Vertical Partitioning for flash and HDD database systems

Davur S. Clementsen and Zhen He
Department of Computer Science and Computer Engineering
La Trobe University
VIC 3086
Australia
dsclementsen@gmail.com, z.he@latrobe.edu.au

July 6, 2010

Abstract

Recent advances in flash memory technology have greatly enhanced the capability of flash memory to address the I/O bottleneck problem. Flash memory has exceptional I/O performance compared to the hard disk drive (HDD). The superiority of flash memory is especially visible when dealing with random read patterns. Even though the cost of flash memory is higher than that of HDD storage, the popularity of flash memory is increasing at such a pace that it is becoming a common addition to the average computer. Recently, flash memory has been made into larger devices called solid state drives (SSD). Although these devices can offer capacities comparable to HDDs, they are considerably more expensive per byte.

Our research aims to increase the I/O performance of database systems by using a small amount of flash memory alongside HDD storage. The system uses a fully vertically-partitioned storage structure where each column is stored separately on either the HDD or SSD. Our approach is to assign the columns into the SSD which will benefit the most from the characteristics of flash memory. We prove this problem is NP-complete and propose an optimal dynamic programming solution and a faster greedy heuristic solution.

A system simulator has been implemented and experiments show that the overall I/O costs can be greatly reduced using only a limited amount of flash memory. The results show the greedy heuristic solution performed similarly to the more expensive dynamic programming solution for the situations tested.

1 Introduction

Data collection is becoming increasingly popular these days. A wide range of software applications use databases to store both current and historical data. As the amount of data stored in a database grows, the queries we perform on the database become ever more complex. This, in turn, puts added stress onto the system and the hardware requirements increase. Advances in CPU speeds far surpass advances made in hard disk drive (HDD) speeds. Therefore, I/O has become a bottleneck for database systems. The relatively slow I/O performance of HDDs is partly due to the dependency of the mechanical read head, therefore, there is always a significant seek latency involved when accessing data at different physical locations.

The popularity of flash memory has grown steadily in recent years with both USB flash drives and flash-based solid state drives (SSD) becoming a common addition to the average computer system. In the remainder of this paper we will use the term SSD and flash memory interchangeably. The advantage of flash memory is that it is a non-mechanical technology, therefore the seek latency involved in accessing data is dramatically lower than HDD. Large capacity flash memory is, however, still significantly more expensive than HDD, therefore storing large scale database systems purely

on flash memory can be extremely expensive. The aim of this research is to achieve enhanced I/O performance with a database system which utilizes both flash memory and HDD storage. We do this by storing only a fraction of the database on flash memory. To obtain the greatest performance possible, we will select the data whose access pattern would benefit the most from being on flash memory. Splitting a database across HDD storage and flash memory is, however, a non-trivial task as the hardware characteristics of flash memory are fundamentally different from those of the standard HDD. Any flash memory storage software should be implemented with these differences in mind.

Most widespread database systems have been designed to use HDD storage. These systems, therefore, utilize the HDD's capability to overwrite data in-place when updating and to a great extent avoid random read access patterns due to the significant seek latency involved. On flash memory this makes little sense and is far from optimal as in-place writes are very expensive for flash memory and random read access is very cheap for flash memory.

The above has motivated a significant body of research on flash memory-based database systems [22, 16, 19, 31, 21, 28, 17, 14]. However, the previous work does not consider the use of a hybrid setup where flash memory and HDD are used together. To our knowledge, the paper by Koltsidas et al.[18] is the only existing work on a hybrid database for flash memory. However, their work disregards the presence of sequential data access and also assumes the entire database can fit on flash memory. The cost of sequential access is much lower than random access for HDD and therefore should play an important role in determining which data is placed on the HDD. In contrast to the work by Koltsidas et. al.[18], we account for sequential data access as our database is designed for situations where only a portion of the database fits on flash memory and we perform our partitioning in the context of a column instead of row store.

In developing a HDD/flash memory hybrid system, we hope to capitalize on the advantages of both while avoiding their disadvantages. The proposed system uses a fully vertically-partitioned storage structure. Each column of every table is stored separately and as such, can be placed in either of the two drives. We seek to find the optimal partitioning of the database in terms of columns across the two drives. Intuitively, a vertically-partitioned database can reduce I/O by only loading the columns that are needed by the query. However, in some cases, a vertically-partitioned database may lead to more disk seeks. For example, 3 attributes are required from 4 tuples of a table. In a non-vertically partitioned database, this will incur at most 4 seeks since all the attributes of each tuple are stored together. However, in a vertically-partitioned database, this may result in 12 seeks since each attribute of each tuple may be stored on a different page. In such a situation, it would be advantageous to store the columns involved on the flash memory since flash memory incurs very small seek latency. We thus take this factor into consideration when assigning columns to drives.

We theoretically prove the drive assignment problem for columns is a NP-complete problem. We then develop a dynamic programming solution which optimally partitions the database over HDD and flash memory. We also propose a greedy solution to the partitioning problem. The partitioning is done while the system is offline.

We conduct a thorough experimental evaluation of the performance of the proposed algorithms. The results of the experiments show that the overall I/O costs can be greatly reduced using only a limited amount of flash memory. In addition, the experiments reveal the greedy solution offers similar I/O cost reduction compared to the dynamic programming solution for the situations tested.

2 Background

This section provides a detailed description of the background information required to understand the problem being studied in this paper. The first section will describe the characteristics of flash memory. The second section will present some of the issues relating to software designed for this technology.

2.1 Flash Memory

Flash memory is non-volatile memory. In contrast to random access memory (RAM), once data is written to flash memory it can not be updated simply by over-writing the same memory location. A characteristic of flash memory is that a write operation must be preceded by an erase operation. The erase operation is expensive in terms of both speed and energy consumption.

These erase operations differ from both read and write operations in more ways than one. Read and write operations can be performed on small sections of flash memory (typically at byte level on NOR memory and in pages of 512-2,048 bytes on NAND). Erase operations, however, can only be performed on blocks of around 32-64 pages, called erase-units (or blocks). Furthermore, erase operations can only be performed a limited number of times on each block before the block becomes unstable and unusable; this is commonly called block-wear.

Flash memory does, however, provide many advantages including very high read speeds, low energy consumption, amazing durability and complete silence with no moving parts.

2.1.1 NOR and NAND flash memory

There are two common types of flash memory. NOR flash memory allows for simple random access at byte granularity through dedicated address and datalines. This has been shown to be particularly suitable for storing programs in execute-in-place memory (XIP), meaning that programs stored in NOR flash can be executed directly, without the need to copy them into RAM. NAND flash memory, on the other hand, does not have dedicated address lines and thus only provides access at page granularity.

The lack of dedicated address lines has also allowed manufacturers to produce cheap, high-density, high-capacity chips. As a result, NAND flash memory is growing in popularity and is seen as an attractive alternative to standard magnetic hard disk storage.

This paper will be focused on NAND technology and unless otherwise stated, the term 'flash' will herein implicitly refer to NAND flash technology.

2.1.2 NAND Operational Characteristics

There are basic differences between NAND flash memory and magnetic hard disk drives.

Asymmetric read versus write cost Traditional magnetic hard disk drives have an almost identical cost for both read and write operations. For flash memory, the write costs are typically higher than read costs. Table 1 [27] gives a comparison of the read and write speeds of two typical HDDs and two SSDs. Due to the higher cost of write operations, these should be avoided whenever possible.

Expensive in-place updates With hard disk drives, data can easily be overwritten by new data. On flash memory, such in-place updates would require entire blocks to be erased, after which both updated data and unchanged data residing in the same blocks would have to be re-written to the flash memory. Because of this, updates are not made in-place. Rather, updated pages are written to a new location and the old pages are marked as invalid. Figure 1 illustrates this process.

Limit number of erases Each block in flash memory has a limited lifespan in terms of the number of write/erase cycles before it becomes unstable and unusable. Newer flash drives have increasingly higher life spans, enduring up to 5 million write/erase cycles. Nevertheless, each erase operation slowly wears out the block. To maintain the functionality of the blocks for as long as possible, page updates should be spread out evenly across all available blocks. This is commonly called wear-leveling.

Device	Random Access Time (ms)	Sequential Read Rate (MB/s)	Sequential Write Rate (MB/s)
WD VelociRaptor HDD	8	74	74
WD Caviar HDD	12.2	85	85
Mtron Flash SSD	0.1	94.6	74.2
SanDisk SSD	0.11	68.1	47.3

Table 1: Table comparing speed of typical HDD and SSD

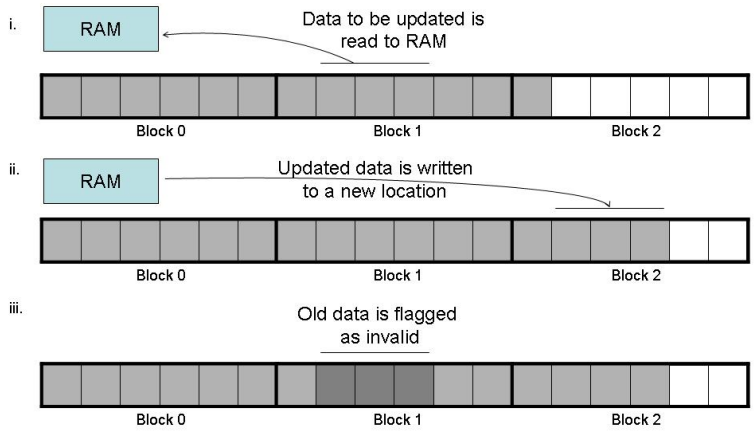

Figure 1: Out-of-place update

2.1.3 Flash-based solid state drives

A solid state drive (SSD) is a non-mechanical storage device that emulates the standard HDD in size, shape and interface. This makes it easy for the drive to be integrated into any system which would otherwise require a HDD. There have been considerable advances in flash based SSD technology over the last few years, with faster and more expensive drives showing great improvements in both speed and capacity. Intel, for example, has a flash-based SSD delivering 250 MB/s read and BiTMICRO Networks has products with 1.6TB capacity [30]. The current high cost means the top of the line drives are beyond the reach of most users. However, the lower range drives are becoming more and more affordable and these can still outperform common HDDs in many situations.

