

Flashing Up the Storage Layer

Ioannis Koltsidas
School of Informatics
University of Edinburgh
i.koltsidas@sms.ed.ac.uk

Stratis D. Viglas
School of Informatics
University of Edinburgh
sviglas@inf.ed.ac.uk

ABSTRACT

In the near future, commodity hardware is expected to incorporate both flash and magnetic disks. In this paper we study how the storage layer of a database system can benefit from the presence of both kinds of disk. We propose using the flash and the magnetic disk at the same level of the memory hierarchy and placing a data page to only one of these disks according to the workload of the page. Pages with a read-intensive workload are placed on the flash disk, while pages with a write-intensive workload are placed on the magnetic disk. We present a family of on-line algorithms to decide the optimal placement of a page and study their theoretical properties. Our system is self-tuning, *i.e.*, our algorithms adapt page placement to changing workloads. We also present a buffer replacement policy that takes advantage of the asymmetric I/O properties of the two types of storage media to reduce the total I/O cost. Our experimental evaluation shows remarkable I/O performance improvement over both flash-only and magnetic-only systems. These results, we believe, exhibit both the potential and necessity of such algorithms in future database systems.

1. INTRODUCTION

Though primarily designed for mobile devices, flash disks are also starting to appear in commodity computer hardware, as their capacity is constantly growing, while their price is proportionally dropping. As of February 2008, flash disks with capacities of 64 or even 128 gigabytes have reached the market. It is expected that in the near future commodity hardware will incorporate not only conventional magnetic disks, but also flash disks. Additionally, operating systems are already providing facilities to take advantage of flash disks [11]. Given these trends, in this paper we investigate how the presence of both types of disk can be taken advantage of when designing I/O-intensive database applications.

Manufacturers have striven to make the existence of flash disks transparent to users by providing them with an I/O interface that is identical to that of magnetic disks. How-

ever, though both kinds of disk have the same interface, their I/O characteristics are widely disparate. Flash disks demonstrate extremely fast random read speeds, but slow random write speeds. At the same time, conventional magnetic disks are more efficient than flash disks at random write patterns, but slower when randomly reading data. For instance, the flash disk we used for our experiments can read pages from random locations more than twenty times faster than a magnetic disk. On the contrary, a magnetic disk can write pages at random locations ten times faster than the flash disk. Our goal is to design a database system that uses both a flash and a magnetic disk as storage media and improve its I/O performance. We do so by combining the fast read speed of the flash disk with the fast write speed of the magnetic disk. Our experimental results show that in such a hardware setup our proposals can have significant and immediate impact.

Flash disks. The most common type of flash memory found in solid state drives (SSDs) is NAND flash. NAND flash disks are typically used in mobile devices and low energy environments, as they are shock resistant, demonstrate low power consumption, and operate completely silently. However, the I/O characteristics of NAND flash disks differ substantially from the characteristics of conventional magnetic platter disks, and need to be taken into account when designing applications for such disks. The most important I/O characteristics of SSDs are:

◊ *I/O interface.* At the operating system level, SSDs behave identically to magnetic drives, as they are accessed through the same I/O interface. Typically, the unit of I/O operations on a flash disk is a sector of 512 bytes, which is equal to the size of a magnetic disk sector (for most magnetic disks).

◊ *No mechanical latency.* Flash disks are purely electronic devices and have no mechanical moving parts. The time needed to access a data item on a flash disk is independent of its position on the physical medium, *i.e.*, access latency does not depend on the access pattern. Additionally, access latency is orders of magnitude less than the random access latency for mechanical disks. Both properties present great opportunities for performance gains over magnetic disks.

◊ *I/O asymmetry.* The electrical properties of flash cells result in read operations being much faster than write operations: when the value of a NAND cell is changed, it takes some time before it reaches a stable state. For most flash disks read speed is twice as fast as write speed.

◊ *Erase-before-write limitation.* The most important limitation of flash disks is that a sector cannot be overwritten: it has to be erased before it can be updated. Moreover, only entire blocks can be deleted, with each block contain-

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, to post on servers or to redistribute to lists, requires a fee and/or special permission from the publisher, ACM.

VLDB '08, August 24-30, 2008, Auckland, New Zealand
Copyright 2008 VLDB Endowment, ACM 000-0-00000-000-0/00/00.

ing many sectors. Updating a sector requires that the whole block to which it belongs be erased and rewritten. Considering that a deletion is more expensive than a read or a write, updating a NAND flash sector becomes quite costly.

Our proposals target commodity hardware equipped with both magnetic and flash disks. While high-end flash disks that completely outperform magnetic disks have recently been announced, such as the *ioDrive* [5], they are not likely to appear in commodity hardware any time soon. High-performance flash disks use *Single Level Cell (SLC)* flash memory that stores one bit of data per cell. The alternative is *Multi Level Cell (MLC)* flash memory that uses four voltage levels and can thus store two bits of data per cell. While MLC flash has twice (or even more) the density of SLC, programming an MLC cell takes much longer [16]. As a result, MLC flash disks have greater capacity at a fraction of the cost of SLC ones, but quite worse write performance. Given the higher storage capacity (comparable to that of magnetic disks) and lower cost of MLC flash disks, commodity hardware is expected to incorporate that type of disk, which clearly cannot outperform magnetic disks with respect to write performance. In this paper we consider such flash disks.

One idea is to use the flash disk as a cache for the magnetic disk, *i.e.*, as an extended buffer. While this design might be reasonable for a file system, it can prove suboptimal for database workloads as it disregards the writing inefficiency of the flash disk. The reading efficiency of the flash disk, on the other hand, is an argument for using it for persistent storage [7]. Given this discrepancy, and considering the growing capacity of flash disks, we propose to use both types of disk at the same level of the memory hierarchy, *i.e.*, a database page can reside *either* on the flash disk *or* on the magnetic disk, but not on both. We present algorithms for optimally placing a page according to its workload. Pages with a read-intensive workload are placed on the flash disk, while pages with a write-intensive workload are placed on the magnetic disk. We propose ways of accurately predicting the workload of a page and of improving the page eviction choices of the buffer manager when it is aware that both kinds of disk are present.

The types of system that can benefit from our proposal include (but are not limited to): (a) (parts of) database systems with well-defined workloads, especially when a portion of the data is very frequently accessed but only scarcely updated *e.g.*, the database catalog, typical access paths, *etc.*; (b) archiving systems, where a percentage of data appearing only in the latest versions are frequently accessed, whereas a larger percentage of the archive (also referred to as the *deep* archive) is infrequently used; (c) file systems, where both kinds of disk are transparently handled by the operating system, but user data is organized according to its I/O workload for maximum efficiency; (d) hybrid hard disks, *i.e.*, magnetic disks that are equipped with flash memory, which they use as non-volatile cache [15]. Our algorithms can be employed by the controllers of such disks to boost performance.

Contributions and organization. In the following sections we present our approach to data management using both a magnetic and a flash disk as storage media. Our main contributions are:

- We propose using the flash disk and the magnetic disk at the same level of the memory hierarchy, *i.e.*, the

flash disk is not used as a cache for the magnetic disk. We show that important performance benefits can be gained with such a design, especially for queries touching large sets of pages with read-intensive workloads.

- We study the problem of optimal placement of each data page (*i.e.*, whether it should reside on the flash or on the magnetic disk) both from a practical and a theoretical perspective. We present a family of on-line algorithms that can be used to dynamically decide the optimal placement of each data page. Our algorithms adapt to changing workloads for maximum I/O efficiency.
- We present a novel buffer pool page replacement algorithm, designed to benefit from the asymmetric properties of the disks. Page replacement is not only decided by the probability of a page being accessed in the future, but also by the I/O cost of evicting the page and re-fetching it into main memory.
- We have implemented all proposed algorithms and conducted an extensive experimental study. Our algorithms can significantly improve I/O performance over magnetic-disk-only and flash-disk-only setups, and for database workloads that frequently occur in practice.

The rest of this paper is organized as follows. Related work is presented in Section 2. The problem statement is given in Section 3, and a family of on-line algorithms for deciding the placement of pages on storage media is presented in Section 4. In Section 5 we present our novel page replacement algorithm to be used by the buffer manager of a hybrid system. The results of our experimental study are given in Section 6, followed by a discussion of the advantages of our design in Section 7. We conclude and present our future research directions in Section 8.

