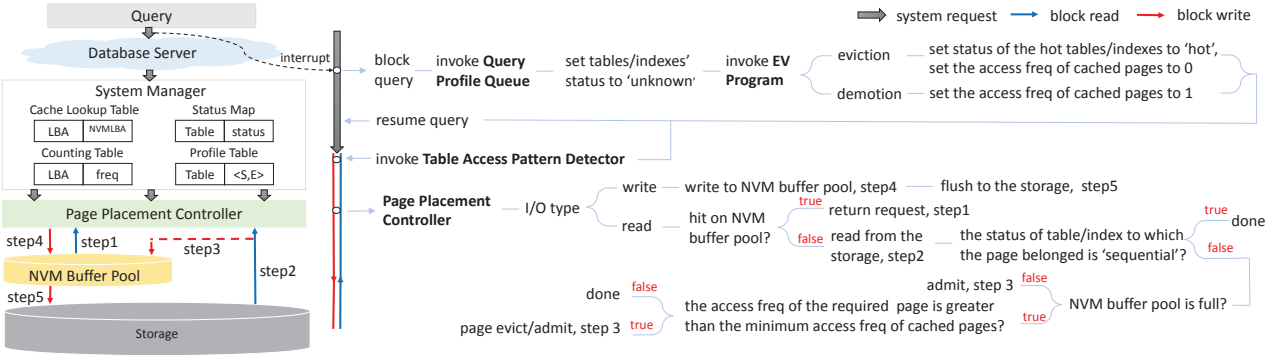


Figure 1: HeuristicDB overview. The left part shows the internal architecture of HeuristicDB, and the right part explains its working processes.

required table or index during query execution. Using the detected information, the page placement controller redirects database block requests either to the NVM buffer pool or to storage according to Rules 1 to 5.

4.1 Metadata Management

In the system, several hash tables are created in memory, as shown in the left part of Fig. 1, to facilitate page placement.

Cache lookup table: Like existing cache algorithms, this helps to look up cached pages. Each item inside this cache lookup table is defined as ⟨LBA, NVMLBA⟩, where LBA is the logical block address of a cached page in storage and is used as a key. The value, NVMLBA, is the logical block address of the cached page in the NVM buffer pool.

Counting table: the counting table is included in the proposed system to avoid recounting the access frequency of a hot page once it has been evicted from the NVM buffer pool. It only records the page identifier of each accessed page and its corresponding access frequency. Each item in the counting table is defined as ⟨LBA, freq⟩.

Status map: this is used to indicate the access behavior of each table or index during a query. An item within this hashmap is defined as ⟨T, Status⟩, where T is a table or index name. During query processing, the status of each table or index is determined as one and only one of the following: *unknown*, *sequential*, *random*, *repeated*, and *hot*.

Profile table: This table is used to facilitate system lookup of items belonging to an accessed page. Each item within this profile table is defined as ⟨T, V⟩, where T is a table or index name and V contains two tuples, ⟨startLBA, endLBA⟩. These two tuples represent the starting and ending LBAs of the corresponding table or index in storage. Note that when content is added/removed to/from a table or index, the address of that content should also be updated consequently.

4.2 System Overview

The left part of Fig. 1 shows the architecture of HeuristicDB. In the system, a database application is running on top. The NVM block device is used as an extension of the database buffer pool. All database tables and indices are preserved permanently in storage.

The right part of Fig. 1 presents the working processes of HeuristicDB. Every time that a query is submitted to a database server, an interrupt signal is generated to invoke *query profile queue* and *EV program*. By evaluating the performance benefit delivered by the cached pages and detecting the existence of hot tables or indices required by both past and newly submitted queries, multiple further operations are triggered. If no performance benefit was delivered in the past and there exists hot tables or indices, all cached pages are set as eviction candidates. The status of the tables or indices required by those two consecutive queries is set to *hot*, and those of the remaining tables or indices required by the newly submitted query are set to *unknown*. Otherwise, a demotion operation is required. Meanwhile, the status of all tables or indices required by the newly submitted query is set to *unknown*. Once the query profile queue and the EV program have terminated, query processing can be resumed. At the same time, *the table access pattern detector* is triggered to run asynchronously to detect the access behavior of each required table or index from the block layer.

In HeuristicDB, *the page placement controller* is assigned to redirect database block requests to their proper destination by following heuristic rules. When a database block request is generated, its corresponding access frequency is updated in *the counting table*. For a block write request, it is preferably written to the block cache, as step 4 in Fig. 1, and then lazily updated to the storage later, as step 5. For a block read request, the NVM buffer pool is queried first. If the required page has been cached, the page placement controller returns the request immediately, as step 1. Otherwise, a cache miss has occurred, and the required page is fetched from storage, as

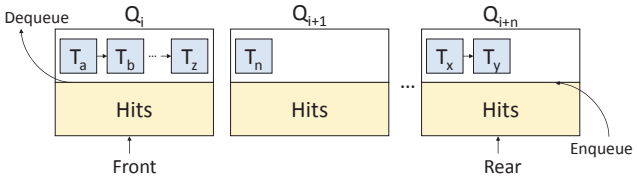


Figure 2: Query profile queue.

step 2. Meanwhile, by combining the collected information, the page is assessed to determine whether it is worth caching in the NVM buffer pool. If the NVM buffer pool is not full, the page can be admitted only if the access behavior of the table or index to which this page belonged is not *sequential*. Otherwise, in addition to the above condition, the access frequency of the required page must be higher than the minimum access frequency for cached pages for it to be admitted into the NVM buffer pool, as in step 3.

