Fine-grained Updates in Database Management Systems for Flash Memory

Zhen He and Prakash Veeraraghavan
Department of Computer Science and Computer Engineering
La Trobe University
VIC 3086
Australia
{z.he, p.veera}@latrobe.edu.au

April 17, 2009

Abstract

The growing storage capacity of flash memory (up to 640 GB) and the proliferation of small mobile devices such as PDAs and mobile phones makes it attractive to build database management systems (DBMSs) on top of flash memory. However, most existing DBMSs are designed to run on hard disk drives. The unique characteristics of flash memory make the direct application of these existing DBMSs to flash memory very energy inefficient and slow. The relatively few DBMSs that are designed for flash suffer from two major short-comings. First, they do not take full advantage of the fact that updates to tuples usually only involve a small percentage of the attributes. A tuple refers to a row of a table in a database. Second, they do not cater for the asymmetry of write versus read costs of flash memory when designing the buffer replacement algorithm. In this paper, we have developed algorithms that address both of these short-comings. We overcome the first short-coming by partitioning tables into columns and then group the columns based on which columns are read or updated together. To this end, we developed an algorithm that uses a cost-based approach, which produces optimal column groupings for a given workload. We also propose a heuristic solution to the partitioning problem. The second short-coming is overcome by the design of the buffer replacement algorithm that automatically determines which page to evict from buffer based on a cost model that minimizes the expected read and write energy usage. Experiments using the TPC-C benchmark[18] show that our approach produces up to 40-fold in energy usage savings compared to the state-of-the-art in-page logging approach.

Keywords: Flash memory, database, caching, buffer replacement, vertical partitioning, and database optimization.

1 Introduction

In recent times, the flash memory has become one of the most prevalent technologies for persistent storage of data, ranging from small USB memory sticks to large solid state drives. The cost of these devices is rapidly decreasing, while the storage capacity is rapidly increasing. This opens up the possibility for databases to store data on flash memory rather than hard disk drives. The following two factors make flash memory more desirable than hard disk drives for data storage:

High Speed. One of the main impediments to faster databases remains the speed of accessing the hard disk drive. The lack of spinning devices make the solid state drive (SSD) up to 100-fold faster than the hard disk drive (HDD) for random access time and similar or better than HDD for sequential read speed. Random access time is critical to the performance of database systems since databases rely heavily on indexes to provide fast data access. Typically, an index lookup involves loading several different pages at different locations on the disk, hence generating many random accesses. However, the sequential write speed of SSD is typically similar or

slower than HDD. The asymmetry between read and write speed of SSDs is one of the main challenges in building a high performing database on SSDs. Table 1 compares the speed differences between typical HDDs and SSDs. The table shows the asymmetry between read and write speeds of SSDs.

Suitable for portable devices. The suitability of flash memory compared to the hard disk for use in small portable devices is apparent due to the fact that it is smaller, lighter, noiseless, more energy efficient and has much greater shock resistance. For small devices that are battery powered, the low energy consumption of the flash memory is of the greatest importance. Table 2 compares the energy consumption characteristics of typical flash drives versus typical HDDs. The table also shows the energy consumption of HDDs are significantly higher than flash memory.

Device	Random Access Time (ms)	Sequential Read Speed (MB/s)	Sequential Write Speed (MB/s)
Western Digital HDD (WD1500ADFD)	8	74.7	74.3
Mtron SSD (SATA/150)	0.1	94.6	74.2
SanDisk SSD (SSD5000)	0.11	68.1	47.3

Table 1: Table comparing speed of typical HDD and SSD[25]

Device	Idle Power Consumption (Watts)	Seek Power Consumption (Watt)
Super Talent 2.5” IDE Flash Drive 8GB	0.07	0.38
Super Talent 2.5” SATA25 Flash Drive 128GB	1.19	1.22
Hitachi Travelstar 7K100 100GB SATA HDD	1.53	3.81
Western Digital Caviar RE2 500 GB SATA HDD	8.76	10.57

Table 2: Table comparing energy consumption of typical flash memory devices with HDDs[8].

Designing a DBMS customized for flash memory requires understanding flash memory energy consumption characteristics. As mentioned above, the most important characteristic of flash memory is that it can not be updated efficiently. To update a particular tuple in a table, we need to erase the entire block (which typically contains more than 2621 tuples[1]) that the tuple resides in and then rewrite the entire contents of the block. This is very costly in terms of energy consumption. A better scheme is to write the updated tuple into a new area on the flash memory. However, this out-of-place update causes the flash memory to become fragmented, i.e. flash memory blocks will contain both valid and invalid tuples.

There is a relatively small amount of existing work [4, 13, 16, 26] on designing DBMSs for flash memory, all of which use storage models customized for the out-of-place update nature of flash memory. However, they do not take full advantage of the fact updates often only involve one or very few attributes of a tuple. For example, in the TPC-C benchmark[18], we found on average each update only modified 4.5% of the tuple. So there is significant savings that can be gained from restricting writes to flash memory to only those portions of the tuple that are updated.

Our main approach to reducing flash memory IO costs is to partition the data into columns. Each group of columns corresponds to a set of attributes of the table which tend to be updated or read together. The result is each tuple is partitioned into sub-tuples, where the attributes for a particular sub-tuple are usually updated or read together. This results in the fact that most of the time, updates can be confined to a small sub-tuple rather than the entire tuple. Since we treat each sub-tuple as an inseparable entity, we do not keep a list of all updates to the sub-tuple, just the location of the most recent version of the sub-tuple. This results, in at most, one page load when the sub-tuple is loaded since

[1] Assuming each tuple is less than 50 bytes and a block is 128 Kbytes.

the entire sub-tuple is kept together on one page. In order to determine the best partitioning of a table into groups of columns, we use a cost model that incorporates read and write costs to flash memory and frequency by which attributes are read or written together. We propose both an optimal and a greedy solution to the partitioning problem.

In this paper, we use an RAM buffer to reduce IO costs. The RAM buffer is much smaller than the entire database. The database is stored persistently on flash memory. The challenge is to optimally use the limited RAM buffer in order to reduce expensive read and write operations to flash memory, as flash memory writes are particularly expensive, as mentioned earlier. Therefore, we need to pay particular attention to optimal caching of updated data in RAM. To this end, we cache data at the fine sub-tuple grain and propose a flash memory customized buffer replacement algorithm which minimizes the amount of data written to flash memory. Despite the importance of caching for improving system performance, previous DBMSs [4, 13, 16, 26] built for flash memory have not focused on creating buffer replacement policies that cater for high write versus read costs of flash memory. In this paper, we propose a buffer replacement algorithm that dynamically adjusts itself in response to changes in frequency of reads versus writes and the cost of reads versus writes to flash memory.

In summary our paper make the following main contributions: 1) we incorporate flash memory characteristics into data partitioning decisions; 2) we cache data at the fine sub-tuple grain, thereby reducing the amount of data written to flash memory; and 3) we propose a buffer replacement algorithm that evicts sub-tuples based on a cost formula customized to the characteristics of flash memory.

We have conducted a comprehensive set of experiments comparing the performance of our approach versus the state-of-the-art in-page update (IPL) approach[16]. The results show that we outperform IPL in all situations tested by up to 40-fold in terms of total energy consumption. The results also show that partitioning tuples into sub-tuples outperform non-partitioning by up to 80% in terms of total energy consumption.

The paper is organized as follows: Section 2 describes the unique characteristics of NAND flash memory; Section 3 surveys the related work; Section 4 describes our caching approach, which includes our novel buffer replacement algorithm; Section 5 formally defines the table partitioning problem; Section 7 describes our solutions to the partitioning problem; Section 8 describes the experimental setup used to conduct our experiments; Section 9 provides the experimental results and analysis of our experiments; and finally Section 10 concludes the paper and provides directions for future work.

2 Characteristics of NAND flash memory

There are two types of flash memory, the NOR and NAND flash memory. Reading from NOR memory is similar to reading from RAM in that any individual byte can be read at a time. Hence, it is often used to store programs that run in place, meaning programs can run on the NOR memory itself without having to be copied into RAM. NAND memory, in contrast, reads and writes at the page grain, where a page is typically 512 bytes or 2 KB. Writing at this coarse grain is similar to hard disk drives. The per megabyte price of NAND memory is much cheaper than NOR flash memory and hence, it is more suitable to use as a secondary storage device. Hence, in this paper we focus on the NAND flash memory. In order to clearly understand the design choices of our system, the reader needs to have detailed understanding of the performance characteristics of NAND flash memory. In this section, we describe these characteristics in detail.

First, we describe the basic IO operations of NAND flash memory.

- **Read:** NAND flash memory works similarly to typical block devices for read operations in the sense that they read at a page grain. Pages are typically 2 KB in size.
- **Write:** Writing is also performed at the page grain. However, writing can only be performed on freshly-erased pages.
- **Erase:** Erase is performed at the block level instead of page level. A block is typically 128 KB in size (fits 64 pages). The implication of this characteristic is that we need to reclaim flash memory space at the block instead of page grain.

Some limitations of NAND flash memory are related to the basic IO operations. These limitations are described as follows:

Operation	Access Time ($\mu s/4KB$)	Energy Consumption ($\mu J/4KB$)
Read	284.2	9.4
Write	1833.0	59.6
Erase	499.2	16.5

Table 3: The characteristics of NAND flash memory when 4KB of data is read/written/erased.

- **Very costly in-place update:** As already mentioned, erases at the block level must precede writes. This means in-place updates are very costly. The consequence is the need to store updates to tuples in a new location. This out-of-place tuple update gives rise to fragmentation (i.e. blocks with mixed valid and invalid tuples). Fragmentation will lead to flash memory space running out quickly. To reclaim space, a block recycling policy needs to be used. Recycling is expensive since it requires reading valid tuples from the block to be recycled and then erasing the block before writing the new content back into the flash memory.
- **Asymmetric read and write operations:** The energy cost and speed of reading and writing to flash memory is very different. Writing typically costs a lot more energy and is much slower. Table 3 taken from [15] shows the energy costs and speed of reading and writing to a typical NAND flash device. It points out that write costs about 6.34 times that of read, which is even higher if you include the erase costs. The implication of higher write costs is that cache management decisions should be more biased toward reducing the number of writes compared to reads.
- **Uneven wear-out:** Flash memory has a limited number of erase write cycles before a block is no longer usable. Typically, this is between and 100, 000 and 5, 000, 000 for NAND flash memory. This means to prolong the life of the NAND flash memory, data should be written evenly throughout the flash memory. This is called wear-leveling.

3 Related Work

We review related work in four main areas: buffer replacement for flash devices; databases built for flash memory; flash translation layer; and column based stores.