2. RELATED WORK

To overcome the “erase-before-write” limitation of NAND flash, flash disks employ a software layer called the *Flash Translation Layer (FTL)*. Its main purpose is to provide logical-to-physical address mapping, power-off recovery, and wear-levelling. Many different algorithms have been proposed for the FTL [4]. In [8], the authors propose maintaining a small number of log blocks as temporary storage for overwrites, with each logical sector being mapped only to a certain log block (block level associativity). This technique substantially improves the random-write performance of flash disks. In [10], the authors present a scheme for fully associative translation between logical sectors and log blocks, thereby improving the space utilization of log blocks. The scheme further improves write performance (by as much as 50%). In [1], it is argued that increasing the amount of volatile RAM on flash disks is the only way to achieve acceptable random access write performance and present a design for flash disks with large volatile buffers.

File system FTL algorithms are not well-suited for database workloads. As shown in [9], random write operations in file systems are mostly required for metadata. When executing typical database workloads, however, DBMSs perform random write operations that are scattered over the whole disk address space. To improve write efficiency for flash-based databases, an *in-page logging (IPL)* scheme is proposed in [9]: changes made to a data page are not written directly to disk, but to log records associated with the page.

Changes are logged on a per-page basis, while each data page and its log records are located in the same physical block of the disk *i.e.*, in the same erase unit. Each erase unit is divided into a number of data pages and a number of log sectors for the log records of the pages. In-memory representation of a page includes an in-memory log sector (of the same size as the flash log sector). When a page is dirtied in memory, its contents need not be written back to disk; only the log records for the page need to be appended to the log sector for the page on disk. When the erase unit is out of free log sectors, the logged changes are applied to the corresponding data pages in the unit. Then, the data pages are written to a new erase unit (that has its log sectors erased). That way, page updates only involve writing already erased log sectors. Simulation results of the IPL scheme show that it improves the random write efficiency of flash disks by an order of magnitude for typical database workloads. The authors also propose an IPL-based recovery mechanism for transactions that minimizes the cost of system recovery. Such logging schemes are orthogonal to our proposals; they can be used complementary and will most likely result in further I/O efficiency.

Research has also been conducted in the area of flash-aware indexing. In [17], the authors propose storing B^+ -tree nodes as a sequence of log records spread over multiple disk blocks. To update a page, one appends log records to this sequence, thereby not having to erase the entire block for each page update. The evaluation of the approach shows that it improves performance both in terms of time and energy efficiency. On the same topic, authors in [12] propose a self-tuning B^+ -tree that provides indexing functionality to the storage manager of a flash-based database system. The index dynamically adapts its storage structure according to the database workload and the underlying storage device. Specifically, nodes of the B^+ -tree are stored either in *log* mode or in *disk* mode. In *disk* mode, the entire node is written in consecutive disk pages, while nodes in *log* mode are stored as log entries, that may be spread over multiple disk pages. Pages in *log* mode are written very efficiently (as updates do not incur overwriting physical blocks), while read operations require all log entries for the page to be gathered, so that the page can be reconstructed. On the contrary, pages in *disk*-mode can be read efficiently (by just reading a page from disk), but writing a *disk* mode page requires erasing the physical block first. Switching between modes incurs a specific cost, therefore authors propose an on-line algorithm to decide the optimal mode of a page.

The algorithm we propose for page placement is also an on-line one. The problem we are solving is an instance of the page migration problem: deciding the optimal node of a network to store a data page, so as to minimize the total cost of serving requests for the page from other nodes of the network. In [2], the authors thoroughly study this problem and present competitive algorithms for different network topologies. In [3], the authors study metrical task systems similar to the one we use to model our system and provide a $(2n - 1)$ -competitive on-line algorithm (n is the number of states of the task system). However, our task system is not metrical, as discussed in Section 3. Page migration in graphs with arbitrary edge distances is studied in [14] where the authors propose a randomized algorithm that approaches 2.62-competitiveness against an oblivious adversary.

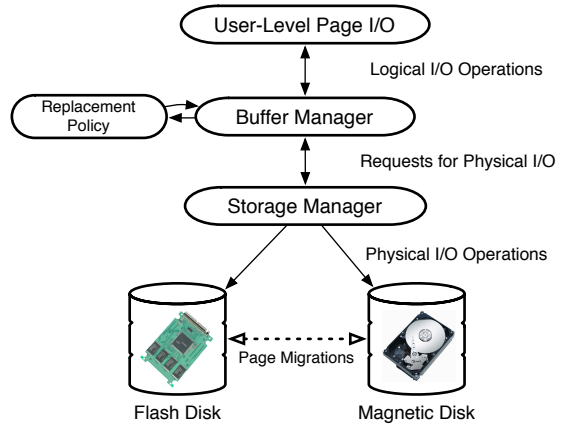


Figure 1: An overview of our system

3. PROBLEM STATEMENT

The two problems we solve are (a) adapting to changing I/O workloads to optimally place pages across the two disks, and (b) making the optimal choice of the page to replace in the buffer pool when such a need arises. The high-level architecture of our system is shown in Figure 1. A magnetic disk and a NAND flash disk operate at the same level of the memory hierarchy. Each data page exists only on the flash, or on the magnetic disk at any given time.

Page placement. The storage manager decides the optimal placement for each page according to the workload of the page. Pages with a read-intensive workload are placed on the flash disk, while pages with a write- or update-intensive workload are placed on the magnetic disk. Thus, reads are faster than a magnetic-disk-only system, and writes are faster than a flash-disk-only system. In this manner the total I/O cost is reduced. The main challenge in this approach is how one can predict the future workload of a page based on past accesses to the page. Also of paramount importance is the ability to self-tune, *i.e.*, adapt the placement choice for each page when its workload changes from read-intensive to write-intensive and vice-versa. Considering that moving a page from one disk to another incurs significant I/O cost, the prediction of a page’s future workload has to be as accurate as possible. Failure to achieve an acceptable level of accuracy means that the I/O cost will be heavily penalized, as the page will migrate from disk to disk before the migration cost has been expensed.

Page replacement. The buffer manager can also benefit from the asymmetric properties of the two different storage media. The efficiency of a buffer manager under a specific workload is determined by the total I/O cost paid for that workload. Unlike traditional systems that use only magnetic disks, this cost in our system depends not only on the number of page misses, but also on the cost of each page miss. The problem then is how to adjust the page replacement policy of the buffer pool to further reduce the I/O cost. Our goal is to improve I/O efficiency by reducing page misses (*i.e.*, the total number of I/O operations), while at the same time reducing the I/O cost of each miss. We achieve the latter by taking into account the I/O cost for each page eviction when choosing the next page to evict.

4. PAGE PLACEMENT

We present a family of algorithms to decide the optimal

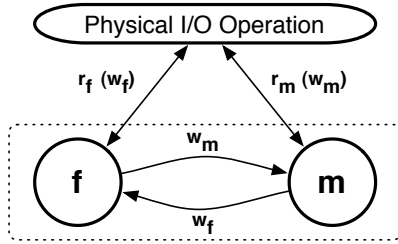


Figure 2: Abstraction as a two-state task system

placement of data pages. We employ a typical buffer pool: pages are fetched on demand from disk to main memory. Read and write operations that are served in main memory are referred to as *logical* hereafter, while ones that reach the disk are referred to as *physical*. Whenever the buffer pool is out of space, a page is selected to be replaced according to the buffer manager’s replacement policy. At that point the system needs to decide the placement of the page, *i.e.*, whether the page will be stored on the flash or on the magnetic disk. The decision is made dynamically and on a per-page basis, it depends only on the history of the page, and is independent of all other pages. Our system keeps track of the location of each page (so that it knows which disk to read it from) and statistics about its workload.

The decision algorithm is an on-line one. We model the decision process for each page as a two-state task system [3], depicted in Figure 2. The two *states* of the system are f and m , representing that a page is on the flash disk or on the magnetic disk, respectively. The cost for reading a random page from the magnetic disk is r_m , while the cost for writing a page to a random position is w_m ; r_f and w_f are the respective costs for the flash disk. The *transition cost* from one state to the other is equal to the cost of writing a page to the other disk (when a transition occurs, the page has already been read). The *tasks* in our task system are I/O operations. The cost of processing a read request is r_f when the system is in state f and r_m when it is in state m (resp. w_f and w_m for write/update operations).

The problem we solve resembles the page migration problem on an arbitrary tree [2]. The key difference is that in our case the page migration cost depends on the direction of the migration. Our proposed solution resembles the algorithm given in [12] as the mode-deciding algorithm. However, one cannot simply adapt that approach if one wants to realistically model the problem we solve. The reason is that one needs to make the crucial distinction between logical and physical I/O operations. The interaction between physical and logical operations is not clear, unless buffer pool and storage management parameters are taken into account. This is due to the actual I/O cost being decided by physical operations, while application-level I/O requests are merely logical. This salient distinction is elegantly captured in our model. As we shall see in Section 6, our results prove this non-trivial extension necessary in an implementation with real-world workloads (and not in a simulation). We present a family of on-line algorithms that have the same structure, but use different cost metrics.

