Cost/Performance in Modern Data Stores

How Data Caching Systems Succeed

David Lomet
Microsoft Research
Redmond, Washington
lomet@microsoft.com

ABSTRACT

Data in traditional "caching" data systems resides on secondary storage, and is read into main memory only when operated on. This limits system performance. Main memory data stores with data always in main memory are much faster. But this performance comes at a cost. In this paper, we analyze the costs of both in-memory operations and secondary storage operations where data is not "in cache". We study the performance impact of cache misses on caching system performance. The analysis considers both execution and storage costs. Based on our analysis, we derive cost/performance results for a data caching system [Deuteronomy and its Bw-tree] and a main memory system [MassTree] to understand where each demonstrates the best cost per operation, what is driving the cost differences, and the scale of the differences. This analysis (1) provides insight into why data caching systems continue to dominate the market; (2) points to higher performance that does not rely on simply increasing main memory cache size; and (3) suggests a path to lower costs and hence better cost/performance.

ACM Reference Format:
David Lomet. 2018. Cost/Performance in Modern Data Stores: How Data Caching Systems Succeed. In *DaMoN'18: 14th International Workshop on Data Management on New Hardware, June 11, 2018, Houston, TX, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3211922.3211927

1 INTRODUCTION

1.1 Traditional Data Management Systems

Traditional data management systems were implemented during the hard disk drive (HDD) era [14, 24, 26]. Such systems assumed that main memory was expensive and limited in size. HDDs were a factor of 50 less expensive per byte. Applications neither required nor expected throughput more than a thousand transactions per sec when executing multiple read and update operations within a transaction [33].

These factors resulted in such systems being designed as data caching systems. That is, data was expected to live on HDDs, and be brought into a main memory cache when the data was operated on. B-trees [3] were the dominant indexing structure. The expectation was that the top levels of the B-tree would fit in cache, but that most of the leaf nodes, where the data resided, would need to be brought into main memory when needed.

As main memories became larger and costs fell, more and more data was cached in main memory, frequently including the hotter data pages themselves. Removing I/O cost from the path to this hotter data, by itself, enabled substantially higher performance. It exposed, however, a new set of bottlenecks to high performance. Concurrency control and recovery (CCR), when most data was accessed from HDDs, had modest execution cost compared with I/O accesses to reach the data. With the I/O cost reduced, CCR costs played a much larger role in limiting performance.

1.2 Main Memory Systems

Database researchers [16, 17, 35] began looking for ways to substantially enhance data management performance. There was particular focus on reducing the cost of CCR. The implicit assumption was that I/O to access data would be rare. Ultimately, CCR techniques were developed that ended up depending, for performance and effectiveness, upon there being no I/O, as high latency operations, particularly within high latency multi-operation transactions, did not fit well with the CCR techniques being proposed.

This led to a leap in the number of in main memory data stores [7, 18, 32], transactional and non-transactional, key value stores and full database systems. These systems led to new levels of performance, where millions of operations/sec became possible, and in some cases millions of transactions/sec. Such systems found acceptance in places that required performance that was difficult or impossible to achieve with traditional systems. To a first approximation, these systems required all the data to be in main memory all the time. The exceptions required that access to data on secondary storage, even SSDs, be very rare.

Main memory systems, with their potential for great performance, led to an explosion of new CCR and data access techniques suitable for such systems. Otimistic and multi-version methods, even serial execution, have been explored. Various forms of latch-free access to system data structures became common. Main memory also permitted data access via main memory pointers. All of this both led to and required very low latencies. Hence, it came at the high cost of main memory permanently committed to the data being managed. [1]

1.3 New Data Caching Systems

There are promising signs that it is possible to achieve much higher throughput for data caching systems than was previously possible.

[1] Two efforts [4, 9] softened this strict requirement, to accommodate data larger than available main memory.

DaMoN'18, June 11, 2018, Houston, TX, USA

ACM ISBN 978-1-4503-5853-8/18/06…$15.00
https://doi.org/10.1145/3211922.3211927

Examples of such systems are RocksDB [28] (an open source system) and Deuteronomy [22]. Both of these systems exploit latch-free access to their data structures. Both dramatically shrink write I/O via the use of log-structuring techniques. For RocksDB, it is the LSM-tree. For Deuteronomy, it is its LLAMA log-structured store. The systems also share the property that "blind" updates do not require the page containing the prior values being updated to be in main memory. Thus, they both use main memory techniques for operational speed in operating on main memory data, and log-structuring techniques to dramatically reduce the number of I/Os.

1.4 What's in the Paper

We look at the performance (throughput as measured in read and/or update operations per second) and the cost/performance of one modern data caching system (Deuteronomy). We only consider the costs of bringing data into main memory in this analysis. We assume, because of log-structuring, that the write cost is an insignificant factor in the level of analysis here. We then compare it to a well-known main memory system, the MassTree [23].

There are several limits to the analysis. The prices of hardware are gleaned from the web, and have wide variability at any time and more over time. Fortunately, it is relative prices that are important, and these change more slowly. Performance results used to determine cost/performance are the result of a limited number of experiments, under our specific system environment. So "mileage" will vary. Despite these caveats, the analysis paints a good "big" picture of data management cost/performance, and provides insights into the underlying "why" as well.

Our cost/performance analysis highlights the unique strength of data caching systems. Because they can move data between high performance but high cost DRAM storage and low performance but low cost flash storage (for example), they can adapt to their workload in a way that other systems cannot easily do. This permits them to choose the most cost effective way to operate on any given piece of data. What that is depends upon how hot the data is. If hot, we want it in DRAM for high performance, if cold, we want it only in flash.

In addition to our cost/performance analysis, we describe techniques used by modern data caching systems, not just Deuteronomy but also RocksDB, that are addressed to reducing the cost of operations. Specific to data caching systems, we focus on how to reduce the cost of using secondary storage. These techniques are helpful in (1) reducing the freqency of I/O and (2) reducing the cost of an I/O access. Both of these directly improve cost/performance.

