A Multi-resolution Block Storage Model for Database Design

Jingren Zhou
Columbia University
jrzhou@cs.columbia.edu

Kenneth A. Ross
Columbia University
kar@cs.columbia.edu

Abstract

We propose a new storage model called MBSM (Multi-resolution Block Storage Model) for laying out tables on disks. MBSM is intended to speed up operations such as scans that are typical of data warehouse workloads. Disk blocks are grouped into "super-blocks," with a single record stored in a partitioned fashion among the blocks in a super-block. The intention is that a scan operation that needs to consult only a small number of attributes can access just those blocks of each super-block that contain the desired attributes. To achieve good performance given the physical characteristics of modern disks, we organize super-blocks on the disk into fixed-size "mega-blocks." Within a mega-block, blocks of the same type (from various super-blocks) are stored contiguously. We describe the changes needed in a conventional database system to manage tables using such a disk organization. We demonstrate experimentally that MBSM outperforms competing approaches such as NSM (N-ary Storage Model), DSM (Decomposition Storage Model) and PAX (Partition Attributes Across), for I/O bound decision-support workloads consisting of scans in which not all attributes are required. This improved performance comes at the expense of single-record insert and delete performance; we quantify the trade-offs involved. Unlike DSM, the cost of reconstructing a record from its partitions is small. MBSM stores attributes in a vertically partitioned manner similar to PAX, and thus shares PAX's good CPU cache behavior. We describe methods for mapping attributes to blocks within super-blocks in order to optimize overall performance, and show how to tune the super-block and mega-block sizes.

1 Introduction

The I/O behavior between main-memory and secondary storage is often a dominant factor in overall database system performance. At the same time, recent architectural advances suggest that CPU performance on memory-resident data is also a significant component of the overall performance [17, 2, 3, 10]. In particular, the CPU cache miss penalty can be relatively high, and can have a significant impact on query response times [2]. Therefore, modern database systems should be designed to be sensitive to both I/O performance and CPU performance.

In this paper, we focus in particular on the storage model used to place data from relational tables on disk. Our goal is to create a scheme that yields good performance for workloads in which operations such as table scans are frequent relative to single-record insertions and deletions.

I/O transfers between memory and disk are performed in units of *blocks* (sometimes also called *pages*). I/O volume, measured in blocks, is a simple measure of an algorithm's I/O complexity. More detailed cost models take into account the physical characteristics of disk devices. For example, sequential I/O is faster than random I/O because the disk head usually does not need to seek.

Relational DBMSs typically pack records into slotted disk pages using the N-ary Storage Model (NSM). NSM stores records contiguously starting from the beginning of each disk page, and uses an offset (slot) table at the end of the page to locate the beginning of each record [13]. Given access to the page identifier (say via an index), a record can be retrieved using a single page of I/O. On the other hand, scans that access just a few columns must retrieve from the disk *all* blocks of the table, even though most of the transferred data is not relevant to the query. NSM has poor cache behavior because it loads the cache with unnecessary data [1].

The Decomposition Storage Model (DSM) [4] was proposed to minimize unnecessary I/O for those queries which only use a small number of attribute values in each record. DSM vertically partitions an n-attribute relation into n sub-relations, each of which is accessed only when the corresponding attribute values are needed. An extra record-id field (surrogate) is needed in each component sub-relation, so that records can be pieced together. Sybase-IQ uses vertical partitioning combined with bitmap indices for data warehouse applications [12]. For table scans involving just a few attributes, DSM requires considerably less I/O than NSM. On the other hand, queries that involve multiple attributes from a relation must spend additional time to join the participating sub-relations together; this additional time can be significant [1]. Single-record insertions and deletions also require many pages of I/O rather than one for NSM (assuming no overflow).

Recent research [17, 2, 3] has demonstrated that modern database workloads are also impacted by delays related to the processor. Data requests that miss in the cache hierarchy are a key memory bottleneck. Loading the cache with useless data wastes bandwidth, pollutes the cache, and possibly forces replacement of information that may be needed in the future.

To address the issue of low cache utilization in NSM, Ailamaki et al. introduce Partition Attributes Across (PAX), a new layout for data records [1]. Unlike NSM, within each page, PAX groups all the values of a particular attribute together on a minipage. During a sequential data access, PAX fully utilizes the cache resources, because only a number of the required attribute's values are loaded into the cache. However, compared with DSM, PAX doesn't optimize the I/O between disk and memory. Like NSM, PAX loads all the pages belonging to the relation into the memory for scans, regardless of whether the query needs *all* or only several of the attributes. Unnecessary attributes' values waste the I/O bandwidth between disk and memory and decrease the efficiency of database buffer pool management. The challenge is to design a storage model with better I/O performance without compromising the nice cache behavior.