4.1 Conservative Algorithm

The first algorithm, which we refer to as *conservative*, is given in Figure 3. For each page in the system, the algorithm maintains a counter C that is updated after each physical operation. The cost of reading the page from the current

Algorithm conservative (Page pg)

1. **if** (pg is a new page)
2. $pg.state \leftarrow m, pg.C \leftarrow 0$
3. After each *physical read* of the page:
4. $pg.C \leftarrow pg.C + (r - r')$
5. Upon eviction of the page:
6. **if** ($pg.dirtybit = 1$)
7. $pg.C \leftarrow pg.C + (w - w')$
8. **if** ($pg.C > w_f + w_m$)
9. $pg.state \leftarrow$ other state
10. $pg.C \leftarrow 0$
11. $pg.dirtybit \leftarrow 1$

Figure 3: The conservative algorithm

disk is r , while the cost of reading the page from the other disk is r' (resp. w and w' for writing). When a page is *physically* read, C is incremented by the cost difference $r - r'$ (line 3), that represents the cost units that would have been saved, had the page been read from the other disk (if $r - r' < 0$ the page was read from the read-efficient disk). The same happens when a dirty page is to be evicted from the buffer pool: the cost counter is incremented by $w - w'$. Upon eviction, C is examined (line 8) and if it is greater than the cost of two migrations ($w_f + w_m$), the page migrates to the other disk (by changing its *state* value and setting its dirty bit to 1 – lines 9 to 11).

The conservativity of the algorithm lies in two points: (a) The algorithm initiates a migration only after the accumulated cost for a page has surpassed $w_f + w_m$. This is the earliest point in time that the algorithm can be sure that the page is not on the optimal storage medium. For the time period during which the counter accumulated the $w_f + w_m$ cost units, the cost would have been less if the page was stored on the other disk. This is because the cost of migrating to the other disk and back has been already reached during the last physical operations and those physical operations would have been served more efficiently by the other disk. (b) The algorithm takes into account only physical operations on pages, not logical ones. The physical cost is the actual cost paid by the system and therefore the conservative algorithm does not try to induce the physical access pattern from the logical one. Rather, it waits until the logical access pattern has been translated into physical accesses. Note that due to the lack of any access history for new pages, they are always written to the magnetic disk for the first time, since the magnetic drive is more write-efficient (line 1 in Figure 3).

An off-line algorithm that knows the exact workload for each page beforehand can decide the optimal placement of the page *i.e.*, it incurs the minimum I/O cost for the workload. An algorithm like *conservative* is 3-competitive with respect to the optimal off-line algorithm, as we prove in the Appendix. Our evaluation shows, however, that the cost of *conservative* remains more than 1.5 times less than the cost of the optimal algorithm for realistic workloads.

4.2 Optimistic Algorithm

Though physical operations capture the actual cost paid by the system, their sequence is dictated by logical operations and the replacement policy of the buffer pool. Moreover, while the page remains in the buffer pool (*i.e.*, between two physical operations on the page) many logical operations

Algorithm optimistic (Page pg)

1. **if** (pg is a new page)
2. $pg.state \leftarrow m$, $pg.reads \leftarrow 0$, $pg.writes \leftarrow 0$
3. No accounting until after the first physical write
4. After each *logical read* of the page:
5. $pg.reads \leftarrow pg.reads + 1$
6. After each *logical write* of the page:
7. $pg.writes \leftarrow pg.writes + 1$
8. Upon eviction of the page:
9. $c_f \leftarrow pg.reads \cdot r_f + pg.writes \cdot w_f$
10. $c_m \leftarrow pg.reads \cdot r_m + pg.writes \cdot w_m$
11. **if** ($(c_f > c_m$ **and** $pg.state = f$) **or**
 $(c_m > c_f$ **and** $pg.state = m$))
12. $pg.state \leftarrow$ other state
13. $pg.reads \leftarrow 0$, $pg.writes \leftarrow 0$
14. $pg.dirtybit \leftarrow 1$

Figure 4: The optimistic algorithm

may occur. The conservative algorithm, will only record two (or one) physical operations on the page, and thus, if the workload changes, it will take many physical operations before conservative adapts. This gives rise to an “optimistic” version of the algorithm that works only on logical page operations and adapts to new workloads as quickly as possible; the algorithm is presented in Figure 4.

For each page pg the optimistic algorithm maintains a read counter ($pg.reads$) and a write counter ($pg.writes$). Each counter is incremented when a logical read or write operation occurs, respectively. These counters hold the total logical read and write operations on the page since its last migration. Upon eviction, the algorithm computes the total cost the system would pay if these operations were physical, for each of the two disks (c_f for the flash disk, c_m for the magnetic one – lines 10 and 11 in Figure 4). The page migrates to the disk with the least total cost, if it is not already there, and its read and write counters are reset (lines 11 to 14). When a new page is created, the algorithm does not account for logical operations until the page has been physically written for the first time (line 3). This is because most newly created pages will be logically written to many times when they are created (e.g. after a B^+ -tree node split). These logical writes do not reflect the normal workload for the page and are therefore not logged.

The optimistic algorithm is not conservative in the number of migrations. It assumes that when the workload of a page changes from read-intensive to write-intensive (or vice-versa), the migration cost will be amortized, *i.e.*, changes to the workload of the page are not frequent. Thus, optimistic adapts quickly to changing workloads but when changes do not last long enough for the migration cost to be expensed, the overall cost paid by the system grows. Our experimental results verify these observations.

Another caveat is that optimistic tries to minimize the cost of future physical operations on the page based on its history of logical operations. Consider a page p having been brought into the buffer pool at time t_1 and evicted at time t_2 , after having been logically read a large number of times: its workload upon eviction is found to be strongly read-intensive and the page will be written to the flash disk. Then, the workload of the page changes to write-intensive and optimistic needs to see some k logical writes on p before deciding it is now write-intensive. If the page is frequently replaced

Algorithm hybrid (Page pg)

1. **if** (pg is a new page)
2. $pg.state \leftarrow m$
3. $pg.lr \leftarrow 0$, $pg.lw \leftarrow 0$, $pg.pr \leftarrow 0$, $pg.pw \leftarrow 0$
4. No accounting until after the first physical write
5. After each *logical read* of the page:
6. $pg.lr \leftarrow pg.lr + 1$
7. After each *physical read* of the page:
8. $pg.pr \leftarrow pg.pr + 1$
9. After each *logical write* of the page:
10. $pg.lw \leftarrow pg.lw + 1$
11. Upon eviction of the page:
12. **if** ($pg.dirtybit = 1$)
13. $pg.pw \leftarrow pg.pw + 1$
14. $q \leftarrow 1 - b/n$
15. $c_f \leftarrow (pg.lr \cdot q + pg.pr) \cdot r_f + (pg.lw \cdot q + pg.pw) \cdot w_f$
16. $c_m \leftarrow (pg.lr \cdot q + pg.pr) \cdot r_m + (pg.lw \cdot q + pg.pw) \cdot w_m$
17. **if** ($(c_f - c_m > w_f + w_m$ **and** $pg.state = f$) **or**
 $(c_m - c_f > w_f + w_m$ **and** $pg.state = m$))
18. $pg.state \leftarrow$ other state
19. $pg.lr \leftarrow 0$, $pg.lw \leftarrow 0$, $pg.pr \leftarrow 0$, $pg.pw \leftarrow 0$
20. $pg.dirtybit \leftarrow 1$

Figure 5: The hybrid algorithm

by the buffer manager until the k logical writes have been served, these k logical writes will have been realized as physical ones (since the page is frequently evicted). Reasons for these frequent evictions include the buffer manager deciding to assign fewer pages to the file p belongs to, or some other file becoming hot, or the time between writes on p being much longer than the time between reads, (*i.e.*, the page becomes cold). In this scenario, not only is the benefit from the migration never realized (since read operations are very scarce after the initial eviction), but also the system pays a very high penalty by writing the page to the flash disk, before the write-intensive workload has been identified.

4.3 Hybrid Algorithm

To minimize the total cost of physical operations, one needs both physical *and* logical operations on data pages to be taken into account. We introduce a hybrid algorithm that combines the strong points of conservative and optimistic, at the same time avoiding their weak points. The basic idea is that a physical operation on a page has more impact on the decision of the algorithm than a logical one. This is because physical operations on a page are typically fewer than logical ones, but at the same time they are the ones to affect the actual cost.