In the area of buffer replacement algorithms for flash memory, there is some existing work. Wu et. al. [28] was among the first to propose buffer management for flash memory. They mainly focused on effective ways of garbage cleaning the flash when its space had been exhausted. However, they also proposed a very simple first in first out (FIFO) eviction policy for dirty pages in RAM.

One approach to buffer replacement is to extend the existing least recently used (LRU) policy so that it becomes biased towards evicting clean pages before dirty pages [22, 23]. The reason we take this approach is that evicting dirty pages is much more expensive than clean pages since it involves an expensive write to flash memory. Park et. al. [22] was the first to propose this approach. The policy first partitions the pages in RAM into the most recently accessed w pages and then partitions the rest. Clean pages in the w least recently accessed pages are first evicted. If there are no clean pages in the w least recently accessed pages, then the normal LRU algorithm is used. However, there was no discussion on how to select the best value for w. Park et. al. [23] extended this work by proposing several cost formulas for finding the best value for w.

Jo et. al. [11] designed a buffer replacement algorithm for portable media players which uses flash memory to store media files. Their algorithm evicts entire blocks of data instead of pages. When eviction is required, they choose the block that has the most number of pages in RAM as the eviction victim, LRU, is used to break ties. This type of eviction policy is suitable for portable media players since writes are more likely to be long sequences, but it is much less suitable for databases since updates in databases are typically more scattered.

All existing work on buffer replacement algorithms mentioned above are designed for operating systems and file systems running on flash memory. They all manage the buffer at the page or block grain. In contrast, our buffer replacement algorithms operate at the tuple or sub-tuple grain and are designed for databases. In Section 4.1, we explain how performance gain can be achieved via caching at the finer tuple or sub-tuple grain.

Another difference between our work and existing buffer replacement algorithms is that we propose a cache management algorithm that balances between clean and dirty data in a way that is independent of any particular buffer replacement policy. We accomplish this by creating two buffers: a clean data buffer and a dirty data buffer. Then, cost models are used to balance the size of the two buffers so that the overall energy usage is minimized. Within each buffer, any traditional buffer replacement algorithm such as LRU, FIFO, etc. can be used.

There is a relatively small amount of existing work [4, 13, 16, 26] on building databases especially designed to run on flash memory. Bobineau et. al. [4] proposed the PicoDBMS which is designed for smart cards. PicoDBMS features a compact storage model that minimizes the size of data stored in flash memory by eliminating redundancies and also features a query processing algorithm that consumes no memory. Sen et. al. [26] proposed an ID-based storage model which is shown to considerably reduce storage costs compared to the domain storage model. In addition, they propose an optimal algorithm for the allocation of memory among the database operators. Kim et. al. [13] proposed the LGeDBMS which is a database designed for flash memory. They adopt the log-structured file system as the underlaying storage model. The above papers focus on storage models used by the DBMSs rather than the use of RAM caches to avoid or minimize reading or writing to the flash memory. In contrast, this paper is focused on efficient algorithms for caching database tuples in RAM and table partitioning in order to minimize reading and writing to flash memory.

The work that is closest to ours is that done by Lee et. al.[16]. They proposed the in-page logging (IPL) algorithm for flash memory based database servers. They use a logging-based approach because, for flash memory-based systems, it is better to write a log record of an update rather than updating the data directly. The reason for this is, in a flash memory-based system, updating a small part of a page inplace requires erasing the entire block the tuple resides in, which is much more expensive than using the logging approach. Their approach is to reserve a fixed set of pages of each block to store a log of updates to the pages inside that block. A log page is flushed when it is full or the RAM buffer is full or when a transaction is committed. When a page p is read from the flash memory, the set of log pages for the block that p resides in is loaded and the updates to page p are applied. When the log pages of a block are exhausted, the data and log pages in the block are merged, thereby freeing up the log pages in the block. This approach has a number of drawbacks. First, loading a page requires loading the log pages which means extra page reads. Second, using a fixed number of log pages per block will result in under-utilization of a large proportion of the flash memory since updates typically do not occur uniformly across all pages of a database. This under-utilization of some parts of the flash memory will mean the log and data pages in the frequently updated blocks will be frequently merged. In contrast, our algorithm suffers from neither of the drawbacks mentioned above.

Operating systems on small devices typically use a flash translation layer (FTL) to manage the mapping from logical to physical block or page ids. To cope with the fact pages can not be written back in-place without erasing an entire block, the FTL writes an updated page to a new location in the flash memory. When there are no more free pages left in flash memory, the garbage collector is used to reclaim space. Most FTL perform wear leveling. Kim et al. [14] proposed a FTL scheme that writes updated pages into a fixed number of log blocks. When all log blocks are used, the garbage collector is used to merge certain blocks to free space. This scheme requires a large number of log blocks since even if only one page of a block is updated, a corresponding log block needs to be created. The Fully Associative Sector Translation (FAST) FTL scheme [17] overcomes this short-coming by allowing the updates to a page to be written into any block. Therefore, a block can contain a mixture of updated pages from different blocks. This added flexibility decreases the frequency of garbage collection. Kawaguchi et al. [12] proposed a FTL that supports the UNIX file system transparently. It used a log-based file structure to write updates sequentially on to the flash memory. The work in this paper does not use any particular FTL scheme but instead manages the reading and writing to the flash memory according to the details specified in Section 4. The reason we do this is that we want better integration between the buffer replacement algorithm and the physical organization of data on the flash memory.

There has been extensive existing work in both vertical [19, 20, 21, 7, 6, 9] and horizontal [3, 29] partitioning of databases. In vertical partitioning, the database tables are partitioned into sets of columns. The columns within each set are stored together. This way IO can be minimized by only loading the column sets which contain the columns needed by the query. In horizontal partitioning sets of tuples of tables that are likely to be accessed together are placed together in the same page. This way pages containing tuples that are not needed by a query do not need to be loaded, thereby reducing the number of IO. All these papers propose various partitioning algorithms to reduce reading costs for data stored on hard disk drives. In contrast, our paper is focused on balancing between the asymmetric reading and writing costs of flash memory and integrating the design of caching algorithms with partitioning to achieve large

overall benefits.

A number of recent vertical partitioning papers have focused on the CStore [1, 27, 10, 2]. The idea is to separate the database into a read optimized store which is column-based and a writable store which is write optimized. Groups of columns are stored in different sorted orders inside the read optimized store. This approach allows cluster indexes to be built on multiple columns. This means the same column may exist in different column groups. Compression techniques are used to keep the total size of the store from becoming too large. All updates first occur in the write optimized store and then, at a later time, batched together and merged with the read optimized store using the tuple mover. These papers mostly focus on improving the read performance of databases using the hard disk as the secondary storage device. In contrast, our paper which focuses on storing data in flash memory, concentrates on grouping different attributes of a tuple together based on the overall benefit in terms of read and update costs of using the flash memory. In addition, we propose intelligent caching algorithms to take maximum advantage of optimal data partitioning to further improve performance.

4 Data Caching

In this section, we will focus on the caching aspect of the database system on flash memory. The core contribution of this paper is to partition and cache data at the sub-tuple grain. The different possible grains of caching are described and compared in Subsection 4.1. Next, in Subsection 4.2, we describe the approach to locating sub-tuples stored on flash memory. Next, in Subsection 4.3, we describe our cache management algorithm. Lastly, in Subsection 4.4, we describe our approach to maintain a balance between clean versus updated tuples in the cache.

4.1 Granularity of caching

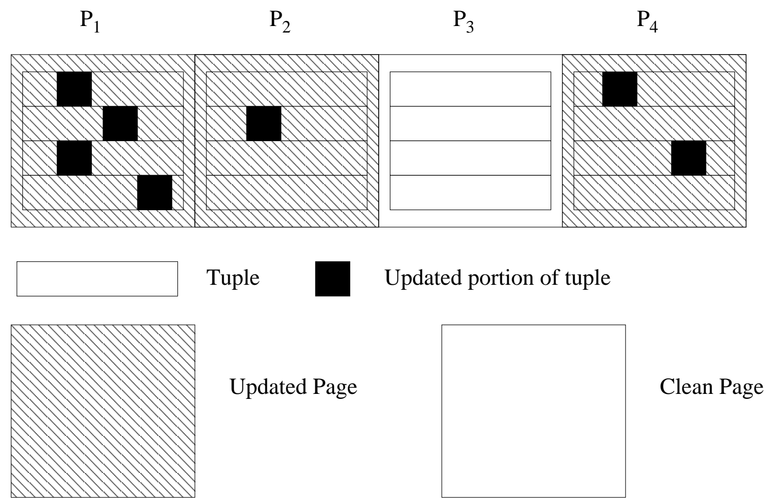
We cache the data at the sub-tuple grain instead of the tuple or page grain. A sub-tuple is a concatenation of a subset of attribute values of a tuple. Figure 1 shows the effect of caching at the three different grains of page, tuple and sub-tuple. Figure 1 (a) shows that when caching at the page grain, even when only one small part of a tuple is updated (as the case for page P_2), the entire page needs to be flushed when evicted from the cache. In the case of the tuple grained caching (Figure 1 (b)), only the tuple that contains updated parts is flushed (as in the case of page P_2, only the second tuple is flushed). In the case of the sub-tuple grained caching (Figure 1 (c)), the system can potentially only flush the parts of the tuples that have been updated (the block squares in the diagram). We have found that in typical database applications, each update usually only updates a small portion of a tuple which suggests sub-tuple grained can save a lot of needless data flushes. For example, in the TPC-C benchmark we found on average each update only modified 4.5% of the tuple. We arrived at this figure from analyzing the trace of a real execution of the TPC-C benchmark.

In this paper, we use the sub-tuple grained caching for the reasons mentioned above. We formally describe the problem of partitioning tuples into sub-tuples in Section 5 and the proposed solutions to the problem in Section 7.

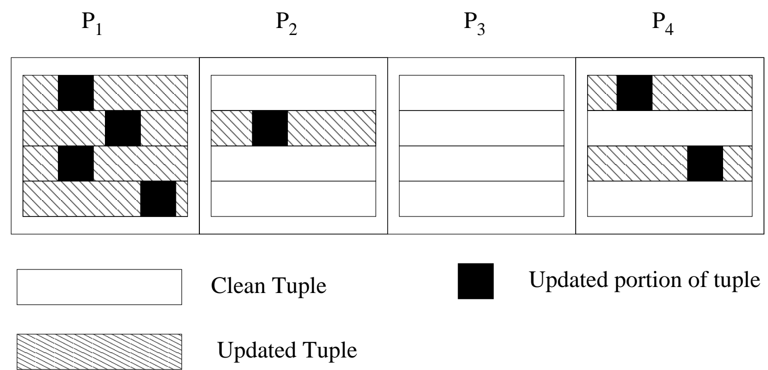
4.2 Locating sub-tuples

In a typical database, there can be a large number of sub-tuples. An important problem is how to find the location of a sub-tuple on flash memory efficiently. Here, location means which page and what offset within the page. Using an indirect mapping table which maps each sub-tuple ID to its location would be prohibitively expensive in terms of memory usage. However, using a direct mapping-based approach where we encode the location of the sub-tuple inside the sub-tuple ID itself will not work when the sub-tuple is updated. This is due to the out-of-place update of the flash memory. This is when a sub-tuple is updated, its old location is invalided and it must be written to a new location when it is flushed from the RAM buffer. Therefore, the tuple ID would need to be updated if the direct mapping approach is used. Updating tuple ID would be prohibitively expensive since it would require updating every reference to the tuple in the database. Hence, excluding the possibility of using a direct mapping only based approach.

