Larger-than-Memory Data Management on Modern Storage Hardware for In-Memory OLTP Database Systems

Lin Ma♠, Joy Arulraj♠, Sam Zhao♦, Andrew Pavlo♠, Subramanya R. Dulloor♣, Michael J. Giardino△, Jeff Parkhurst♣, Jason L. Gardner♣, Kshitij Doshi♣, Col. Stanley Zdonik♦
♠Carnegie Mellon University, ♣Intel Labs, △Georgia Institute of Technology, ♦Brown University
{lin.ma, jarulraj, pavlo}@cs.cmu.edu
{subramanya.r.dulloor, jeff.parkhurst, jasonx.l.gardner, kshitij.a.doshi}@intel.com
giardino@ece.gatech.edu, {sam, sbz}@cs.brown.edu

ABSTRACT

In-memory database management systems (DBMSs) outperform disk-oriented systems for on-line transaction processing (OLTP) workloads. But this improved performance is only achievable when the database is smaller than the amount of physical memory available in the system. To overcome this limitation, some in-memory DBMSs can move cold data out of volatile DRAM to secondary storage. Such data appears as if it resides in memory with the rest of the database even though it does not.

Although there have been several implementations proposed for this type of cold data storage, there has not been a thorough evaluation of the design decisions in implementing this technique, such as policies for when to evict tuples and how to bring them back when they are needed. These choices are further complicated by the varying performance characteristics of different storage devices, including future non-volatile memory technologies. We explore these issues in this paper and discuss several approaches to solve them. We implemented all of these approaches in an in-memory DBMS and evaluated them using five different storage technologies. Our results show that choosing the best strategy based on the hardware improves throughput by 92–340% over a generic configuration.

1. INTRODUCTION

In-memory DBMSs provide better throughput and lower latency for OLTP applications because they eliminate legacy components that inhibit performance, such as buffer pool management and concurrency control [17]. But previous work has shown that the performance of an in-memory DBMS will drop by up to 66% when the dataset grows larger than the memory capacity, even if the working set fits in memory [22]. Enabling an in-memory DBMS to support databases that exceed the amount of available memory is akin to paging in operating systems (OS). When the system runs out of memory, the OS writes cold data pages out to disk, typically in least-recently-used (LRU) order. When an evicted page is accessed, it trips a page fault that causes the OS to read the page back in. If the OS does not have any free memory for this new page, it will have to evict other pages first before the requested page can be brought back. Virtual memory is problematic for a DBMS because it does not know whether or not a page is in memory and thus it has no way to predict when it will hit a page fault. This means that the execution of transactions is stalled while the page is fetched from disk. This problem has been known since the early 1980s [23] and more recently has even affected newer DBMSs like MongoDB that rely on OS-level memory management (pre-WiredTiger) [6].

Several approaches have been developed to allow in-memory DBMSs to support larger-than-memory databases without sacrificing the higher performance that they achieve in comparison to disk-oriented systems. The crux of how these techniques work is that they rely on the skewed access patterns exhibited in OLTP workloads, where certain data tuples are "hot" and accessed more frequently than other "cold" tuples. For example, on a website like eBay, new auctions are checked and bid on frequently by users near the auction's end. After the auction is closed, it is rarely accessed and almost never updated. If the auction data is moved to cheaper secondary storage, the system can still deliver high performance for transactions that operate on hot in-memory tuples while still being able to access the cold data if needed at a later point in time.

But how the DBMS should use secondary storage for its cold data is highly dependent on the properties of the underlying hardware. Thus, in this paper, we identify policies that are crucial to the performance of an OLTP DBMS with a larger-than-memory database. To evaluate these design choices, we implemented all of them in the H-Store [3] DBMS and analyzed them using different OLTP workloads. We deployed the system with five different storage devices that all have distinct performance characteristics: (1) HDD, (2) shingled magnetic recording HDD, (3) NAND-based SSD, (4) emulator for 3D-XPoint like technologies, and (5) byte-addressable NVRAM. Our experimental results show that by properly accounting for the hardware's properties in the DBMS for cold-data storage, we can achieve up to 3× higher throughput.

2. STORAGE TECHNOLOGIES

We first summarize the different storage technologies that are available today. Let's assume that the DBMS read/writes tuples from these devices with a block-granularity. When a transaction accesses a cold tuple stored on the device, the DBMS needs to read the entire block containing that tuple back into memory.

Hard-Disk Drive (HDD): Modern HDDs are based on the same high-level design principles from the 1960s: a magnetic platter

DaMoN'16, June 26-July 01 2016, San Francisco, CA, USA
© 2016 ACM. ISBN 978-1-4503-4319-0/16/06...$15
DOI: http://dx.doi.org/10.1145/2933349.2933358

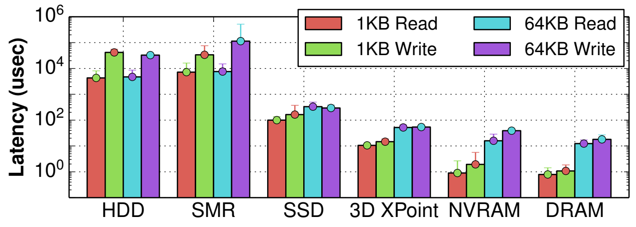


Figure 1: Storage Comparison – Evaluation of the storage devices using a microbenchmark that simulates a workload of reading and writing cold tuples in an in-memory DBMS.

spins and an arm reads data off of it with a block-granularity (typically 4 KB). Since moving the platter and the arm is a mechanical process, these drives are the slowest of all the devices except for the SMR HDDs. As sequential reads and writes to the device do not require the arm to be re-positioned, they are faster than random accesses. The main advantage of HDDs is that they have a high data density and consequently offer more storage space per dollar.