The probability that a logical operation will not be realized as a physical one is proportional to the size of the buffer pool. Let n is the number of pages in a file and b the number of pages the buffer manager has dedicated to the file: the probability that a logical operation on a page will be served in-memory is b/n . The probability that a logical operation on a page will affect the total I/O cost is equal to the probability of the logical operation to result in a physical one. Thus, the probability that a logical operation will have an impact on the I/O cost is equal to $(1 - b/n)$. We use this probability to scale the impact of a logical operation.

The hybrid algorithm, shown in Figure 5, maintains four counters per page: lr and lw count logical reads and writes since the last migration, respectively; pr and pw count phys-

ical reads and writes since the last migration, respectively. Newly created pages are written to the magnetic disk and counters are not modified until the page has been written for the first time (lines 1-4). For each logical or physical operation on the page, the corresponding counter is incremented (lines 5-10). Upon eviction, the algorithm computes the total cost physical and logical operations would incur for each disk (lines 16-17). As mentioned, the cost of logical operations is scaled by $1 - b/n$. A page migrates to the other disk if the accumulated cost for the current disk surpasses the cost for the other disk by $w_f + w_m$ cost units (line 18).

Accounting for logical operations when deciding the placement of a page allows hybrid to recognize changes in the workload of the page very early, as does optimistic. However, hybrid is not as eager as optimistic to trigger page migration. It decides that a page should migrate only after it is certain that the page is on the wrong disk (*i.e.*, the cost of migrating to the other disk and back has been already paid). In that sense, hybrid resembles conservative. By taking into account physical costs (*i.e.*, actual costs) the system has a realistic view of the effect of the buffer pool on logical operations.

5. PAGE REPLACEMENT

The buffer manager’s page replacement policy plays an important role in the system’s I/O efficiency. Consider a page that is chosen for eviction. The I/O cost of this choice depends on whether the page is dirty, and whether it will be referenced by a logical operation in the future. If the page is dirty, then it has to be physically written and the cost of writing is added to the total cost. If the page is referenced by some operation in the future, then it has to be physically read and the total cost grows by the reading cost.

When using one or more disks with the same read/write speeds (*e.g.*, magnetic disks), the I/O efficiency of the replacement policy is determined by how well the policy can predict which page will be referenced in future I/O operations. In our case the write speed of the flash disk is two orders of magnitude less than its read speed. Moreover, read and write speeds for the magnetic disk are one order of magnitude less than the read speed of the flash disk and one order of magnitude greater than its write speed. Consider a page p_1 that is stored on the flash disk and is not dirty at a given point in time, and page p_2 that is stored on the magnetic disk and is also not dirty. These two pages are the ones that have the least probability of being accessed by future requests among all pages in the buffer pool. If p_1 is chosen for eviction, at the next read operation on p_1 the system will pay the cost of reading from the flash disk; if p_2 is chosen, the system will pay the cost of reading from the magnetic disk, which is more than 20 times higher. The best choice for eviction is p_1 : if p_2 has not been evicted by the next access to it, the system will have avoided $r_m - r_f \simeq r_m$ cost units (if both pages have been evicted by the time of their next access the cost is the same). We therefore propose a novel buffer management policy that decides on page replacement not only based on access frequency, but also based on the I/O cost that the replacement is likely to induce. The proposed replacement policy exploits the asymmetric properties of the two media to further improve performance and complements the role of the theoretical model of Section 4.

The buffer pool is logically divided into two segments: the *time* segment and the *cost* segment, as shown in Figure 6. Pages in the time segment are sorted on their timestamp

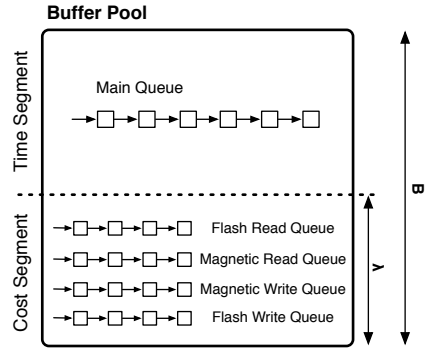


Figure 6: Buffer Pool Segments

(*i.e.*, the time of their last access). Pages in the cost segment are sorted on their cost of eviction. If the buffer pool is B pages in size, the cost segment size is λB , $\lambda \leq 1$. Of all pages in the buffer pool, the cost segment contains the λB least recently used ones, at any given time. According to our replacement policy, the next page to be evicted is always selected from the cost segment, while new pages fetched from one of the disks are always inserted in the time segment.

The *eviction cost* of a page is r_f if the page is on the flash disk ($r_f + w_f$ if it is dirty) and r_m if it is on the magnetic disk ($r_m + w_m$ if dirty). Pages in the time segment are sorted on their timestamp in typical Least-Recently-Used (LRU) fashion. Implementation-wise, a queue (termed *main* queue) is maintained with the timestamps of pages and pointers to them. The front of the queue always refers to the page with the minimum timestamp (*i.e.*, the least recently used page). When a page is accessed (and therefore has the greatest timestamp), it is put in the back of the queue.

The cost segment consists of four queues, one for each eviction cost class, as shown in Figure 6: (a) the flash read queue (FRQ) holds pointers to non-dirty pages that are on the flash disk, (b) the flash write queue (FWQ) holds pointers to dirty pages on the flash disk; (c) the magnetic read queue (MRQ) and (d) the magnetic write queue (MWQ) hold clean and dirty pages on the magnetic disk, respectively. Each queue holds its elements sorted on their timestamp, just like the main queue.

The buffer manager maintains a hash index on pages by their page identifier. If a page is in the buffer pool, a lookup in this index returns the queue element that represents the page (which can be an element of any one of the five queues). On access, a page is inserted into the buffer pool as in algorithm `fetchPage` of Figure 7. A hash index lookup is performed to check if the page is in the buffer pool (line 1). If it is in the pool and in the main queue (*i.e.*, in the time segment), it is given the current timestamp and moved to the back of the queue (lines 2-4). If it is in a queue of the cost segment, it is removed from that queue, given the current timestamp and inserted to the back of the main queue (lines 5-8). Then, the least recently used page of the time segment (*i.e.*, the front of the main queue) is removed from the main queue and inserted to the back of the cost segment queue that holds pages with the same eviction cost (lines 9-10).

If the page is not in the buffer pool it is read from the disk on which it resides. If the pool is full, a page is evicted using algorithm `evictPage` of Figure 8 (line 13). The requested page is read from disk, given the current timestamp, and added to the back of the main queue (*i.e.*, in the time seg-

Algorithm fetchPage (PageId pid)

1. $pg \leftarrow \text{hash_lookup}(pid)$
2. **if** (pg found in main queue)
3. give pg a new timestamp
4. move pg to the back of main queue
5. **else if** (pg found in cost segment queue q)
6. remove pg from q
7. give pg a new timestamp
8. add pg to the back of main queue
9. $pg' \leftarrow$ the front element of the main queue
10. insert pg' to the back of cost segment queue q' that holds pages with cost $\text{evict_cost}(pg')$
11. **else**
12. **if** (buffer pool is full)
13. **evictPage** ()
14. read pid from disk into pg
15. give pg a new timestamp
16. **if** (size of main queue $< (1 - \lambda)B$)
17. add pg to the back of main queue
18. **else**
19. insert pg to the back of cost segment queue that holds pages with cost $\text{evict_cost}(pg)$

Figure 7: Algorithm fetchPage

ment). If the size of the time segment is less than $(1 - \lambda)B$, it means there is room in the time segment so the page is inserted there. Otherwise, all new pages will be inserted into the cost segment, until the pool is full. Thus, when the pool becomes full, the size of the time segment is $(1 - \lambda)B$ and the size of the cost segment is λB . After page eviction, the size of the time segment is less than $(1 - \lambda)B$ (*i.e.*, $(1 - \lambda)B - 1$) and the size of the cost segment stays λB .

Page evictions are decided by algorithm **evictPage** shown in Figure 8. The page to be evicted is the front element from the non-empty queue that holds pages with the least eviction cost (lines 2-5). If the page is dirty (*i.e.*, it comes from either MWQ or FWQ), it is written to disk (line 6). Then, it is removed from the queue in which it resided and deleted from main memory (lines 7-8). Finally, the least recently used page of the time segment is removed and placed into the cost segment, by being appended to the back of the appropriate queue (lines 9-10). The cost segment maintains the same size and a free page is created, so the page to be read after the eviction can be inserted in the time segment.