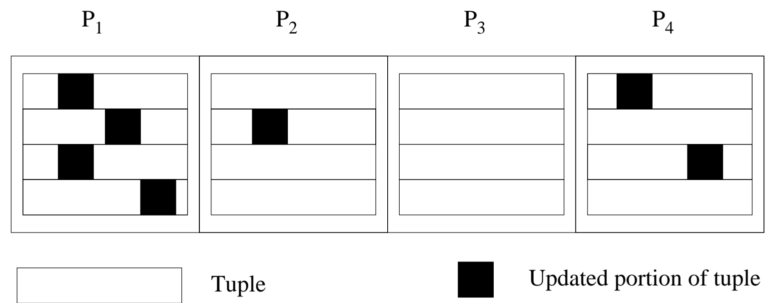
To address this challenge, we use a combination of direct and indirect mapping. All sub-tuples start off using direct mapping. However, when a sub-tuple is moved by either the garbage collector or the fact it is updated, an indirect mapping entry is inserted in an indirect mapping table. When the system is offline (eg. at night), the sub-tuples are rearranged in the flash memory so that direct mapping can be used again and the indirect mapping table



(a) Page grained caching



(b) Tuple grained caching



(c) Sub-tuple grained caching

Figure 1: The effects of caching at different grains.

is deleted. We keep the number of entries in the indirect mapping table small by reducing the amount of sub-tuples moved by the garbage collector. This is done by separating sub-tuples that are likely to be updated versus those that are unlikely to be updated in the flash memory. This separation is performed by the partitioning algorithm of Section 7. This results in large portions of the flash memory containing effectively read-only sub-tuples which will never be touched by the garbage collector and will not be updated and therefore no indirect mapping entries are needed. In our experiments using the TPC-C benchmark with 584259 sub-tuples, and running 2500 transactions (the default settings for our experiments), we found there were only 34289 entries in the indirect mapping table. This means only 5.9% of the sub-tuples of the entire database had entries in the indirect mapping table.

At run-time, sub-tuples are located by first checking if an entry exists in the indirect mapping table for the sub-tuple. If the mapping exists, then the indirect mapping table is used to locate the sub-tuple, otherwise direct mapping is used.

4.3 Cache management algorithm

In this section, we describe the algorithm used to manage the RAM cache designed to minimize reads from and costly evictions to flash memory. When the buffer is full and a page is to be loaded, some data from the buffer needs to be evicted to make room. To this end, we have designed a flash memory customized buffer replacement algorithm. The main novelty in the algorithm is the idea of logically splitting the cache into two parts, each with a different maximum size limit. One part is called the clean sub-tuple cache and the other the dirty sub-tuple cache. Each cache internally uses a traditional buffer replacement algorithm such as LRU to keep the data stored inside within its assigned maximum size limit. Figure 2 shows an example splitting of RAM into the two caches diagrammatically. We dynamically vary the size of the clean versus dirty sub-tuple caches so that we maintain the optimal balance between evicting clean versus dirty sub-tuples. Evicting clean sub-tuples is much cheaper than dirty sub-tuples since it only involves discarding the sub-tuples from RAM, but evicting dirty sub-tuples requires writing them into flash memory which is much more expensive. However, making the clean cache size very small would result in the very frequent reloading of popular sub-tuples. This would result in excessive flash memory read IO. We use the clean versus updated ratio ($CVUR$) to set the size of the clean versus dirty caches. For example, given a maximum total cache size of 1MB and a $CVUR$ of 0.2, the clean cache would be 0.2 MB and the dirty cache would be 0.8 MB. $CVUR$ is dynamically readjusted at every eviction according to a number of different factors, such as how frequent reads are compared to writes, the cost of read versus write, etc. The details on how $CVUR$ is set is shown in Section 4.4.

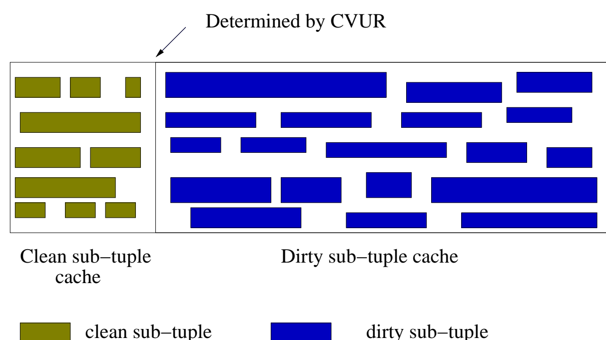


Figure 2: Example shows the clean and the dirty sub-tuple caches.

Figure 3 shows the algorithm run when a cache miss occurs and therefore a page needs to be loaded from flash memory. Since data can not be updated in place, it must be written to a different area of flash memory, and the old data becomes invalid. A page therefore can contain many invalid sub-tuples. When a page is loaded, only the valid sub-tuples contained in it are cached. Therefore, we need to free up enough RAM to store all its valid sub-tuples. Line 2 is where we determine the amount of free space needed and we store it in the variable L. Lines 4 and 5 show where we recompute and use the $CVUR$ ratio. Line 5 shows the criteria used to decide when to flush updated sub-tuples or evict clean sub-tuples. Updated sub-tuples are evicted when the ratio of clean versus updated sub-tuples currently

in the buffer is greater than the clean versus updated ratio threshold. This is when there is more updated sub-tuples than what is deemed optimal in the buffer, then updated sub-tuples are evicted rather than clean sub-tuples. If updated tuples need to be evicted, a flash block is found according to lines 6 to 15.

The algorithm in Figure 3 invokes all the cache management components. Our cache management system contains three main components. These components are described as follows:

- **Flash block recycling manager:** When a new flash memory block is required but there is no free component, this component is used to pick a used block for recycling. This component is invoked in line 8 of the algorithm shown in Figure 3. There are many existing policies for choosing the block to be recycled from the file system literature[5, 24, 28]. Two popular existing policies are the *greedy* [28] and the *cost-benefit* [5] policy. The greedy policy selects the block with the largest amount of garbage for recycling whereas the cost benefit policy also considers the age and number of times the block has been previously erased. The greedy policy has been shown to perform well on uniform distributed updates but performs poorly for updates with high locality[28]. The cost-benefit policy has been shown to outperform greedy when locality is high[5]. Any of these policies can be used for this component.

- **Updated sub-tuples eviction manager:** This component first chooses which updated sub-tuples from the dirty cache to write into the flash memory and then writes it into block f (where f is defined in line 8 of the algorithm). This component is invoked in line 16 of the algorithm shown in Figure 3. It is triggered when the RAM buffer is full and the total size of clean tuples divided by total size of updated tuples is *below* $CVUR$. This effectively means the dirty sub-tuple cache will overflow if we do not evict from it. Any existing buffer replacement algorithm such as LRU, CLOCK, etc. can be used internally here to determine which sub-tuples are to evicted. This component evicts first U dirty sub-tuples suggested by the buffer replacement algorithm for eviction, where U is chosen such that the total size of the evicted sub-tuples is at least L but not any bigger than necessary.

- **Clean sub-tuples eviction manager:** This component is used to decide which clean sub-tuples from the clean sub-tuple cache to discard and then evicts them. This component is invoked in line 18 shown in Figure 3. It uses the same technique as the updated sub-tuples eviction manager except it evicts clean rather than updated sub-tuples.

4.4 Maintaining balance between clean versus updated sub-tuples

At the heart of effective cache management for flash memory DBMSs is the correct balance between keeping clean versus updated data in memory. The reason is the asymmetric cost of writing versus reading. As mentioned in the previous section, we maintain the balance by dynamically adjusting the size of the clean versus dirty sub-tuple caches via the clean versus updated ratio (CVUR). We dynamically compute CVUR based on past usage statistics and flash IO costs.

In our approach, CVUR is determined based on the cost of reading an average sized sub-tuple multiplied by the probability of a read versus the cost of writing an average sized sub-tuple multiplied by the probability of a write. So when the system observes there are more frequent reads of large clean sub-tuples, it will make the clean sub-tuples buffer larger.

The equation below formally defines how CVUR is determined:

$$\begin{aligned} CVUR &= \frac{\text{expected cost per byte of loading a clean sub-tuple after it has been evicted}}{\text{expected cost per byte of evicting a dirty sub-tuple}} \\ &= \frac{P_r \times CBB_r \times AVG_{cs}}{P_w \times CBB_w \times AVG_{us}} \end{aligned} \tag{1}$$

where P_w and P_r are the stationary probability of sub-tuple write and read requests, respectively (estimated by dividing the number of write/read requests to any sub-tuple by the total number of requests), CBB_w is the cost per byte of

```
Load_Page ( p: requested page, B: maximum buffer size)
1. load p from flash memory into RAM
2. // let L = total size of valid sub-tuples in p
3. if (current buffer size + L > B)
4.     compute CVUR using Equation 1
5.     if (total size of clean sub-tuples / total size of updated sub-tuples < CVUR)
        // we should evict from the dirty sub-tuples cache.
6.       // let f be a flash block used to store the evicted updated sub-tuples
7.       if ( flash memory == full )
8.           f = flash resident block picked by recycling manager
9.           load f from flash memory
10.          copy valid sub-tuples from f into the RAM buffer
            // we need to free more RAM in order to fit the copied valid sub-tuples
11.          increase L by the amount of valid sub-tuples copied
12.          erase f from flash memory
13.       else
14.          f = a flash block with at least one free page
15.       end if
16.       evict at least L amount of updated sub-tuples into f
17.   else
        // we should evict from the clean sub-tuple cache.
18.       evict at least L amount of clean sub-tuples
19.   end if
20. end if
21. copy valid sub-tuples of p into the RAM buffer
22. discard p from RAM
```

Figure 3: Algorithm for managing memory when a cache miss occurs and thus a page needs to be loaded from flash memory.

writing an updated sub-tuple into flash memory, CBB_r is the cost per byte of reading a sub-tuple from flash memory, AVG_{cs} is the average size of clean sub-tuples, and AVG_{us} is the average size of updated sub-tuples.