The hope is that our community will re-focus its efforts away from pure performance to a focus on cost/performance instead. Main memory data systems have had only a modest impact on the database market. In re-focusing on data caching systems, with their ability to flexibly control storage and execution costs, the work that we do as a community should have wider relevance and impact.

2 DATA CACHING PERFORMANCE

2.1 Two Operation Forms

A caching data system operation that does not find its data in cache is much slower and more costly than one operating on cached data. In addition to executing the operation logic, it must issue a read I/O, switch the execution to other work during I/O latency, and then switch back to continue the operation. Cost increases because of the increased execution path and the secondary storage access. We want to determine the relative cost of these operations and the impact of executing a mixed workload of in-cache (main memory or *MM*) operations and operations that miss in the cache and need to access secondary storage (secondary storage or *SS*) operations.

We start by considering the result of executing a mixed workload of operations, based on relative performance R of *SS* operations compared to *MM* operations. Performance here is measured as the execution time that one core needs to complete an operation. This is not the same as latency since we do not include time waiting for I/O to complete, only the time the core spends executing the operation. This performance is a measure of the computational load of the operation. And it is this, for a processor bound workload, that determines the per core throughput in operations/sec.

R is not constant as modern processor performance does not depend soley on execution path, but rather varies with workload and is affected by how warm processor caches are. But it varies in a limited range. We use R to provide insight into system performance and cost.

2.2 Mixed Operation Performance

What relative performance can we expect when I/Os are present in the mix of operations? Let F be the fraction of *SS* operations in the total set of operations. Let PF be the number of ops/sec we achieve when the fraction of *SS* operations equals F. $P0$ is the ops/sec when no operations are *SS* (all are *MM* and $F = 0$). The execution time (secs/op) for an *MM* operation is $\frac{1}{P0}$. For an *SS* operation it is $R * \frac{1}{P0}$ since *SS* operations consume R times the processor execution time. Thus, the weighted average execution time for an operation that is part of a mix of *SS* and *MM* operations is:

$$\frac{1}{PF} = (1 - F) * \frac{1}{P0} + F * R * \frac{1}{P0} \quad (1)$$

Performance is the inverse of execution time. We find it by solving for PF, i.e., by inverting equation 1. So, for a mix of operations, we can expect:

$$PF = P0 * \frac{1}{(1 - F) + F * R} \quad (2)$$

Performance drops toward *SS* performance as the fraction F of *SS* operations increases. F is the cache miss ratio cited in caching discussions. At a cache miss ratio of 1, the Bw-tree runs at $\frac{1}{R}$ of in-memory performance, i.e. at *SS* operation performance. We worked hard to reduce the execution cost of an I/O operation in Deuteronomy by introducing user level I/O [15]. We then ran experiments where we monitored the number of *MM* and *SS* operations. Our success in reducing *SS* operation cost shows up in these experimental results, which we use to derive R. Equation 2 above associates performance with R. We can solve this for R, and with our experimental results, we can determine a range of values for R.

$$R = 1 + \frac{1}{F} * \left[\frac{P0}{PF} - 1\right] \quad (3)$$

From our experiments, we conclude that over most of the range of access rates we tested, R was $5.8 \pm 30\%$. R was outside of this range when the I/O path was very cold. Another potential problem

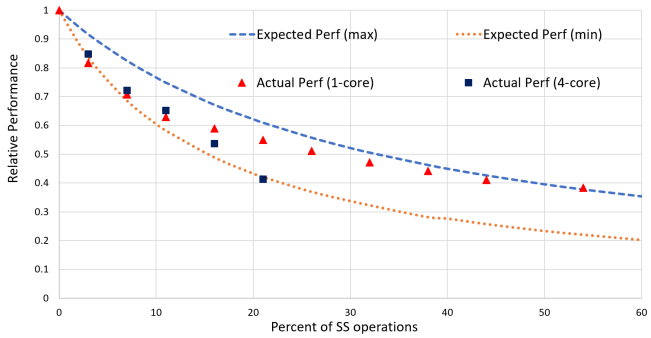

Figure 1: Relative performance (y-axis) of a mixed workload of *MM* and *SS* operations (x-axis) in percent of the operations that are *SS* operations.

(we saw evidence for it) in determining a reasonable value for R occurs when the system becomes I/O bound. Our analysis assumes that is not the case.

Figure 1 shows a range of relative performance versus the fraction F of *SS* operations. The two dotted lines show the range of values for $R = 5.8 \pm 30\%$. The figure also shows the actual 1-core and 4-core experimental results, which fall in the region between the two R curves. This graph captures the intuition that as slow *SS* operations that include an I/O are mixed with fast *MM* operations, that one needs to expect performance to decline toward the performance of the slower operations, as slow operation execution squeezes out opportunities to execute the faster operations. We will use R in our subsequent cost analysis, and there use the value $R = 5.8$. But keep in mind that this is a simplifying assumption. Also remember that this is the optimized Deuteronomy R. Conventional operating system I/O path is longer and will produce a larger R.

Given a certain fixed hardware configuration, one will *ALWAYS* achieve higher performance by keeping all data in main memory and using all processing power to execute *MM* operations. The fall-off in performance when data has to be brought into the main memory cache is substantial, even for highly optimized I/O. So why bother with a data caching system? That is answered in the next section.

3 DATA CACHING COST/PERFORMANCE

A data caching system can adapt for lowest cost depending upon load as follows.

(1) Like many database systems, it can assign more or fewer hardware cores to the execution of a workload.
(2) Unique to data caching systems, it can move data between main memory and secondary storage, changing its mix of *MM* vs *SS* operations.

The analysis here determines the relative costs of *MM* vs *SS* operations, which depend upon how hot the data is.

3.1 Infrastructure Factors

We "rent" resources for operation execution. Thus we need rental costs, i.e. the cost/sec, derived by dividing price paid by lifetime L (in seconds). We assume that all infrastructure has the same lifetime, permitting us to ultimately eliminate L from our equations. We include lifetime here since it makes the discussion easier.