2.2 Flash Software

2.2.1 Garbage Collection

With many out-of-place updates, the number of invalid pages increases rapidly while the amount of available free space decreases. To address this, garbage collection is used to reclaim space occupied by invalid pages. Almost all software that works on top of flash memory implements garbage collection to some extent.

An invalid page can only be reclaimed by erasing the entire block in which it resides. As there might be valid pages

also residing in the same block, there are two options when a block has been selected for erasure. The valid pages can either be read to memory, the block can be erased and the valid pages written back to the block. Alternatively, the valid pages can be transferred to another block with free space and then the block erasure can proceed. With the prior option, a system failure after the block has been erased but before the valid pages have been written back to the block would cause data to be lost. To ensure data integrity, the latter option is therefore preferred. As a result, garbage collection must be performed while there are still a number of free blocks available.

A block containing only invalid pages can be erased without the additional copying overhead. On the other hand, blocks containing only valid pages could be left out of the garbage collection procedure. The issue of wear leveling does however arise when updates are not spread evenly across the blocks of flash memory. For example, some blocks never have their pages updated, therefore they do not contain any invalid pages. With this in mind, the garbage collector is occasionally given the added task of transferring valid pages to a new block from time to time [3, 32] - even though this is not strictly garbage collection.

2.2.2 Flash Translation Layer

The most common file systems were written to run on a HDD. These file systems, therefore, have features to make use of HDD's sequential access architecture while also addressing HDD's limitations such as seek latency. As the characteristics of flash memory are so fundamentally different, these file systems do not work optimally on the media. The Flash Translation Layer (FTL) [4, 5] is a technology designed to solve this problem.

FTL is a software layer between the file system and the flash memory hardware. FTL provides the file system with an interface to the flash memory that is identical to that of the common HDD. As a result, common file systems such as FAT, ATA or SATA can be used with flash memory. FTL performs updates out-of-place, garbage collection, wear leveling and error detection transparent to the file system. This is done by mapping virtual addresses to physical addresses. Although FTL addresses the need for wear leveling on flash memory, it does not address the issue of the skewed read/write speed ratio.

2.2.3 Flash-based File Systems

One method of achieving a flash file system is to map a common file system on top of FTL, as mentioned in the previous section. An alternative method is to build a custom file system directly on top of the flash hardware. These include JFFS [3] (Journaling Flash File System), JFFS2 [32], and YAFFS [25] (Yet Another Flash File System) which are log structured file systems. With these log structured file systems, all new and updated data are appended to a log in a purely sequential fashion. This approach was originally proposed for HDD on which write speeds were greatly improved while read performance suffered from added seek latency for reading data that is scattered across the disk. This technique is, however, ideal for flash memory as updates have to be performed out-of-place and random access reads do not suffer from high seek latency.

2.2.4 Flash Database Systems

As with traditional file systems, traditional database systems have been designed to maintain data on HDD storage. One simple example of converting these databases to run on flash memory is to use them on top of the FTL. However, this is not optimal because these databases are designed for hardware that can perform in-place updates cheaply with a preference for sequential access compared to random access. They tend to update small fractions of many pages rather than concentrating large updates to just a few pages. Take, for example, the product database of a store. After every processed purchase, the *in stock count* of the purchased items is updated. This will result in many pages being updated. Having such a database system on top of a flash translation layer will cause excessive garbage collection which, in turn, will severely affect the performance of the system. Therefore, traditional databases need

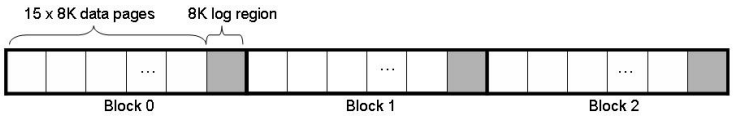

Figure 2: IPL on flash memory with 16 x 8K data pages per erase unit

to be redesigned in order to be flash memory friendly. We will survey the existing literature on flash-based database systems in Section 3.

3 Related Work

This section will present a review of some of the existing work on flash memory databases and vertical partitioning in databases to increase I/O performance.

3.1 Flash-based database systems

Much research has been conducted on different issues relating to database systems running on flash storage.

LGeDBMS, developed by Kim et al. [16], introduced the log structure to the flash-based database systems. This was done following the lead of flash file system developers who had already discovered that this structure had its advantages when used on flash memory [3, 32, 25]. When updates occur on a log structured file system, data itself is not changed, rather updates are appended to a log. When reading data from the database, the updated data is generated by merging its original form with any related updates found in the log. The major concern with the log structure (originally intended to run on magnetic HDD) is the cost of the added read operations needed to read the original data, followed by scanning the log for related updates. With flash memory however, this seems like an ideal way to avoid expensive write operations. In doing so, sacrificing a few extra read operations which are relatively inexpensive may be acceptable. One problem with this is that as the log grows very large, the cost of the number of reads needed to scan the log overshadows the benefits of decreased write costs.

One alternative flash-based database system is the In-Page Logging (IPL) system [19]. They take an approach closely related to the log structured file systems mentioned earlier. However, instead of keeping the log purely linear, separate logs are kept in a log region at the end of each block/erase unit (see Figure 2 for an illustration). As flash memory does not have seek latency on random write operations, there is no need to keep the log entries purely sequential. This approach effectively increases read speeds, as searching sequentially through the entire log for related updates is no longer necessary.

A pre-determined number of pages at the rear of each block/erase unit is reserved for the log of the remaining pages in the block. As a result, the log does not grow out of control and each read operation needs only read the requested page and the small log at the end of the same block. This does, on the other hand, introduce two issues: firstly, at the block where the data is written often, the log might fill up quite easily; secondly, fixed size logs on every block will be a waste of storage space on blocks that contain data that is never updated.

In the event of the log filling up, the specialized garbage collector is triggered. The garbage collector reads the entire block to memory, merges the original data with its associated update entries in the log and consequently writes the now up-to-date pages to an empty block. The long term cost of this operation on different workloads is not presented. It is not hard to see, however, that only workloads where updates are evenly spread over the whole database will take full advantage of the IPL approach. Workloads with high temporal locality or "hot data" which gets updated a lot while other data does not get updated at all will suffer greatly from frequent merge operations needed in the garbage collection as well as wasting a lot of storage space for logs on blocks that do not get updated.

It is clear that no single storage structure will work optimally for all storage hardware or all workloads. Addressing this leaves us two options when developing a database system. The system must either have self-tuning capabilities or the system should be set up to target a specific type of hardware and/or workload.

Nath and Kansal [22] identify the fact that none of the existing database systems will work optimally for all storage hardware or all database workloads. With this in mind, they propose FlashDB; a system which uses a variation of a B+-tree that integrates both the traditional storage structure (disk mode) and a log based storage structure (log mode). FlashDB also features a self-tuning mechanism that dynamically decides which of the storage structures (or modes) is optimal for each node. In this way, the data in nodes that are rarely queried but frequently updated can be stored in log mode while nodes that are often queried are saved in disk mode.

Li et. al. [21] proposed the FD-tree which is write optimized for the flash disk. The design consists of a small B+-tree that fits in RAM and multiple levels of sorted runs on flash disk. Index lookups may incur more disk reads but this is offset by converting random writes to sequential ones through merges into the lower runs. Results show the index outperform B+-tree variants on both update and search intensive workloads.

Shah et. al. [28] propose the use of the column-based storage layout for flash databases. Their idea is to convert traditional sequential I/O algorithms to ones that use a mixture of sequential and random I/O to process less data in less time. To this end, they propose the RAndom Read Efficient Join (RARE-join) algorithm which is a hash-based join algorithm that uses random reads to retrieve less data than hybrid hash joins. However, their work only applies to binary joins and they did not implement their solution.

Tsirogiannis et al. [31] improved on [28] by proposing FlashJoin which performs pipelined multi-way joins. They also use a column-based storage layout. The implementation described in the paper uses a hash-based join kernel that employs the hybrid-hash join algorithm. However, their system allows any other join algorithm to be used instead. They implemented their algorithm into PostgreSQL. The results show FlashJoin significantly reduces the amount of memory and I/O needed for each join in the query. However, their work is solely focused on joins and therefore does not consider database updates. Like FlashJoin, we also use a column-based storage layout. However, our work contrasts with theirs in a number of important aspects. First, we consider updates and therefore attempt to reduce write I/O. Second, we consider a hybrid setup in which the SSD may not fit all the columns. Third, we do not focus on join algorithms.