We use Equation 1 to define $CVUR$ as the ratio of expected cost of loading clean sub-tuples versus expected cost of evicting dirty sub-tuples. We use expected cost since this allows us to effectively use the average cost of loading clean sub-tuples versus evicting dirty sub-tuples in past as a prediction for future costs.

The cost per byte of writing a dirty sub-tuple into flash memory CBB_r is computed simply by the following:

$$CBB_r = \frac{CPL}{PS} \tag{2}$$

where CPL is the cost of loading a page of flash memory and PS is the size of a page in bytes. However, computing CBB_w is not so simple since it needs to include such factors as: the cost of recycling a block (occurs when writing onto a block that contains some valid data) and erasing a block. The reason is writes must occur on pages that have been freshly erased and erasing is done at the block level. We consider two situations: 1) the data is written onto a recycled block (occurs when there are no empty blocks) and 2) the data is written to an empty block. The following equation is used to compute CBB_w:

$$\begin{aligned} CBB_w &= \begin{cases} \dfrac{\text{cost of recycling block}}{\text{number of free bytes on block reused}} & \text{if recycled block} \\[2ex] \dfrac{\text{cost of erasing and writing block to flash memory}}{\text{number of bytes in a block}} & \text{otherwise} \end{cases} \\ &= \begin{cases} \frac{CBL+CEB+CWB}{BS-VTS} & \text{if recycled block} \\[1ex] \frac{CEB+CWB}{BS} & \text{otherwise} \end{cases} \end{aligned} \tag{3}$$

where CBL is the cost of loading a block from flash memory, CEB is the cost of erasing a block, CWB is the cost of writing a block into flash memory, BS is the size of a block in bytes and VTS is the total size of valid sub-tuples in the recycled block in bytes. In order to recycle a block, we need to first load it into memory and then erase it and then place the valid sub-tuple back into the block. Therefore, the amount of space that we actually recover from recycling a block is the size of the block minus the size of the valid sub-tuples in the block.

$CVUR$ is computed completely dynamically, meaning it is computed every time eviction is required. This is because it can be computed very cheaply since the terms in Equation 1 can be kept up-to-date incrementally. P_r and P_w are trivial to compute incrementally by simply keeping a counter of the number of sub-tuple reads and writes. CBB_r is a constant and CBB_w is estimated by assuming the total size of valid sub-tuples in the recycled block (VTS of Equation 3) is the same as in the previously recycled block. This assumption is reasonable since most garbage collection algorithms recycle the block with the most amount of invalid data. Typically, the amount of invalid data on consecutive recycled blocks (blocks with most invalid data) should be similar.

5 Tuple Partitioning Problem Definition

As mentioned in Section 4.1, we partition tuples into sub-tuples in order to reduce the amount of data updated. In this section, we formally describe the tuple partitioning problem.

Before the formal definition, we first give an intuitive description of the problem. We wish to find the best column grouping such that the total cost (for both read and write) of running a given workload is minimized. This grouping decision should be based on which columns are frequently read or updated together. Grouping frequently read columns together minimizes the number of flash page loads. Columns are frequently read together in databases, such as the following example: a customer's contact details are retrieved. In this case, the columns containing the address and phone numbers of the customer are read together. Grouping columns which are frequently updated together minimizes the amount of data written into the flash memory since data is dirtied at the sub-tuple grain. An example to illustrate the advantage of updating the attributes (columns) of a sub-tuple all together is the following: suppose a sub-tuple

has ten attributes but only one of the attributes is updated. This will cause the entire sub-tuple of ten attributes to be marked dirty and later to be flushed to flash memory. Hence, in this example, it is better to store the updated attribute into a single sub-tuple. Our problem also incorporates the fact a column X can be updated together with column Y in one transaction and later read together with column Z in a different transaction. Therefore, in our problem definition, we minimize a cost function that considers all group accesses.

For simplicity of exposition, we define the problem in terms of partitioning the set of attributes $A = \{a_1, a_2, ., a_n\}$ of a single table T. However, this is at no loss to generality since we partition the tables independent of each other. In this paper, we assume every tuple fits into one page of flash memory.

Our aim is to find non-overlapping partitions of A such that the total energy cost of updates and reads to the flash memory are minimized. To this end, we are given a query workload Q_w, estimates on cache miss rates and costs per byte of reading and writing to the flash memory. Q_w is specified in terms of the number of times subsets of A are updated or read together (within one query). Cache miss rates are estimated based on the stationary probability each reading or writing of a subset of attributes of A will generate a cache miss. Finally, costs per byte of reading and writing to flash memory are computed as that specified in Section 4.4.

We provide a formal problem definition. Let $Q_w = \{< s, N_r(s), N_u(s) > |\forall s \in S\}$, where S is the set of all subsets of A which have been either read and/or updated together in one query, $N_r(s)$ and $N_u(s)$ are the number of times s have been read and updated together in one query, respectively. For example, consider a table with three attributes, in which the first and second are updated together 3 times and the second and third read together 2 times. For this example, $Q_w = \{< \{a_1, a_2\}, 0, 3 >, < \{a_2, a_3\}, 2, 0 >\}$. Although in theory there can be a large number of attribute subsets that can be read and/or updated together ($|S|$ is large), in practice $|S|$ is typically small since typically, database applications have a small set of different queries which are repeatedly run with nothing but the parameter values varying. This is true since most database applications issue a query from a form that the user fills in. In this case, the query format is fixed and the only changes are the parameter values.

We define the reading cost $RC(x)$ for attribute set $x \subset A$ as follows:

$$\begin{aligned} RC(x) &= CPL \sum_{s \in S_r(x)} \text{expected number of cache misses incurred when reading } s \\ &= CPL \sum_{s \in S_r(x)} N_r(s) P_m \end{aligned} \tag{4}$$

where $S_r(x)$ is the attributes sets in S which have been read together and contain x, CPL is the cost of loading a page from flash memory as was first used in Equation 2 and P_m is the stationary probability of a cache miss. The reading cost is computed in terms of the expected number of cache misses because requests for attributes that are RAM resident do not incur and IO costs.

Note, according to the above equation, the probability of a cache miss is the same for any subset of attributes $s \in S$. That is P_m is defined independent of s. This assumption is necessary to make the cost estimation practical because accurately modeling the probability of a cache miss for every s independently requires us to model the order by which requests are issued and the exact way the buffer replacement algorithm works. The reason is the probability of incurring a page miss is dependent on all of these factors. Therefore, in this paper, we make the common practical assumption that queries follow the independent identically distributed (IID) reference model and hence we make P_m independent of s. We describe how P_m is estimated in Section 6. Note although P_m does not reflect the fact that some attribute sets are more "hot" than others the cost formula (Equation 4) as a whole does. This is because the read cost is dependent on $N_r(s)$ which is the number of times the attribute set s is read.

We define the updating cost $UC(x)$ for attribute set $x \subset A$ as follows:

$$
\begin{aligned}
UC(x) &= \sum_{s \in S_u(x)} \text{expected number of times } s \text{ is flushed} \times \text{cost of flushing } s \\
&= \sum_{s \in S_u(x)} N_u(s) P_f \times SB(x) CBBS2_w
\end{aligned}
\tag{5}
$$

where $N_u(s_u)$ is the number of times s_u is updated, P_f is the stationary probability that an update of a sub-tuple will cause it to be flushed to flash memory before it is updated again, $SB(x)$ is the total size of attribute set x in bytes, $CBBS2_w$ is the cost per byte of flushing a sub-tuple to flash memory. We again assume the IID reference model between queries. Hence, we assume P_f does not depend on which attributes s are updated. This is for the same reasons as for P_m of Equation 4.

The definition of $CBBS2_w$ is similar to the definition of $CBBS_w$ of Equation 3 except that here, we do not know if the sub-tuple will be flushed to a recycled block or new block and also we do not know how much valid sub-tuples are left in the recycled block. Hence, we take the average value of these variables from previous runs. $CBBS2_w$ is defined as follows:

$$
CBBS2_w = \left(\frac{CEB + CWB + CBL}{BS - AVTS}\right)\alpha + (1 - \alpha)\left(\frac{CEB + CWB}{BS}\right) \tag{6}
$$

where CEB, CWB, CBL, BS, are the same as those defined for Equation 3, $AVTS$ is the average size of all valid sub-tuples for previously recycled blocks, α is the fraction of recycled versus non-recycled blocks used in the past.

Given the definitions above, the partitioning problem is defined as finding the best set of non-overlapping attribute sets $\{b_1, b_2, ., b_i, ., b_p\}$ of the attributes in A such that the following cost is minimized:

$$
cost(\{b_1, b_2, ., b_i, ., b_p\}) = \sum_{i=1}^{p} (RC(b_i) + UC(b_i)) \tag{7}
$$

subject to the constraint that the attribute sets $\{b_1, b_2, ., b_i, ., b_p\}$ need to be non-overlapping and must include every attribute in A.

Equation 7 gives the total read and write cost for a given non-overlapping set of attribute sets. This is simply done by summing the read and write cost of each of the constituent attributes sets (b_i). We can do this because none of the sets overlap each other, hence there is no double counting of costs when we sum them up.

6 Efficient Statistic Collection and Estimation

In this section, we describe one efficient technique for collecting and estimating the statistics $N_r(s)$, $N_u(s)$, P_m, P_f used in Section 5. However, the problem definition and algorithms proposed in the previous sections are general with respect to how the statistics are collected and estimated. As was done in Section 5, we assume the statistics collected are for a table T. Again, this is at no loss to generality.

We first explain how the $N_r(s)$ and $N_u(s)$ statistics are collected. We do the following for each query executed: run the query and then identify all the tuples read or updated by the query. For each tuple read, we record all the attributes read during the query and then we increment $N_r(s)$ for that set of attributes. We do the same for each updated query.

We now explain how the statistics $N_r(s)$ and $N_u(s)$ are stored. We build a hash table for each table in the database, where the key of the hash table is s represented by a string of binary bits. The length of the string is the number of attributes in the table. Each binary value represents the presence or absence of a particular attribute. The data that the key points to is $N_r(s)$ or $N_u(s)$. As already mentioned in Section 5, the cardinality of S is typically small so only small hash tables are required.

We estimate P_m by assuming reading different subsets of attributes of tuples have equal probability of generating cache misses which is similar for P_f for the reasons mentioned in Section 5. We use the following simple formula to

estimate $P_m(s)$ and $P_f(s)$:$P_m(s) = N_m/TN_r$ and $P_f(s) = N_f/TN_u$, where N_m is the total number of cache misses while reading sub-tuples of the table T and TN_r is the total number of sub-tuple reads for table T. N_f is the total number of sub-tuples of table T flushed and TN_u is the total number of sub-tuple updates for table T. Multiple reads of the same sub-tuple in one query is considered as a single read and similarly for updates, since we also consider multiple reads of the same tuple in one query as one read when computing $N_r(s)$ and $N_u(s)$.