Shingled Magnetic Recording Drive (SMR): SMR is a newer HDD recording technology that is designed to increase the storage density and overall per-drive storage capacity [26]. The SMR drive can pack more data onto the platter by using narrower tracks than a HDD. However, narrower tracks can cause writes to overwrite data stored on nearby tracks. The SMR drive must repair the data on contiguous tracks on each write to protect them. So, writes are slower and more unpredictable than a regular HDD. However, the read performance of SMR drives is similar to that of HDDs.

Solid-State Drive (SSD): These devices differ from HDDs in that they do not have any moving parts. Instead, they use NAND storage cells (sometimes refered to as "flash") that retain data even after power is cut. These devices also provide block-level access granularity to the data (also typically 4 KB). SSDs are currently 3–10× more expensive per GB than an HDD, but their read and write latencies are up to 1300× and 1000× lower, respectively [9]. Another major challenge with them is that each storage cell can only be written to a fixed number of times before it wears out.

3D XPoint SSD (3DX): 3D XPoint [1] is one type of non-volatile memory (NVM) that has 1000× lower latency and 1000× higher endurance than flash-based SSDs. Unlike in DRAM where each memory cell needs a transistor for selection, this technology uses perpendicular wires to connect submicroscopic storage columns that can be addressed by selecting their top and bottom wires. Current 3DX devices have 10× higher density than DRAM and export a PCIe NVMe interface with an SSD form factor.

NVRAM: Non-volatile random-access memory (NVRAM) represents a broad class of byte-addressable persistent storage technologies. This includes 3DX, PCM [21], memristors [25], and STT-MRAM [14]. In contrast to the other devices that use PCIe or SATA interfaces, future NVRAM storage will use DIMM slots to deliver lower latencies and higher bandwidths to the CPU(s). They are predicted to provide lower read/write latencies and better endurance than existing SSDs. They will also have higher densities and larger capacity than DRAM. NVRAM's latency is expected to be 2–8× higher than DRAM latency [8], and they do not require periodic refreshing unlike DRAM.

To better understand the performance characteristics of these five devices, we use a microbenchmark that simulates the access patterns of cold-data storage for an in-memory DBMS. We use a disk-resident hash table similar to the one used in anti-caching [12]. We first load a database of 10m tuples that are 1 KB each. We then execute 5m operations (50% reads / 50% writes) with a highly skewed Zipfian distribution against the hash table using a single thread to avoid the overhead of concurrency control. Each operation accesses either 1 or 64 tuples. We use direct I/O to avoid OS-level caching and invoke the appropriate synchronization primitive (e.g., `fsync`, `pcommit` [8]) after each write operation. We execute three trials per device and report the average latency of the operations. We describe these hardware devices in Sect. 5.

The results in Fig. 1 show that the difference between reads and writes for all the devices is 1.8–93.8%. The difference is more prominent for magnetic drives than other devices. The time taken to perform a 64 KB read/write is similar to that required for a 1 KB operation on HDD/SMR, whereas on DRAM/NVRAM these operations take up to 20× more time. The access latency of 3DX is up to 10× lower than that of the NAND-based SSD. SMR exhibits larger variance on writes compared to other devices. We note that the performance of NVRAM is similar to that of DRAM.

We contend that in-memory DBMSs need to employ the hardware-specific strategies for cold-data storage. But at the same time there are other policies that are independent of the hardware. Given this, we first describe these independent policies in Sect. 3. Then in Sect. 4 we discuss the policies that are closely tied to the device. For each policy, we summarize how they are used in existing DBMSs that support larger-than-memory databases: (1) H-Store's anti-caching [12], (2) Apache Geode's overflow tables [2], (3) Microsoft's Project Siberia [16] in Hekaton, (4) a variant of VoltDB from EPFL [22], and (5) MemSQL's external tables [4].

3. HARDWARE INDEPENDENT POLICIES

We now discuss DBMS implementation policies that are not affected by the underlying non-volatile device for cold data storage.

3.1 Cold Tuple Identification

In order to move data out of memory to non-volatile storage, the DBMS must identify which tuples to evict based on some metric. This issue has been thoroughly studied in buffer managers for disk-oriented DBMSs and thus there are many techniques that can be borrowed from them. In general, there are two approaches: (1) *on-line identification* and (2) *off-line identification*.

The on-line approach maintains internal data structures that track the usage information of tuples. The DBMS updates the data structure whenever tuples are accessed, and identifies the coldest tuple directly from the data structure when it needs to evict data. The major concern of this approach is that maintaining fine-grained tracking information for all of the tuples in a database can be costly, so in practice certain approximation techniques like sampling can be applied. This approach is used in H-Store and Apache Geode [2].