The costs of interest then are:

- \$M: cost/byte of main memory (DRAM). The cost/byte/sec is $\frac{\$M}{L}$.
- \$Fl: cost/byte of SSD memory (flash [10]). The cost/byte/sec is $\frac{\$Fl}{L}$.
- \$P: the cost of the processor. $\frac{\$P}{L}$ is its cost/sec.
- \$I: the cost of an SSD's I/O capability. This comes bundled with the SSD so we use a couple of methods for deriving this cost (see below). $\frac{\$I}{L}$ is its cost/sec.

3.2 Operations and Their Cost/Sec

We analyze costs of our two forms of operation: *MM* and *SS*. The costs are of two forms, storage costs and execution costs. To determine the storage cost per second, we divide the total cost by the lifetime L. To determine the execution cost, we multiply the cost for the execution of one operation by the number N of operations/second.

Main Memory Operation Costs: *MM* operations find their data in the DRAM cache. We assume that the data is also on secondary storage for durability. Further, we assume that the data caching system storage is organized in pages of size P_s. For most data caching stores, a page is the unit of storage we can transfer between cache and secondary storage. *MM* operations have costs:

Storage:

- rental of a page of main memory (DRAM) (per second): $\frac{\$M}{L} * P_s$.
- rental of secondary storage, needed for durability (per second): $\frac{\$Fl}{L} * P_s$, for the same page size P_s.

Execution:

- rental of the processor for one operation, determined experimentally by dividing $\frac{\$P}{L}$ by *ROPS*, the number of *MM* operations we execute in a second (from our experiments).

Secondary Storage Operation Costs: *SS* operations find their data on secondary storage, avoiding main memory rental at the cost of reading data from secondary storage to perform an operation. *SS* operations have costs:

Storage:

- rental of secondary storage (per second): $\frac{\$Fl}{L} * P_s$, for page size P_s.

Execution

- rental of the processor for one operation, determined experimentally by dividing $R * \frac{\$P}{L}$, the cost of the processor for one second by *ROPS*.
- rental of one SSD I/O access to bring data into main memory: by dividing $\frac{\$I}{L}$, the cost per second for the SSD I/O capability by *IOPS*, the maximum number of I/O operations per second we achieve (in our experiments) using the SSD.

To determine the cost/sec for executing a given number N of operations/sec, we multiply the execution cost/operation above by N, yielding execution cost per second. We then add it to the

storage cost per second. Bringing these costs together then yields the following:

$$\$MM = \frac{1}{L} * \left[P_s * (\$M + \$Fl) + N * \frac{\$P}{ROPS}\right] \quad (4)$$

$$\$SS = \frac{1}{L} * \left[P_s * \$Fl + N * \left\{\frac{\$I}{IOPS} + R * \frac{\$P}{ROPS}\right\}\right] \quad (5)$$

Because we are interested in relative costs, lifetime L will always cancel out. So we drop L from subsequent equations and discussion, with the understanding that when we discuss cost for an operation, there is an implicit $\frac{1}{L}$ factor.

4 COSTS

4.1 Hardware Costs

For our cost computation, we need infrastructure costs, performance quantities *IOPS* and *ROPS*, and page size P_s. *ROPS* come from our 4-core experiments. *IOPS* come from experiments we ran on our Samsung SSD. Our pages vary in size but P_s averages $2.7 * 10^3$ bytes. Deuteronomy pages are variable in size and approximately 100% utilized, and we set maximum page size at 4K bytes. B-tree pages have average utilization of just under 70%. The combination of B-tree-style page splitting and 4K pages explains our average page size P_s. In Section 2.2, we determined that the relative cost of *SS* vs *MM* operations is around $R \approx 5.8$. Below are estimates for the costs of our server infrastructure. Note again that costs change continuously and vary across vendors, and between server and desktop. So cost estimates are necessarily rough.

Processor:

- Costs: $\$M = \$5 * 10^{-9}$; $\$P = \$3 * 10^2$;
- Performance: $ROPS = 4 * 10^6$. This is our experimentally determined read operation rate.

SSD:

- Costs: $\$Fl = \$.5 * 10^{-9}$; \$I is the difference between SSD cost and flash storage cost for our .5TB drive: $\$I = \$300 - \$250 = \$50$. [2]
- Performance: $IOPS = 2.0 * 10^5$, our experimentally determined maximum I/O rate.

4.2 Our 5-minute Rule

At low access rates, operation cost is dominated by storage cost. At high rates, it is dominated by execution cost. Here's why? At low ops/sec, *SS* is cheaper than *MM* because *SS* (flash) storage cost is cheaper than *MM* (DRAM + flash) storage cost by a factor of about 11X. However, *SS* execution cost is more costly than *MM* execution cost by a factor of about 12X. Thus, at sufficiently high ops/sec, it is the execution cost that dominates. At that point, it becomes cheaper to store data in main memory and execute *MM* operations on it than it is to keep it solely on flash and execute *SS* operations on it.

We can now derive the relative costs of the two flavors of operations in a caching data management system. From this, we can calculate the breakeven point- i.e., how many op/sec N we need to execute for the costs to be equal. For lowest cost cache management, we switch between MM and SS operations when the number of operations per second reaches this value.

We set costs equal and solve for N. But solving for $1/N$ results in a simpler equation. Let $\frac{1}{N} = T_i$, the interval between operations at this operation execution rate.

$$T_i = \frac{1}{N} = \frac{1}{\$M * P_s} * \left[\frac{\$I}{IOPS} + (R-1) * \frac{\$P}{ROPS}\right] \quad (6)$$

T_i at the value calculated with this formula yields our updated "5-minute rule". We include in our analysis not just the trade-off between the I/O access cost and main memory (first term) but also the processing cost of executing the I/O path (second term), an **additional cost**. The number of IOPS provide by an SSD has dramatically increased, reducing the cost of an I/O. This means that CPU cost for the I/O becomes a larger part of the overall cost of operations, and begins to play a more important role in determining breakeven point T_i. Further, including storage costs in overall operation costs permits us to assess these costs at points away from the breakeven point.