In this paper, we introduce a new storage model called MBSM (Multi-resolution Block Storage Model) which takes care to address both the I/O performance and cache utilization in main-memory. It is similar to PAX in that attributes are stored columnwise as physically contiguous array segments. As a result, it shares PAX's good cache behavior. It is different from PAX in that it only loads pages with referenced attributes' values from disk. It also considers disk characteristics by placing the data carefully on disk to facilitate fast sequential I/O.

Disk blocks are grouped into "super-blocks," with a single record stored in a partitioned fashion among the blocks in a super-block. The intention is that a scan operation that needs to consult only a small number of attributes can access just those blocks of each super-block that contain the desired attributes. To achieve good performance given the physical characteristics of modern disks, we organize super-blocks on the disk into fixed-size "mega-blocks." Within a mega-block, blocks of the same type (from various super-blocks) are stored contiguously. The cost of reconstructing a record from its partitions is small.

Experiments show that MBSM outperforms competing approaches such as NSM, DSM and PAX, for I/O bound decision-support workloads consisting of scans in which not all attributes are required. The average scan query cost for a workload based on the TPC-H benchmark is 70% less with MBSM than with either PAX or NSM, and comparable with DSM, which has high record reconstruction cost. For insertions and deletions of single records into the Lineitem table of TPC-H, the cost is 40% less for MBSM than for DSM.

The rest of this paper is organized as follows. Section 2 presents an overview of the related work and surveys current disk technology. Sections 3 and 4 explain our new storage model in detail and analyze it storage requirements. Section 5 lists the changes required of conventional database systems to use MBSM. Section 6 evaluates MBSM on both a synthetic workload and a workload based on the the TPC-H decision-support benchmark. We conclude in Section 7.

2 Related Work

This section describes the advantages and disadvantage of three data placement schemes: NSM, DSM and PAX (shown in Figure 1), and briefly outlines the disk technology which is important to our design.

2.1 The N-ary Storage Model

Traditionally, a relation's records are stored in slotted disk pages [13] obeying the N-ary Storage Model (NSM). NSM stores records sequentially on data pages. Each record has a record header, offsets to the variable-length values, and other implementation-specific information. Fixed-length attribute values are stored first, followed by an array of offsets and a mini-heap containing the variable-length attribute values. Each new record is typically inserted into the first available free space starting at the beginning of the page. Records may have variable lengths, and therefore a pointer to the beginning of the new record is stored in the next available slot from the end of the page. For fixed-sized records, an array of bits, one per slot, is stored at the end of the page to keep track of free slot information.

2.2 The Decomposition Storage Model

A "full DSM" method partitions an n-attribute relation vertically into n sub-relations to improve the density of relevant information for queries. Each sub-relation contains two attributes, a logical record ID (or *surrogate*) and an attribute value. Sub-relations are stored as independent relations in slotted pages, enabling each attribute to be scanned separately.

DSM exhibits high I/O and cache performance on decision support workloads known to utilize a small percentage

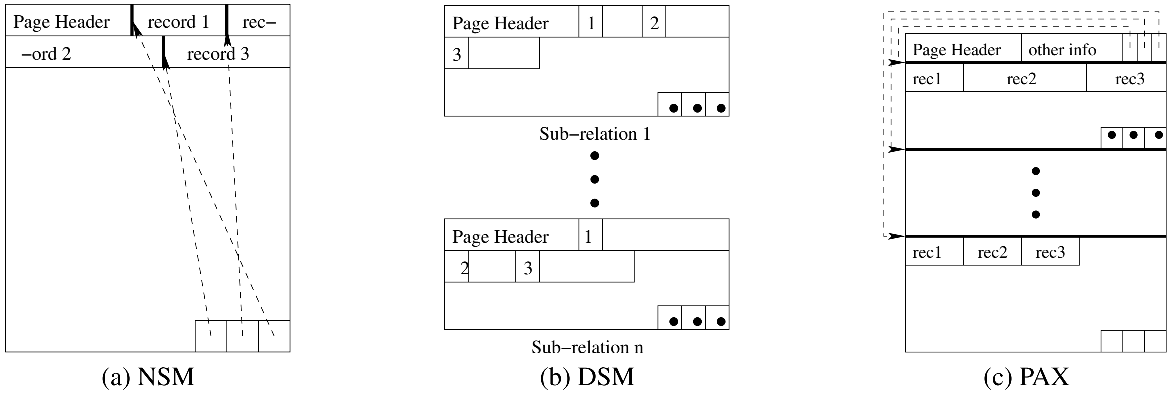


Figure 1. Three Data Placement Schemes

of the attributes in a relation. Sybase-IQ uses vertical partitioning combined with bitmap indices for data warehouse applications [12]. In addition, DSM can improve cache performance of main-memory database systems, assuming that the record reconstruction cost is low [3]. For queries that involve multiple attributes from each participating relation, the database system must join each sub-relation on the surrogate to reconstruct the partitioned records. The time spent joining sub-relations increases with the number of attributes in the result relation and can be significant [1].