He and Veeraraghavan [14] proposed a partitioning and caching algorithm for flash memory databases. They partition database tuples into subtuples which are then cached at this fine grain. This reduces write I/O by allowing the system to selectively assemble updated subtuples into an empty page and then flushing the entire page from RAM when a page eviction is required from the buffer. The results showed the proposed system outperformed IPL[19] by up to 40 fold. Our work differs from theirs by considering a hybrid setup in which the flash memory may not be large enough to fit all the data. We focus on determining if columns should be stored on SSD or HDD, whereas their work is focused on caching data at the fine sub-tuple grain.

Sang-Won Lee et. al.[20] examine the applicability and potential impact of flash memory as a secondary storage device for different components of database systems. They performed an extensive set of experiments and found flash memory can achieve an order of magnitude performance improvement for transaction processing compared to hard disk drives for such activities as transaction logging, accessing data from rollback segments and temporary table spaces. The aim of their work is fundamentally different from us. They measure the effect of replacing hard disk drives with flash memory for different functions of databases and do not propose any new algorithms, in contrast we propose partitioning algorithms to speed up databases using both flash memory and hard disks together.

Kim et. al.[17] recently proposed the page-differential logging approach to efficiently handle updates for flash memory based databases. It improves over both traditional page-based and log-based approaches by only writing the difference between the original page in flash memory and the up-to-date page in main memory and only computing the page differential when the page needs to be flushed. The proposed method allows existing disk-based DBMSs to be reused as flash-based DBMSs just by modifying the flash memory driver. The results show the proposed system significantly outperforms existing page-based and log-based solutions. Our work contrasts from this in that we focus

on the speeding up databases using both flash memory and hard disk drives together.

Koltsidas and Viglas [18] proposed a dynamic self-tuning storage structure intended for systems with both HDD and SSD available. They propose a system where each page is allocated space on both drives, but at any given time a node resides only on one of the two drives. To determine which device is better suited for a given page at any particular time, a family of algorithms is presented. Their work has three notable drawbacks. First, it assumes the SSD drive can fit the entire database. Second, it does not distinguish between sequential and random data access. The cost of sequential access is much lower than random access on hard disk drives, thus treating all access as random access will result in more data being stored in the SSD rather than the hard disk. Third, it does not consider vertical partitioning which has been shown to offer significant performance improvements in traditional hard disk databases. In contrast, we propose a system that works with a limited SSD size, incorporates sequential data access costs and uses vertical partitioning to achieve improved system performance.

3.2 Vertical partitioning

The idea of vertical partitioning was proposed in the early days of databases as a means to increase I/O performance. Much work was done in the seventies and eighties exploring different approaches to find the optimal partitions. Hoffer et. al. [15], Navathe et. al. [23, 24] and Cornell and Yu [9] all propose vertical partitioning algorithms that use transactional affinity as the dominant criteria. The first three of the aforementioned do this by gathering statistics on a representative workload to discover which attributes are commonly referenced together and the partitions are derived from this information. Cornell and Yu, however, approached it as an integer programming optimization problem, iteratively calculating the total workload cost of possible binary partitions.

AutoPart [26] is a more recent algorithm for database partitioning. Similar to the previously mentioned approaches, it is also based on a representative workload. In some ways, its approach is the opposite to that of Cornell and Yu [9]. Autopart partitions the tables into atomic one attribute fragments, then iteratively examines the different fragment combinations to find the combinations that will reduce the total cost of the representative workload. Experimental evaluation of this algorithm shows improvements on resulting partitions of up to an order of magnitude. The algorithm can also optionally create overlapping partitions which is shown to further increase the retrieval speeds at the cost of additional storage requirements.

In 1985, Copeland and Khoshafian [8] made a detailed analysis of the attributes of a column-oriented storage model in which a table is stored as a set of files containing (surrogate, attribute value) pairs. They advocate the model, pointing out numerous benefits and challenging the general consensus that the row store model is far superior. Workloads, where queries commonly contain only a fraction of a table's attributes, can benefit greatly from the use of a column-oriented model. The higher cost of accessing many different attributes is mainly due to hard disks' seek latency involved in accessing each column file separately. It is also noted that this latency could be avoided in the future as RAM memory prices are on a steady decline. In the event that the workload is write intensive however, the row oriented structure is highly preferable. Delete and insert operations are especially expensive on the column-oriented structure as all column files will have to be altered separately. Insertion cost can, however, be amortized with the use of an in-memory buffer.

As part of a recent renewed interest in the column-oriented storage model [6, 29, 2, 1], Harizopoulos et. al. [13] re-analyze this model as a read-optimized storage structure. Based on analysis of the different storage structures alone, column-oriented model makes better use of disk bandwidth in most circumstances but the row-oriented model is still preferable for workloads with non-selective queries and/or narrow tuples. However, further optimizations proposed to the column-oriented model (such as the ability to operate directly on compressed data [2] or vectorized processing [7]) give the column-oriented model added appeal which is not included in the analysis.

The recent column-oriented storage model mentioned above is C-Store [29]. C-Store is a read-optimized database system that comes with a read-optimized column-oriented store (RS) and a separate write-optimized row-oriented store (WS) to which all updates are initially added. A tuple mover then moves updated tuples from WS to RS in bulk.

C-Store can also store multiple copies of each column in separate sort orders to achieve higher retrieval performance. With clever usage of compression and dense-packing of data, C-Store is able to provide a redundant schema in less space than both a commercial row store product and a commercial column store product.

4 Problem Definition

Our goal is to minimize the overall I/O cost of running a given database workload on a system that uses two types of storage devices (a magnetic disk HDD and a flash memory SSD). We will assume that the HDD storage capacity is sufficient to hold the entire database while the available SSD storage capacity may be limited. This is a reasonable assumption as most modern day computers are equipped with a high capacity HDD while the capacity of the average SSD is still relatively small. The problem is thus: "using a limited amount of flash memory and a HDD that can fit the entire database, how best to reduce the overall I/O cost for processing a given workload on a given database"?

Database partitioning is already an extensive area of research. The hardware used to store each of the partitions has, however, not been the primary issue. For the purpose of partitioning a database across distinct storage hardware, the problem will differ to some extent depending on what partitioning method is used. We have made the decision for the purpose of this dissertation to explore the use of a vertically-partitioned database. Extending the research to explore other partitioning options could be done at a later stage.

For the database, we will be using a vertically-partitioned storage model fully decomposed into individual column files. The use of this model has shown promising results in increasing I/O performance for HDD-only database systems. Also, its main disadvantage, the added seek latency involved in data retrieval on HDD, will mainly affect the data on the HDD as the seek latency of flash memory is much lower.

Keeping true to the column-oriented storage model, each column will be saved in a single file on one drive or the other where possible. However, because the size of a database column is not constant, a column placed on the limited capacity SSD may need to expand when there is no more room left. In this event, the column will be allocated space on the HDD for consequent new records.

With this in mind, we will now give a more formal definition of the problem.
Given the following:

- the set of all columns of the database, C.
- a sequence of requests $R =< r_1, r_2, r_3, ..., r_m >$, where each request r_i contains a request type [Read / Scan/ Update / Insert / Delete] and the id(s) of the affected table/column(s) and tuple(s),

$$r_i =< requesttype, tableid, columnid, tupleid >,$$

- a buffer replacement algorithm
- HDD and SSD specifications, i.e. the cost of each type of I/O operation.
- the maximum size of the RAM buffer

our goal is to find an algorithm that partitions the columns, C, into an HDD partition and an SSD partition such that:

- the I/O cost of processing the sequence of requests, R, is minimized.
- the total size of the columns assigned to the SSD is less than the capacity of the SSD.

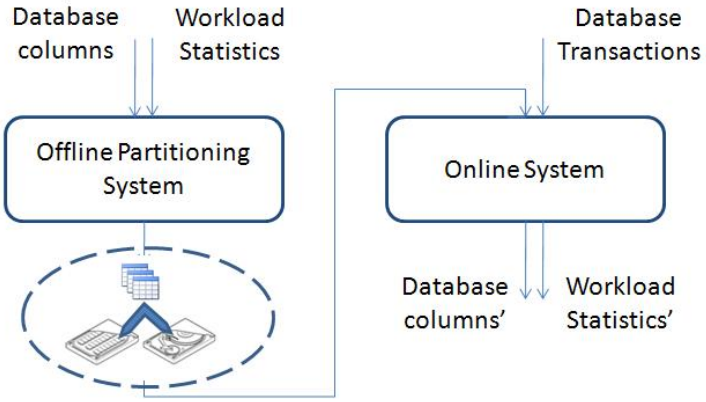

Figure 3: System Overview

5 Methodology

5.1 System Overview

Our solution to the problem defined in the previous section can be viewed as two disjoint subsystems. The components of the system and their interaction are illustrated in Figure 3. The two subsystems, the Offline Partitioning System and the Online System, will be discussed in detail in the following sections.

In our solution the offline system is responsible for deciding which device (flash or HDD) each column of the database should be placed in. Given partitions determined offline the online system runs the queries without migrating the data. The online system also collects the workload statistics needed by the offline system. We reserve dynamic partitioning (migrating columns between SSD and HDD online) as an area of future work. Therefore the majority of the methodology section is devoted to explaining how the offline system determines which storage device each column is placed in.