7 Tuple Partitioning Solutions

In this section, we provide two alternative solutions to the problem outlined in Section 5. The first is an algorithm that finds the optimal solution with high run-time complexity and the second is a greedy solution that offers a good tradeoff between run-time complexity and quality of solution. Both solutions are preceded by an initial partitioning.

7.1 Important Observations

From the problem definition of Section 5, it can be seen the following two observations are evident:

Observation 1 creating larger sub-tuples tend to decrease read costs;

Observation 2 creating smaller sub-tuples tend to decrease write costs.

The first observation is evident from the fact the lowest read cost is achieved when all attributes of the same tuple are grouped together into one sub-tuple since this means no matter which set of attributes are read together, at most one cache miss is generated per attribute set read per tuple. In contrast, if n attributes are accessed together and each attribute was stored in a different sub-tuple, then potentially n pages may need to be loaded per tuple if each sub-tuple was stored on a different page. Due to the out-of-place update behavior of the system, sub-tuples of the same tuple are often placed in different pages due to different update times.

The second observation is evident from the fact the lowest write cost is achieved when each attribute of each tuple is partitioned into different sub-tuples. This way, updating any set of attributes of a tuple will result in only dirtying those attributes concerned. In contrast, for sub-tuples made up of all the attributes of the tuple, any single attribute update will result in dirtying all the attributes of the tuple.

Example 1 (Observation 1) This example shows how a larger attribute set can reduce the read cost according to Equation 7. In this example, we are only interested in the read cost and hence we assume there are no updates. Suppose we are given a table with four attributes and the following workload: $Q_w = \{attr_1, attr_2, attr_3\}$, where $attr_1 =< \{a_1, a_2, a_3\}, 5, 0 >, attr_2 =< \{a_2, a_3\}, 7, 0 >, attr_3 =< \{a_3, a_4\}, 8, 0 >$. Consider the following two ways of partitioning the table: $p_1 = \{\{a_1\}, \{a_2\}, \{a_3\}, \{a_4\}\}$ and $p_2 = \{\{a_1, a_3\}, \{a_2, a_4\}\}$. According to Equation 7 $cost(p_1) = CPL\ (5+7+8)\ P_m = 20\ CPL\ P_m$ and $cost(p_2) = CPL\ (5+7+8)\ P_m + CPL\ (5+7+8)\ P_m = 40\ CPL\ P_m$. From this example, we can see the single larger *attribute set* p_1, has smaller read cost compared to the two smaller *attribute set* p_2. □

Example 2 (Observation 2) This example shows how partitioning into smaller attribute sets can reduce the write cost according to Equation 7. In this example, we are only interested in the write cost and hence, we assume the workload has only update queries. Suppose we are given a table with four attributes each of size SI and the following workload: $Q_w = \{attr_1, attr_2, attr_3\}$, where $attr_1 =< \{a_1, a_2, a_3\}, 0, 3 >, attr_2 =< \{a_2, a_3\}, 0, 5 >, attr_3 =< \{a_3\}, 0, 2 >$. Consider the following two ways of partitioning the table: $p_1 = \{\{a_1, a_2\}, \{a_3, a_4\}\}$ and $p_2 = \{\{a_1\}, \{a_2\}, \{a_3\}, \{a_4\}\}$. According to Equation 7, $cost(p_1) = (3+5)\ P_f\ 2\ SI\ CBBS_w + (3+5+2)\ P_f\ 2\ SI\ CBBS_w = 36\ SI\ P_f\ CBBS_w$ and $cost(p_2) = 3\ P_f\ SI\ CBBS_w + (3+5)\ P_f\ SI\ CBBS_w + (3+5+2)\ P_f\ SI\ CBBS_w = 21\ P_f\ SI\ CBBS_w$. From this example, we can see each attribute residing in a different sub-tuple p_2, has smaller write cost compared to the two larger sub-tuples of p_1. □

It can be seen the two observation, advocate conflicting partitioning strategies. Therefore, our partitioning algorithms must balance the conflicting concerns in order to achieve minimum cost in terms of Equation 7.

7.2 Initial Partitioning

We initially partition the attributes of A into non-overlapping subsets so that all attributes that have been accessed (either updated or read) together in the workload Q_w are grouped into the same attribute set and all sets of attributes that are accessed in isolation from the rest are in separate attribute sets. Then, either the algorithm described in Section 7.3 or Section 7.4 can be applied to each non-overlapping subset disjointedly at no loss to quality of the solution. This approach reduces the run-time of the algorithms by reducing the search space for candidate attribute sets.

Figure 4 shows an example initial partitioning of a set of 8 attributes. Notice that attribute sets 2 and 3 have only one attribute each since they are always accessed in isolation from the other attributes. In this example, we can simply concentrate on further partitioning attribute set 1 and thereby reduce the run-time of the algorithm.

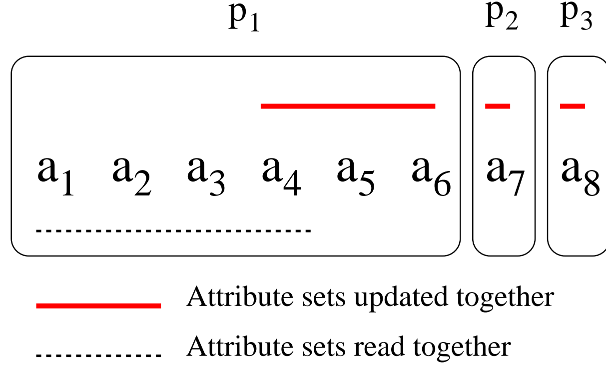


Figure 4: Example initial partitioning.

Theorem 1 states that the attribute sets produced from the initial partitioning preserve the optimality of the attribute sets in terms of cost as defined in Equation 7. Before defining the theorem, we first define accessed together (Definition 1) and transitively accessed (Definition 2).

Definition 1 : *Attribute $x \in A$ is* accessed together *with attribute $y \in A$, (xRy), if there exists $s \in S$ such that $x \in s \wedge y \in s$.*

Definition 2 : *Attribute $x \in A$ is* transitively accessed together *with attribute $y \in A$ within attribute set c, $(xT_c y)$, if there exists some n attributes, $\{\gamma_1, \gamma_2 \cdots \gamma_n\} \subset c$ such that $xR\gamma_1 R\gamma_2 \cdots R\gamma_n Ry$.*

Theorem 1 : *Let the initial attribute set C be a partitioning of A into non-overlapping attribute set $c \in C$ such that any two attributes in c are transitively accessed together and no attributes from different attribute sets are accessed together. Then, the optimal attribute sets produced from further partitioning C has the same cost as the optimal attribute sets produced starting from A, where cost is that defined by Equation 7.*

Proof: We prove this theorem by first proving that further partitioning C can lead to minimum read cost. Next, we prove that further partitioning C can lead to minimum write cost. According to Equation 7, the read cost is minimized when all attributes that are read together are placed in the same attribute set. The reason is in Equation 4, if x only contains one attribute in s, then the expected cache miss for s needs to be added. The way to minimize read cost is to group the attributes that have been read together s into the same attribute set x since this results in counting cache miss for each s only once. The definition of attribute sets C in Theorem 1 states all attributes transitively accessed together are placed in the same attribute set. Therefore, using C as the initial partitioning does not prevent minimum read cost from being reached. According to Equation 7, minimum write cost occurs when each attribute is placed in a separate attribute set. Since C is produced from only the initial partitioning, it can be further partitioned to achieve minimum write cost. Therefore, the further partitioning of C can achieve minimum cost according to Equation 7. □

Algorithm 5 shows the simple algorithm used to perform the initial partitioning. The algorithm produces an initial partitioning conforming to that defined in Theorem 1. Lines 2 to 5 ensure that all and only those attributes

that are transitively accessed together are in the same attribute set. This is because line 3 initially splits all attributes into separate attribute sets and lines 3 to 5 merge all and only those attribute sets that contain attributes that have been accessed together. The run-time complexity of the algorithm in Figure 5 is $O(|S| + |A|)$ since line 2 takes $|A|$ operations to separate each attribute to a separate attribute set and lines 3 to 5 perform $|S|$ merges.

```
Initial_Partition ( A: all attributes of table T, S: subsets of A which were read or updated
                    together )
1. // let C store the eventual resulting attribute sets
2. initially place each a ∈ A in a separate attribute set in C
3. for each s ∈ S
4.     merge the attribute sets in C which contain at least one element from s
5. end for
6. return resultant attribute sets C
```

Figure 5: Algorithm for initial partitioning of the attributes

7.3 Optimal Partitioning

In this section, we describe an optimal partitioning algorithm for the problem defined in Section 5. The algorithms further partition each of the initial attribute sets generated from Section 7.2 separately. The resultant attribute sets is proven to be optimal by Theorem 2.

The optimal algorithm starts by splitting each attribute set produced from the initial partitioning $c \in C$ into maximal elementary attribute sets. Intuitively maximal elementary attribute sets are a partitioning of c such that the minimum cost according to Equation 7 can be achieved by only the optimal merging of the maximal elementary attribute sets. This way, we can reduce the search space for the candidate attribute set since the number of maximal elementary attribute sets is typically smaller than the number of attributes in c. We define elementary attribute sets as follows:

Definition 3 : *An* elementary partition E *further partitions a set of initial attributes* c *into attribute sets* $e \in E$ *so the following property holds: the attributes in each* e *are* always updated *together or none of the attributes in* e *are ever updated.*

Definition 4 : A maximal elementary partition R *partitions a set of attributes* c *into elementary attribute sets so the following property holds: merging any pair of attributes sets* $x \in R$ *and* $y \in R$ *creates a resultant partition that is no longer an elementary partition.*

Figure 6 shows an example of maximal elementary partitioning 8 attributes. In the example, no pair of attributes from the first four attributes are *always* updated together. Hence, the first four attributes are partitioned into separate four elementary attribute sets. However, attributes 5 and 6 are always updated together and therefore, they are grouped into the same elementary attribute set. Attributes 7 and 8 are never updated and therefore are grouped into the same elementary attribute set. The elementary partition is maximal because merging any pair of the attribute sets creates a resultant partition that is not an elementary partition.