5 System Implementations

This section presents the detailed system implementation of the proposed four mechanisms in HeuristicDB.

5.1 Query Profile Queue

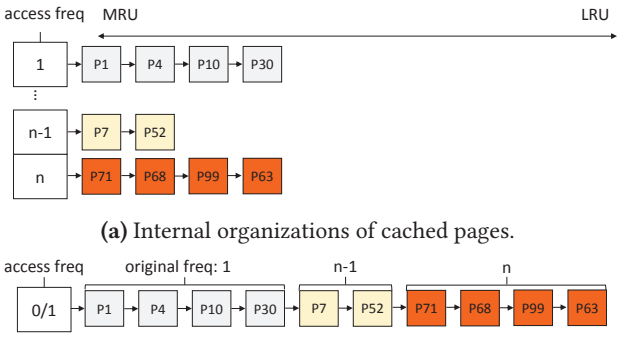
The query profile queue is developed to analyze the profiles of past and newly submitted queries and send either an eviction or a demotion request to the EV program according to Rules 6 and 7.

A queue data structure, as shown in Fig. 2, forms part of the proposed system. Each item in the queue contains two main types of variables used for query profiling: (1) a table linked list, which is used to record the tables or indices required by the corresponding query; and (2) a performance variable, which is called hits and is used to record the number of hits since the query has run.

Whenever a query is submitted, a query profile block is inserted at the rear of the queue. The initial value of the performance variable (i.e., hits) is zero. Then, the previous query's profile information is extracted from one block ahead of the rear of the queue. If the value of the performance variable inside that block is zero and there exists a table or index required by both the previous and the newly submitted queries, an eviction request is generated. The status of the specific table or index required by both queries is set to *hot*. Otherwise, a demotion request is generated. Because only information about the previous and newly submitted queries is needed, we set the size of the profile queue to two. In addition, because multiple queries can run concurrently for a certain time, the number of cache hits during that time is counted in the profile block of the last submitted query.

5.2 Eviction and Demotion Program

The EV program is triggered after an eviction or a demotion request is received. Before explaining its working



(a) Internal organizations of cached pages.

(b) Access freq equals to 0 for eviction and 1 for demotion.

Figure 3: Internal organizations of cached pages and EV operations demonstration.

mechanism, this subsection provides a detailed discussion of how cached pages are organized in the NVM buffer pool.

In HeuristicDB, accessed pages are prioritized based on their access frequency. To efficiently evict the coldest page from the NVM buffer pool when a replacement is required, a least frequently used (LFU) data structure is used. In the proposed system, all cached pages are managed by a hash table, as shown in Fig. 3a. Each item within this hash table is defined as ⟨access freq, ⟨linked list⟩⟩. The access freq is used as a key to record the access count. Pages with the same access count are assigned to the same linked list. When a page admission is needed, the target page is inserted in the head, the most recently used (MRU) place, of the corresponding linked list. When a page replacement is required, the coldest cached page is evicted from the back, the least recently used (LRU) place, of the lowest non-empty linked list.

When performing an eviction request, the EV program does not evict all cached pages; instead, their access frequency is set to zero. The detailed operations are described below. A new item with an access frequency of zero is created first if it does not already exist in the LFU data structure. All items in the LFU data structure are then scanned from the lowest to the highest access frequency in sequence. If the linked list inside an item is not empty, that linked list is first removed and then re-inserted at the back of the number zero linked list. These operations are repeated until all items inside the hash table have been scanned. After the eviction, as shown in Fig. 3b, the most frequently used pages stay in the least recently used place of the number zero linked list. This design enables HeuristicDB to replace more easily pages that were hot in the past but are rarely used in newly submitted queries. On the other hand, if those pages remain hot in the newly submitted query, since they are still in the NVM buffer pool and will be assigned with high priority, no performance degradation occurs.

For a demotion request, the EV program performs similar operations as for an eviction request. The only difference is

that it inserts all cached pages into the number one linked list. After an eviction or demotion has been completed, the pages in the NVM buffer pool are synchronized to *the counting table*. The counting table then contains LBA for cached pages only, and their corresponding access frequency is zero if the eviction is performed. Otherwise, it is one.

5.3 Access Pattern Detector

To determine the access behavior (i.e. sequential/random/repeated) of each table or index required by a query, an access pattern detector is developed. The detailed detection metrics are as the below.

The detector concludes a table or index is accessed repeatedly and sets its status to *repeated*, if a page inside it is found to be reused. Here, we say a page is reused if its access frequency is larger than one.

For determining whether a given table or index is accessed sequentially or randomly, the detector computes its corresponding *Sequential Ratio*. The Sequential Ratio is defined as the number of pages accessed sequentially divided by the total number of pages accessed from that table or index. The system sets the status of a table or index to *sequential* if its Sequential Ratio exceeds 90%, otherwise to *random*. If a file is accessed sequentially, most pages inside that file are also accessed sequentially. Therefore, instead of deciding after an entire table or index had been accessed, in the present study, only the first 1% of pages accessed are monitored to determine the access behavior of a table or index to minimize the extra overhead imposed by this detection program. The number of sequentially accessed pages in each table or index is reset to zero whenever a new query is submitted, and the number is increased by one if a page is read using the database *read-ahead mechanism* [8].