The Offline Partitioning System takes as input the database to be partitioned and workload statistics. The algorithm then uses this information to allocate each column to one of the available storage options. The output of this system is the set of column to drive mappings which would result in the optimal I/O performance when processing a database workload with workload statistics similar to the given input. The system will then place the columns in the assigned drives according to the drive mapping determined.

How often the offline partitioning algorithm is used would vary greatly depending on factors such as changes in the workload, changes in the data tables, and the performance requirements in the specific environment.

5.2 Offline Partitioning System

The task of the offline system is to solve the partitioning problem defined in Section 4. First, we define the workload statistics input for this subsystem. Second, we prove the problem is NP-complete and then we present two algorithms for solving the problem. One solution is a dynamic programming algorithm that is suitable for most situations and can guarantee finding the optimal solution. The second is a greedy heuristic algorithm that is faster and more suitable if the database is especially large.

5.2.1 Workload Statistics

Some knowledge of the database workload is needed in order to partition the database across the two drives. To obtain accurate information on which requests will require disk I/O, it is not sufficient to consider the database transactions alone. The buffering technique used by the system must also be considered, as not all read requests will incur a read from disk. Furthermore, the offline partitioning system will consider the placement of each individual column and as such, the information on these requests is required at column grain. In other words, the required workload statistics should include the number of each different type of I/O requests incurred on each column.

Unfortunately, the fact that a database is a dynamic and ever changing entity complicates things. For example, a column stored on the SSD may grow over time. As the SSD capacity is limited, this needs to be taken into consideration. The proposed system will try not to split a column across two drives if it can be avoided. The expansion of columns is dealt with by expanding the column file on the drive it was stored on when needed. However, if the column is stored on the SSD and the SSD has reached full capacity, then subsequent column expansions are stored on the HDD. This means a column which grows over time *may* be stored across both drives. In particular, *added data* (data that has been added since the database was partitioned) may be stored on the HDD even if the column was assigned to the SSD by the offline system. Since the added pages of a column may be on the HDD, whereas the original pages of the column may be on the SSD, the cost of reading and writing to the added pages may be different from the original pages. To address this, the system will require workload statistics on the number of each type of I/O request incurred on the original data and added data separately, for each column. The required workload statistics are listed here for clarity. For each column u of the database, we will keep the following statistics:

- number of original pages read from index lookup ($N_{oir}(u)$)
- number of pages read from index lookup for added pages ($N_{air}(u)$)
- number of column scans before any addition on the column ($N_{bcs}(u)$)
- number of column scans after the first addition to the column ($N_{acs}(u)$)
- number of original pages read from column scan ($N_{osr}(u)$)
- number of added pages read from column scan ($N_{asr}(u)$)
- number of writes to original pages ($N_{ow}(u)$)
- number of writes to added pages ($N_{aw}(u)$)

Note our workload separates statistics of index lookups from column scans since index lookups are random accesses whereas column scans are sequential accesses. The cost of random accesses differs significantly from sequential accesses for the HDD as shown in Table 1.

5.2.2 NP-complete Proof

In this section, we prove that our partitioning problem is NP-complete by mapping the knapsack problem to the partitioning problem. We first describe the knapsack problem and then show how it can be mapped to our problem. Once this is done we can use solutions previously developed for the knapsack problem to solve our problem. In Section 5.2.3 we describe a dynamic programming solution which solves our NP-complete problem in pseudo-polynomial time. Section 5.2.4 describes a greedy solution which produces less optimal results but is faster.

Proving a problem Π is NP-complete can be done by showing the existence of a (polynomial) transform from a known NP-complete problem Π' to the problem at hand $T : \Pi' \Rightarrow \Pi$.

The knapsack problem is a proven NP-complete problem. Given a set of items which individually have a value and a physical size, the knapsack problem lies in finding the most valuable subset of items that will fit into a knapsack of

a specified size. Formally, this is defined as follows [11]:

INSTANCE: Finite set U, for each $u \in U$ a size $s(u) \in Z^+$ and a value $v(u) \in Z^+$, and positive integers B and K.

QUESTION: Is there a subset $U' \subseteq U$ such that $\sum_{u \in U'} s(u) \leq B$ and such that $\sum_{u \in U'} v(u) \geq K$?

The partitioning problem lies in allocating each column of a database to one of two drives. If we instead consider it as allocating a subset of the columns to the SSD, then intuitively, the partitioning problem would conform to the definition of the knapsack problem. This would, in fact, be the same problem once we make the assumption that the HDD capacity is sufficient to hold the entire database and thus only the SSD capacity is constrained.

We define the I/O reduction resulting from placing column u into the SSD called $I/OR(u)$. This definition is used by our NP-complete proof. The definition of $I/OR(u)$ is as follows:

$$I/OR(u) = cost_{SSD_{column}}(u) - cost_{HDD_{column}}(u) \tag{1}$$

where $cost_{SSD_{column}}(u)$ is the I/O cost that the workload on column u would incur if the column was placed on the SSD. It is defined as follows:

$$\begin{aligned} cost_{SSD_{column}}(u) \quad = \quad & (N_{oir}(u) + N_{osr}(u)) \times (cost_{SSD_{read}} + cost_{SSD_{rand}}) + (N_{air}(u) + N_{asr}(u)) \times cost_{HDD_{read}} \\ & + (N_{air}(u) + N_{acs}(u)) \times cost_{HDD_{seek}} + N_{ow}(u) \times (cost_{SSD_{write}} + cost_{SSD_{rand}}) + \\ & + N_{aw}(u) \times (cost_{HDD_{seek}} + cost_{HDD_{write}}) \end{aligned} \tag{2}$$

The workload variables used in the above equation are defined in Section 5.2.1. $cost_{SSD_{read}}$ is the cost of a page read on the SSD, $cost_{SSD_{write}}$ is the cost of a page write on the SSD and $cost_{SSD_{rand}}$ is the cost of random access on the SSD. $cost_{HDD_{seek}}$, $cost_{HDD_{read}}$ and $cost_{HDD_{write}}$ are the cost of a seek, page read and page write on the HDD, respectively. Note the above equation is an overestimate of the actual cost since we assume all added data are written into the HDD. However, when there is leftover space on the SSD, some added pages may be written onto the SSD, in which case, this assumption would not be correct. However, typically the partitioning would maximize the use of the SSD and hence the amount of space left on the SSD would be very small.

$cost_{HDD_{column}}(u)$ is the I/O cost that the workload on column u would incur if the column was placed on the HDD. It is defined as follows:

$$\begin{aligned} cost_{HDD_{column}}(u) \quad = \quad & (N_{oir}(u) + N_{air}(u)) \times (cost_{HDD_{seek}} + cost_{HDD_{read}}) + (N_{osr}(u) + N_{asr}(u)) \times cost_{HDD_{read}} + \\ & (N_{bcs}(u) + N_{acs}(u)) \times cost_{HDD_{seek}} + (N_{ow}(u) + N_{aw}(u)) \times (cost_{HDD_{seek}} + cost_{HDD_{write}}) \end{aligned} \tag{3}$$

$cost_{HDD_{read}}$ is the cost of reading a page from the HDD after seeking to the location of the page; the other variables have the same definition as that of Equation 2. Note for both of the above equations, we assume writes are random writes since these writes are resultant from page flushes when pages are evicted from the buffer.

Theorem 1 : *The partitioning problem is NP-complete.*

PROOF:

- *Let* Π *be the partitioning problem.*
- *Let* Π' *be the knapsack problem.*
- *The mapping* $T : \Pi' \Rightarrow \Pi$ *is straightforward and can clearly be performed in polynomial time.* U *is the set of all columns in the database,* U' *is the set of columns which when placed on the SSD will result in the greatest overall reduction in I/O costs,* $s(u)$ *is the size of column* u *and* $v(u)$ *is computed using Equation 1.*

□

As we are dealing with an NP-complete problem, it may be difficult to find an optimal solution. However, dynamic programming solutions exist which make it possible to solve a knapsack problem in pseudo-polynomial time. Furthermore, the number of columns in a database tends to be limited in most cases. Therefore, a dynamic programming algorithm would, in such cases, find the optimal solution within an acceptable time frame. Nonetheless, we propose two different algorithms: a dynamic programming algorithm and a greedy heuristic approach. The latter would be used in the event that finding a fast solution is of higher importance than finding the optimal solution. Both approaches are described in the subsequent sections.

5.2.3 Dynamic Programming Partitioning

In this section we describe how our partitioning problem can be solved using dynamic programming. We first give an intuitive description of the solution. The idea is to divide and conquer. First, break the problem into the smallest possible sub-problem and then solve that small problem. Next, use the solutions to the smallest problem to construct the answer for the next largest problem. This process is repeated until the entire problem is solved. In this context our problem of determining the best location for the database columns can be solved as follows. First limit the SSD capacity to size 1 page and then place into that very small SSD the set of columns which will result in lowest overall I/O costs for our given workload. In most cases the solution will be zero columns since typically all columns will be greater than 1 page in size. Next do the same but with a SSD capacity of 2 pages. This is repeatedly done until we reach the actual SSD capacity of the device. The reason we progressively solve the problem from smallest to largest SSD capacity is that when solving the larger SSD capacity problems we can use the solutions for smaller SSD capacities to help us. For example suppose we want to find the optimal solution that includes column C_i when the SSD capacity is k. The solution must be the optimal set of columns for the smaller problem of SSD with capacity $k - size(Col_i)$ plus column Col_i. Note the solution for SSD capacity $k - size(Col_i)$ can not include Col_i. This is how we iteratively build the solution to the problem. The remainder of the section provides a more detailed description of the algorithm.