An alternative "partial DSM" [4, 5] partitions each relation based on an attribute affinity graph, which connects pairs of attributes based on how often they appear together in queries. Highly connected attributes are then stored in the same partition (using NSM for each partition and requiring one surrogate per partition).

2.3 The Partition Attributes Across Model

PAX vertically partitions the records within each page, and groups values for the same attribute together in minipages [1]. Compared with NSM, PAX demonstrates high data cache performance because values from the same column are loaded together into each cache line. Compared with DSM, PAX cache performance is better because there are no surrogates stored along with attributes. PAX has low record reconstruction cost after data is in memory. But, like NSM, PAX incurs more disk accesses than DSM when queries only involve a fraction of the attributes. Since the information content of a PAX page is the same as that of an NSM page, the effort required to reimplement an NSM-based database system to use PAX is small.

2.4 Disk Technology

In this section, we begin with an overview of disk characteristics that are helpful for our later design. For a detailed introduction to disk drives, see [15, 9].

Access time is the metric that represents the composite of all specifications reflecting random performance positioning in the hard disk. The most common definition is that access time is the sum of *command overhead time*, *seek time*, *settle time* and *rotational latency*. The *Track-to-Track Seek Time* is the amount of time that is required to seek between adjacent tracks. It is much smaller than the *Average Seek Time* from one random track (cylinder) to any other.

A seek is composed of a *speedup*, where the arm is accelerated until it reaches half of the seek distance or a fixed maximum velocity, a *coast* for long seeks, where the arm moves at its maximum velocity, a *slowdown*, where the arm is brought to rest close to the desired track. The settle time refers to the amount of time required, for the heads to stabilize sufficiently for the data to begin to be read. The rotational latency is the waiting time for the desired block to rotate under the head. Very short seeks are dominated by the settle time. Short seeks (less than 200 – 400 cylinders) spend almost all of their time in the constant-acceleration phase, and their time is proportional to the square root of the seek distance plus the settle time. Long seeks spend most of their time moving at a constant speed, taking time that is proportional to distance plus a constant overhead. As disks become smaller and track densities increase, the fraction of the total access time attributed to the settle phase increases.

	Seagate Cheetah	Quantum Atlas
Capacity (GB)	18.35	36.7
RPM	15K	10K
Avg. Rot. Latency (ms)	2	3
Avg. Seek Time (ms)	3.9	4.7
Adjacent Track Seek (ms)	0.5	0.6

Table 1. Disk Specifications

For the experimental results in this paper, we are using two state-of-the-art SCSI Ultra160 disks: the Seagate Cheetah X15 and the Quantum Atlas 10K II. Table 1 lists some of the specifications of both disks. Detailed descriptions can be found in [16, 11].

3 Super-Blocks

In this section, we introduce our new strategy for placing tables on disk for fast I/O performance. The idea is to partition a relation into *Super-Blocks*. Each super-block consists of a fixed number of pages. Values for one attribute go into only some of a super-block's pages, not into all of the pages. This facilitates queries which involve just a few attributes of the relation because only the pages which really store those attributes' values are required, thus incurring fewer I/O requests than NSM or PAX. The values for the same attribute are placed contiguously in each page within a super-block. So our new storage model has similar cache behavior to PAX. On the other hand, there are no DSM-like surrogates within each page. The cost of reconstruction of the partitioned records is low. All the attributes for one record can be found in one super-block. To take advantage of the high speed of disk sequential I/O, we group super-blocks into *Mega-Blocks*. Pages, from different super-blocks, which store values for the same attribute(s) are placed contiguously within mega-blocks. Mega-blocks will be discussed in Section 4.

We assume we have a relation with n attributes, each attribute having size $s_i, 1 \leq i \leq n$. We assume for now that each attribute is fixed-sized. We will discuss variable-sized attributes in Section 3.3 and Section 7.