5.4 Page Placement Controller

The page placement controller is developed to redirect database block requests to the appropriate destination according to Heuristic Rules 1 to 5. During query execution, when a block request is generated, the page placement controller first updates its access frequency. For a block write request, the page placement controller checks space availability in the NVM buffer pool for page allocation. If there is no space left, the coldest cached page is evicted. The target page is then written into the NVM buffer pool and flushed into the storage device asynchronously later.

For a block read request, if the required page has already been cached, let us suppose in the i-th linked list in Fig. 3a, the controller removes it from that linked list and then reinserts the page in the MRU place of the i+1-st linked list. Otherwise, a cache miss happens, and the required page is read from storage. An extra evaluation for page admission is also required. The detailed criteria for page admission have been discussed in subsection 4.2. When a page replacement is required, the LRU page in the lowest non-empty linked list is evicted to release extra space. The page placement controller then inserts the new incoming page into the MRU place in the corresponding linked list.

6 Trace Based Evaluation

This section describes a trace-based evaluation that is conducted on the proposed system. Six baseline replacement algorithms, including the least recently used (LRU), the most recently used (MRU), the least frequently used (LFU), the adaptive replacement cache (ARC) [37], multi-queue (MQ) [54], and TAC [19], are also tested for comparison. For MQ and TAC, their tuning parameters are configured as described in [54] and [19], respectively. The block traces of the database workload are collected by blktrace [16].

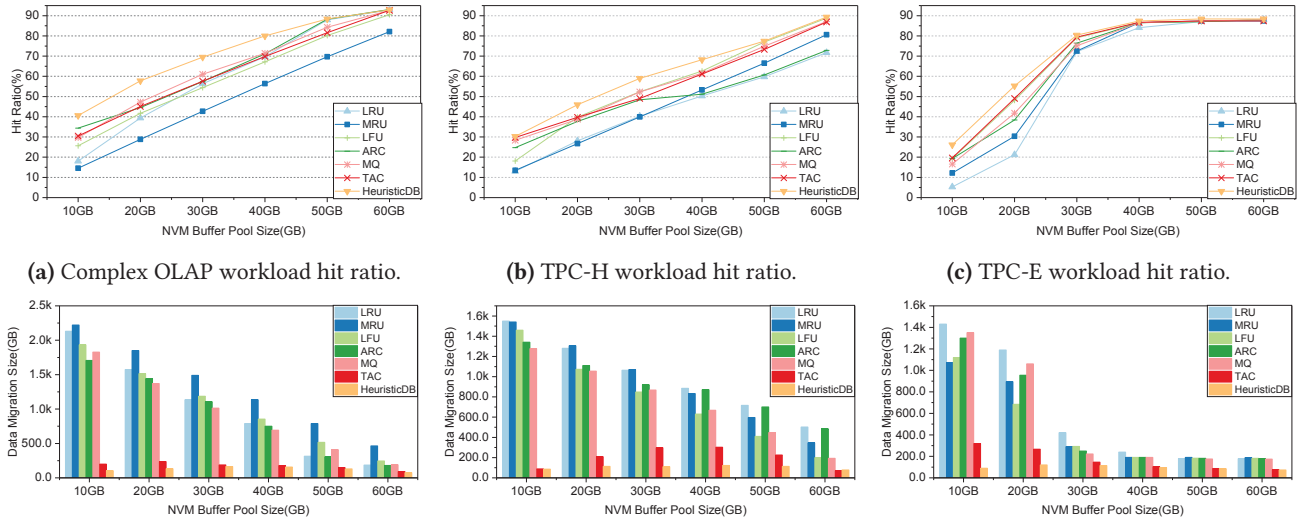
Two database benchmarks, TPC-H [1] and TPC-E [2], are used in the experiments. TPC-H is an OLAP benchmark consisting of eight tables with corresponding indices and a variety of ad-hoc queries. In the experiment, the Sacle Factor (SF) is set to 60, and after all data have been loaded into storage, the TPC-H database size is raised to 90GB. TPC-E is an OLTP benchmark consisting of 33 tables with corresponding indices and diverse concurrent transactions. For this experiment, a 7000-customer database is chosen, and after all data have been loaded into storage, the TPC-E database size is increased to 97GB. The query condition of each query/transaction are generated randomly to make the hot spots of the TPC-H/E benchmarks change dynamically.

Two metrics are used to evaluate the performance of HeuristicDB and the other six baseline replacement algorithms: (1) the cache hit ratio; and (2) the data migration size between storage and the NVM buffer pool. The evaluations are conducted under three database workloads, including a complex OLAP query workload, a TPC-H workload, and a TPC-E workload. It is assumed that the NVM buffer pool is empty before each evaluation. Its size is set to values from 10 to 60 GB with a granularity of 10 GB. A performance gain analysis is provided in the last.

6.1 Complex OLAP Workload

Processing high-complexity OLAP queries usually requires massive block operations from storage. As a result, I/O time becomes a main bottleneck in the database system [49, 50]. This subsection describes an evaluation of how HeuristicDB and the other baselines work for a complex OLAP workload.