The algorithm we use to solve the knapsack problem is shown in Algorithm 1 adapted from [12]. The input to this algorithm is the set of all columns C and the SSD specifications of the SSD and the result is the columns to put on HDD C_{HDD} and a set of columns to put on the SSD C_{SSD} which will give the optimal performance. The algorithm will first construct a table of the optimal solution to all possible sub-problems of this partitioning problem, that is, a table with the optimal benefit which can be achieved by using any smaller SSD capacity $0 \leq d \leq Size(SSD)$ considering only the first j columns in C where $0 \leq j \leq n$.

To make the following description as short and concise as possible, we will first define a couple of keywords. Let $SP(j, d)$ be the sub-problem where only the first j elements are considered and the SSD capacity is d. Additionally, $Opt(j, d)$ is the benefit of the optimal solution to $SP(j, d)$ and $Included(j, d) \rightarrow true, false$ indicates whether or not column c_j is included in the optimal solution to $SP(j, d)$. Now the algorithm starts by indicating the obvious, the optimal benefit when considering 0 columns with any capacity d is 0 (Lines 1 to 3), next consider one additional column c_j at a time. Obviously, if the capacity $d < Size(c_j)$, then c_j will not be part of the optimal solution to $SP(j, d)$ and $Opt(j, d)$ is unchanged from $Opt(j - 1, d)$ (Lines 5-7). For each $d \geq Size(c_j)$, we consider whether or not c_j should be put on SSD or left on the HDD. c_j will be included in the solution to $SP(j, d)$ if and only if the optimal solution of $SP(j - 1, d)$ is smaller than $SP(j, d - Size(c_j)) + Value(c_j)$ (Lines 4 - 17). Note the value for each column refers to the reduction in I/O costs resulting from placing the column on the SSD instead of HDD as specified by Equation 1. Now the table of all the optimal benefit values for each sub-problem is complete including the optimal benefit value for the complete problem. However, only the optimal benefit is known, not the set of columns included. Note $Included(j, d)$ only shows whether c_j was a part of the solution to $SP(j, d)$, not the optimal solution for the complete problem. To assemble the solution set, the algorithm will go through the table backwards. At each step, if $Opt(j, d) = Opt(j - 1, d)$, then c_j is clearly not part of the solution and c_j is added to C_{HDD} and then $SP(j - 1, d)$ is revisited. However, if $Opt(j - 1, d - Size(c_j)) + Value(c_j) > Opt(j - 1, d)$ then c_j is included and added to C_{SSD}.

Algorithm 1: Dynamic Programming Partitioning Algorithm

Input : C : the set of columns, SSD
Output: C_{SSD} : the set of columns to be put on the SSD,
C_{HDD} : the set of columns to be put on the HDD

```
1  for d = 0 to Capacity(SSD) do
2  |  Opt(0, d) ⟵ 0
3  end
4  for j = 1 to |C| do
5  |  for d = 0 to Size(c_j) − 1 do
6  |  |  Opt(j, d) ⟵ Opt(j − 1, d)
7  |  end
8  |  for d = Size(c_j) to Capacity(SSD) do
9  |  |  if Opt(j − 1, d − Size(c_j)) + Value(c_j) > Opt(j − 1, d) then
10 |  |  |  Opt(j, d) ⟵ Opt(j − 1, d − Size(c_j)) + Value(c_j)
11 |  |  |  Included(j, d) ⟵ true
12 |  |  else
13 |  |  |  Opt(j, d) ⟵ Opt(j − 1, d)
14 |  |  |  Included(j, d) ⟵ false
15 |  |  end
16 |  end
17 end
18 j ⟵ |C|
19 d ⟵ Capacity(SSD)
20 while j > 0 do
21 |  if Included(j, d) then
22 |  |  Add c_j to C_SSD
23 |  |  d ⟵ d − Size(c_j)
24 |  else
25 |  |  Add c_j to C_SSD
26 |  end
27 |  j ⟵ j − 1
28 end
```

In this small scale example adapted from [12], we have an SSD capacity of 9 and 7 columns with the benefit values and sizes listed in Table 2.

c_i	c_1	c_2	c_3	c_4	c_5	c_6	c_7
$value(c_i)$	6	5	8	9	6	7	3
$size(c_i)$	2	3	6	7	5	9	4
$\frac{value(c_i)}{size(c_i)}$	3	1.67	1.33	1.29	1.2	0.78	0.75

Table 2: A small scale partitioning example

The table of the optimal benefits for each sub-problem that the dynamic programming algorithm would generate for this example is displayed in Table 3 and the values shown in bold indicate the sub-problems revisited to determine where the columns are assigned. Note that the optimal solution is storing c_1 and c_4 on the SSD with a total benefit of 15.

d f	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0
2	0	**6**	**6**	**6**	6	6	6	6
3	0	6	6	6	6	6	6	6
4	0	6	6	6	6	6	6	6
5	0	6	11	11	11	11	11	11
6	0	6	11	11	11	11	11	11
7	0	6	11	11	11	12	12	12
8	0	6	11	14	14	14	14	14
9	0	6	11	14	**15**	**15**	**15**	**15**

Table 3: Optimal solution benefits $Opt(j, d)$ computed by Algorithm 1

5.2.4 Greedy Partitioning

The partitioning problem is about getting the best value from a limited capacity SSD. Intuitively, one might be inclined to choose the columns with the highest value-to-size ratio and repeatedly select the column with the best value to size ratio from the remaining columns, the reason being that we want to squeeze as much value into a confined space as possible and the elements with the highest value-to-size ratio have the most compressed value. Note the value for each column refers to the reduction in I/O costs resulting from placing the column on the SSD instead of HDD as specified by Equation 1. The greedy partitioning algorithm (Algorithm 2) does just that. The columns are first sorted in descending order according to their value to size ratio i.e.

$$\frac{Value(c_1)}{Size(c_1)} \geq \frac{Value(c_2)}{Size(c_2)} \geq ... \geq \frac{Value(c_n)}{Size(c_n)}$$

The algorithm will then consider each column in order, storing the column on SSD if there is still room for it, otherwise the column is stored on HDD.

Now we revisit the small scale example from before. For clarity, the columns have already been sorted by descending value to size ratio. The greedy partitioning algorithm will store columns c_1, c_2, and c_7 on the SSD and thus, get a total value of 14. The optimal solution would, however, be selecting columns c_1 and c_4 and thereby obtaining a total value of 15. If obtaining this suboptimal solution is not acceptable, then dynamic programming partitioning would be

Algorithm 2: Greedy Partitioning Algorithm

Input : C : the set of columns, SSD
Output: C_{SSD} : the set of columns to be put on the SSD,
C_{HDD} : the set of columns to be put on the HDD

```
1  Sort C in descending order of value to size ratio
2  FreeSpace(SSD) ⟵ Capacity(SSD)
3  foreach c ∈ C do
4  |  if Size(c) ≤ FreeSpace(SSD) then
5  |  |  Add c to C_SSD
6  |  |  FreeSpace(SSD) ⟵ FreeSpace(SSD) − Size(c)
7  |  else
8  |  |  Add c to C_HDD
9  |  end
10 end
```

chosen. However, the dynamic programming approach has a runtime complexity of $O(nC)$ where n is the number of columns and C is the capacity of the SSD in terms of number of pages. Therefore, in the case where the SSD is very large, the greedy algorithm which has a complexity of $O(n \log n)$ where n is the number of columns is faster.

5.3 Online System

The online system processes queries with the added awareness of the dual storage device environment that we are addressing. It consists of a query processor which handles database queries that address tables, columns and records. The query processor handles these by looking up the location of the needed columns and recording it in the offline provided column-to-disk mapping and sending the appropriate I/O requests to the buffer system. The buffer system will translate the requests to the HDD at the page grain and check if the page is in memory or if it needs to be loaded from disk. If necessary, the buffer system will then send I/O requests to the hardware. Additionally, this system has a statistics collection module with which both the query processor and the buffer system interact. Both the query processor and the buffer system need to pass on information about the request to acquire the workload statistics mentioned in Section 5.2.1. The query processor will notify the statistics collection module of which tables, columns and/or records are requested and the buffer system will send notifications on which drive I/O requests are incurred (if any).

6 Experimental Setup

This section will describe the environment which was setup to test and evaluate the performance of the proposed system. First, the simulator will be described, then the chosen benchmark will be briefly described and finally, the algorithms in the experimental setup will be described.