The off-line approach uses a dedicated background thread to analyze the DBMS's write-ahead log to compute the tuples' access frequencies. This information is then provided to the DBMS, which then uses it to identify the cold tuples. Hekaton and EPFL's variant of VoltDB adopt this technique. It is more computationally efficient than the on-line approach, but requires more storage space to log the access tuples. It also only provides access frequency information after the analysis of all the log records, whereas the on-line approach can provide identification result in real-time.

3.2 Evicted Tuple Meta-data

The next design choice is what information the DBMS should keep in memory for the evicted tuples. This is required to handle queries that access data that does not reside in memory.

One solution is to leave a marker to represent an evicted tuple in memory. This is the approach used in H-Store. After a tuple is moved to secondary storage, a special "tombstone" tuple is created in its place that contains the storage location of the evicted tuple. The DBMS then updates every index that references that tuple to point to the tombstone. Every tuple has a flag in its header to indicate whether it is a tombstone so that the DBMS can determine whether it needs to retrieve the original data from cold data storage whenever the tuple is accessed. This only reduces memory usage if the size of the tombstone (64 bits) is less than the original tuple.

Instead of using tombstones, the DBMS can use a space-efficient probabilistic data structure (e.g., Bloom filter) to track whether a tuple exists on the secondary storage. The DBMS checks the filter before it executes a query to determine whether the data it needs has been evicted. If the filter indicates that some of the tuples it needs are in the cold-data storage, the DBMS fetches the data back into memory. These filters use less memory than tombstones but may report false positives that cause unnecessary reads to secondary storage. This approach is employed in Hekaton [16].

The most memory efficient approach is for the DBMS to keep no information about evicted tuples and instead let the OS manage the cold data using virtual paging [22]. EPFL's variant of VoltDB analyzes transactions' access patterns and moves cold tuples to a memory region that the OS is allowed swap out to secondary storage. In the end, however, it is the OS that decides when to move the data. Because the DBMS has no information about the physical location of a tuple during query execution, it can incur long stalls when a transaction accesses a tuple that is not in memory.

Using tombstones enables the DBMS to track evicted tuples without affecting transactions that operate on hot tuples, but it requires the most memory. Filters consume less memory, but increase the computational overhead as the DBMS has to check the filters for each query. Virtual memory requires no extra computation or memory, but the DBMS is unaware of data location and is unable to control what happens when an evicted tuple is accessed.

3.3 Eviction Timing

The last policy is when the DBMS should evict cold tuples from memory out to non-volatile storage. The simplest mechanism is for DBMS to monitor the database's memory usage and begin the eviction process when it reaches a threshold defined by the administrator. The DBMS can continue to move data until it goes below this threshold. If this approach is used with on-line identification, like in H-Store or Geode, then the DBMS can trigger the eviction process immediately when it detects that it is running out of memory. With off-line identification, however, it must first wait for the analysis process to finish before it can start evicting tuples.

If the DBMS uses virtual memory (as in EPFL's variant of VoltDB), then the OS decides to swap the cold data to secondary storage when it runs out of physical memory. The DBMS cannot explicitly instruct the OS to invoke the eviction process and thus it cannot control the exact amount of memory used by the DBMS.

Lastly, the DBMS can entirely defer the decision about when to move data out of memory to a human administrator. As an example, the DBA can declare a table to be in on-disk column-based stores in MemSQL when they think available memory is not enough.

4. HARDWARE DEPENDENT POLICIES

We next discuss three policies that are tightly coupled to the storage technology used for the DBMS's cold-data storage. We analyze how the characteristics of the hardware device relate to each of these policies.

4.1 Cold Tuple Retrieval

The first policy is how the DBMS should move tuples back into DRAM from the secondary storage. Again, we assume that the DBMS reads from and writes to the storage device in blocks.

One method is to abort the transaction that touches cold tuples, merge the tuples asynchronously into memory, and then restart the transaction. We call this method *abort-and-restart* (AR). This removes the overhead of reading the data out of a transaction's critical path, which is important if the device has a high read latency. It also enables the DBMS to use larger blocks for cold storage because data is read in a background thread, which benefits HDDs and SMR HDDs where sequential access is much faster than random access. The disadvantage of this method, however, is that it introduces additional overhead from aborting transactions. Restarting a long-running transaction can also be costly if the DBMS aborts that transaction near the end of its execution.

The alternative approach is *synchronous retrieval* (SR) where the DBMS stalls a transaction that accesses evicted tuples while the data is brought back into memory. There is no additional overhead with respect to aborting and restarting the transaction, but the retrieval of data from the secondary storage delays the execution of other transactions. This delay can be significant for a DBMS that uses a coarse-grained locking concurrency control scheme (e.g., H-Store/VoltDB [24]). In other schemes, this stall increases both the amount of time that a transaction holds a lock and the likelihood that validation will fail. Thus, the SR policy is ideal when using smaller eviction block sizes on devices with low latencies.