The experimental workload is constructed based on five types of complex TPC-H queries where performance is limited by I/O time [49]. These are Queries 3, 7, 9, 13, and 20. To avoid cases where a replacement algorithm works well only for a specific type of query that consumes most of the



(a) Complex OLAP workload hit ratio. **(b)** TPC-H workload hit ratio. **(c)** TPC-E workload hit ratio.

(d) Complex OLAP workload migration size. **(e)** TPC-H workload migration size. **(f)** TPC-E workload migration size.

Figure 4: Experimental results of trace based evaluation.

block operations in the workload and is therefore producing a misleading conclusion, in the experiment, the number of each type of query is inversely proportional to its execution time. In addition, to avoid a query's results being cached because it has been run before, in the experiment, different query conditions are used for the same type of query. Using these principles, 20 complex OLAP queries are generated in random order and issued by the database system. Fig. 6a lists the generated query sequence.

As Fig. 4a shows, HeuristicDB outperforms all other baselines on the cache hit ratio. Most significantly, it maintains about a 10% higher cache hit ratio relative to the second-best replacement algorithm, MQ, in a large cache size range from 20GB to 40GB. In contrast, TAC, one of the main competitors of HeuristicDB, delivers an even worse cache hit ratio than ARC when the NVM buffer pool size is set to 10GB. HeuristicDB also work well on data migration size. As shown in Fig. 4d, HeuristicDB migrates only half as much data as TAC with NVM buffer pool sizes equal to 10GB and 20GB. The performance advantage of HeuristicDB disappears gradually after the NVM buffer pool size rise to 50GB. This is because a larger NVM buffer pool leads to more hot pages being cached, and the performance difference for each replacement algorithm is reduced.

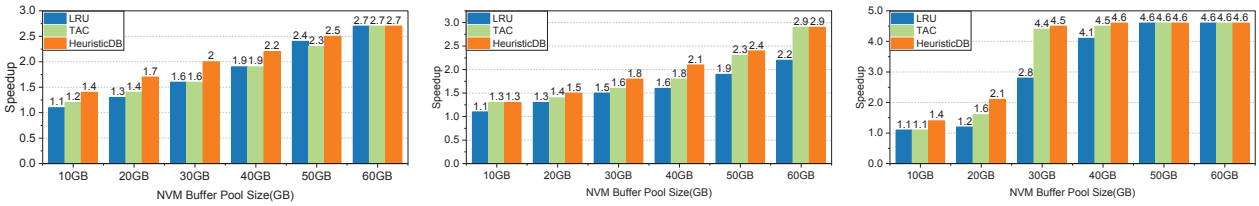
6.2 TPC-H Workload

This subsection describes a performance evaluation of HeuristicDB on a TPC-H query sequence. As shown in Fig. 4b, HeuristicDB outperforms all other baselines and delivers a cache hit ratio about 7% higher than the second-best replacement algorithms, TAC and LFU, when the NVM buffer pool size is set to 20GB, 30GB, and 40GB. In addition, there are two more interesting observations to be made. First, for the TPC-H workload, MQ can not maintain its second-highest cache hit ratio as it does for the complex OLAP workload. Instead, TAC and LFU share that distinction. Second, ARC delivers a highly unstable performance improvement on cache hit ratio. For instance, when the NVM buffer pool size is less than 40GB, ARC delivers a cache hit ratio up to 11% higher than MRU. However, as the NVM buffer pool size continues to rise to 50GB and 60GB, ARC delivers an 8% lower cache hit ratio than MRU.

Finally, HeuristicDB achieves a significant reduction in data migration size. As shown in Fig. 4e, when the NVM buffer pool size is equal to 30GB and 40GB, HeuristicDB migrates 3X less data between storage and the NVM buffer pool than TAC.

6.3 TPC-E Workload

To validate the effectiveness of HeuristicDB on the OLTP workload, a five-hour block trace of the TPC-E benchmark is collected and used for performance evaluation. The experimental results are plotted in Figs. 4c and 4f. From these results, when the NVM buffer pool size is set to 10GB and 20GB, HeuristicDB provides a 6% higher cache hit ratio than the second-best replacement algorithm, LFU, and migrates 3.6X less data than TAC. As the NVM buffer pool size increases, the performance gap between HeuristicDB and other baselines decreases. The reason is that most of the block operations inside the TPC-E benchmark are generated for accessing the 28GB *trade* table. When the NVM buffer pool size is greater than 28GB, most of the frequently used pages have already been cached and thus HeuristicDB and the other baselines produce similar performance.



(a) Speedup for complex OLAP workload. **(b)** Speedup for TPC-H workload. **(c)** Speedup for TPC-E workload.

Figure 5: The speedup of HeuristicDB and other baselines. The execution time and transaction rate for complex OLAP, TPC-H, and TPC-E workloads of the case without NVM buffer pool are 34993.22sec, 20507.77sec, and 59.3 TpsE, respectively.