6.1 Simulation Setup

The experiments were conducted on a system that simulated HDD and SSD drives. The specifications of the simulated drives were set according to the parameters in Table 4 [27]. Unless otherwise specified, the simulated HDD and SSD were setup to simulate the specifications of the average HDD and the average SSD. With the system simulator, it was

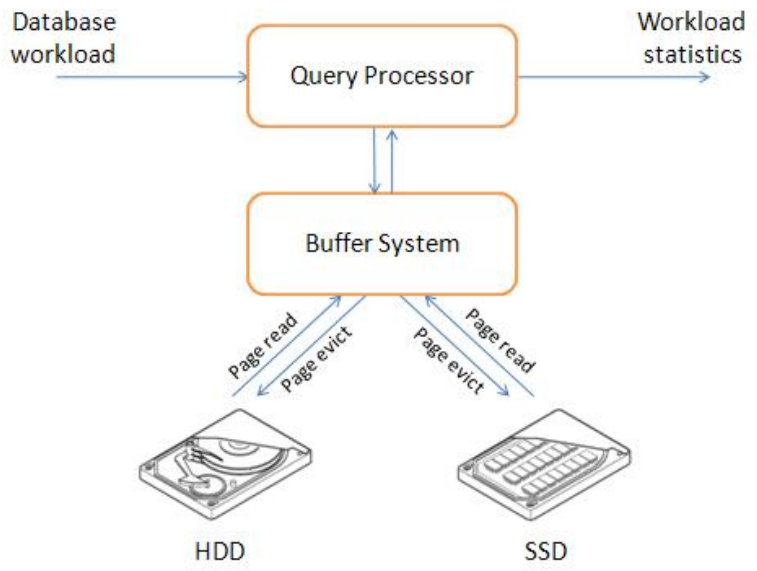

Figure 4: Online System Overview

possible to simulate the system as if it was running on a variety of hardware combinations. The parameters that were varied in the experiments described in the next section were disk capacities, RAM buffer size and disk I/O costs to match different hardware.

6.2 Benchmark

The workload that was tested on the system was generated according to the TPC-C benchmark [10]. This benchmark is a popular means of comparing the performance of database systems designed for on-line transaction processing. TPC-C has been designed to represent a typical workload of a standard OLTP database system.

We generate a workload and then use the first half of the transactions for offline partitioning and the remainder are used as the online workload.

6.3 Algorithm Setup

Each of our experiments compared the performance of four algorithms. These four algorithms are described as follows:

Dynamic Programming Partitioning. This algorithm is described in detail in Section 5.2.3. Here, each column is mapped to one of the two storage options in such a way that the overall I/O cost of running a workload which has the given statistics is minimized. The algorithm finds the optimal solution according to the cost formula.

Greedy Partitioning. This algorithm is described in detail in Section 5.2.4. This algorithm uses a greedy heuristic to try to find a solution which also minimizes the overall I/O cost similar to the dynamic partitioning algorithm. The solution is not, however, guaranteed to be optimal in this algorithm. This algorithm is included to show that the overall I/O cost can still be reduced in the situation when finding the optimal solution using the dynamic programming algorithm is too costly. Refer to Section 5.2.4 for the full details.

Parameter	Default Value	Varied
HDD Random Access Latency	12.2ms	Yes
HDD Read Rate	85.0MB/s	Yes
HDD Write Rate	85.0MB	Yes
HDD Page Size	4KB	No
SSD Random Access Latency	0.11ms	Yes
SSD Read Rate	68.1 MB/s	Yes
SSD Write Rate	47.3 MB/s	Yes
SSD Page Size	4KB	No
SSD Capacity	20% of DB	Yes
Buffer Capacity	5% of DB	Yes
Number of Transactions	2500	Yes

Table 4: Default Simulation Parameters

HDD Only. This algorithm simply allocates all columns to the HDD. This algorithm is included as a comparison showing the I/O cost that a given workload would incur if the system only had HDD storage.

SSD Only. This algorithm simply allocates all columns to the SSD, simulating an SSD-only system. This algorithm would intuitively be expected to produce better results than all the other algorithms. However, it is important to keep in mind that this requires a large SSD that fits all the data which has a high monetary cost.

6.4 Computing Platform

The software platform we used was cygwin running on windows XP. The hardware platform used was an Intel Celeron laptop with 256 KB level 2 cache and 1.25 GB RAM and a 60 GB of HDD.

7 Experimental results

To evaluate the system, four experiments were conducted to compare the different algorithms in a variety of settings. The first experiment reported the results when the SSD size was varied. The second experiment varied the buffer capacity. The third experiment compared the algorithms when different SSD and HDD devices were used. Finally, the fourth experiment measured the time used by both the dynamic programming and greedy algorithms to partition the database.

7.1 Varying SSD Capacity Experiment

The aim of this experiment is to compare the benchmark results of the system running on varying SSD capacities. Figure 5 summarizes the results of the experiment in terms of the total execution time. As shown on the graph, both the optimized dynamic programming partitioning and the greedy partitioning provide significant reductions in overall I/O costs with a small SSD. For example, with only 20% of the database stored on the SSD, the resulting overall reduction was around 70% compared to a system with HDD only. As a comparison, we can also note that the cost reduction if the entire database is stored on SSD would be more than 90%. For this, however, we would need an SSD of sufficient capacity to hold the entire database and that could be costly.

We can also note that both of our algorithms provide very similar results. The greedy approach is not guaranteed to provide optimal results in all circumstances. However, databases usually tend to contain small width columns (eg. primary indexes on columns such as student number) which are accessed most frequently and hence, have the highest cost. This makes it straightforward for the greedy algorithm to decide to place these small width columns with high costs onto the SSD. This placement decision causes the greedy algorithm to achieve near optimal performance.

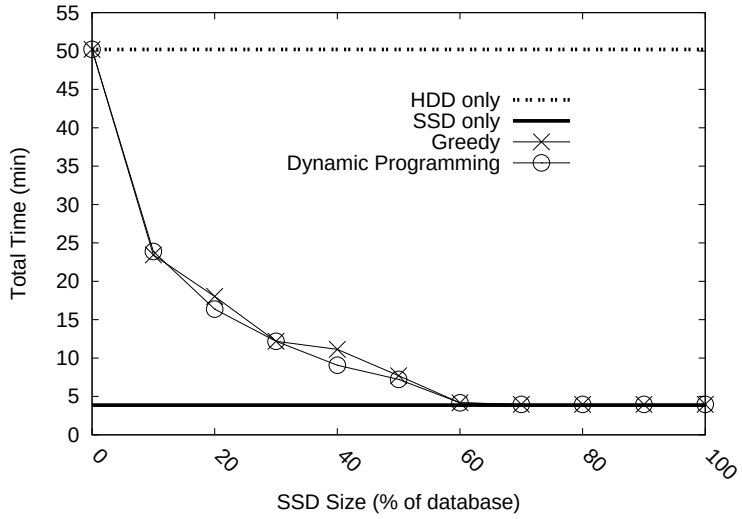

Figure 5: Total time results for the varying SSD capacity experiment

Figure 6 shows the results for the random access latencies of the HDD and SSD when SSD capacity is varied. Note the results for HDD access latency and total time (Figure 5) are very similar since HDD random access time is so high it dwarfs all other costs. When comparing the SSD random access latency, Dynamic Programming incurs slightly higher costs than Greedy since it is able to more fully utilize the SSD.

Figure 7 shows the results for the read time of the HDD and SSD when SSD capacity is varied. The results show very similar trends compared to random access latency results.

Figure 8 shows the results for the write time of HDD and SSD when SSD capacity is varied. The results show that when a small amount of SSD becomes available, Dynamic Programming and Greedy algorithms shift most of the writes onto the SSD. The reason is the columns that get the most writes are also the ones that get the most random reads and therefore those columns are moved onto the SSD to save random read costs. The random read costs overrides the writes costs when considering which drive to place the columns in because in our benchmark there are much more reads compared to writes.

7.2 Varying Buffer Capacity Experiment

The second experiment was conducted to analyze the benefit of the system when the buffer capacity is varied. The SSD capacity for this experiment was set to 20% of the database size. Figure 9 summarizes the total time results of the experiment. We note that not surprisingly, the benefit of our algorithm is minimized when the buffer is large enough to hold all the referenced pages. However, when the buffer size is small, our algorithms offer significant performance improvement over HDD only, the reason being that a larger buffer effectively minimizes the disk I/O operations needed.

It is important to note that in these experiments, our algorithms were only allowed to place 20% of the database

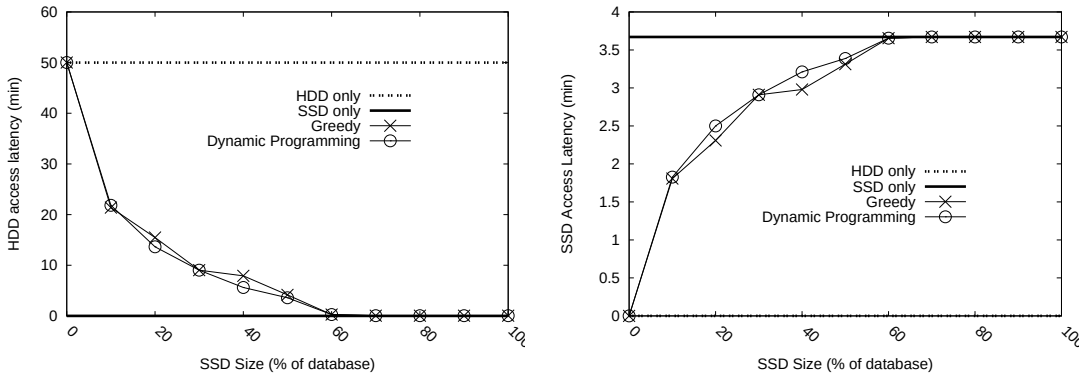

Figure 6: HDD and SSD random access latency results for the varying SSD capacity experiment