4.2 Merging Threshold

Another important policy is where the DBMS should put tuples when they are brought back into memory. It could merge them back into the regular table storage (i.e., heap) immediately after a transaction accesses it so that future transactions can use it as well. This is important if the cost of reading from the device is high. But given that OLTP workloads are often skewed, then it is likely that the data that was just retrieved is still cold and will soon be evicted again. This thrashing will degrade the DBMS's performance.

One way to reduce this oscillation is to delay the merging of cold tuples. When a transaction accesses evicted tuples, the DBMS stores them in a temporary in-memory buffer instead of merging them back into the table. Then when that transaction finishes, the DBMS discards this buffer and reclaims the space. This avoids the thrashing problem described above, but requires that the DBMS also track the access frequency of evicted data so that when access pattern changes and certain cold data becomes hot, the DBMS can bring them back into the regular table. The storage overhead of this tracking information should be minimal. Our implementation of this approach maintains a count-min sketch to approximate the access frequency of evicted data because it only uses a small amount of storage space that is proportional to the total frequency of all tracked items [11]. The DBMS needs to perform more reads from the secondary storage when the merging threshold is higher. Thus, this setting benefits storage technologies with lower read costs.

4.3 Access Methods

Up until now, we have assumed that the DBMS manages cold tuples in a block-oriented manner. That is, the system reads and writes data from the secondary storage device in batches that are written sequentially. Although this mitigates the high latency of storage device by reducing the number of reads/writes, it causes the DBMS to almost always read more data in than it actually needs whenever a transaction accesses a single evicted tuple.

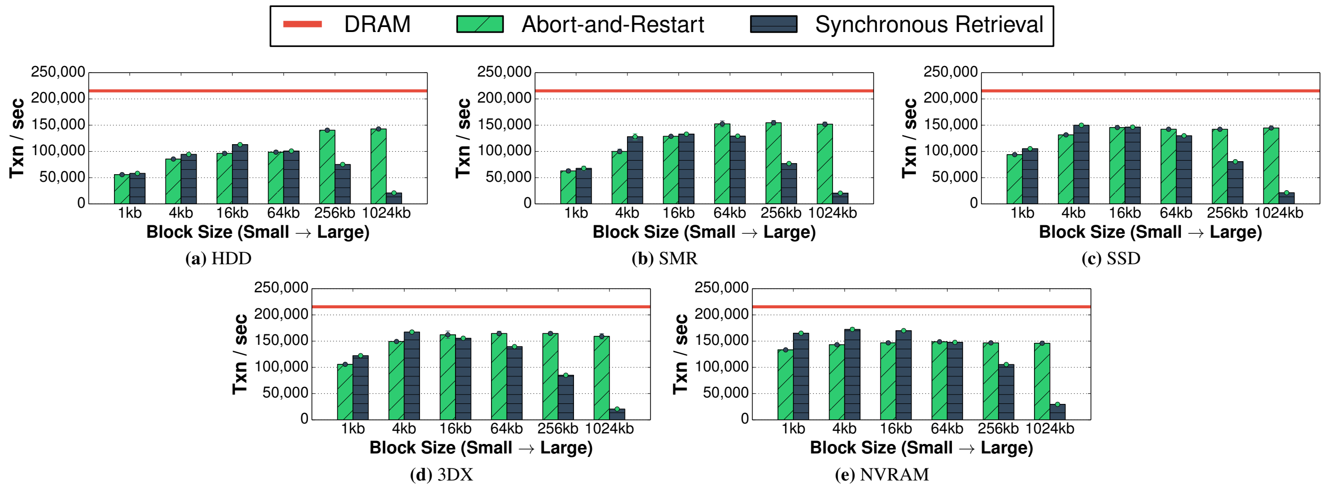


Figure 2: Cold Tuple Retrieval – Throughput for the YCSB workload in H-Store with anti-caching when using different storage devices with the two retrieval policies from Sect. 4.1. For each device and retrieval configuration, we scale up the size of the cold tuple blocks from 1 to 1024 KB.

This block-oriented model could be inappropriate for the future NVRAM storage that is able to support byte-level operations. Instead of organizing tuples into blocks, an alternative approach is to map a portion of the DBMS's address space to files on storage devices using `mmap` system call and then move the cold data to the mapped region. This is similar to the approach taken in EPFL's variant of VoltDB [22] except that the DBMS controls the flushing process (rather than the OS). We adopt this approach by using a filesystem designed for byte-addressable NVRAM (PMFS) [15]. This enables the DBMS to operate directly on NVRAM-resident data as if it existed in DRAM, thereby obviating the need for merging it.

5. EXPERIMENTAL ANALYSIS

In this section, we present our analysis of how hardware affects the choice of the policies the DBMS adopts. For cold tuple retrieval and merging policies, we show the effect of them on each hardware type and discuss what the better policy is for each of them. For access method policies, we restrict our analysis to NVRAM as it is the only device that can bypass the OS's filesystem API for byte-level access. We finish with an evaluation of the benefits of choosing policies that are tailored for each device.

In these experiments, we use a single machine with a dual-socket Intel Xeon E5-4620 @ 2.60 GHz (8 cores per socket) and 4×32 GB DDR3 RAM (128 GB total). This single server supports the five storage devices that we described in Sect. 2:

HDD: Seagate Barracuda (3 TB, 7200 RPM, SATA 3.0)

SMR: Seagate Archive HDD v2 (5 TB, 5900 RPM, SATA 3.1)

SSD: Intel DC S3700 (400 GB, SATA 2.6)

3DX: This is a PCIe emulator from Intel Labs that supports latencies that closely matches the real hardware prototype.