6.4 Performance Gain Analysis

From the results, HeuristicDB delivers a significantly enhanced cache hit ratio and incurs much less data migration than the other alternatives. Most significantly, the proposed system maintains the best performance for all tested database workloads. The performance advantages of HeuristicDB can be attributed to two design principles. First, by considering the characteristics of database block requests, the tables and indices required by the query are prioritized based on their reusability, and only the hottest pages from a high priority table or index are cached. This enables HeuristicDB to cache the most profitable pages in the NVM buffer pool to deliver maximum performance gains with less data migration during query execution. Second, the adaptive EV program enables HeuristicDB to evict or demote cached pages and admit new incoming hot pages more efficiently as the access pattern of a workload changes. In contrast, by ignoring the characteristics of database block requests and lacking an efficient demotion program for cached pages, the performance improvement provided by the prior work is limited and varies greatly as database workloads change or the NVM buffer pool size changes.

7 Real System Performance Evaluation

This section describes the implementation of a HeuristicDB prototype on an empirical system for performance evaluation. For comparison, we also implement LRU, which is the most widely used replacement algorithm, and TAC, which is a hybrid storage management algorithm for the RDBMS. In the test, the same workloads are used as in the previous section. We run TPC-H with one query stream, and TPC-E is configured with 7000 customers. Two metrics, including execution time and transaction rate, are used to evaluate the performance of tested algorithms.

7.1 Implementation Details

LRU, TAC, and HeuristicDB are implemented based on Flashcache [38], a block cache component for the Linux kernel. During query processing, the database server runs on top of the system. When block requests are sent to the storage layer, Flashcache maps them either to the NVM buffer pool or to storage based on its internal replacement algorithm.

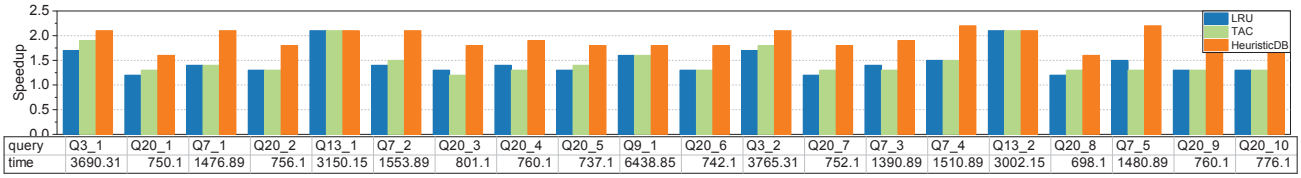
The experiments are conducted on a Dell Precision Tower 3620 workstation equipped with a four-Intel i7-6700 3.4 GHz CPU and 16 GB of memory. The Ubuntu 18.04 LTS operating system is installed on the workstation. Two types of storage devices are used, an Intel 168 GB DC P4800X Optane SSD [5] and an Intel 2TB DC P3700 NAND SSDs [4]. The Optane SSD acts as an NVM buffer pool, and the NAND SSDs are used as persistent storage. MySQL 5.7 [6] with the InnoDB storage engine [12] is running on the system. The database buffer pool is set to 4 GB. In the experiment, after each test run, the MySQL database application is restarted to clean up both the main memory and the NVM buffer pool.

7.2 Experimental Results

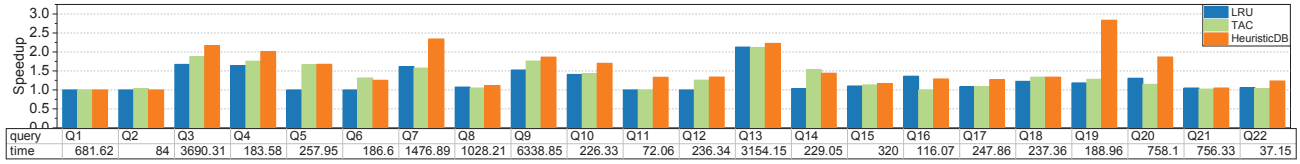
Fig. 5 illustrates the performance speedup of the three replacement algorithms for the case without the NVM buffer pool. For each type of workload, different performance metrics are used. For the OLAP workloads, including complex OLAP and TPC-H, the total execution time to complete the workload is measured. For the OLTP workload, TPC-E, the number of transactions executed per second (TpsE) is measured. From the results, HeuristicDB delivers the highest performance accelerations compared with the other baselines. This observation matches the results described in Section 6.

As shown in Fig. 5a, for the complex OLAP workload, HeuristicDB improves database performance over LRU and TAC by up to 25% when the NVM buffer pool size is set to 30GB. In addition, HeuristicDB maintains the best performance for all NVM buffer pool sizes. In contrast, TAC shows unstable improvement in performance. For instance, when the NVM buffer pool is set to 50GB, TAC delivers even worse performance than LRU. For the TPC-H workload, as shown in Fig. 5b, for an NVM buffer pool size of 40GB, HeuristicDB provides 17% and 31% higher performance than TAC and LRU, respectively.

HeuristicDB also works well for the OLTP workload. As shown in Fig. 5c, HeuristicDB maintains the best performance for the TPC-E benchmark. It enhances the transaction rate over LRU and TAC by 75% and 31%, respectively



(a) Complex OLAP workload internals.



(b) TPC-H workload internals.

Figure 6: The speedup of different replacement algorithm to each query inside the OLAP workload. We record the execution time (in seconds) of each query of the case without NVM buffer pool buffer below each figure.

when the NVM buffer pool is set to 20GB. However, because more hot pages are cached with continued increases in NVM buffer pool size, the performance advantages of HeuristicDB relative to other baselines gradually disappear.