The value of λ affects the efficiency of the algorithm. For simplicity, consider that pages in the buffer pool are not dirty. As λ grows, the number of magnetic disk pages in the buffer pool grows, while the number of flash disk pages in the buffer pool shrinks (because FRQ pages are evicted first, they will typically be found only in the main queue which decreases in size with increasing λ), *i.e.*, the hit probability of a magnetic disk page increases, but so does the miss probability of a flash disk page. Let H_m denote the increase of magnetic page hits (with respect to simple LRU) and M_f denote the increase of flash page misses. Then, performance is improved so long as $r_m \cdot H_m > r_f \cdot M_f \Rightarrow H_m > \frac{r_f}{r_m} M_f$. The buffer manager can keep track of these quantities and adapt λ accordingly in real time (this can be straightforwardly extended to account for dirty pages). In Section 6.4 we experimentally verify the effect of λ on performance.

Observe that for each page access or page eviction the complexity of our algorithm is constant in the size of the

Algorithm evictPage ()

1. Page pg ;
2. **if** (FRQ is not empty) $pg \leftarrow$ front of FRQ
3. **else if** (MRQ is not empty) $pg \leftarrow$ front of MRQ
4. **else if** (MWQ is not empty) $pg \leftarrow$ front of MWQ
5. **else** $pg \leftarrow$ front of FWQ
6. **if** (pg is dirty) write pg to disk
7. remove pg from the queue it belongs to
8. delete pg
9. $pg' \leftarrow$ front of main queue
10. insert pg' to the back of cost segment queue q' that holds pages with cost $\text{evict_cost}(pg')$

Figure 8: Algorithm evictPage

buffer pool. All operations on queues are $O(1)$ and both **fetchPage** and **evictPage** incur a constant number of operations on queues (one at least, two at most). Additionally, each hash index lookup is also $O(1)$. The complexity of our algorithm is only greater than the complexity of LRU by some constant c . As we show in Section 6, however, our algorithm is much more I/O efficient than LRU.

6. EXPERIMENTAL STUDY

We implemented our algorithms to evaluate their performance under various workloads. Our system consists of a storage manager and a buffer manager and uses B^+ -trees for storing data. Though we have implemented other file structures as well (*i.e.*, heap files and linear hash files), we only present results with B^+ -trees since they are the most commonly used database structures and make our presentation more succinct. Moreover, B^+ -trees have the extra property of exhibiting both random access patterns (*e.g.*, when descending the levels of the tree) and sequential ones (*e.g.*, when scanning the leaves). The system was implemented in C++ and was running on an Intel Pentium 4 box clocking at 2.26GHz with 1.5GB of physical memory. The operating system was Debian GNU/Linux with the 2.6.21 kernel. The system has two magnetic disks and a flash disk. Our system and the operating system ran from one of the magnetic disks. The other magnetic disk (referred to simply as the magnetic disk hereafter) and the flash disk were used to store data pages. The magnetic disk was a 300GB Maxtor Diamond-Max 6L300R0 with 16MB of cache memory. The flash disk was a Samsung MCAQE32G5APP, an MLCNAND flash disk with a capacity of 32GB. Both disks were connected to the system using the IDE interface. To reduce the effects of operating system caching we used both storage media as raw devices. Therefore, the operating system did not cache data pages, pages were never double buffered and our system had absolute control of physical I/O operations.

Metadata. As discussed in Section 4, the storage manager keeps accounting information for each page. For the **conservative** algorithm this information is nine bytes per page, of which one byte represents the state of the page and the rest hold a 64-bit integer that represents the accumulated cost for the page. The **optimistic** algorithm needs one byte for the state of the page and eight bytes for two integers counting logical reads and writes, for a total of nine bytes. The **hybrid** algorithm requires one byte for the state and sixteen bytes for four 32-bit integers counting logical and physical reads and writes. For a data file of size n bytes, the metadata is $\lceil \frac{n}{4096} \rceil \cdot 9$ bytes for **conservative** and **optimistic** and $\lceil \frac{n}{4096} \rceil \cdot 17$

Operation	Time (CPU cycles $\times 10^3$)	Cost units
Flash read	915	1
Flash write	108987	118
Magnetic read	21470	23
Magnetic write	10983	12

Table 1: I/O costs

bytes for hybrid, if 4096-byte pages are used. To reduce this amount, one can increase the page size. However the size of extra data is negligible for most practical purposes, as it is three orders of magnitude less than the size of the data. For instance, for the hybrid algorithm (which has the largest requirements), the metadata for a 10GB table are only 44MB. All accounting information is stored on the hard disk with the operating system (*i.e.*, file pages contain only raw data).

We assume the capacity of either disk is enough to hold all data placed on it. The address for a page is the same for both disks (*i.e.*, we only used the first 32GB of the magnetic disk) and no explicit mapping is necessary. In a real deployment, metadata for files, pages, page mappings, and free space on each of the disks would be maintained. This is necessary even for systems using just one disk, so standard file system techniques could be used for that purpose without any additional overhead. To keep the experimental study as simple as possible we chose not to implement these structures, since they would not affect our measurements. Given the current capacities of flash drives, it is conceivable that the flash disk is not large enough to accommodate all read-intensive pages. For such cases, ranking algorithms are required to capture the utility of each read-intensive page being kept on the flash disk. We are currently working on such ranking primitives.

Raw performance of disks. We measured the read costs for each disk by computing the average of 10^6 read requests of 4096 bytes each at random offsets on the disk. Requested page offsets span the whole disk address space; this is particularly important for the magnetic disk, as measured costs need to reflect the average rotational latency of a read. We similarly measured the average time for random writes for each disk. The results are shown in Table 1. The second column is the measured average times, while the third column is the costs normalized by the read time of the flash disk. The flash disk was 23 times faster than the magnetic disk at reading random pages; the magnetic disk was 10 times faster than the flash disk at writing to random locations. It is clear from the relative cost differences that when pages are placed on the correct medium (according to their workload), I/O cost will significantly drop – almost regardless of the access pattern.

Datasets and workloads. We tested the efficiency of our system under a multitude of workloads. The record layout consisted of a key that was sixteen bytes long and a payload of eighty bytes. The B^+ -tree contained one million records and had a size of 140MB. To minimize the effect of on-disk caches, the pages of the tree were not stored consecutively on disk, but separated by nine-page intervals (*i.e.*, the pages of the B^+ -tree spanned 1.4GB). We experimented with trees and buffer pools of various sizes. Across all configurations the results were consistent. To avoid repetition, and due to space limitations, we present the results for the aforementioned B^+ -tree size and a buffer pool of 20MB. We choose to show the results for this setup as it is more in line with the discrepancy between disk and main memory capacities; one can expect a difference of three orders of magnitude

between the two in current configurations. The workloads on the B^+ -tree consisted of reads, *i.e.*, lookups and range queries, and writes, *i.e.*, insertions and updates. Each insertion or update to the B^+ -tree results in the destination leaf being both read and written, and internal nodes on the path from the root of the tree to the leaf being at least read and potentially written (*i.e.*, in the case of a split). We focus on these simple operations as we aim to show that the placement of the page on the right medium (in addition to our buffer pool replacement policy) is what primarily makes a difference. Our algorithms prove this point true for such basic workloads; it will most certainly continue to hold for any complex workload that will surely use all these primitives.

6.1 Impact of using both disks

In the first set of experiments we measure the performance improvement gained by using both types of disk over using only one. We ran the same set of queries in three different setups: (a) using only the flash disk, (b) using only the magnetic disk, and (c) using both disks. In all cases the conservative algorithm decided the placement of pages. Since the workload of a page does not change, the conservative algorithm gives the least performance improvement among all three algorithms. Additionally, we used LRU as the buffer pool replacement policy as it is applicable in all three setups (the policy proposed in Section 5 would favor the two-disk setup). In the first experiment, we executed a set of 50,000 read queries (80% of which were lookups and 20% range queries) that targeted all leaf nodes of the B^+ -tree. We executed this set of queries 15 times (emptying the buffer pool after each execution) and measured the wall clock time of each run. Results are shown in Figure 9 (a) with the query set run shown on the x -axis and total execution time shown on the y -axis. Our system is shown as “ M/F ”; the system using only the flash disk is shown as “ F ”; and the system using only the magnetic disk is shown as “ M ”.