We determine T_i is approximately 45 seconds at breakeven (the updated "5 minute rule" [1, 11, 12]). With time between page accesses > 45 seconds, it is less costly to evict the page and execute a secondary storage operation than to retain the page in memory and use a main memory operation. The smaller the breakeven time, the sooner the execution cost is improved by evicting the page. The cost trade-off between *MM* and *SS* operations is shown in Figure 2. Thus, we are better off with data on flash and adding additional processors to help in the execution than we are to increase our use of main memory for the data cache when data is sufficiently cold. It takes a lot of cold data in the cache before the cache hit ratio changes very much.

A data caching system can use the breakeven point for guidance in choosing the lower cost operation. The fact that this breakeven point has shrunk of late is evidence that SSD IOPS are increasing in number and declining in cost relative to DRAM costs. And this is sufficient to overcome the fact that Bw-tree performance on main memory operations is dramatically higher than that of classic data caching systems.

5 MAIN MEMORY VS DATA CACHING

A similar analysis to the preceding section can be done for any main memory database system, or caching database system. And, as we shall see, there can be a similar cross-over point where relative cost advantage changes from one to the other system. In this section, we compare the cost/performance of the Bw-tree to MassTree [23], a main memory key-value store. The Bw-tree exploits main memory technology like latch-free access to shared resources to dramatically improve performance relative to classic caching stores.

5.1 MassTree Cost Analysis

Trading space for time can be useful for improving performance, but how much space for how much performance? We consider this in examining the Bw-tree fully cached vs the MassTree. As with the analysis in the preceding section, we would like to understand the cost implications of our performance results.

MassTree has lower execution cost, while the Bw-tree has lower storage cost. This is not a "paging" situation. We need to consider

[2] If a user buys an SSD for its IOPS, then we need to attribute the cost of the SSD entirely to IOPS, in which case $\$I = \$300$. We do not pursue this here.

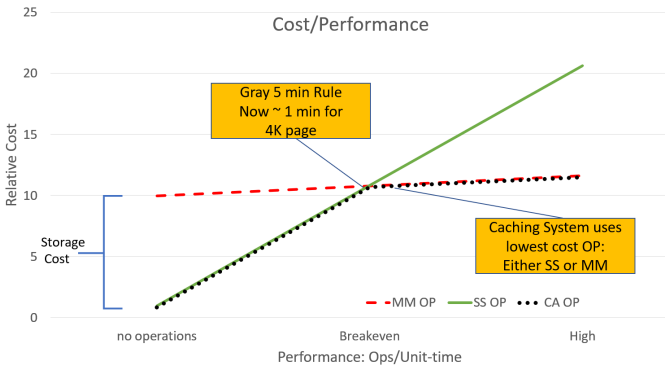

Figure 2: The costs of main memory operations and secondary storage operations as access rates change indicate where each type of operation should be used. The cost intersection is where the operation costs are equal, and is our derivation of Gray's 5-minute rule.

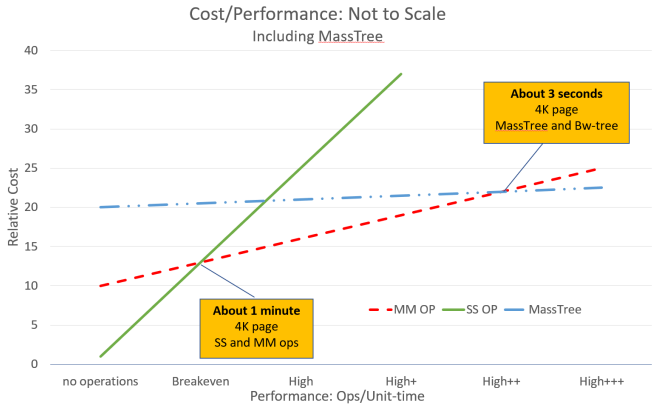

Figure 3: Comparison of the operation costs for the Bw-tree and MassTree. The cost breakeven point depends on the database size. See the discussion.

the costs for the entire main memory footprint of each system. With that in mind, our previous "page size P_s" becomes simply database size S. Since we are only comparing main memory operations, we remove secondary storage costs. We capture the costs of operations in the two systems below. Deuteronomy main memory operation cost is denoted by $\$DM$, MassTree main memory cost by $\$MTM$. Both operations have maim memory cost and processor cost. We can simplify operation costs from Equation 4 to express these costs. We express costs for MassTree in terms of their storage and execution costs relative to the Bw-tree.

- Bw-tree:
  $\$DM = T_i * S * (\$M) + \frac{\$P}{ROPS}$
- MassTree:
  $\$MTM = T_i * M_x * S * (\$M) + \frac{\$P}{P_x * ROPS}$

where M_x is the memory expansion for the MassTree and P_x is its performance gain, both observed to be greater than one. Setting the costs equal and solving for T_i gives us

$$T_i = \frac{1}{S} * \left[\frac{\$P}{ROPS} * \frac{1}{\$M}\right] * \frac{P_x - 1}{P_x * (M_x - 1)} \quad (7)$$

Aside from M_x and P_x, other costs were provided in the preceding subsection. Based on our read-only experiments in the 4-core case, with the Bw-tree configured for main memory, $P_x \approx 2.6$ and $M_x \approx 2.1$. (Let me remind the reader that this is a point experiment.) Then we have:

$$T_i = \frac{1}{Size} * (8.3 * 10^3) \quad (8)$$

5.2 Comparative Costs

With a database of 6.1GB (the Bw-tree footprint in our experiment), $T_i = 1.37 * 10^{-6}$, which translates to an access rate of $\approx .73 * 10^6$. The access rate must scale with database size. Thus for a 100GB database, the access rate would need to be about $12 * 10^6$ ops/sec before MassTree would have lower costs. Relating this to Bw-tree page size of 2.7 KB page size analysis, the time between accesses T_i would need to be less than 3.1 seconds before its cost per operation for data on that page would be lower. The Bw-tree can lower its overall costs further by evicting cold pages at a T_i of 45 seconds when it is used as a data caching system. These results are shown in Figure 3.

6 REDUCING SECONDARY STORAGE ACCESSES

Reducing the impact of I/O on data caching system performance is key to keeping costs low. Avoiding I/O is one aspect of this. Caching of pages that are referenced by a workload is commonplace for traditional data caching systems, and such pages are retained, usually in some approximation of LRU, to avoid the need to repeatedly spend an I/O access to bring them in again. However, there are new techniques that modern data caching systems use in addition to this that are very effective. We illustrate this by describing some of the ways that Deuteronomy and RocksDB reduce the number of I/Os that need to be executed.