We also quantify the overhead of HeuristicDB. The experimental results show that the extra memory space consumed by HeuristicDB for storing the metadata of TPC-H and TPC-E is 57MB and 61MB, respectively. Compared with the capacity of host memories, the footprint of HeuristicDB is minimal. In addition, the extra time spent by the query profile queue and the EV program of HeuristicDB for each query/transaction inside the TPC-H/E workloads is only 0.5%-2% of the total execution time. This low overhead is because those programs run only at the beginning of a query and are completed quickly. As a result, the computation overhead of HeuristicDB is trivial.

7.3 Internals on the OLAP Workload

Compared with OLTP, which accesses the database using an index and retrieves a tiny portion of data to execute a transaction, an OLAP query has higher complexity and requires many more block operations. Therefore, in most cases, I/O time is the main bottleneck for OLAP queries [27–29].

This subsection describes the detailed internals of the OLAP workload. The speedup achieved by HeuristicDB and the other two baselines on each query inside both the complex OLAP and the TPC-H workloads is plotted in Fig. 6. In the experiment, the NVM buffer pool is set to 30GB. To show query complexity clearly, the execution time of each query for the case without the NVM buffer pool is recorded below the figure. From the results, HeuristicDB delivers the highest performance acceleration for almost all queries inside both the complex OLAP and the TPC-H workloads. This is because HeuristicDB adaptively adjusts the priorities of cached pages and tracks hot tables and indices continuously when a query is submitted. In addition, it discriminates tables, indices, and pages based on their reuse probability. These designs enable HeuristicDB to preserve the most profitable pages in the NVM buffer pool during every single query to deliver the highest performance acceleration. In contrast, the performance improvement of TAC on each single OLAP query is unstable. For instance, as shown in Fig. 6a, TAC experiences performance degradation and even delivers lower speedup than LRU for queries Q20_3, Q20_4, Q7_3, and Q7_5. A similar result can be observed for the TPC-H workload, as shown in Fig. 6b, for queries Q7, Q8, Q13, Q16, Q20, Q21, and Q22.

8 Conclusion

In this paper, we present HeuristicDB, a new approach for using an NVM block device as an extension of the database buffer pool. By considering the characteristics of database block requests, seven rules that associate database (block) requests with the proper quality of service are proposed. To support the system implementation of our proposed rules, four programs are then developed inside HeuristicDB. Intensive experiments are conducted to evaluate the effectiveness of our proposed design. The experimental results show that the proposed HeuristicDB delivers up to 75% higher performance than tested baselines with only trivial memory and execution time overheads.

Acknowledgments

This work was supported in part by the Center for Research in Intelligent Storage (CRIS), which is supported by National Science Foundation grant no. IIP-1439622 and member companies. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

References

[1] 2001. TPC Benchmark H. http://www.tpc.org/tpch/.

[2] 2007. TPC Benchmark E. http://www.tpc.org/tpce/.

[3] 2009. TeraData Virtual Storage System. http://assets.teradata.com/resourceCenter/downloads/Brochures/EB5944.pdf?processed=1.

[4] 2014. Intel SSD DC P3700 Series. https://www.intel.com/content/www/us/en/products/memory-storage/solid-state-drives/data-center-ssds/dc-p3700-series.html.

[5] 2018. Intel Optane SSD DC P4800X Series. https://www.intel.com/content/www/us/en/solid-state-drives/optane-ssd-dc-p4800x-brief.html.

[6] 2019. MySQL 5.7. https://dev.mysql.com/doc/refman/5.7/en/.

[7] 2019. Persistent memory development kit. https://https://github.com/pmem/pmdk.

[8] 2020. Optimizing InnoDB Disk I/O. https://dev.mysql.com/doc/refman/5.7/en/optimizing-innodb-diskio.html.

[9] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D Davis, Mark S Manasse, and Rina Panigrahy. 2008. Design tradeoffs for SSD performance.. In *USENIX Annual Technical Conference*, Vol. 57.

[10] Mohamed Arafa, Bahaa Fahim, Sailesh Kottapalli, Akhilesh Kumar, Lily P Looi, Sreenivas Mandava, Andy Rudoff, Ian M Steiner, Bob Valentine, Geetha Vedaraman, et al. 2019. Cascade lake: Next generation intel xeon scalable processor. *IEEE Micro* 39, 2 (2019), 29–36.

[11] Joy Arulraj, Andy Pavlo, and Krishna Teja Malladi. 2019. Multi-tier buffer management and storage system design for non-volatile memory. *arXiv preprint arXiv:1901.10938* (2019).

[12] Ryan Bannon, Alvin Chin, Faryaaz Kassam, Andrew Roszko, and Ric Holt. 2002. Innodb concrete architecture. *University of Waterloo* (2002).

[13] Charles Andrew Bell and Sven Sandberg. 2012. *Expert MySQL*. Vol. 3. Springer.

[14] Curtis Neal Boger, John Francis Edwards, Randy Lynn Egan, and Michael S Faunce. 2006. Metadata manager for database query optimizer. US Patent 6,996,556.

[15] Michael L Brundage and Andrew E Kimball. 2006. Query optimizer system and method. US Patent 7,146,352.