NVRAM: This is another emulator from Intel Labs that supports byte-addressable reads/writes [15]. We configure the emulator's latency to be 4× that of DRAM [8].

We implemented the three hardware dependent policies in the anti-caching component [12] for the H-Store DBMS [18]. Anti-caching performs on-line cold data identification by keeping tombstones in memory to represent evicted tuples, and monitoring its memory usage. We note that the hardware dependent policies that we examine in this analysis are equally useful in other DBMSs.

5.1 Cold Tuple Retrieval

For our first experiments, we use the YCSB [10] workload comprised of 90% read transactions and 10% write transactions. We start with comparing the cold tuple retrieval policies: (1) abort-and-restart (AR), and (2) synchronous retrieval (SR). We also vary the size of the eviction block for each of these two policies. We use a 10 GB database with 1.25 GB DRAM available to the DBMS for hot tuple storage. Each YCSB tuple is 1 KB. The workload generator is configured to use a highly skewed distribution for transactions' access patterns (*factor*=1.0). Under this distribution, 90% of the transactions are accessing only 10% of the tuples.

We now discuss the results for each of the storage devices shown in Fig. 2. Each graph also includes the upper-bounds performance measurement of when the database fits entirely in DRAM.

HDD: The results in Fig. 2a show that the AR policy achieves the best performance with large block sizes because HDD's read latency is high and it is faster to perform sequential writes. Likewise, because of their high write latencies, reducing the number of writes by using larger blocks improves the performance. These larger blocks make reading data take longer, but it does not affect the throughput with AR since the retrieval is outside the critical path of transactions. The SR policy, however, stalls the DBMS from executing transactions for a long time when they access evicted data.

SMR: In Fig. 2b, we see that the SMR's performance trend with different block sizes is similar to HDD's because these two devices have similar characteristics. The AR policy also achieves the best performance with large block sizes. Another observation is that the larger variance in the read/write latency does not affect the SMR's performance that much as compared to HDD.

SSD: Fig. 2c shows that larger block sizes do not provide better performance with SSDs like with HDDs because sequential writes are not much faster. As we showed in Fig. 1, writing a 64 KB block is 80% slower than writing a 1 KB block with a SSD. Smaller block size enables finer-grained data retrieval, which is beneficial to the SR policy. But using a size less than 4 KB hurts performance because it is smaller than what the hardware supports. The best performance of the two policies are similar because the overhead of aborting and restarting transactions negates the advantage of non-blocking data retrieval. Nevertheless, the performance with the AR policy is more stable across different block sizes.

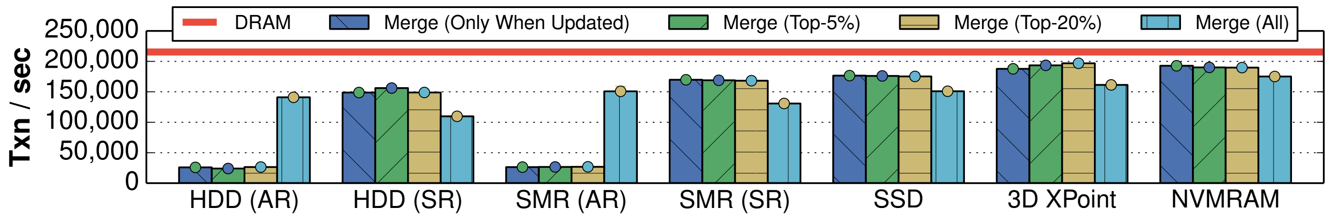
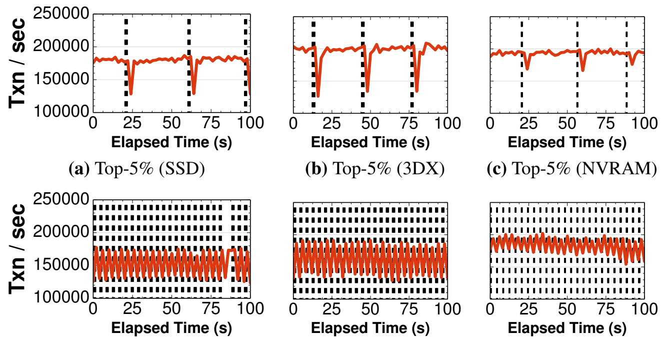


Figure 3: Merging Threshold – Throughput for YCSB in H-Store with anti-caching under different merge threshold policies.



(a) Top-5% (SSD) **(b)** Top-5% (3DX) **(c)** Top-5% (NVRAM)

(d) Merge All (SSD) **(e)** Merge All (3DX) **(f)** Merge All (NVRAM)

Figure 4: Merging Threshold (Sustained Throughput) – Throughput for YCSB in H-Store under different merge threshold policies. The vertical bars denote when the DBMS evicts tuples out to secondary storage. The y-axis represents the real-time throughput of the DBMS in every second.