6.1 Log-Structuring for Reduced Writes

Deuteronomy's LLAMA storage manager [21] organizes storage in a "log-structured" manner, first described in log-structured file systems(LFS) [30]. LLAMA writes very large buffers containing a large number of pages to secondary storage in a single write. While log-structuring was originally intended as a way to deal with the very low rate and very high cost of hard disk accesses, log-structuring techniques have proven useful for flash drives as well, where access rates can still be a limiting factor, and the processor execution cost of an I/O remains significant.

Deuteronomy's log-structured implementation is not a classic log-structured file system but it shares major characteristics of one, with updated pages being accumulated into large write buffers to reduce the number of writes. And like LFS, it maintains a map of where (logical) pages, relocated on every write, are stored, i.e. the mapping table shown in Figure 4. Deuteronomy uses its Bw-tree to find logical pages and the LLAMA cache manager, which operates along the lines of a conventional page cache, to find the corresponding physical page.

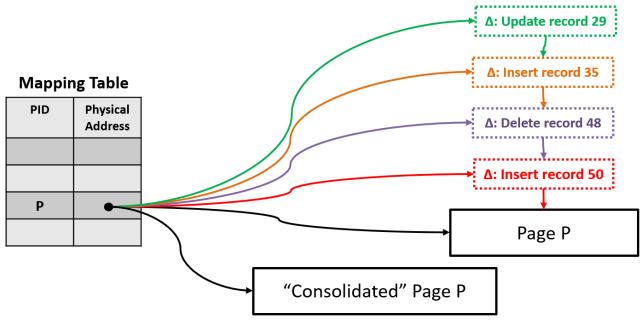

Figure 4: Deuterony's log-structured store using a mapping table to locate updated pages that move on every update to secondary storage. The mapping table also accommodates blind delta updates.

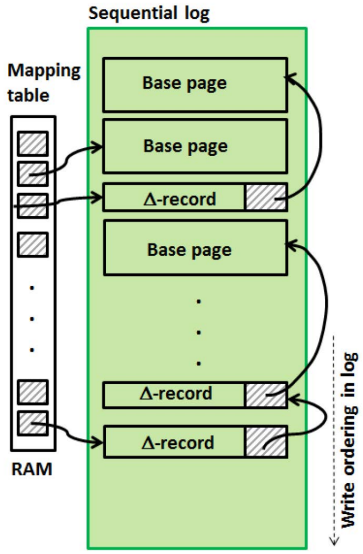

Figure 5: Deuterony's log-structured store uses variable size pages and need only store delta updates when the base page has previously been stored.

Unlike a conventional log-structured fixed block store, Deuteronomy can, by shrinking the storage needed for an update, reduce the cost of updating secondary storage. It does this in two ways (see Figure 5).

(1) Variable size pages: Only the storage consumed by data in the page needs to be written. In a B-tree setting, storage utilization is typically about $ln(2)$ or about 69%. Around 30% less storage is consumed when compared with a fixed block store.
(2) Delta updates: When a (base) page has been previously stored, it is only necessary to store its delta updates to capture a new page state on secondary storage.

RocksDB also takes a log-structured approach, using the LSM-tree [25, 27]. LSM maintains multiple trees, including an in-memory tree, where all updates are initially "accepted". As needed, it merges the in-memory tree with the smallest of the secondary storage trees, and subsequently will merge the other trees as they reach some storge limit. Updates to all secondary trees are done only via this merge process, turning all writes into large writes and keeping storage utilization of the secondary storage trees high.

Log-structured stores always append new versions of pages to the "log". So out-of-date versions need to be garbage collected to keep storage utilization of secondary storage acceptable. This presents an interesting trade-off. Garbage collection can be turned on when system load is low to reduce long term storage footprint, and hence storage costs. However, at high execution load, garbage collection might be delayed to save the compute cycles of GC. This also improves GC efficiency by letting the storage for old versions increase, hence increasing the reclaimed space per segment when GC eventually executes.

6.2 Blind Updates to Avoid I/O

Deuteronomy's Bw-tree can post an update as a delta update to the mapping table it maintains to find the pages in its log-structured store (see Figure 4). This can be done without the need to immediately incorporate the new (delta) update into the main body of the (base) page. Delta updating enables high performance for latch-free updating.

Blind updates are updates that do not depend on the prior state of a record. Delta updating enables high performance blind updates that do not require that the base data page be present in main memory [31]. When Bw-tree index pages, used to locate the correct data page are cached in main memory, I/O can be completely avoided as a blind update does not need to read the data page being updated. The update delta can be posted simply to the mapping table entry for the data page. RocksDB also avoids an I/O by posting a blind update into its LSM main memory tree without reading any secondary storage tree's page.

In Deuteronomy, blind updates are particularly useful and low cost. Even a user with an ordinary update to a record acts like a blind updater. The transactional component issues a read to the data component (its Bw-tree). Later, the updated record is posted back to the Bw-tree as a blind update. A timestamp is used to properly order the updates. When the update is handled as a delta update, we simply prepend the delta to its appropriate page. A reader, using the timestamps will select the record version it needs. Thus, all transactional updates are blind updates at the Bw-tree. This same technique is used during recovery– indeed, there is no difference in how updates are handled during normal operation and during recovery.