Within each page in a super-block, the page structure is the same as PAX's. Attributes are stored in a partitioned fashion, in contiguous array segments called minipages. When dealing with just fixed-sized attributes, we can calculate exact offsets for the minipages so that each minipage holds exactly the same number of records when the page is filled to capacity. Page reorganization does not occur, and we do not even need to store explicit pointers to the minipages in the page header.

3.1 A First Step

As a starting point, imagine that we use a super-block with n component pages, one attribute per page. We place values for an attribute in one and only one of the n pages. Within each page, the values are stored contiguously, with an array of bits stored at the end of the page to keep track of free slots. The page structure is similar to a fixed-sized DSM page, except that we don't have surrogates for records. The record reconstruction cost is low since matching attributes can be found from their offsets, and there is no join needed. We stop inserting new records into a super-block once one of the component pages is full. Figure 2(a) shows a filled four-page super-block for a relation with four attributes. The shaded region represents empty space and the dashed lines within a page represent value boundaries. The super-block contains 3 records. The super-block is full only because the first page, which stores the largest attribute, is full.

Pages in a super-block can be accessed independently. With proper information, the database can access only the pages for the required attributes. The super-block has good cache behavior due to the high value density. The record reconstruction cost is low since matching attributes can be found from their offsets, and there is no join needed.

We measure the quality of this solution in terms of the average query time and the space needed to represent the relation. The space *overhead* is easily measured as the proportion of wasted space in a page that is lost due to fragmentation. In Figure 2(a), suppose we have R records to store. Then the space needed for the whole table is $4R/3$ pages. Since the data can actually fit in $3R/4$ pages without fragmentation (Figure 2(c)), the fragmentation overhead is $4R/3 - 3R/4 = 7R/12$ pages total, and the fragmentation constitutes $7/16$ of each super-block.

The query time depends on the query workload. In Figure 2(a), a scan of any single attribute requires the reading of one quarter of all pages, i.e., $R/3$ pages. A scan of k attributes requires $kR/3$ pages in this scheme, $1 \leq k \leq 4$.

Note that the fragmentation and the query time are positively correlated. Although disk space is cheap these days, the issue is that the extra space resulting from fragmentation consumes precisous I/O bandwidth. If there is a lot of fragmentation, queries that read pages having wasted space need to read more pages. As a result, we expect that a solution with small average query time (our primary goal) will also have small fragmentation overhead (our secondary goal), and vice-versa.

3.2 A Second Step

The method of Section 3.1 works when s_i is roughly the same for all i. If not, it could yield a large fragmentation within many blocks of each super-block. For example, if $s_i + s_j \leq s_k$, it would better to put both the ith and jth attributes' values in one page and construct a super-block with $n-1$ pages.

As shown in Figure 2(b), the reorganized super-block has less fragmentation. In particular, the fragmentation now constitutes just $1/4$ of each super-block. This reorganization does not change the time required for any single-attribute queries. However, a query that requires both attributes 2 and 3 is now significantly cheaper. For example, the scan query asking for exactly attributes 2 and 3 now takes $R/3$ pages rather than $2R/3$ pages.

At this point, we are able to formulate an optimization problem. For simplicity, we phrase the optimization problem as trying to minimize fragmentation. In later sections, we will reformulate the problem as one of minimizing average query time. Suppose that p is the number of pages