As expected, F is much faster than M for reads. The performance of our system is initially equal to that of M , since all pages are first on the magnetic disk. A large number of frequently accessed pages (*e.g.*, the internal nodes of the B^+ -tree and some hot leaf pages) migrate to the flash disk during the 4th execution of the query set, while the remaining read-intensive leaf node pages migrate to the flash disk during the 7th execution. Since we are using the conservative algorithm the point in time at which pages migrate depends only on the number of physical accesses to the page. Next, we executed a set of 50,000 insert/update queries (30% insertions, 70% updates) using the same buffer pool size. Results are shown in Figure 9 (b). In this case, F is one order of magnitude slower than M , as expected. Our system has initially the same performance with M . At the 4th execution the pages storing the internal nodes of the B^+ -tree migrate to the flash disk and performance improves slightly (due to the number of insertions being relatively small, internal nodes have mostly a read workload). Had the size of the buffer pool been too small to fit all internal node pages, the performance boost would have been much greater.

Next, we generated mixed query sets including both read and write queries. In the first set, 40% of pages are read-only, 40% are write-only and the remaining 20% have a 50% probability of being read and a 50% probability of being updated. Results are shown in Figure 10 (a). Then, we altered the query set, so that 70% of the pages are read-only and

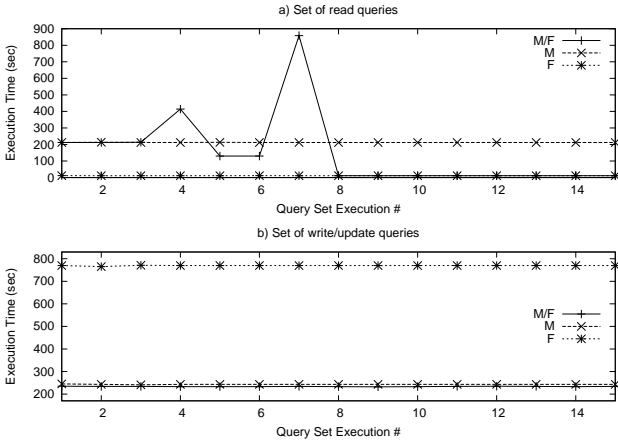


Figure 9: Read-only and write-only sets of queries

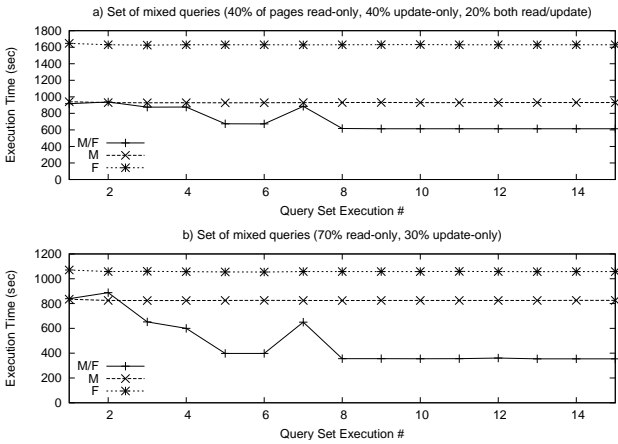


Figure 10: Execution of mixed queries

30% are update-only. The results are shown in Figure 10 (b). As the ratio of pages with a read workload grows, the performance of both F and M/F improves, while the performance of M remains almost constant. Clearly, using both a magnetic and a flash disk is more I/O efficient than using only one type of disk, provided that pages are placed on the disk that best suits their workload.

6.2 Comparison of page placement algorithms

We then moved on to study how well the page placement algorithms adapt to changing workloads. We created a set of 100,000 read queries and a set of 100,000 update queries. Using these two query sets, we created two different B^+ -tree query sequences and executed them using the conservative, optimistic and hybrid algorithms. Additionally, each query sequence was executed using the optimal placement for each page, which we computed off-line. The difference between the two query sequences is the frequency with which the page workload changes. In the first sequence, the set of read queries is executed 10 times, followed by 10 executions of the update query set; then, the read query set is executed again 20 times. In the second sequence, 3 executions of the read query set are followed by 3 executions of the update query set and vice-versa for a total of 18 query set runs. The buffer pool was emptied after each execution. Neither sequence is very likely to occur in real-world workloads; however, they highlight the difference between the three algorithms and their relationship to the optimal one in terms of

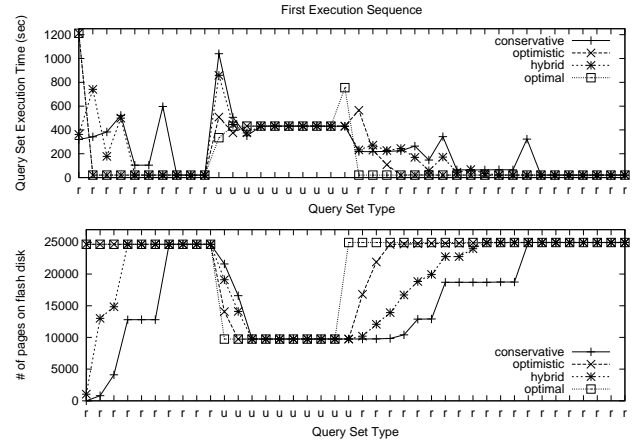


Figure 11: Infrequently changing workloads

their adaptability to changing workloads. The results of the two execution sequences are shown in Figure 11 and in Figure 12 respectively. On the x -axis, r 's stand for read query set executions and u 's stand for executions of the update query set. In addition to showing the total execution time on the y -axis, we also show an alternative plot with the y -axis denoting the number of pages being placed on the flash disk by the different placement algorithms, as this gives a more succinct picture of their decisions.

The first sequence shows that optimistic performs nearly optimally, while conservative is the slowest algorithm to adapt to workload changes. The performance of hybrid lies between the performance of optimistic and conservative, *i.e.*, hybrid adapts to workload changes more gracefully than optimistic, but more eagerly than conservative. Updates have a higher impact than reads on the decisions of the algorithms. This is due to update costs for the two disks differing by 107 cost units, while read costs differ by 22 cost units. This is why all algorithms adapt very quickly to the update workload. Also, observe that conservative and hybrid adapt very quickly to the initial read workload, but they adapt much more slowly after the execution of the ten update query sets. This is due to the cost threshold of $w_f + w_m$ having to be surpassed before a migration is triggered. When the workload changes from read- to update-intensive, some pages are read from the flash disk and written to the magnetic disk, which is the best case in terms of I/O efficiency. This explains why the first execution of the update query set, after the 10 executions of the read query set, is executed faster than the following 9 update query executions. Note that the total time for conservative was 9,784 seconds, for optimistic it was 7,003 seconds, for hybrid it was 8,338 seconds, while executing the queries according to the optimal off-line algorithm took 6,366 seconds. The time for executing this sequence only on the magnetic disk was 13,920 seconds and 12,760 for executing it only on the flash drive, amounting to a substantial improvement in all cases.

In the second sequence of runs, the workload changes every three executions. Since the cost of a page migration from the magnetic disk to the flash disk is not expensed by the three successive read query set executions, the optimal placement for pages is on the magnetic disk (except for the internal nodes of the B^+ -tree). For this reason, the optimal algorithm only places internal node pages on the flash disk. The conservative algorithm places only a small number of

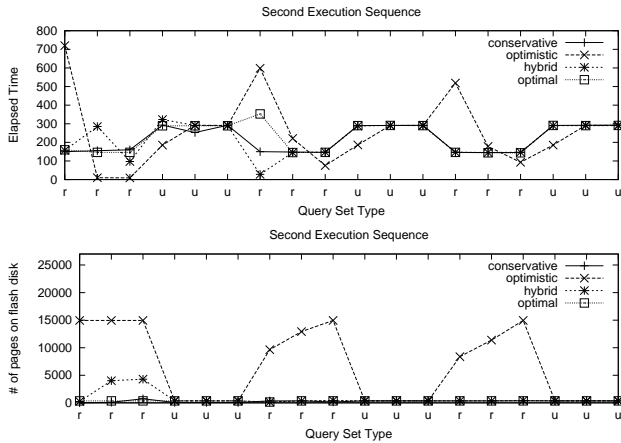


Figure 12: Alternating workloads execution

pages on the flash disk during the third read query set execution, but places them back on the magnetic disk after the first set of update queries. The **optimistic** algorithm eagerly places pages on the flash disk, which incurs a great cost when the workload changes. Of course, not all read pages are placed on the flash disk by **optimistic**, but only the ones that are logically read many times by the read query set. For **hybrid**, some pages migrate to the flash disk during the second and third query set executions. However, after the first three update query sets, all pages migrate back to the magnetic disk (except for the internal node pages of the B^+ -tree). The total times were 3,932 seconds for **conservative**, 4,560 seconds for **optimistic**, and 4,060 seconds for **hybrid**, while executing the queries according to the optimal algorithm took 3,920 seconds. The total time for executing this sequence using only the magnetic disk was 4,150 seconds and 6,552 seconds when using only the flash disk.