Theorem 2 : *The minimum* $cost(opt(R)) = cost(opt(c))$*, where cost is that defined by Equation 7,* $opt(R)$ *is the minimum cost partition produced from optimally merging the maximal elementary partition* R *of* c *and* $opt(c)$ *is the minimum cost partition produced when given the freedom to merge any attributes of* c *in any way.*

Proof: We consider the attribute sets that are read and updated together separately in our proof. We start by proving that S_r the subset of attribute sets in S which are read together can be ignored. The reason is the elementary attribute sets in R can be further combined to arrive at $opt(R)$. According to Equation 7 and 4, larger attribute sets will always result in the same or smaller read costs since the same $s_r \in S_r$ may overlap any attribute set that contains the attributes in s_r, however given the freedom to combine any attribute sets you can always combine them such that the resultant

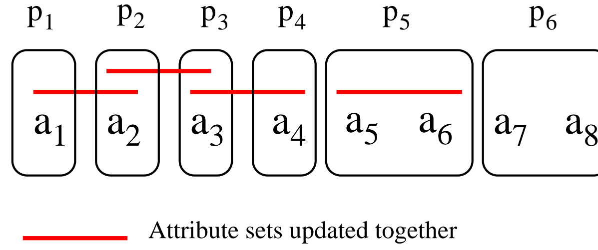


Figure 6: Example elementary partition.

partition covers s_r entirely and hence results in the minimum read cost for s_r.

We now consider S_u the subset of attribute sets in S which are updated together. According to the definition of elementary partition, each $r \in R$ can either: 1) exactly overlap one element $s_u \in S_u$ and does not overlap the other elements of S_u; or 2) does not overlap any elements of S_u at all. For the first case, update cost is minimized in terms of update costs incurred for s_u since exactly the attributes in s_u are updated when s_u is updated, no extra attributes are updated. The second case does not affect update cost since it means none of the attributes in r are updated at all. Therefore, first partitioning c into a maximal elementary partition and then finding the optimal combination of them into attribute sets will result in optimal partitioning. □

Figure 7 shows the algorithm that partitions c into a maximal elementary partition. The algorithm first puts all the attributes into a single attribute set and then iterate through each set of attributes that are updated together (lines 3 - 13). Then in lines 6 - 11 the algorithms iterates through all the current attribute sets r in R and partition the ones that overlap the current s_u into two halves: the attributes of r that overlap s_u; and the attributes of r that do not overlap s_u. Figure 8 shows an example of partitioning an attribute set r into two maximal attribute sets k_1 and k_2. In the example $(r \cap s_u) = k_2 = \{a_3, a_4\}$ (line 8) and $(r - s_u) = k_1\{a_1, a_2, a_5, a_6\}$ (line 9). The result is K stores the two elementary attribute sets k_1 and k_2 which are maximal since they can not be merged to create a larger elementary attribute set.

```
Create_Maximal_Elementary_Partitions(c: an element of C, S_u: subset of S which
                                     were updated together)
1. // let R store the resulting maximal elementary partition.
2. initialize R to have one attribute set which contains all attributes in c.
3. for each s_u ∈ S_u
4.     // let K store the set of current elementary attribute set
5.     initialize K to be empty
6.     for each attribute set r ∈ R
7.         if (r ∩ s_u ≠ 0)
8.             place the attribute set (r ∩ s_u) into K
9.             place the attribute set (r − s_u) into K
10.        end if
11.    end for
12.    R = K
13. end for
14. return R
```

Figure 7: Algorithm for creating elementary partition.

Figure 9 shows the optimal partitioning algorithm. The algorithm first uses the initial partitioning algorithm (Figure 5) to partition the attributes into sets that can be further partitioned separately without losing the possibility of finding the optimal solution. Second, the algorithm iterates through the set of initial attribute sets individually and further partitions them (lines 5 - 11). For each of the initial attribute sets $c \in C$ the maximal elementary attribute set (Figure

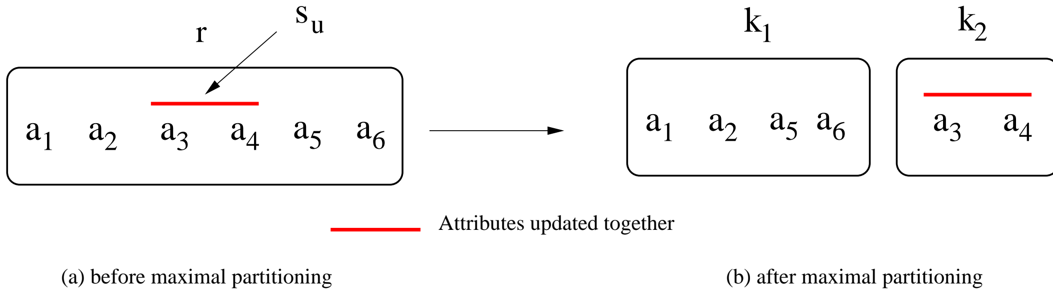


Figure 8: Example maximal partition.

7) is computed. Then, in line 10, we find the partition of c that produces the minimum cost according to Equation 7 among all possible ways of combining elementary attribute sets in R.

```
Optimal_Partition(A: all attributes of table T, S_u: subset of S which were updated together)
1. // Let C be a set of attribute sets produced from the initial partitioning algorithm.
2. C = Initial_Partition(A)
3. // Let H store the resultant optimal partition of A
4. Initialize H to be empty
5. For each c ∈ C do
6.    // Let R store maximal elementary partition of c.
7.    R = Create_Maximal_Elementary_Partitions(c, S_u)
8.    // Let AllPart(R) be the set of all possible groupings of the sets in R
9.    e = min_{b∈AllPart(R)}(cost(b))
10.    place the attribute set e into H
11. end for
12. return H
```

Figure 9: Algorithm for performing the optimal partitioning.

Theorem 3 : *The algorithm in Figure 9 produces an optimal partition of the attributes of A according to Equation 7.*

Proof: According to Theorem 1, further optimal partitioning the initial attribute sets produced from line 2 of the algorithm will lead to an optimal solution. The algorithm then partitions each of the initial attribute sets into maximal elementary attribute sets in line 7. According to Theorem 2, the optimal partition can be obtained by the optimal combining of the maximal elementary attribute sets. In line 9 of the algorithm, every combination of maximal elementary attribute sets are considered and the one with the minimum cost is selected. Therefore, the algorithm in Figure 9 produces the optimal partition. □

Theorem 4 : *The worst case run-time complexity of the algorithm in Figure 9 is $O(|S|+\sum_{c\in C}(|c||S_u|+\sum_{i=1}^{|R|} S_2(|R|,i)))$, where $|S|$ is the number of attribute sets that are read or updated together, $|c|$ is the number of attributes in the attribute set c, $|S_u|$ is the number of sets of attributes that are updated together, $|R|$ is the number of maximal elementary attribute sets for c and $S_2(|R|,i)$ is the stirling numbers of the second kind function. $S_2(|R|,i)$ returns the number of distinct partitions of a set with $|R|$ elements into i sets.*

Proof: The $|S|$ is the time-complexity for the initial partitioning algorithm (Figure 5) used in line 2 of Figure 9. Then for each of the initial attribute sets $c \in C$, the following number of operations are performed: $|c||S_u|$ which is the worst case run-time complexity for the maximal elementary partition algorithm (Figure 7), since in the worst case R

has $|c|$ elements in it; and $\sum_{i=1}^{|R|} S_2(|R|, i)$ is the number of operations it takes to consider all possible attribute sets of $|R|$ elements.

□

In practice, the average run time overhead is low since the initial partitioning and maximal elementary partitioning typically make $|R|$ small, hence the $\sum_{i=1}^{|R|} S_2(|R|, i)$ is relatively small and the rest of the terms in the time complexity are polynomial.

7.4 Greedy Partitioning

The high worst case time complexity of the optimal algorithm is due to iterating through all possible combinations of maximal elementary partitions. Although this guarantees the optimal solution, it may be too slow to be useful for the situation where the maximal elementary partitioning has many attribute sets. Hence, we propose a greedy solution to the problem which offers a near optimal solution with much lower run-time complexity.

Figure 10 shows the algorithm for the greedy partitioning. The idea is to first put each maximal elementary attribute set into a separate attribute set (line 7) and then find the best pair of attribute sets to merge (lines 8 - 19). The reason for starting with the maximal elementary partition is that maximal partitioning is fast and does not prevent the optimal partition from being found. Continuously grouping together the best pair of attribute sets (the pair that produces the lowest cost) produces a near optimal solution without having to perform an expansive exhaustive search. If the cost of the best pair is higher or equal to not grouping any, then partitioning ends (line 17) otherwise group the best pair (lines 14 - 15). Repeat until no pair gives lower cost (line 17) or all attributes are in one attribute set (line 9).

```
Greedy_Partition(A: all attributes of table T, S_u: subset of S which were updated together)
1. // Let C be a set of attribute sets produced from the initial partitioning.
2. C = Initial_Partition(A)
3. // Let H store the resultant partition of A
4. Initialize H to be empty.
5. For each c ∈ C do
6.     // Let k store resultant partition of c
7.     k = Create_Maximal_Elementary_Partitions(c, S_u)
8.     bestNotMergeCost = cost(k)
9.     while (|k| > 1)
10.        // Let Pairs(k) be the set of all possible pairs of attribute sets in k merged
11.        // Let Merged(k, p) be the partition produced when p ∈ Pairs(k) is merged
12.        e = min_{p∈Pairs(k)}(cost(Merged(k, p)))
13.        if (cost(Merged(k, e)) < bestNotMergeCost)
14.           bestNotMergeCost = cost(Merged(k, e))
15.           merge the two attribute sets in k that correspond to the attribute sets in e
16.        else
17.           exit while loop
18.        end if
19.    end while
20.    place the attribute sets in k into H
21. end for
22. return H
```

Figure 10: Algorithm for performing the greedy partitioning.

Theorem 5 : *The worst case run-time complexity of the algorithm in Figure 10 is* $O(|S|+\sum_{c\in C}(|c||S_u|+|r|^3))$*, where* $|S|$ *is the number of attributes sets that are read or updated together,* $|c|$ *is the number of attributes in the attribute set* c*,* $|S_u|$ *is the number of sets of attributes that are updated together,* $|r|$ *is the number of maximal elementary attribute sets in* c*.*

Proof: The $|S|$ term is the run-time complexity for the initial partitioning algorithm (Figure 5) used in line 2 of Figure 10. Then, for each of the initial attribute sets $c \in C$ the following amount of operations are performed: $|c||S_u|$ which

Parameter	Value
Page size	2 KB
Block size	128 KB
Flash memory size	109 MB
RAM size	1 MB

Table 4: Simulation parameters used in the experiments

is the worst case run-time complexity for the maximal elementary partition algorithm (Figure 7), since in the worst case R has $|c|$ elements in it; and $|r|^3$ is the worst case run-time complexity when each of the $|r|$ maximal elementary attribute sets are merged, since in this case $O(|r|^2)$ pairs of the maximal elementary attribute sets are considered for each of the $|r|$ merges.