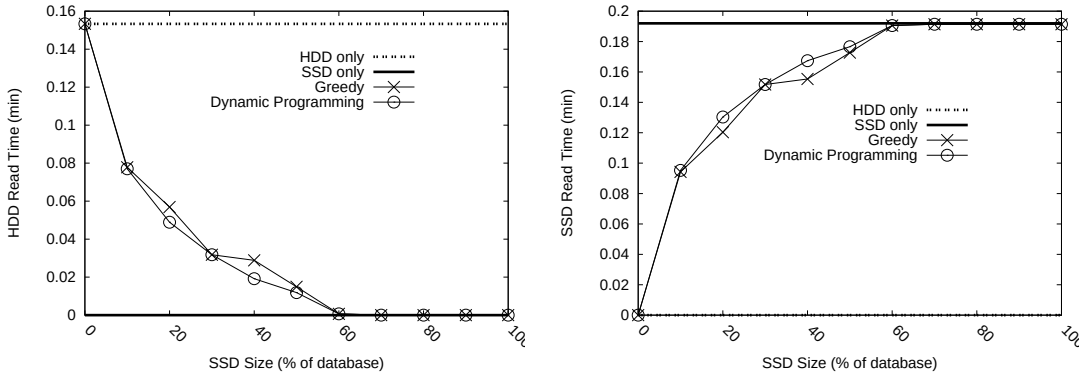

Figure 7: HDD and SSD read time results for the varying SSD capacity experiment

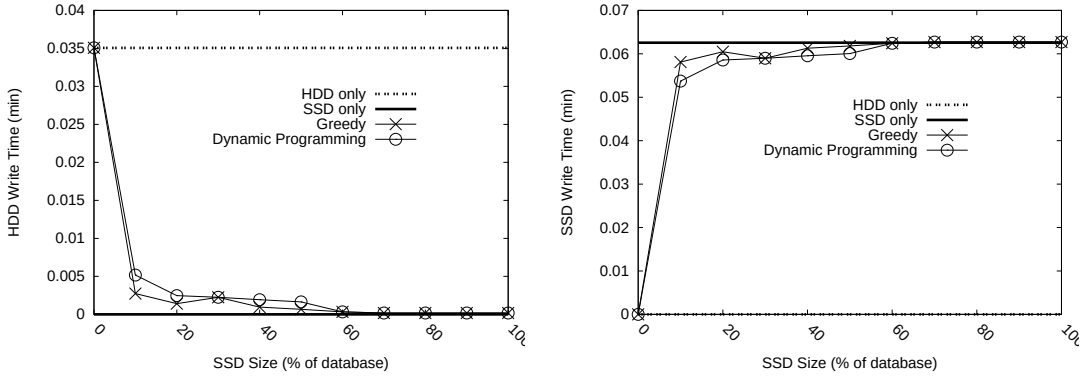

Figure 8: HDD and SSD write time results for the varying SSD capacity experiment

on the SSD. Despite this fact, our algorithms are still able to outperform HDD-only, for a large range of the results (buffer sizes between 0% and 60% of database size). This shows that it is beneficial to use a hybrid setup, even when RAM size is relatively large.

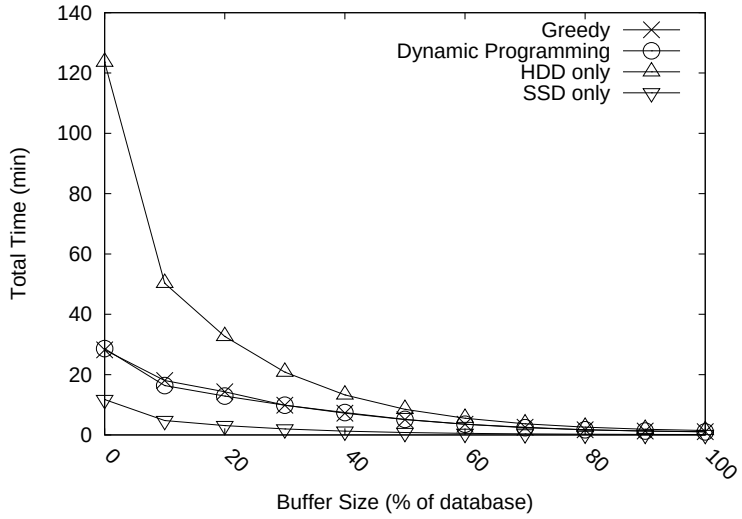

Figure 9: Total time results for the varying buffer capacity experiment

Figure 10 shows the results for the random access latencies of the HDD and SSD when buffer capacity is varied. The results again show the random access latency of the HDD is the dominate cost since these costs are almost the same as the total time results.

Figure 11 shows the results for read time of the HDD and SSD when buffer capacity is varied. The results show that when buffer size is zero the SSD read costs is higher than the HDD read costs for Greedy and Dynamic Programming. This is caused by a large portion of the frequently read columns being placed on the SSD. The frequently accessed columns are cache resident most of the time when the buffer size is greater than zero. Hence the SSD read costs are only high when the buffer size is zero.

Figure 12 shows the results for write time of the HDD and SSD when buffer capacity is varied. Again it can be seen that Dynamic Programming and Greedy are very effective at placing the frequently written pages on the SSD. This is again because the frequently read pages are also the frequently written pages and are thus the columns the algorithms placed on the SSD.

7.3 Varying Hardware Specifications Experiment

The third experiment was conducted to see how the system perform using different hardware. We simulated the system running on average and good hardware. The HDD representing the average and good HDD were the WD Caviar and the WD VelociRaptor respectively. The SSDs representing the average and good SSD were SanDisk SSD and Mtron Flash SSD respectively. The hardware specifications of these are listed in Table 1 in Section 2. The default buffer and SSD capacities were used. The configurations used are defined as follows:

Avg-Avg The system was simulated using an average speed HDD and an average speed SSD.

Avg-Good The system was simulated using an average speed HDD and a fast SSD.

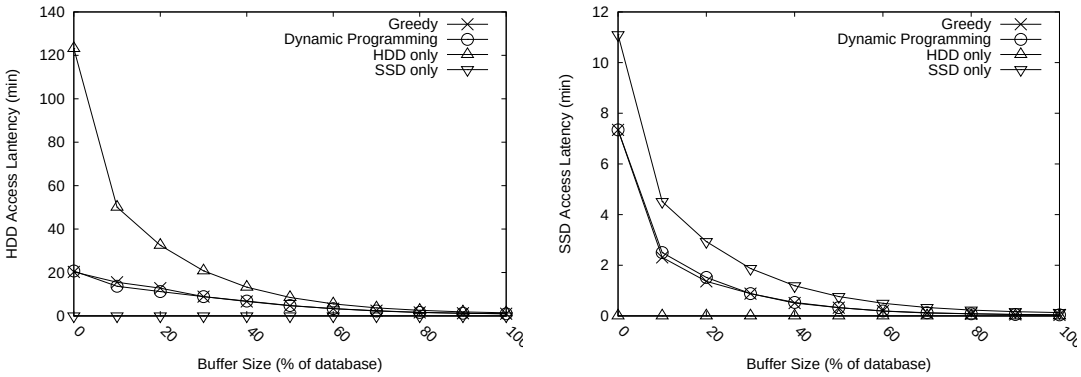

Figure 10: HDD and SSD random access latency results for the varying buffer capacity experiment

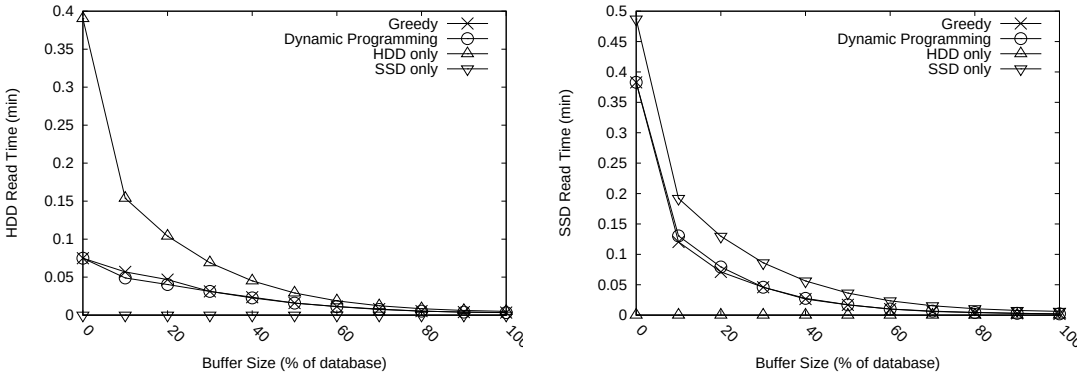

Figure 11: HDD and SSD read time results for the varying buffer capacity experiment

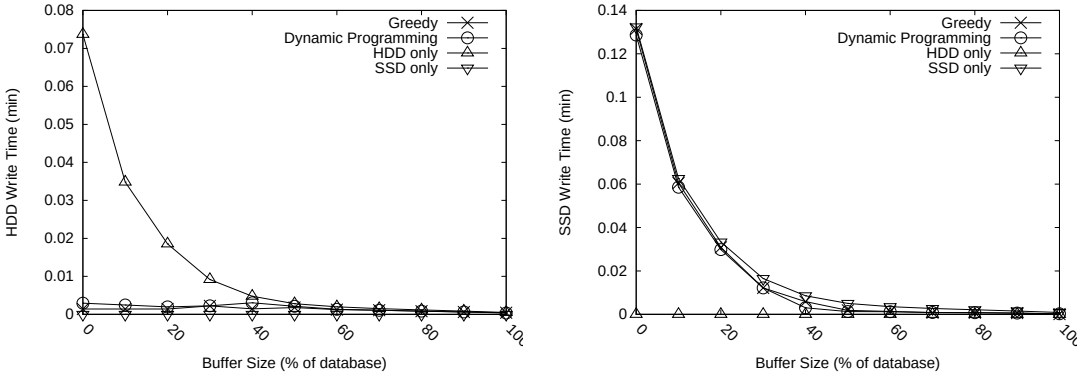

Figure 12: HDD and SSD write time results for the varying buffer capacity experiment

Good-Avg The system was simulated using a fast HDD and an average speed SSD.