Of all on-line algorithms, **optimistic** gives the greatest performance improvement when it makes the right decisions. However, when workload changes do not last long enough for the migration cost to be paid off, **optimistic** introduces extra I/O cost due to wrong migration decisions. On the contrary, **conservative** decides migrations only after workload changes persist for a number of future accesses. Thus, **conservative** is less likely to make the wrong decision and does not migrate pages with frequently changing workloads. For this reason, however, it improves performance less than **optimistic**. The **hybrid** algorithm, by taking into account the decision criteria of both **optimistic** and **conservative**, manages to balance its adaptivity between the aggressive behavior of **optimistic** and the defensive behavior of **conservative**. Thus, **hybrid** is more I/O-efficient than **conservative**, without taking the risks of **optimistic** that could lead to very poor performance. Therefore, we believe that **hybrid** is the most appropriate algorithm to decide on page migration and we focus on that algorithm for the remaining sections.

6.3 Mixed workloads

For the next set of experiments we created query sets that have mixed-type queries. We picked a range of record key values (which we refer to as the *interesting* set) and performed B^+ -tree operations on these key values with a pre-determined probability. All other records had equal 50% read and update probabilities. All records (both the ones in the interesting set and all remaining) had the same prob-

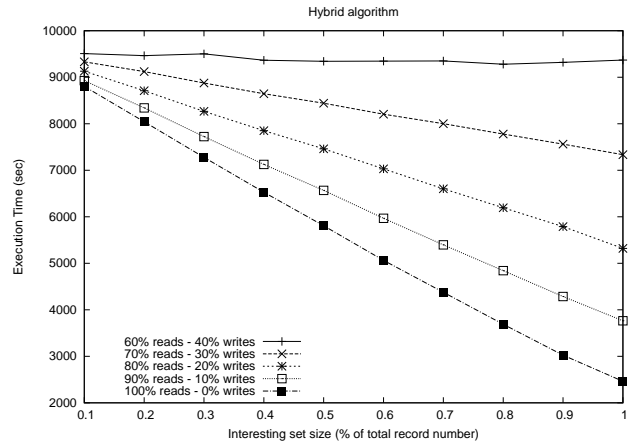


Figure 13: Performance in mixed workloads

ability of appearing in the query set. No set of pages was made artificially hot and the only thing that changed was the ratio between reads and updates among the pages in the interesting set. We varied the read and update probabilities, as well as the range of key values for records in the interesting set. We executed 1,000,000 queries using the hybrid algorithm using a 20MB buffer pool and measured the total execution time. Results are shown in Figure 13.

Performance improves when the workload of the interesting set pages becomes more read-intensive. For a given number of records in the interesting set, the number of pages that migrate to the flash disk is constant across workloads with different read/update probabilities. However, as the update probability grows, pages of the interesting set (most of which are on the flash disk) become more frequently updated, thus decreasing the performance gain of having them on the flash disk. When the workload is more than 70% read-intensive, one can see that performance improves as the size of the interesting set increases. This is because more pages migrate to the flash disk where read operations are more efficient. One can also see that when records in the interesting set are updated more than 30% of the time, performance does not improve as the size of the interesting set grows. This is due to leaf nodes that store records of the interesting set not migrating to the flash disk. In this case only internal nodes are placed on the flash disk and thus performance is slightly improved over a system employing only a magnetic disk. When using the magnetic disk only, execution time is comparable to that of the "60% read - 40% write" workload for all workloads and interesting set sizes (with only slight deviations). When using only the flash drive, execution time is much higher due to the pages not belonging to the interesting set being updated 50% of the time (except when the entire dataset is interesting).

6.4 Buffer pool replacement policy

In the last set of experiments, we measured the impact of the buffer replacement policy proposed in Section 5. We experimented with two different query sets, using different values for λ . In the first query set (query set A), 50% of B^+ -tree leaf nodes have a read workload and the remaining 50% of leaf nodes are read and updated with equal probability. Therefore half of the leaf pages are placed on the flash disk and the remaining are placed on the magnetic disk (of course, internal pages are placed on the flash disk, since no

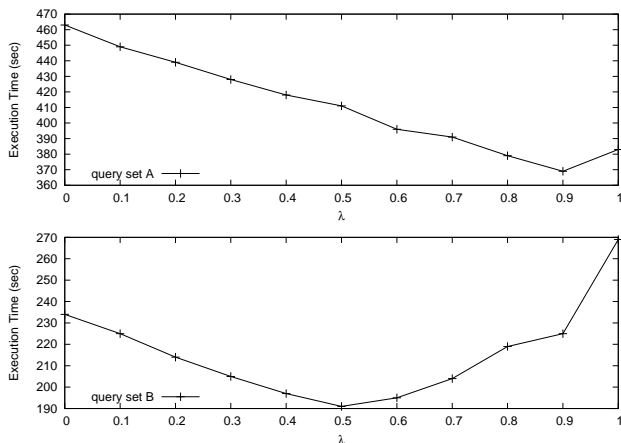


Figure 14: Replacement policy for different λ

insertions occur). All measurements are taken after pages have been placed on the appropriate disk. All leaf nodes have an equal probability of being referenced by a query. Query set *A* consists of 100,000 queries. We measured the total execution time as we varied λ and plotted the results in the top part of Figure 14. When $\lambda = 0$, the replacement policy degenerates to simple LRU, while when $\lambda = 1$ the whole buffer pool is used as a cost segment, with page replacement decided only by the eviction cost of a page.

One can see that performance improves as the value of λ increases, for values of λ up to 0.9. This is because pages on the flash disk and pages on the magnetic disk are accessed with the same probability and therefore $H_m \simeq M_f$ (using the terms of Section 5). When pages for internal nodes of the tree (stored on the flash disk) do not fit in the time segment, *i.e.*, for $\lambda > 0.9$, they are evicted in favor of pages on the magnetic disk. Then, each leaf page access requires 3 physical flash read operations (the depth of the tree was 4), and thus performance degrades. Performance with $\lambda = 1$ remaining better than with simple LRU has to do with flash pages not being particularly hot. The performance benefit of our replacement policy reaches 21% in this case.

In the second set of queries (query set *B*), 10% of the B^+ -tree leaf nodes have a read workload, while the rest are read and updated with equal probability. The set consists of 150,000 queries. However, only 15,000 queries access leaf pages on the magnetic disk, while the remaining 135,000 access leaf pages that are on the flash disk (*i.e.*, 10% of all leaf pages); thus, 10% of the B^+ -tree leaf pages are hot. The results, as we varied λ , are shown in the bottom part of Figure 14. The best performance is for $\lambda = 0.5$: 15% better than the performance of LRU. As λ grows greater than 0.5 performance degrades. More than half of the buffer pool fills up with magnetic disk pages that are scarcely accessed, while for most accesses to flash disk pages (which amount to 90% of all accesses) a miss occurs. When internal node pages do not fit in the time segment (for some $\lambda > 0.9$), performance again degrades and becomes even worse than the performance of LRU.

7. CURRENT DESIGN ADVANTAGES

We present some advantages of the current design that one can employ to further improve the performance of a system with a mixed disk setup.

Deferred page migrations. Most solid state disks, including the one we used in our experiments, are equipped with

a DRAM buffer. The buffer is partitioned into a number of segments (a typical size for each segment is 512KB to 1MB). Its purpose is to temporarily hold the contents of updated blocks (erase units) in order to avoid some erase operations, *i.e.*, to act as a write cache. Each DRAM segment can store a number of *contiguous* blocks. Thus, when writes to sectors are sequential, page sectors that belong to the same block are buffered in a DRAM segment. When the segment becomes full, all updates to the sectors of the block are performed with a single erase operation. Our benchmarks show that writing pages sequentially to the flash disk is over 10 times faster than writing them in a random fashion.

Our system can take advantage of this to further reduce the I/O cost and improve response time. During normal operation, the system can only *mark* pages that should migrate to the flash disk and perform all write operations on the magnetic disk. Then, migrations can be executed sequentially and in the background *e.g.*, when the system load is lower, or when execution of the query that marked them for migration has finished. Such a strategy is sensible for pages that are scarcely updated, or else the benefit of using the flash disk is cancelled.

Accelerating flash reads. As shown in Section 6.3, the improvement of our system shrinks as the frequency of updates to the flash disk pages grows. To minimize this effect one can employ the logging techniques of [9, 12], which are complementary to ours. The combination of both techniques will most likely lead to increased I/O efficiency.