□

8 Experimental Setup

In this section, we will describe the details of the experimental setup that we used to conduct the experiments for this paper.

8.1 Simulation

Our experiments were conducted on a simulation of the NAND flash memory device. The simulation modeled the NAND flash memory characteristics according to those described in Section 2. In particular, we used the energy consumption characteristics of Table 3. The other parameters of the simulation are described in Table 4. Unless otherwise specified, the parameter values described in Table 4 are those used in the experiments.

8.2 Benchmark

The experiments were conducted using the TPC-C benchmark. We have modeled the TPC-C benchmark in the same way as that of [18]. However, to test varying amounts of locality, we used three random distributions whenever random values such as customer ID, order ID, etc. need to be selected. The three random distributions used are described as follows:

Uniform uniform random distribution. The distribution with lowest amount of locality.

Default the random distribution used in the TPC-C specifications.

Gaussian a distribution that exhibits the highest locality. It is created by first using uniform random distribution to find three centers in a given one-dimensional range of values. The centers are used as centers of a gaussian distributed random number generator. The gaussian distribution is confined to have a variance of one and a range of 0.1% of the full range. The following describes which of the three centers is used whenever a random number needs to be generated. To model temporal locality, a chosen center is used for 100 transaction before the next center is chosen. When choosing the next center, zipf distribution with a z parameter of two is used.

We modeled all five transactions in the TPC-C benchmark. We used uniform random distribution to select which of the five transactions to run at each turn.

8.3 Algorithm Settings

In the experiments, we have compared the results of four algorithms and they are described as follows:

IPL This is the in-page logging algorithm from Lee et. al.[16]. The paper describes two versions of the algorithm, one that provides full recovery and another that does not. We simulated the non-recovery version because our algorithm does not support recovery either and therefore this is a fairer comparison. We used the default parameters specified in the paper except we used a page size of 2KB instead of 512 bytes and 4 log pages per block instead of 16. The reason for this is that NAND flash memory typically has a page size of 2KB and since the pages are bigger, less log pages are required per block. We used the least recently used buffer replacement policy for IPL.

Tuple-Grain This algorithm uses our caching architecture as described in Section 4 but caches at the tuple instead of sub-tuple grain. This algorithm is included to allow us to determine the benefits caching at the sub-tuple grain instead of tuple grain.

ST-Optimal This is our algorithm using the optimal partitioning algorithm described in Section 7.3 to partition tuples into sub-tuples. We first run the algorithm using a training workload to partition the tuples into sub-tuples. Then, we run the testing workload on the partitioned tuples to generate the results. The difference between the training and testing workloads is the random seed used.

ST-Greedy This is set up the same way as ST-Optimal except the greedy partitioning algorithm described in Section 7.4 is used instead.

Tuple-Grain, ST-optimal and ST-Greedy all used the cache management algorithm proposed in Section 4.3. The flash block recycling manager used the greedy policy. The updated and clean eviction managers used the least recently used policy.

9 Experimental Results

We have conducted three experiments comparing our sub-tuple based partitioning algorithms against the existing IPL algorithm. The first experiment compares the algorithms when the RAM size is varied. The second experiment compares the algorithms as the number of transactions increases. The third experiment compares the algorithms when the amount of locality in the workload varies.

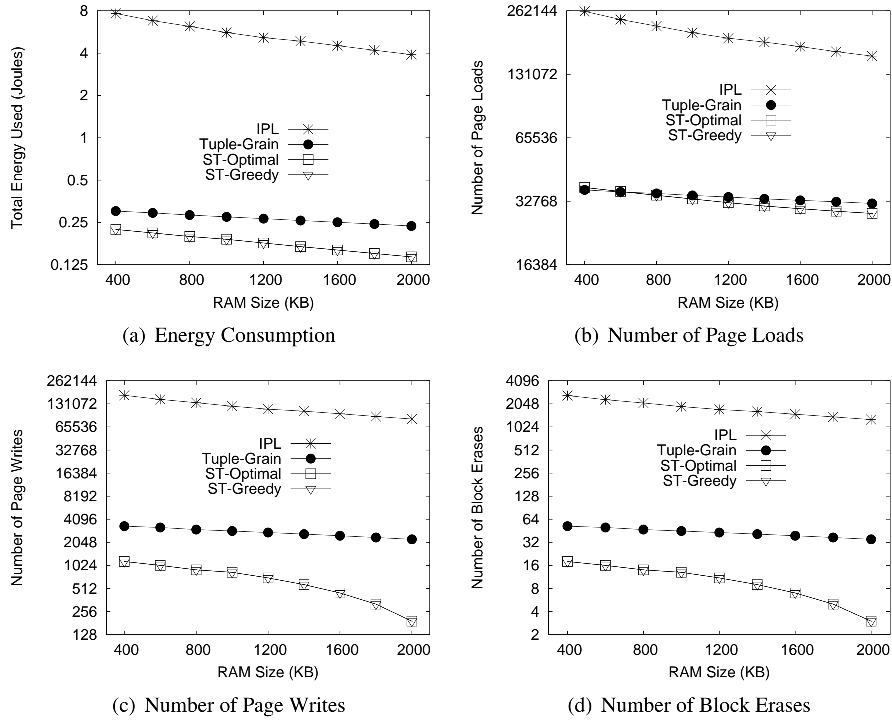
9.1 Varying RAM Size Experiment

This experiment compares how the algorithms perform when the available RAM size increased. For the experiment, we used the default workload setting of the TPC-C benchmark (see Section 8.2). In this experiment we ran 2500 transactions.

Figure 11 shows the results of this experiment. All the results are shown using $\log_2$ scale on the y-axis since the performance difference between our sub-tuple partitioning algorithm and IPL is very large. The graphs report the results of the following four metrics: energy consumption; number of page loads; number of page writes; and number of block erases. For all the metrics measured, our sub-tuple partitioning algorithms (ST-Greedy and ST-Optimal) outperform IPL by a very large margin. ST-Greedy and ST-Optimal outperform IPL by up to 32-fold for total energy consumption. The most significant difference occurs for the number of pages writes, with ST-Greedy and ST-Optimal outperforming IPL by up to 500-fold. There are three reasons for this: 1) IPL does not allow updates to different blocks to be mixed into the same log page, the consequence being that there will be many log pages that are only partially full when it needs to be flushed which results in more page flushes; 2) ST-Greedy and ST-Optimal write updates to tuples at the sub-tuple grain which means the amount written per update is very small; and 3) ST-Greedy and ST-Optimal use the dynamic clean versus dirty sensitive buffer replacement algorithm developed in this paper, whereas the buffer replacement algorithm used by IPL does not distinguish between clean and dirty pages.

Figure 11 (b) shows ST-Greedy and ST-Optimal also outperform IPL for reads. This is because whenever a data page is loaded in IPL, any log pages for the block that the data page resides in must also be loaded. ST-Greedy and ST-Optimal do not have log pages and therefore do not load them when loading a data page.

The results show that ST-Greedy and ST-Optimal outperform Tuple-Grain by up to 80% for total energy consumption. However, Tuple-Grain performs very similarly to ST-Greedy and ST-optimal for the number of page reads but



(a) Energy Consumption (b) Number of Page Loads

(c) Number of Page Writes (d) Number of Block Erases

Figure 11: Results of varying RAM size. The y-axis is scaled by $\log_2$

the sub-tuple partitioning algorithms significantly outperform Tuple-Grain for number of page write and block erases. The reason for the similar performance for read is due to observation 1 of Section 7.1, namely creating larger tuples tends to decrease read costs. Therefore, for read cost sub-tuple partitioning can not really do much to outperform the naive approach of simply grouping all attributes into the same attribute set. The reason that the sub-tuple partition algorithms can slightly outperform Tuple-Grain for read cost is that the sub-tuple partition algorithms can move those attributes that are never read into a separate attribute set which will never end up being loaded and occupying RAM space. This means that the sub-tuple partitioning algorithm makes more efficient use of RAM. When comparing the number of pages written, ST-Greedy and ST-Optimal outperform Tuple-Grain by up to 10-fold. This is due to the fact that ST-Greedy and ST-Optimal write updates to tuples at the sub-tuple grain which means the amount written per update is smaller than for Tuple-Grain which may, for example, write out an entire tuple of 21 attributes even if only one attribute is updated.

The results show that ST-Greedy and ST-Optimal's performance are almost identical. We also found that they both only take a few seconds to perform the partitioning. This is because they both do the same initial partitioning (Figure 5) and maximal elementary partitioning (Figure 7). In our experiments, we found the maximal elementary attribute sets produced at the end of these two partitioning algorithms contain a very small number of attribute sets (often just one), which means there is a very small search space to explore in terms of finding the best combination of maximal elementary attribute sets. How to find the best combination of maximal elementary attribute sets is where ST-Greedy and ST-Optimal differ.

9.2 Varying Number of Transaction Experiment

This experiment compares the algorithms as the number of transactions processed increases. For the experiment, we again used the default workload setting of the TPC-C benchmark (see Section 8.2).

Figure 12 shows the results of this experiment. Again all the results are shown using $\log_2$ scale on the y-axis. The results reflect the same trends as that of the varying RAM size experiment, namely the following: the sub-tuple

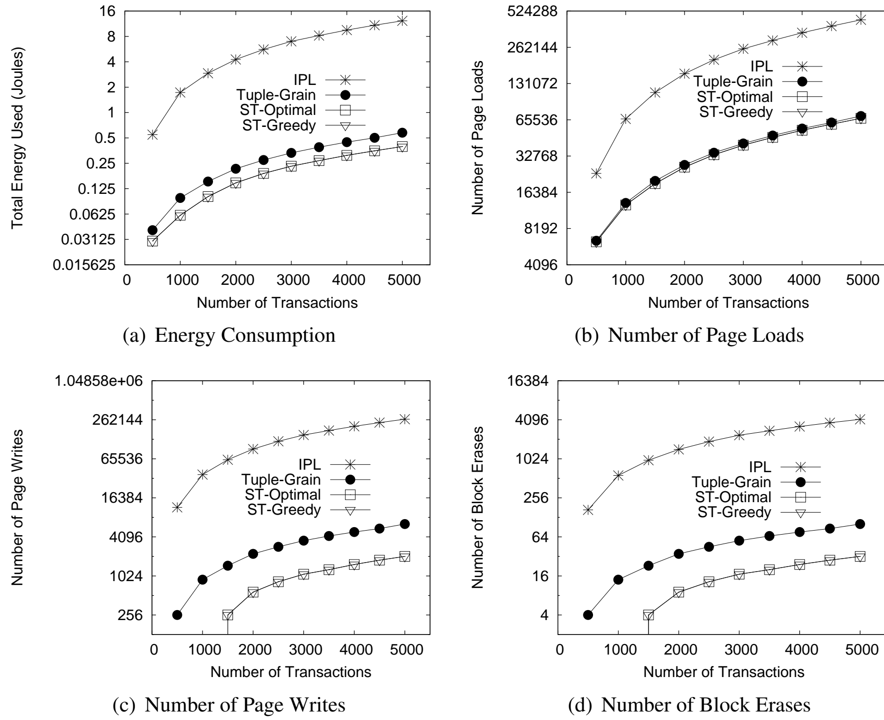


Figure 12: Results of varying number of transactions. The y-axis is scaled by $\log_2$

partitioning algorithm consistently significantly outperforms IPL across all cases tested and for all metrics measured (up to 40-fold for total energy consumption); sub-tuple partitioning outperforms Tuple-Grain; and ST-Greedy performs almost identically to ST-Optimal. The trends can be explained by the same reasons as for the varying RAM size experiment.