6.3 Record Caching

The great thing about a record cache is that storage for a record is typically much smaller than the storage needed for a page, perhaps less than 10%. Smaller storage cost expands the access frequency range where main memory operations are less costly. The result is improved performance because main memory operations can then be used more frequently, while continuing to operate at the lowest

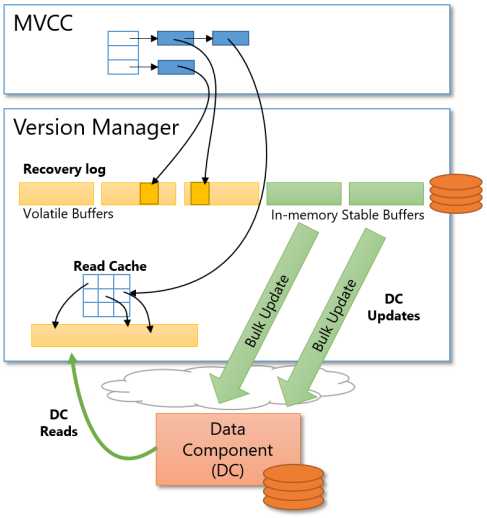

Figure 6: Deuterony's TC maintains recovery log and an MVCC version store. These stores are used as a record cache by the TC. Records read by transactons are also retained in a Read Cache.

cost point as well. Equation 6 for T_i includes the quantity P_s in its denominator. Thus, when there are 10 records in a page, the record breakeven $T_i = 10\%$ minutes instead of about one minute for the page.

Deuteronomy's Bw-tree can direct LLAMA to keep delta updates in main memory even when evicting a base page. Thus the deltas for the page remain in main memory, serving as a record cache. RocksDB's LSM main memory tree likewise serves as a record cache. When a record being read is found among the delta updates of a page, even though the base page may have been evicted, the read can be satisfied without a read I/O.

Deuteronomy's transaction component (TC) carries record caching a step further [22]. Multi-version concurrency control shrinks the frequency of conflicting operations. Instead of using proxies for the multiple versions, the TC uses the versions themselves. The TC uses the redo log records on the recovery log as an updated record cache by retaining the recovery log buffers in main memory, including after the log may have been flushed. Records that are read from the data component are retained in a separate log-structured read cache. This is shown in Figure 6. Thus, the multi-version concurrency control mechanism's hash table also functions to access a record cache at the TC. When there is a cache hit at the TC, not only is an I/O avoided, but even going to the data component to look for the record in the Bw-tree is avoided.

7 IMPROVING SECONDARY STORAGE COST/PERFORMANCE

7.1 Reducing I/O Execution Cost

Secondary storage (SS) operation cost are substantially higher than main memory (MM) operation execution cost. Figure 2 starkly illustrates this. Shrinking the cost of SS operations results in shorter

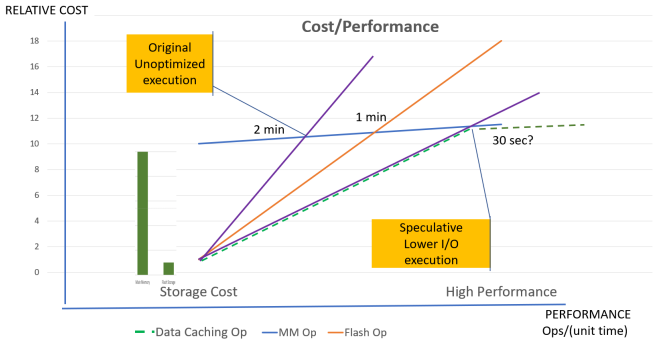

Figure 7: The impact of changing the execution cost of secondary storage operations on cost/performance.

breakeven times, enabling data, in a cost effective manner, to be evicted from main memory earlier. This section discusses two paths for doing this.

7.1.1 Optimizing I/O Execution Path. The cost of executing an operation that requires an I/O is high. Not only is it necessary to execute code directly related to the I/O but, given current SSD latencies, we need to execute a thread context switch so that we do not lose even more performance by stalling the thread. There are a number of factors that make this more expensive than perhaps it needs to be. A significant one is the common requirement to invoke the operating system to execute the I/O operation. Two problems with this are:

(1) execution needs to cross the application:system protection boundary. This is more expensive than simply a procedure call, and may even involve switching execution to another thread.
(2) a data management system knows more about how to deploy threads for its own work than does the OS. Hence, it can in principle make better scheduling choices than the OS.

So if we can execute an SS operation, including its I/O, in user mode, we are in a position to deal with both these problems. Intel has recently released a Software Performance Development Kit (SPDK) that enables the management of I/O at the user level [15]. We have exploited the SPDK to bring the SS operation execution path into our (user level) code. And the result was reduction in the execution path for an I/O of about a third. The execution ratio between SS and MM operations droped from about 9X to an average of about 5.8X. This has a large impact on data caching system performance, as suggested by Figure 1, which illustrates how performance declines as the fraction of SS operations increases.

Surely as important, improving I/O execution path performance improves data caching cost/performance. Figure 7 illustrates the effect this. Improving SS execution performance reduces the slope of the line representing SS costs, lowering costs over a wide performance spectrum and reducing further the breakeven crossover point. Figure 7 illustrates how cost/performance over a wide performance range is reduced by reducing I/O costs.

7.1.2 Falling Price of SSD IOPS. We probably would not be taking data caching systems seriously without the introduction of

SSDs, which dramatically increased the IOPS and reduced the cost of each I/O operation. This trend is continuing. Our experiments were mostly performed on Samsung flash SSDs supporting 300K IOPS. Samsung is now selling flash SSDs that support 500K IOPS at a similar price point. This represents a decline of about 40% in the cost of an SSD I/O operation. This is on top of the historical huge decline in IOPS cost over the past few years.

The cost of IOPS directly impacts the cost of an SS operation. Hence, we are profiting from the improved hardware being delivered by major flash vendors. But other opportunities exist that arise from the Open Channel SSD initiative [5, 6]. An OCSSD gives system builders the opportunity to install their own code in the SSD controller, opening up the possibility of further cost reductions from optimizing SSDs to the needs of the data management system.

7.2 Reducing Storage Costs

Facebook has a huge amout of data and when that data is accessed, it needs high performance and low latency. But any specific piece of data is accessed infrequently. Thus, hosting that data in main memory is not cost/performance effective. It is less costly to apply additional processing power to operate on data brought in from SSD's when the data is used. Facebook is operating at the left hand side of Figure 2.

Indeed, even with data residing most of the time on SSD, costs were high. So Facebook decided to compress its data to further lower its storage costs [8]. Data compression adds processor execution cost to the cost of operations because such data needs to be decompressed for use. So here, execution costs are increased to reduce storage cost. And this is a good trade-off for Facebook.