Good-Good The system was simulated using a fast HDD and a fast SSD.

The results are shown on Table 5. The table shows the time spent on each of the following operations: random access latency on HDD; reads on HDD; writes on HDD; random access latency on SSD; reads on SSD; writes on SSD; and total time.

It is clear that the HDD random access cost is always the dominant factor and the difference between a decent non-expensive SSD and a high cost SSD is not noticeable when HDD is also used. The reason for this is that the random access cost difference between the average and good SSD is small (similarly between the average and good HDD). However, the random access cost difference across drive types is large.

Metric	Avg-Avg	Avg-Good	Good-Avg	Good-Good
HDD random access latency(sec)	805.4	805.4	528.2	528.2
HDD read (sec)	2.9	2.9	3.3	3.3
HDD write (sec)	0.14	0.14	0.16	0.16
SSD random access latency(sec)	151.7	137.9	151.7	137.9
SSD read (sec)	7.9	5.7	7.9	5.7
SSD write (sec)	3.5	2.2	3.5	2.2
Total (sec)	971.574	954.3	694.7	677.5

Table 5: Varying hardware specifications results

7.4 Partitioning time

In this experiment we measured the time that the Dynamic Programming and Greedy algorithms to partition the database. Although the partitioning is done offline we still want to keep the time to be within acceptable limits. The parameter settings used were all the default settings described in Table 4. Both the Dynamic Programming and the Greedy algorithms took less than 1 minute to partition the data. Since this is done offline, it is a very acceptable time. The reason that both algorithms are so fast is that they both have low computational complexity. The Greedy algorithm has a complexity of $O(n\ log\ n)$ and Dynamic Programming has a complexity of $O(n\ c)$. Where n is the number of columns in the database and c is capacity of the SSD.

8 Conclusion

Our aim was to develop a database system to run on systems which have both HDD and SSD storage available, a system which could achieve enhanced I/O performance by using a limited amount of flash memory. We achieved this by vertically partitioning the database across the two drives. The partitioning problem was mapped to the knapsack problem which is a known NP-complete problem. To solve the problem, we implemented both a dynamic programming algorithm which would always produce an optimal solution and a greedy heuristic solution.

We experimentally tested both algorithms. The results showed that the partitioning algorithms proposed significantly outperformed HDD-only, using only a small amount of flash memory for a variety of situations. Among the dynamic programming and greedy variations of the partitioning algorithms, the results were similar.

For future work, we will explore online partitioning. This is important since workload usage patterns often change with time and an offline approach, such as the one proposed by this paper would not be able to adapt to the changes

in a timely manner. In contrast, an online partitioning algorithm will be able to more quickly adapt to changing access patterns. A variation of the proposed greedy algorithm would be more suitable for such a setting. Also, the system could include a more customized buffering method. Research in flash memory aware buffering has been done in parallel to our research[14]. Lastly a more thorough exploration of the performance of the proposed algorithms with different benchmarks is an area of future work.

9 Acknowledgements

This work is supported under the Australian Research Council's Discovery funding scheme (project number DP0985451). We would also like to thank the anonymous reviewers for their comments and suggestion. They have help us greatly improved the quality of this paper.

References

[1] D. Abadi, D. J. D. D. S. Myers, and S. Madden. Materialization Strategies in a Column-Oriented DBMS. *ICDE 2007. IEEE 23rd International Conference on Data Engineering, 2007*, pages 466 – 475, 2007.

[2] D. Abadi, S. Madden, and M. Ferreira. Integrating compression and execution in column-oriented database systems. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, pages 671–682, New York, NY, USA, 2006. ACM.

[3] Axis Communications. Journaling Flash File System. http://developer.axis.com/software/jffs/, [Accessed March 26, 2008].

[4] A. Ban. Flash File System. U.S. Patent 5 404 485, March 8, 1995.

[5] A. Ban. Flash File System optimized for page-mode flash technologies. U.S. Patent 5 937 425, August 10, 1999.

[6] P. A. Boncz, S. Manegold, and M. L. Kersten. Database architecture optimized for the new bottleneck: Memory access. In *VLDB Conference*, pages 54–65, 1999.

[7] P. A. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR*, pages 225–237, 2005.

[8] G. P. Copeland and S. N. Khoshafian. A decomposition storage model. *SIGMOD Rec.*, 14(4):268–279, 1985.

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

[10] T. P. P. Council. Tpc benchmark c standard specification revision 5.9, June 2007.

[11] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[12] D. P. Hans Kellerer, Ulrich Pferschy. *Knapsack Problems*. Springer, 2004.

[13] S. Harizopoulos, V. Liang, D. J. Abadi, and S. Madden. Performance tradeoffs in read-optimized databases. In *VLDB '06: Proceedings of the 32nd International Conference on Very Large Databases*, pages 487–498. VLDB Endowment, 2006.

[14] Z. He and P. Veeraraghavan. Fine-grained updates in database management systems for flash memory. *Information Sciences*, 178(18):3162–3181, 2009.

[15] J. A. Hoffer and D. G. Severance. The use of cluster analysis in physical data base design. In *VLDB '75: Proceedings of the 1st International Conference on Very Large Databases*, pages 69–86, New York, NY, USA, 1975. ACM.

[16] G.-J. Kim, S.-C. Baek, H.-S. Lee, H.-D. Lee, and M. J. Joe. LGeDBMS: a small DBMS for embedded system with flash memory. In *VLDB '06: Proceedings of the 32nd International Conference on Very Large Databases*, pages 1255–1258. VLDB Endowment, 2006.

[17] Y.-R. Kim, K.-Y. Whang, and I.-Y. Song. Page-differential logging: an efficient and dbms-independent approach for storing data into flash memory. In *SIGMOD Conference*, pages 363–374, 2010.

[18] I. Koltsidas and S. D. Viglas. Flashing up the storage layer. In 514-525, editor, *VLDB*, 2008.

[19] S.-W. Lee and B. Moon. Design of flash-based DBMS: an in-page logging approach. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, pages 55–66, New York, NY, USA, 2007. ACM.

[20] S.-W. Lee, B. Moon, C. Park, J.-M. Kim, and S.-W. Kim. A case for flash memory ssd in enterprise database applications. In *SIGMOD Conference*, pages 1075–1086, 2008.

[21] Y. Li, B. He, Q. Luo, and K. Yi. Tree indexing on flash disks. In *ICDE*, 2009.

[22] S. Nath and A. Kansal. FlashDB: dynamic self-tuning database for NAND flash. In *IPSN '07: Proceedings of the 6th International Conference on Information Processing in Sensor Networks*, pages 410–419, New York, NY, USA, 2007. ACM.

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

[24] S. B. Navathe and M. Ra. Vertical partitioning for database design: a graphical algorithm. *ACM SIGMOD Record*, 18(2):440–450, 1989.

[25] A. One. YAFFS: Yet another Flash file system. http://www.aleph1.co.uk/yaffs/, [Accessed March 26, 2008].

[26] S. Papadomanolakis and A. Ailamaki. AutoPart: automating schema design for large scientific databases using data partitioning. In *Proceedings of the 16th International Conference on Scientific and Statistical Database Management(SSDBM)*, pages 383 – 392, Pittsburgh, PA, USA, 2004. Carnegie Mellon Univ.

[27] P. Schmid and A. Roos. Samsung, Ridata SSD Offerings Tested. Toms Hardware Web Site. http://www.tomshardware.com/reviews/solid-state-drives,1745.html, 2007 [Accessed April 25, 2008].

[28] M. A. Shah, S. Harizonpoulos, J. L. Wiener, and G. Graefe. Fast scans and joins using flash drives. In *Proceedings of the Fourth International Workshop on Database Management on New Hardware*, pages 17–24, 2008.

[29] M. Stonebraker, D. J. 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 *VLDB '05: Proceedings of the 31st International Conference on Very Large Databases*, pages 553–564. VLDB Endowment, 2005.

[30] StorageSearch.com. The Fastest Solid State Disks (SSDs). http://www.storagesearch.com/ssd-fastest.html, 2008 [Accessed April 26, 2008].

[31] D. Tsirogiannis, S. Harizopoulos, M. A. Shah, J. L. Wiener, and G. Graefe. Query processing techniques for solid state drives. In *SIGMOD*, 2009.

[32] D. Woodhouse. JFFS: The journaling flash file system. http://sources.redhat.com/jffs2/jffs2.pdf, 2001 [Accessed March 26, 2006].