Sequential access patterns. A typical access pattern of a database system is a sequential scan. The magnetic disk is more efficient at both reading and writing sequential data. It is conceivable to have the query engine supply hints to the buffer and storage managers whenever such patterns are encountered. As in [6, 13], the buffer manager can use sequential access hints not only for page replacement but also for page placement. In particular, it can employ sequential access costs in the page placement algorithm as opposed to random access ones, thereby favoring the magnetic disk and ensuring that sequentially accessed pages do not migrate to the flash disk.

Replacement algorithm. Our replacement algorithm uses LRU replacement for the pages in the time segment. However, when a page has to be moved from the time segment to the cost segment, the page to be moved can be chosen using *any* replacement algorithm. Therefore we expect the performance benefit seen in our experiments, in which LRU is used, to also be seen if we compare our algorithm to any other page replacement algorithm, provided we use the same replacement algorithm for the pages of the time segment.

8. CONCLUSIONS AND FUTURE WORK

With the capacity of flash disks rapidly increasing and their price dropping, commodity hardware in the near future will most likely include both a flash disk and a magnetic one. In this paper we presented a storage management scheme for such mixed systems. Our system takes advantage of the fast read speed of the flash disk and the fast write speed of the magnetic disk to reduce the total I/O cost of a database workload. Our technique dynamically places pages with read-intensive workloads on the flash disk and pages with write-intensive workloads on the magnetic one. We proposed, theoretically studied, implemented, and tested three different on-line algorithms that adapt well to

changing workloads. Furthermore, we have devised a novel buffer pool replacement policy that exploits the asymmetric properties of the two disks to reduce the total I/O cost. An experimental evaluation has shown that our techniques can reduce the I/O cost by a significant factor, especially when hot pages have read-intensive workloads.

We next plan to investigate how the ideas of [9, 12] can be applied to our design to further improve the efficiency of writing to the flash disk. An interesting problem is how page placement can be statically decided when the query workload is known in advance. We also plan to investigate how the query engine can provide workload and/or access pattern hints to the storage manager and to what extent this could benefit I/O performance. Finally, we plan to study ranking algorithms to capture the utility of each read-intensive page being kept on the flash disk, to enable our system to operate with flash disks that are too small to accommodate all read-intensive pages.

9. REFERENCES

- [1] A. Birrell *et al.* A design for high-performance flash disks. *SIGOPS Oper. Syst. Rev.*, 41(2), 2007.
- [2] D. L. Black and D. D. Sleator. Competitive algorithms for replication and migration problems. *Technical Report CMU-CS-89-201*, 1989.
- [3] A. Borodin, N. Linial, and M. E. Saks. An optimal on-line algorithm for metrical task systems. *J. ACM*, 39(4), 1992.
- [4] T.-S. Chung *et al.* System software for flash memory: A survey. In *EUC*, 2006.
- [5] Fusion-IO, Inc. The ioDrive. <http://fusionio.com>.
- [6] G. Graefe. Query Evaluation Techniques for Large Databases. *ACM Comput. Surv.*, 25(2), 1993.
- [7] G. Graefe. The five-minute rule twenty years later, and how flash memory changes the rules. *DAMON*, 2007.
- [8] J. Kim *et al.* A space-efficient flash translation layer for compactflash systems. *Transactions on Consumer Electronics.*, 2002.
- [9] S.-W. Lee and B. Moon. Design of flash-based DBMS: An in-page logging approach. In *SIGMOD*, 2007.
- [10] S.-W. Lee *et al.* A log buffer-based flash translation layer using fully-associative sector translation. *Trans. on Embedded Computing Sys.*, 2007.
- [11] Microsoft Corp. Windows Vista Operating System: ReadyBoost.
- [12] S. Nath and A. Kansal. FlashDB: Dynamic Self-Tuning Database for NAND Flash. In *IPSN*, 2007.
- [13] M. Stonebraker. Operating system support for database management. *Commun. ACM*, 24(7):412–418, 1981.
- [14] J. Westbrook. Randomized algorithms for multiprocessor page migration. In *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, volume 7, pages 135–150, 1992.
- [15] Wikipedia. Hybrid drive. http://en.wikipedia.org/wiki/Hybrid_drive.
- [16] Wikipedia. Multi level cell. http://en.wikipedia.org/wiki/Multi-level_Cell.
- [17] C.-H. Wu, T.-W. Kuo, and L. P. Chang. An efficient b-tree layer implementation for flash-memory storage systems. *Trans. on Embedded Computing Sys.*, 6(3), 2007.

APPENDIX

We will show that **conservative** is 3-competitive *w.r.t.* the optimal off-line adversary, *i.e.*, at any time t , $CONS(t) < 3OPT(t) + C_0$, where $CONS(t)$ is the total cost incurred by **conservative** up to that time, $OPT(t)$ is the total cost incurred by the optimal on-line algorithm, and C_0 is a constant. First we will show that **conservative** is 3-competitive in a metric space *i.e.*, when the migration cost is the same for both directions and equal to $K = \frac{w_f + w_m}{2}$. A migration happens when $C = w_f + w_m = 2K$ (to comply with this our algorithm could trivially set $C = 2K$ when $C > 2K$; this would yield the same decisions). The accumulated cost of the page we are running the algorithm for is C ; at time t suppose that **conservative** is in state s_1 , while optimal is in state s_2 . We define a potential function $\phi(t)$ as follows:

$$\phi(t) = \begin{cases} 2C & \text{if } s_1 = 2s_2 \\ 3K - C & \text{otherwise} \end{cases}$$

Observe that $\phi(t) \geq 0$ and $\phi(0) = 0$. Also $CONS(0) = OPT(0) = 0$. For each possible event at a time t , we will show that $\Delta_{CONS} + \Delta_\phi \leq 3\Delta_{OPT}$, in which Δ_X indicates the change in the value of X as a result of the event. By summing over all events we obtain the desired inequality (since $\phi \geq 0$). Possible events are:

1. Transition of **conservative**. Then $\Delta_{CONS} = K$ and $\Delta_{OPT} = 0$. Before the transition $C = 2K$ holds, and after the transition $C = 0$ holds. Also:
 $\Delta_\phi = \phi(t+1) - \phi(t) = (3K - 0) - 2 \cdot 2K = -K$, if $s_1 = s_2$ and
 $\Delta_\phi = \phi(t+1) - \phi(t) = 0 - (3K - 2K) = -K$, if $s_1 \neq s_2$.
In both cases $\Delta_{CONS} + \Delta_\phi = K - K = 0 \leq 3\Delta_{OPT} = 0$.
2. Transition of the optimal off-line algorithm. Then, $\Delta_{CONS} = 0$ and $\Delta_{OPT} = K$. Also:
 $\Delta_\phi = (3K - C) - 2C = 3K - 3C \leq 3K$, if $s_1 = s_2$ and
 $\Delta_\phi = 2C - (3K - C) = 3C - 3K \leq 6K - 3K = 3K$, if $s_1 \neq s_2$.
Hence $\Delta_{CONS} + \Delta_\phi = 0 + 3K = 3K = 3\Delta_{OPT}$.
3. The last event is serving a read/write request. Let c_1 be the cost of serving the request in state s_1 and c_2 in state s_2 . Then $\Delta_{CONS} = c_1$ and $\Delta_C = c_1 - c_2$. If $s_1 = s_2$, $\Delta_{OPT} = c_1$ and $\Delta_\phi = 2 \cdot \Delta_C \leq 2c_1$ since $\Delta_C = c_1 - c_2 \leq c_1$. Thus:
 $\Delta_{CONS} + \Delta_\phi \leq c_1 + 2c_1 = 3c_1 = 3\Delta_{OPT}$
If $s_1 \neq s_2$, then $\Delta_{OPT} = c_2$ and $\Delta_\phi = -\Delta_C = c_2 - c_1$. Therefore:
 $\Delta_{CONS} + \Delta_\phi \leq c_1 + c_2 - c_1 = c_2 \leq 3c_2 = 3\Delta_{OPT}$

Thus **conservative** is 3-competitive for the symmetric graph H with transition costs equal to K for both states. We will show that **conservative** is also 3-competitive for our asymmetric graph G that has a transition cost of w_f when moving to f and w_m when moving to m . Since the algorithm is the same (and starts from the same state), it will perform the same transitions in G as it would in H , in which case it would be 3-competitive. However, two consecutive transitions on H would cost the same as they cost on G (because the cost of a cycle is $w_f + w_m$ in both graphs). Thus, the algorithm has the same cost in both graphs if it performs an even number of transitions, and an extra cost of $w_f - \frac{w_f - w_m}{2}$ for G and an odd number of transitions. This extra cost is constant, so **conservative** is 3-competitive in G as well. \square