Figure 8 shows graphically how this extra "compressed data" secondary storage operation (CSS) fits together with secondary storage operations (SS) and main memory operations (MM). Do not be misled by the small range shown in the figure. The amount of data on the left side of the graph can be enormous. Even modest unit cost differences have significant impact on total cost.

A system supporting compression can choose the lowest cost operation under a very wide range of access data rates. At the extreme left, the very cold data has very low storage cost using CSS operations. Toward the middle, the system can switch to uncompressed data and use SS operations. And when the data is in active use (right side of the figure), the system can switch to main memory operations (on cached data).

Data compression might be usefully employed for main memory data as well, used as a way to avoid I/O costs. A main memory operation on compressed data would surely have more expensive execution cost than a main memory operation on uncompressed data. But it would have lower storage cost as well. It also might have a higher storage cost than data on an SSD, but with lower execution cost. So its total cost (execution cost plus storage cost) might well be lower than either of these alternatives. Further, if one is going to compress data destined for secondary storage in any event, then staging the compressed data so that it lives a bit longer in main memory may well be very cost/performance effective.

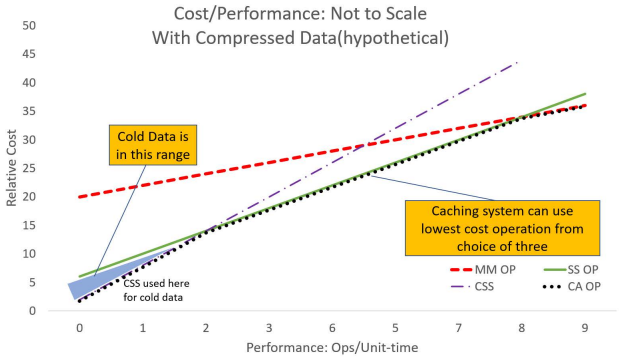

Figure 8: Data compression can reduce the storage cost for data below that of uncompressed data, sometimes substantially. This is illustrated here schematically in the same way as with other comparisons (all numbers are hypothetical).

8 DISCUSSION

8.1 Data Management Goal

The goal of data management systems is to create users value. With businesses, value translates to economic value. Low cost is an important part of this, but it is not the whole story. This is especially true when time-value of results is significant. One example of this is in the financial industry where there is a very significant value for being quickest to execute. So one might choose to operate a data caching system with, e.g., all data in cache even though this does not minimize cost/performance, when very high performance and/or low latency is essential to high "value". Indeed, a high performance main memory system, even one with a large main memory footprint, might create the most value in some of these cases.

But most applications are not like that. Interactive latency is a requirement for many, but latencies in the 10's vs 100's of microseconds is of no consequence to "value". For this common scenario, focusing on good performance at low cost is usually the right goal. And here, performance means throughput. Data caching systems are designed for that goal. This is why even new vendors are building data caching systems [2].

8.2 New Technology

Non-volatile memory (NVRAM) looks like a promising technology (that has been "waiting in the wings" now for a number of years). NVRAM is expected to cost less than DRAM, but more than flash, while having performance that is also between DRAM and flash. And, of course, it provides data persistence as well. There are two possible roles for NVRAM.

- **SSD:** With NVRAM cost higher than flash, it is unlikely to displace flash in SSDs. The cost of accessing an SSD is high largely because of the execution cost of an I/O, so little access cost is saved. And low storage cost is critical for SSDs, so flash has the advantage.
- **Main (or extended) Memory:** NVRAM provides data persistence. And, with a cost less than DRAM, if it can perform close to DRAM, it might make inroads in the main memory area. NVRAM performance is critical in how this plays out.

If too much performance is lost, systems will move hot data to DRAM to restore performance. However, even if that is done, fetching data from NVRAM has much lower cost and performance impact than an *SS* operation which needs I/O.

8.3 Old Technology

Hard disk drives (HDDs) have not disappeared. The best of them have increased IOPS and reduced latency, especially over the past few years. Maximum IOPS is now over 200, with latency around 5 ms. These disks are expensive (on a cost/byte basis) compared with more commodity disks, whose latency is about twice that at 10 ms, and IOPS around 100. But even with the higher performance and lower latencies, HDDs cannot compete with flash drives. When a system is executing 10**6 ops/sec or so, 1000 operations might execute in a millisecond (and 5000 within the latency of an HDD). Further, even less than a small fraction of 1% of operations needing to access secondary storage quickly saturates an HDD. If 10 I/O accesses/sec are required per transaction, this suggests that no more than 20 transactions/second can be supported by a hard disk. So HDDs are not a useful technology any more for high performance data stores. As Jim Gray suggested in 2006 [13], "disk is tape, flash is disk".

However, there are a number of remaining uses for hard disks. In particular, their low storage cost may make them ideal for backups and archiving, where high access rates are not needed. Further, some very large data analytics tasks process data sequentially. And hard disk sequential performance is fine. So these tasks are also appropriate for hard disks. These use cases further demonstrate that "disk is tape". These applications should be susceptible to the kind of analysis that is done here. Very high access costs can be tolerated when frequency of access is low and storage needs are high.

8.4 Conclusions

We have been discussing cost/performance. That is a more economically important topic than sheer performance. And storage costs are a very big part of overall costs, especially since most data is cold. Furthermore, the hot data set typically changes over time. So managing data cost effectively means being able to reduce storage costs when data is cold, and reduce execution cost when it is hot. That is exactly what data caching systems are designed to do.

Modern data management systems such as RocksDB [28] and Deuteronomy [20–22] are designed for high main memory performance AND efficient use of secondary storage. Further, they try to reduce the cost of moving data between main and secondary storage, and to optimize exploitation of data when it lands in the cache to reduce the need for added I/O. These systems do not match main memory system performance but provide great performance at low cost. The key to their low cost is caching data when hot and evicting it when it becomes cold. That is, they are data caching systems.