[16] Alan D Brunelle. 2006. Block i/o layer tracing: blktrace. *HP, Gelato-Cupertino, CA, USA* (2006).

[17] Geoffrey W Burr, Matthew J Breitwisch, Michele Franceschini, Davide Garetto, Kailash Gopalakrishnan, Bryan Jackson, Bülent Kurdi, Chung Lam, Luis A Lastras, Alvaro Padilla, et al. 2010. Phase change memory technology. *Journal of Vacuum Science & Technology B, Nanotechnology and Microelectronics: Materials, Processing, Measurement, and Phenomena* 28, 2 (2010), 223–262.

[18] Mustafa Canim, George A Mihaila, Bishwaranjan Bhattacharjee, Kenneth A Ross, and Christian A Lang. 2009. An object placement advisor for DB2 using solid state storage. *Proceedings of the VLDB Endowment* 2, 2 (2009), 1318–1329.

[19] Mustafa Canim, George A Mihaila, Bishwaranjan Bhattacharjee, Kenneth A Ross, and Christian A Lang. 2010. SSD bufferpool extensions for database systems. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 1435–1446.

[20] Jaeyoung Do, Donghui Zhang, Jignesh M Patel, David J DeWitt, Jeffrey F Naughton, and Alan Halverson. 2011. Turbocharging DBMS buffer pool using SSDs. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*. ACM, 1113–1124.

[21] Assaf Eisenman, Darryl Gardner, Islam AbdelRahman, Jens Axboe, Siying Dong, Kim Hazelwood, Chris Petersen, Asaf Cidon, and Sachin Katti. 2018. Reducing DRAM footprint with NVM in Facebook. In *Proceedings of the Thirteenth EuroSys Conference*. ACM, 42.

[22] Hiroki Fujii, Kousuke Miyaji, Koh Johguchi, Kazuhide Higuchi, Chao Sun, and Ken Takeuchi. 2012. x11 performance increase, x6. 9 endurance enhancement, 93% energy reduction of 3D TSV-integrated hybrid ReRAM/MLC NAND SSDs by data fragmentation suppression. In *2012 symposium on VLSI circuits (VLSIC)*. IEEE, 134–135.

[23] Frank T Hady, Annie Foong, Bryan Veal, and Dan Williams. 2017. Platform storage performance with 3D XPoint technology. *Proc. IEEE* 105, 9 (2017), 1822–1833.

[24] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R Dulloor, et al. 2019. Basic performance measurements of the intel optane DC persistent memory module. *arXiv preprint arXiv:1903.05714* (2019).

[25] John R Jameson, Philippe Blanchard, John Dinh, Nathan Gonzales, Vasudevan Gopalakrishnan, Berenice Guichet, Shane Hollmer, Sue Hsu, Gideon Intrater, Deepak Kamalanathan, et al. 2016. Conductive bridging RAM (CBRAM): then, now, and tomorrow. *ECS Transactions* 75, 5 (2016), 41.

[26] Jaeyong Jeong, Sangwook Shane Hahn, Sungjin Lee, and Jihong Kim. 2014. Lifetime Improvement of {NAND} Flash-based Storage Systems Using Dynamic Program and Erase Scaling. In *12th {USENIX} Conference on File and Storage Technologies ({FAST} 14)*. 61–74.

[27] Insoon Jo, Duck-Ho Bae, Andre S Yoon, Jeong-Uk Kang, Sangyeun Cho, Daniel DG Lee, and Jaeheon Jeong. 2016. YourSQL: a high-performance database system leveraging in-storage computing. *Proceedings of the VLDB Endowment* 9, 12 (2016), 924–935.

[28] Sungchan Kim, Hyunok Oh, Chanik Park, Sangyeun Cho, and Sang-Won Lee. 2011. Fast, energy efficient scan inside flash memory SSDs. In *Proceeedings of the International Workshop on Accelerating Data Management Systems (ADMS)*.

[29] Sungchan Kim, Hyunok Oh, Chanik Park, Sangyeun Cho, Sang-Won Lee, and Bongki Moon. 2016. In-storage processing of database scans and joins. *Information Sciences* 327 (2016), 183–200.

[30] Michael Kund, Gerhard Beitel, C-U Pinnow, Thomas Rohr, Jorg Schumann, Ralf Symanczyk, K Ufert, and Gerhard Muller. 2005. Conductive bridging RAM (CBRAM): An emerging non-volatile memory technology scalable to sub 20nm. In *Electron Devices Meeting, 2005. IEDM Technical Digest. IEEE International*. IEEE, 754–757.

[31] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. 2017. Strata: A cross media file system. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 460–477.

[32] Tapio Lahdenmaki and Mike Leach. 2005. *Relational Database Index Design and the Optimizers: DB2, Oracle, SQL Server, et al.* John Wiley & Sons.

[33] Stefan Lai. 2003. Current status of the phase change memory and its future. In *IEEE International Electron Devices Meeting 2003*. IEEE, 10–1.

[34] Xuhui Li, Ashraf Aboulnaga, Kenneth Salem, Aamer Sachedina, and Shaobo Gao. 2005. Second-Tier Cache Management Using Write Hints.. In *FAST*, Vol. 5. 9–9.