3DX: The results in Fig. 2d are similar with SSD's except that the overall performance is slightly higher. According to Fig. 1, the 3DX is 5.5–10× faster than the SSD, but this does not significantly improve the performance when the workload is highly skewed and the cold storage is accessed infrequently. In Sect. 5.4, we show results when the workload distribution is more uniform.

NVRAM: Assuming that NVRAM's latencies will be comparable to DRAM, we would expect that the overhead of the AR policy is higher than with SR. Indeed, Fig. 2e shows that the advantage to use larger blocks to reduce random writes is insignificant. The best policy for NVRAM is to use SR with the smallest block size. We note that the SR's performance with 1 KB block is slightly lower than with 4 KB block. This is because reducing the number of random writes still has little benefit given the latency of NVRAM is 4× higher than DRAM, and thus writing 4 KB at once to NVRAM is faster than writing 1 KB four times.

5.2 Merging Threshold

We next analyze the impact of merging thresholds on the DBMS's performance with YCSB. When an evicted tuple is accessed but not modified by a transaction, the DBMS determines whether a tuple should either be put into a temporary buffer or merged back into the table based on its access frequency. If a tuple is updated, then the DBMS always merges it back into the table because otherwise it has to be written back to the secondary storage. The four policies that we evaluated are (1) to only merge when a tuple is updated, (2) to only merge when a tuple is the top 5% most accessed, (3) to only merge when a tuple is the top 20% most accessed, and (4) to always merge a tuple. We use one Count-Min Sketch per table to track the cold tuple access frequencies; each Sketch uses ∼100 KB of memory for a table with 10m tuples. We use the same workload settings as in Sect. 5.1, and pick the best retrieval policy and block sizes for each device based on those results.

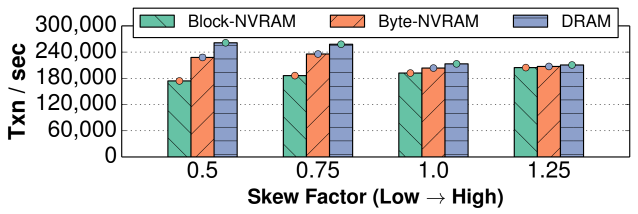


Figure 5: Access Methods – Comparison between block-based or tuple-based access methods for cold-data storage using DRAM and NVRAM.

HDD & SMR: The first thing to notice is that the AR policy does not work well for these devices with selective merging because it increases the number of reads from secondary storage. This is especially harmful when using 1024 KB blocks. For comparison, we also test the performance of SR policy with HDD/SMR with 16 KB blocks. The result in Fig. 3 shows that for that policy the merging threshold can help the performance of the DBMS improve to a similar throughput as with AR policy.

SSD, 3DX, & NVRAM: The results in Fig. 3 show that selectively merging data always improves performance with the three solid-state devices. This is because the DBMS does not have to keep moving data in and out of secondary storage. On average, setting the merging threshold to the top 5% frequently accessed tuples increases the eviction interval by 6× for all the three devices. To measure this effect, Fig. 4 shows times-series graphs of the sustained throughput of the DBMS when it only merges the top 5% of evicted tuples back into memory. These graphs also indicate that the DBMS has smallest throughput drop with NVRAM when data is evicted compared to the other devices.

5.3 Access Methods

We now investigate the impact of access method policies with NVRAM. We use the same YCSB workload as in the previous two sections, except we vary the skew setting in the transaction's access patterns from low (*factor*=0.5) to high (*factor*=1.25).

The results in Fig. 5 show that directly accessing NVRAM improves overall performance, especially for low-skew workloads. When the latency of the persistent device is comparable to DRAM, the overhead of tuple-to-block transformation and the file-system cache becomes significant. The performance difference with higher skew is minor because the data exchange between memory and secondary storage is infrequent. Because the H-Store DBMS we use for evaluation executes queries serially at disjoint partitions each of which is handled by a single thread, the throughput of the DBMS is limited by the CPU speed when the number of quries operate on a specific partition is significantly larger than others. This leads to the performance degradation when the workload skew is higher. The takeaway from this experiment is that future DBMSs will want to use NVRAM as an extension of their address space rather than treating it as just a faster SSD.

	Block Size	Retrieval	Merging	Access Method
Default	1024 KB	AR	Merge-All	Block-level
HDD	16 KB	SR	Top-5%	Block-level
SMR	16 KB	SR	Top-5%	Block-level
SSD	4 KB	SR	Top-5%	Block-level
3DX	4 KB	SR	Top-5%	Block-level
NVRAM	N/A	SR	Top-5%	Byte-level

Table 1: Policy configurations for the different storage devices.