Data caching systems are succeeding in the market because of better cost/performance, despite main memory systems demonstrating higher (sometimes significantly higher) performance. But there is another message in this, illustrated by RocksDB and Deuteronomy. It is that it is possible to apply technology, perhaps "borrowed" from main memory systems, to dramatically increase data caching system performance. And additional work in this direction has already appeared [19]. There is a research agenda here that might succeed in producing a "one size" system that fits all [34], or at least one that fits a very large part of the data management market, with very high performance and low cost/performance.

ACKNOWLEDGEMENTS

Several colleagues have made substantial contributions to Deuteronomy's transactional and data components. These include (in time order of participation) Sudipta Sengupta, Justin Levandoski, Ryan Stutsman, Rui Wang, James Hunter, and Umar Minhas. Phil Bernstein provided comments that greatly improved the clarity and precision of the presentation. Remaining errors are, of course, all mine.

REFERENCES

[1] R. Appuswamy, R. Borovica-Gajic, G. Graefe, and A. Ailamaki: The Five-minute Rule Thirty Years Later and its Impact on the Storage Hierarchy, ADMS, 2017
[2] Amazon Aurora https://aws.amazon.com/rds/aurora/details/
[3] R. Bayer and E. M. McCreight, "Organization and Maintenance of Large Ordered Indices," *Acta Inf.*, vol. 1, no. 1, pp. 173–189, 1972.
[4] J. DeBrabant, A. Pavlo, S.Tu, M. Stonebraker, S. Zdonik: Anti-Caching: A New Approach to Database Management System Architecture. PVLDB 6(14): 1942-1953 (2013)
[5] LightNVM: The Linux Open-Channel SSD Subsystem. FAST 2017: 359-374.
[6] P. Bonnet: What's Up with the Storage Hierarchy? CIDR: 2017.
[7] C. Diaconu, C. Freedman, E. Ismert, P. Larson, P. Mittal, R. Stonecipher, N. Verma, M. Zwilling: Hekaton: SQL server's memory-optimized OLTP engine. SIGMOD 2013: 1243-1254
[8] S. Dong, M. Callaghan, L. Galanis, D. Borthakur, T. Savor, M. Strum: Optimizing Space Amplification in RocksDB. CIDR 2017
[9] A. Eldawy, J. Levandoski, P. Larson: Trekking Through Siberia: Managing Cold Data in a Memory-Optimized Database. PVLDB 7(11): 931-942 (2014)
[10] Flash file system: https://en.wikipedia.org/wiki/Flash_file_system
[11] J. Gray, G. R. Putzolu: The 5 Minute Rule for Trading Memory for Disk Accesses and The 10 Byte Rule for Trading Memory for CPU Time. SIGMOD 1987: 395-398
[12] J. Gray, G. Graefe: The Five-Minute Rule Ten Years Later, and Other Computer Storage Rules of Thumb. SIGMOD Record 26(4): 63-68 (1997)
[13] J. Gray: Tape is Dead, Disk is Tape, Flash is Disk, RAM Locality is King, jimgray.azurewebsites.net/talks/flash_is_good.ppt, 12, 2006.
[14] IBM DB2 https://en.wikipedia.org/wiki/IBM_Db2
[15] Intel: Introduction to the Storage Performance Development Kit (SPDK) https://software.intel.com/en-us/articles/introduction-to-the-storage-performance-development-kit-spdk
[16] 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, PVLDB 1(2): 1496-1499 (2008).
[17] A. Kemper, T. Neumann, J. Finis, F. Funke, V. Leis, H. Mülhe, T. Mühhlbauer, W. Rödiger: Processing in the Hybrid OLTP & OLAP Main-Memory Database System HyPer. IEEE Data Eng. Bull. 36(2): 41-47 (2013)
[18] J. Lee, M. Müehle, N. May, F. Faerber, V. Sikka, H. Plattner, J. Krüger, M. Grund: High-Performance Transaction Processing in SAP HANA. IEEE Data Eng. Bull. 36(2): 28-33 (2013)

[19] Viktor Leis, Michael Haubenschild, Alfons Kemper, Thomas Neumann LeanStore: In-Memory Data Management Beyond Main Memory ICDE 2018
[20] J. Levandoski, D. Lomet, and S. Sengupta, The Bw-Tree: A B-tree for New Hardware Platforms, ICDE 2013, pp. 302–313.
[21] J. Levandoski, D. Lomet, S. Sengupta. LLAMA: A Cache/Storage Subsystem for Modern Hardware. PVLDB 6(10): 877-888 (2013).
[22] J. Levandoski, D. Lomet, S. Sengupta, R. Stutsman, R. Wang: High Performance Transactions in Deuteronomy. CIDR 2015.
[23] Y. Mao, E. Kohler, R. T. Morris. Cache Craftiness for Fast Multicore Key-Value Storage. In EuroSys, 2012, pp. 183-196.
[24] Microsoft SQL Server https://en.wikipedia.org/wiki/Microsoft_SQL_Server
[25] P. E. O'Neil, E. Cheng, D. Gawlick, E. J. O'Neil. The Log-Structured Merge-Tree (LSM-Tree). in *Acta Inf.* 33(4): 351-385 (1996)
[26] Oracle Database https://en.wikipedia.org/wiki/Oracle_Database
[27] R. Sears, R. Ramakrishnan: bLSM: a general purpose log-structured merge tree. SIGMOD 2012: 217-228
[28] RocksDB: A persistent key-value store for fast storage environments. http://rocksdb.org/
[29] LevelDB http://leveldb.org/
[30] M. Rosenblum and J. Ousterhout, "The Design and Implementation of a Log-Structured File System," *ACM Trans. Comput. Syst.*, 10(1), 26–52 (1992).
[31] D. Shukla et al: Schema-Agnostic Indexing with Azure DocumentDB. in *PVLDB* 8(12): 1668-1679 (2015)
[32] M. Stonebraker, A. Weisberg: The VoltDB Main Memory DBMS. IEEE Data Eng. Bull. 36(2): 21-27 (2013)
[33] TPC: History and Overview of the TPC. http://www.tpc.org/information/about/history.asp
[34] M. Stonebraker, U. Cetintemel, "One size fits all": an idea whose time has come and gone. ICDE 2005.
[35] A. Ailamaki, Database Architectures for New Hardware. VLDB 2004.