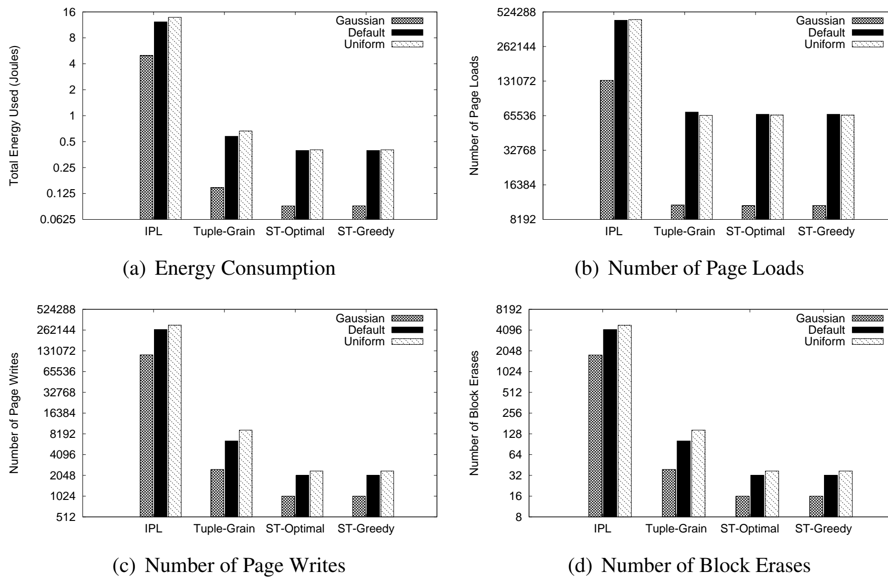
9.3 Varying Workload Locality Experiment

In this experiment, we have compared the algorithms across the three different workloads described in Section 8.2. The workloads contain different amounts of locality. This experiment was conducted by running 5000 transactions.

Figure 13 shows the results of this experiment. Again, all the results are shown using $\log_2$ scale on the y-axis. There are two key observations that can be made from this experiment. Firstly, the results show that the sub-tuple partitioning algorithms significantly outperform IPL for all workloads, up to 40-fold for the total energy consumption metric. This is for the same reason as for the previous experiments. Secondly, the results show that all algorithms perform better when there is a high amount of locality in the workload (Gaussian). This is due to its ability to fit a larger proportion of the working set in RAM when the locality is high.

10 Conclusion

In this paper, we have introduced a novel partitioning based approach to overcome the key flash memory limitation of expensive in-place updates. The partitioning is used to localize the portion of a tuple dirtied to only those attributes that are involved in the update. This reduces the amount of data dirtied which, in turn, reduces the expensive writes to flash memory. We have formulated the partitioning problem as one that minimizes the cost incurred for read and write to flash memory. We have proposed both an optimal and a greedy solution to the problem. The results show that the greedy algorithm performs almost identically to the optimal algorithm.



(a) Energy Consumption (b) Number of Page Loads

(c) Number of Page Writes (d) Number of Block Erases

Figure 13: Results of varying locality in the workload. The y-axis is scaled by $\log_2$

Another important contribution that this paper makes is the development of a buffer replacement algorithm that dynamically adjusts itself to minimize the combined read and write costs to flash memory. The idea is to separate the cache into clean and dirtied regions and then to dynamically adjust the size of the two regions, based on expected read versus write costs.

We have conducted an extensive performance comparison of our approach versus the state-of-the-art IPL algorithm. The results demonstrate that our approach significantly outperforms IPL for all situations and metrics tested. This can be mainly attributed to the use of partitioning into sub-tuples to minimize writes and the dynamic self-adjusting buffer replacement of our algorithms.

An important area of future work is to explore ways of incorporating the ideas in this paper into a flash DBMS that supports full data recovery. The system will need to make updates persistent when a transaction commit occurs. Another area of future work is to modify the partitioning problem to focus on placing data on a hybrid system that has both flash memory and a hard disk drive. The idea is to determine how to vertically partition the data into sub-tuples to benefit a hybrid system. We can then decide which sub-tuples to place on flash memory versus hard disk drive. The partitioning algorithm proposed is static, meaning all partitioning must be done offline. Exploring effective dynamic partitioning algorithms will be a useful direction for future research. Another area for future research is to modify the cache management algorithm proposed for use in flash based virtual memory systems instead of databases. There should be minimal changes required for this purpose. Finally, developing horizontal partitioning algorithms for flash databases will a very interesting area of future research.

11 Acknowledgements

This work is supported under the Australian Research Council's Discovery funding scheme (project number DP0985451).

References

[1] D. J. Abadi, S. Madden, and M. Ferreira. Integrating compression and execution in column-oriented database systems. In *Proceedings of ACM-SIGMOD*, 2006.

[2] D. J. Abadi, D. S. Myers, D. J. DeWitt, and S. R. Madden. Materialization strategies in a column-oriented dbms. In *Proceedings of ICDE*, 2007.

[3] S. Agrawal, V. Narasayya, and B. Yang. Integrating vertical and horizontal partitioning into automated physical database design. In *Proceedings of ACM SIGMOD*, pages 359–370, 2004.

[4] C. Bobineau, L. Bouganim, P. Pucheral, and P. Valduriez. PicoDBMS: Scaling down database techniques for the smartcard. In *Proceedings of VLDB*, pages 11–20, 2000.

[5] M. L. Chiang, P. C. H. Lee, and R. C. Chang. Managing flash memory in personal communication devices. In *Proceedings of the 1997 International Symposium on Consumer Electronics*, pages 177–182, 1997.

[6] G. P. Copeland and S. N. Khoshafian. A decomposition storage model. In *Proceedings of ACM SIGMOD*, pages 268–279, 1985.

[7] D. W. Cornell and P. S. Yu. An effective approach to vertical partitioning for physical design of relational databases. *IEEE Transaction to Software Engineering*, 16(2):248–258, 1990.

[8] G. Gasior. Super talent's SATA25 128GB solid-state hard drive: capacity to spare by performance? http://techreport.com/articles.x/13163/1, September 2007.

[9] Y.-F. Guang and C.-H. Van. Vertical partitioning in database design. *Information Sciences*, 86(1-3):19–35, 1995.

[10] S. Harizopoulos, V. Liang, D. Abadi, and S. Madden. Performance tradeoffs in read-optimized databases. In *Proceedings of VLDB*, 2006.

[11] H. Jo, J. Kang, S. Park, J. Kim, and J. Lee. FAB: Flash-aware buffer management policy for portable media players. *IEEE Transactions on Consumer Electronics*, 52(2):485–493, 2006.

[12] A. Kawaguchi, S. Nishioka, and H. Motoda. A flash-memory based file system. In *Proceedings of the USENIX Technical Conference*, pages 155–164, 1995.

[13] G. Kim, S. Baek, H. Lee, H. Lee, and M. J. Joe. LGeDBMS: a small DBMS for embedded system with flash memory. In *Proceedings of VLDB*, pages 1255–1258, 2006.

[14] J. Kim, J. M. Kim, S. Noh, S. L. Min, and Y. Cho. A space efficient flash translation layer for compact flash systems. *IEEE Transactions on Consumer Electronics*, 48(2):366–375, 2002.

[15] H. Lee and N. Chang. Low-energy hetergeneous non-volatile memory systems for mobile systems. *Journal of Low Power Electronics*, 1(1):52–62, 2005.

[16] S.-W. Lee and B. Moon. Design of Flash-Based DBMS: An in-page logging approach. In *Proceedings of ACM SIGMOD*, pages 55–66, 2007.

[17] S.-W. Lee, D.-J. Park, T.-S. Chung, D.-H. Lee, S. Park, and H.-J. Song. A log buffer-based flash translation layer using fully-associative sector translation. *ACM Transactions on Embedded Computing Systems*, 6(3):18, 2007.

[18] S. T. Leutenegger and D. Dias. A modeling study of the tpc-c benchmark. In *Proceedings of ACM SIGMOD*, pages 22–31, 1993.

[19] S. Navathe, G. Ceri, G. Wiederhold, and J. Dou. Vertical partitioning algorithms for database systems. *ACM Transaction on Database Systems*, 9(4):680–710, 1984.

[20] S. Navathe and M. Ra. Vertical partitioning for database design: A graphical algorithm. In *Proceedings of SIGMOD*, 1989.

[21] S. Papadomanolakis and A. Ailamaki. Autopart: Automating schema design for large scientific databases using data partitioning. In *Proceedings of 16th Internal Conference on Scientific and Statistical Database Management*, 2004.

[22] C. Park, J. Kang, S. Park, and J. Kim. Energy-aware demand paging on NAND flash-based embedded storages. In *Proceedings of ISLPED*, pages 338–343, 2004.

[23] S. Park, D. Jung, J. Kang, J. KIM, and J. Lee. CFLRU: A replacement algorithm for flash memory. In *Proceedings of CASES*, pages 234–241, 2006.

[24] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, 1992.

[25] P. Schmid and A. Roos. Samsung, ridata ssd offerings tested. Tom's Hardware Web Site, December 2007. http://www.tomshardware.com/2007/12/17/solid_state_drives/.

[26] R. Sen and K. Ramamritham. Efficient data management on lightweight computing devices. In *Proceedings of ICDE*, pages 419– 420, 2005.

[27] M. Stonebraker, D. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-store: A column oriented DBMS. In *Proceedings of VLDB*, pages 553–564, 2005.

[28] M. Wu and W. Zwaenepoel. eNVy: A non-volatile, main memory storage system. In *Proceedings of ACM ASPLOS*, pages 86–97, 1994.

[29] B. Zeller and A. Kemper. Experience report. exploiting advanced database optimization features for large-scale SAP R/3 installations. In *Proceedings of VLDB*, 2002.