[35] Hongjun Lu, Yuet Yeung Ng, and Zengping Tian. 2000. T-tree or b-tree: Main memory database index structure revisited. In *Proceedings 11th Australasian Database Conference. ADC 2000 (Cat. No. PR00528)*. IEEE, 65–73.

[36] Tian Luo, Rubao Lee, Michael Mesnier, Feng Chen, and Xiaodong Zhang. 2012. hStorage-DB: heterogeneity-aware data management to exploit the full capability of hybrid storage systems. *Proceedings of the VLDB Endowment* 5, 10 (2012), 1076–1087.

[37] Nimrod Megiddo and Dharmendra S Modha. 2004. Outperforming LRU with an adaptive replacement cache algorithm. *Computer* 37, 4 (2004), 58–65.

[38] Paul Saab Mohan Srinivasan. 2010. Flashcache. https://github.com/facebookarchive/flashcache.

[39] Elizabeth J O'neil, Patrick E O'neil, and Gerhard Weikum. 1993. The LRU-K page replacement algorithm for database disk buffering. *Acm*

Sigmod Record 22, 2 (1993), 297–306.

[40] Jiaxin Ou, Jiwu Shu, Youyou Lu, Letian Yi, and Wei Wang. 2014. EDM: An endurance-aware data migration scheme for load balancing in SSD storage clusters. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. IEEE, 787–796.

[41] Moinuddin K Qureshi, Vijayalakshmi Srinivasan, and Jude A Rivers. 2009. Scalable high performance main memory system using phase-change memory technology. In *Proceedings of the 36th annual international symposium on Computer architecture*. 24–33.

[42] Reza Salkhordeh, Mostafa Hadizadeh, and Hossein Asadi. 2018. An Efficient Hybrid I/O Caching Architecture Using Heterogeneous SSDs. *IEEE Transactions on Parallel and Distributed Systems* 30, 6 (2018), 1238–1250.

[43] Kyungjune Son, Kyungjun Cho, Subin Kim, Gapyeol Park, Kyunghwan Song, and Journ Kim. 2018. Modeling and signal integrity analysis of 3D XPoint memory cells and interconnections with memory size variations during read operation. In *2018 IEEE Symposium on Electromagnetic Compatibility, Signal Integrity and Power Integrity (EMC, SI & PI)*. IEEE, 223–227.

[44] Gokul Soundararajan, Vijayan Prabhakaran, Mahesh Balakrishnan, and Ted Wobber. 2010. Extending SSD Lifetimes with Disk-Based Write Caches.. In *FAST*, Vol. 10. 101–114.

[45] Ralf Symanczyk, Jan Keller, Michael Kund, Gerhard Muller, Bernhard Ruf, Paul-Henri Albarede, Serge Bournat, Laurent Bouteille, Alexander Duch, et al. 2007. Conductive bridging memory development from single cells to 2Mbit memory arrays. In *2007 Non-Volatile Memory Technology Symposium*. IEEE, 71–75.

[46] H-S Philip Wong, Simone Raoux, SangBum Kim, Jiale Liang, John P Reifenberg, Bipin Rajendran, Mehdi Asheghi, and Kenneth E Goodson. 2010. Phase change memory. *Proc. IEEE* 98, 12 (2010), 2201–2227.

[47] Shiqin Yan, Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Andrew A Chien, and Haryadi S Gunawi. 2017. Tiny-tail flash: Near-perfect elimination of garbage collection tail latencies in NAND SSDs. *ACM Transactions on Storage (TOS)* 13, 3 (2017), 1–26.

[48] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. 2020. An empirical guide to the behavior and use of scalable persistent memory. In *18th {USENIX} Conference on File and Storage Technologies ({FAST} 20)*. 169–182.

[49] Jinfeng Yang, Bingzhe Li, and David J Lilja. 2020. Exploring Performance Characteristics of the Optane 3D Xpoint Storage Technology. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)* 5, 1 (2020), 1–28.

[50] Jinfeng Yang and David J Lilja. 2018. Reducing Relational Database Performance Bottlenecks Using 3D XPoint Storage Technology. In *2018 17th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/12th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*. IEEE, 1804–1808.

[51] Zhengyu Yang, Morteza Hoseinzadeh, Allen Andrews, Clay Mayers, David Thomas Evans, Rory Thomas Bolt, Janki Bhimani, Ningfang Mi, and Steven Swanson. 2017. AutoTiering: automatic data placement manager in multi-tier all-flash datacenter. In *2017 IEEE 36th International Performance Computing and Communications Conference (IPCCC)*. IEEE, 1–8.

[52] Ting Yao, Yiwen Zhang, Jiguang Wan, Qiu Cui, Liu Tang, Hong Jiang, Changsheng Xie, and Xubin He. 2020. MatrixKV: Reducing Write Stalls and Write Amplification in LSM-tree Based {KV} Stores with Matrix Container in {NVM}. In *2020 {USENIX} Annual Technical Conference ({USENIX}{ATC} 20)*. 17–31.

[53] Qian Zhao and Hao Chen. 2010. PCI Express interface. US Patent 7,673,092.

[54] Yuanyuan Zhou, James Philbin, and Kai Li. 2001. The Multi-Queue Replacement Algorithm for Second Level Buffer Caches.. In *USENIX Annual Technical Conference, General Track*. 91–104.