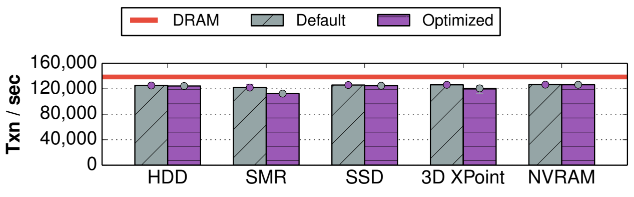
5.4 Additional Workloads

Lastly, we measure the performance of the DBMS using all best policy configurations for each storage device on other OLTP workloads (see Table 1). We compare each optimized configuration with a single "default" configuration from the original anti-caching paper [12]. For HDD and SMR, the default configuration performs the same as their optimized policies in YCSB, but we still test them for comparison. The tables in each database are partitioned in such way that there are only single-partition transactions [20]

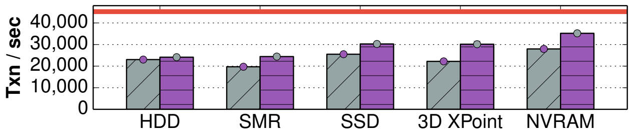
Voter: This benchmark simulates a phone-based election application. It is designed to saturate the DBMS with many short-lived transactions that each updates a small number of tuples. Fig. 6a shows that the DBMS's performance with the optimized and generic configurations is similar for all devices. That is because Voter's transactions insert tuples that are never read. Thus, the DBMS's cold-data component just writes tuples out to secondary storage, which does not occur too frequently since the tuples are relatively small (0.23 KB). The performance of the optimized strategy is slightly lower than that of the default configuration, because writes are less sequential with smaller block sizes.

TPC-C: This benchmark is the industry standard for evaluating the performance of OLTP systems [27]. It consists of five transaction types that simulate an order processing application, but only one of them accesses evicted tuples (`OrderStatus`) and it is only 4% of the workload. Thus, the results in Fig. 6b show that there is not a significant improvement in performance for most devices because the movement of data back-and-forth between memory and secondary storage is not as frequent in TPC-C as it is in YCSB. It is only with 3DX and NVRAM do we see a more prominent improvement over the generic configuration (36% and 26%, respectively).

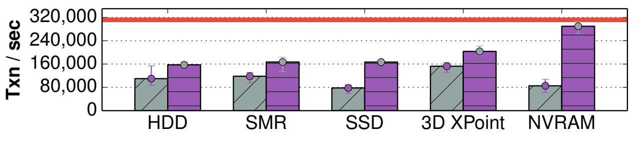
TATP: This last workload simulates a caller location system used by cellphone providers. In Fig. 6c we see that the optimized configurations always outperform the default configuration. This is for two reasons. First, the TATP workload is not skewed, and thus there are more transactions that access evicted data than in TPC-C or Voter. This amplifies the advantage of hardware-optimized policies. Second, the tuple sizes in TATP (0.18–0.57 KB) are much smaller than in YCSB, and each transaction may access multiple evicted tuples. This makes using smaller eviction block sizes ideal because the DBMS is able to retrieve evicted tuples with a finer granularity. Further, the throughput of the default policy is unpredictable because the DBMS is unable to bring back the evicted tuples to memory fast enough. This shows that faster storage devices do not always improve performance if they are used incorrectly.



(a) Voter



(b) TPC-C



(c) TATP

Figure 6: Additional Workloads – Throughput measurements for H-Store with anti-caching when using the optimal hardware-dependent policy configuration for each storage device compared to a default configuration.

6. RELATED WORK

The main challenge in supporting larger-than-memory databases on an in-memory DBMS lies in accessing the data stored on a secondary storage device without slowing down the regular in-memory operations. That is, allowing the DBMS to move data back and forth between memory and disk without relying on a buffer pool and other legacy components [17]. We now describe how previous systems achieve this. There are other DBMSs that claim to support larger-than-memory databases (P*TIME and SolidDB), but we are unable to find documentation on their techniques.

The *anti-caching* architecture for the H-Store [3] DBMS moves cold tuples from DRAM to a disk-resident hash table [12]. The original version tracks the LRU information on a per-tuple basis using a linked list that is embedded in their headers. Storing the list in this manner reduces the storage overhead and improves cache locality by avoiding a separate data structure. To evict data, the DBMS invokes a special transaction that blocks other transactions while it combines the coldest tuples into a block and writes them out to the anti-cache. The DBMS maintains a tombstone for each evicted tuple that stores where the data can be found in the hash table. When a transaction attempts to access an evicted tuple through its tombstone, the DBMS aborts the transaction and then asynchronously fetches the requested tuple in a separate thread. Once the data is brought back into memory, the DBMS removes the tombstone, updates index pointers, and then re-executes the transaction.

Microsoft's *Project Siberia* [16] for Hekaton [13] identifies what tuples to evict in a background thread that analyzes the DBMS's logs [19]. This avoids the overhead of having to maintain a tracking data structure that is updated during execution. The cold tuples are moved to secondary storage using a special migration transaction that is composed of insert and delete operations. Instead of using tombstones, Siberia maintains a Bloom filter for each index to track evicted tuples. Range queries are supported by another special kind of filter [7]. The DBMS needs to check both the regular indexes and filters for each query. An evicted tuple is merged back into memory only when it is updated by a transaction, otherwise it is stored in a private cache and then released after that transaction terminates.

Researchers at EPFL created a variant of VoltDB that used the OS's virtual memory to support large-than-memory databases [22]. The DBMS divides its address space into pinned and unpinned regions (using `madvise`). The former is used to store cold tuples that the OS is allowed to evict, whereas the pinned region is for hot data that the OS is prohibited from moving to disk. Similar to anti-caching, the DBMS collects statistics about how transactions access tuples. However, it uses a separate background thread to analyze this information off-line. Instead of explicitly moving the data to disk as done in anti-caching and Siberia, the DBMS moves the cold tuples to the unpinned region. The advantage of this approach is that the DBMS does not need to track evicted tuples. It is susceptible, however, to stalls due to page faults.

Apache Geode supports the ability to declare an in-memory table as evictable. It identifies cold tuples simply based on their insert order (FIFO). Evicted tuples are written out to a log file stored on HDFS and the DBMS maintains some additional meta-data in-memory to avoid false negatives.

The current version of MemSQL supports external tables [5]. When an in-memory table gets too large, the DBA can manually extract cold tuples and then reload them into a column-based table stored on an SSD. These tables appear to the application as a separate logical table that can be used together with in-memory data for queries. This means that a developer has to rewrite their application's queries if they want to combine data from the in-memory and the external tables.

7. CONCLUSION

This paper presented policies for managing cold data storage in an in-memory DBMS. We outlined policies that are both independent and tightly coupled to the underlying storage technology. We then evaluated these policies in the H-Store DBMS using five different hardware devices. Our results showed that tailoring the strategy for each storage technology improves throughput by up to 3× over a generic configuration. We found that smaller block sizes and synchronous retrieval policy are generally good choices for storage devices that have low access latencies, such as SSD, 3DX, and NVRAM. Limiting the number of cold tuples that are merged back into table storage is effective on reducing the throughput oscillation. Finally, the performance of the DBMS with NVRAM can be as good as with DRAM if treated correctly.

ACKNOWLEDGMENTS

This work was supported (in part) by the Intel Science and Technology Center for Big Data and the National Science Foundation (Award CCF-1438955).

For questions or comments about this paper, please call the CMU Database Hotline at `+1-844-88-CMUDB`.

8. REFERENCES

[1] 3D XPoint Technology Revolutionizes Storage Memory. http://www.intel.com.

[2] Apache Geode. http://geode.incubator.apache.org/.

[3] H-Store. http://hstore.cs.brown.edu.

[4] MemSQL. http://www.memsql.com.

[5] MemSQL – Columnstore. http://docs.memsql.com/4.0/concepts/columnstore/.

[6] MongoDB: New Storage Architecture. https://www.mongodb.com/blog/post/whats-new-mongodb-30-part-3-performance-efficiency-gains-new-storage-architecture.

[7] K. Alexiou, D. Kossmann, and P.-Å. Larson. Adaptive range filters for cold data: Avoiding trips to siberia. *Proceedings of the VLDB Endowment*, 6(14):1714–1725, 2013.

[8] J. Arulraj, A. Pavlo, and S. R. Dulloor. Let's talk about storage & recovery methods for non-volatile memory database systems. In *SIGMOD*, pages 707–722, 2015.

[9] F. A. Colombani. HDD, SSHD, SSD or PCIe SSD-which one to choose? StorageNewsletter, April 2015.

[10] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *SoCC*, pages 143–154, 2010.

[11] G. Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

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

[13] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: SQL Server's Memory-Optimized OLTP Engine. In *SIGMOD*, pages 1–12, 2013.

[14] A. Driskill-Smith. Latest advances and future prospects of STT-RAM. In *Non-Volatile Memories Workshop*, 2010.

[15] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson. System software for persistent memory. In *Eurosys*, 2014.

[16] A. Eldawy, J. Levandoski, and P.-Å. Larson. Trekking through siberia: Managing cold data in a memory-optimized database. *Proceedings of the VLDB Endowment*, 7(11):931–942, 2014.

[17] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker. OLTP through the looking glass, and what we found there. In *SIGMOD*, pages 981–992, 2008.

[18] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. 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. *Proc. VLDB Endow.*, 1(2):1496–1499, 2008.

[19] J. J. Levandoski, P.-A. Larson, and R. Stoica. Identifying hot and cold data in main-memory databases. In *ICDE*, 2013.

[20] A. Pavlo, C. Curino, and S. Zdonik. Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems. In *SIGMOD*, pages 61–72, 2012.

[21] S. Raoux, G. W. Burr, M. J. Breitwisch, C. T. Rettner, Y.-C. Chen, R. M. Shelby, M. Salinga, D. Krebs, S.-H. Chen, H.-L. Lung, et al. Phase-change random access memory: A scalable technology. *IBM Journal of Research and Development*, 52(4.5):465–479, 2008.

[22] R. Stoica and A. Ailamaki. Enabling efficient OS paging for main-memory OLTP databases. In *DaMon*, 2013.

[23] M. Stonebraker. Operating system support for database management. *Commun. ACM*, 24(7):412–418, 1981.

[24] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era: (it's time for a complete rewrite). In *VLDB*, pages 1150–1160, 2007.

[25] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams. The missing memristor found. *nature*, 453(7191):80–83, 2008.

[26] A. Suresh, G. Gibson, and G. Ganger. Shingled Magnetic Recording for Big Data Applications. Technical report, 2012.

[27] The Transaction Processing Council. TPC-C Benchmark (Revision 5.9.0). http://www.tpc.org/tpcc/, June 2007.