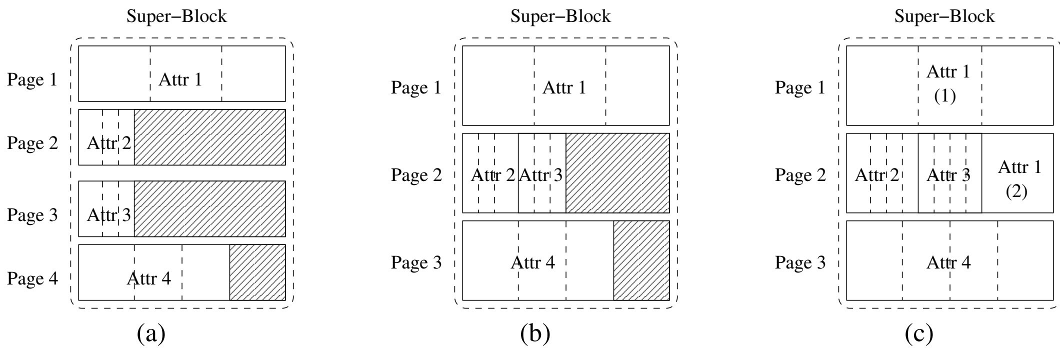


Figure 2. Super-Block Structure

available in a super-block. The optimization problem is "Given different s_i, how does one place attribute values into a p-page super-block in a balanced way to get the least fragmentation?". Unfortunately, this is a well-known NP-complete problem. For a given p, this problem is usually referred to as parallel machine scheduling [7] (the attributes are jobs, the attribute sizes are processing time, the goal is to minimize the schedule length). There are several approximation algorithms for this problem. If we greedily place an arbitrary attribute on the least loaded page, we get a 2-approximation [6], i.e., an approximation that is guaranteed to be within a factor of 2 of optimal. If we first sort the attributes (largest size first), and repeatedly place the next attribute on the least loaded page, we get a 4/3-approximation [8]. Note that in addition to performing the optimization described above, we are also able to optimize p to find a good layout scheme. We defer the discussion of how to choose p until Section 4. We remark that even when the number of attributes u is small, exhaustive search takes time of the order of $min(u, p)^u$, which is likely to be infeasible; when we consider splitting attributes in the next section, the search space becomes even larger.

In subsequent sections, we use this 4/3-approximation algorithm as the basis for a heuristic attribute placement method. As our optimization criteria become more complex, it becomes much more difficult to give theoretical bounds on the quality of the solutions. Our expectation is that even though the optimization criteria become more complex, the underlying nature of the optimization remains the same, and so extensions of the 4/3-approximation algorithm will generate good solutions.

3.3 A Third Step

If one attribute size is much larger than the total size of the rest of the attributes, even the optimal placement can not guarantee small fragmentation. For example, in the TPC-H database benchmark [18], some relations have very large "comment" attributes. In Figure 2(b), the first attribute size is large, meaning that when the first page is full, there is still much empty space in the rest of the pages.

In this case, we may choose to place the largest attribute's value in more than one page in a super-block. Our revised algorithm is given below:

- For a given p and $S = \sum_{i=1}^{n} s_i$, divide any attribute whose size is larger than $\frac{S}{p}$ into several $\lceil \frac{S}{p} \rceil$-sized sub-attributes, plus one extra smaller sub-attribute if necessary.
- Use the 4/3-approximation algorithm to place the resulting attributes.

Although we split large attributes into small sub-attributes in choosing plans, the real attribute values are not split vertically. Those pages which are designed to store an attribute's values are filled in order. During the insertion, the attribute's values are first placed into the first page until it is full, then we move to the next page, and so on. The order is important. As we will discuss in Section 5, the database uses the order information to compute in which page a specific record's attribute is stored.

Figure 2(c) shows the results of the previous relation after such a reorganization. The first attribute's values are stored in the first two pages (the first page is filled first). The super-block now stores four records, instead of three. As a result, there is zero fragmentation overhead. A scan of attribute 4 now takes $R/4$ pages rather than $R/3$, and a similar observation holds for a query that scans both attributes 2 and 3. On the other hand, a query that scans attribute 1 becomes more expensive, requiring $R/2$ pages rather than $R/3$, since both pages 1 and 2 need to be consulted. The net effect on average query time depends on the query workload. We will formulate average query time as our optimization criterion in Section 3.4.

For variable-length attributes, we choose the largest possible size as the size of the attribute, since we don't have any prior-knowledge of its size distribution. By using the largest sizes of variable-length attributes, we know in advance how many records will be placed in each super-block, and we can avoid having to deal with overflows. On the other hand, we waste some space when variable-length attributes are large and are often smaller than their maximum size.

3.4 Cost-Based Super-Block

We now change our optimization criterion from fragmentation to average query time. For decision-support applications, we can optimize the super-block placement based on known workloads. Existing algorithms [4, 5] partition each relation based on an attribute affinity graph, which connects pairs of attributes based on how often they appear together in queries. The affinity of two attributes is defined as the number of queries in the workload in which both attributes are referenced. These algorithms cannot be used directly to guide our super-block placement because they may generate sub-relations of different size, thus incurring much fragmentation in a super-block. Sometimes, it is impossible to compose a plan with both optimal attribute affinity and optimal fragmentation. Instead, we use attribute affinity heuristics to guide our partition.

First we develop a score function for each placement plan for the given workload. We assume that the workload consists of queries whose access patterns amount to scans of the referenced tables for the referenced attributes. This choice is reasonable for a decision-support application, where such queries are common, and are generally the most expensive. Suppose the cost of sequential data access is proportional to the number of disk pages read. Given the number of records in the relation, the total cost to read one page from each super-block is given by

$$disk\ pages_{read} = \frac{total\ number\ of\ records}{records\ per\ page}$$

$$records\ per\ page = \frac{page\ size}{\max_i(\sum attribute\ sizes\ in\ page_i)}$$

We define M as the maximum sum of the attribute sizes in any page of a super-block. The total I/O cost is thus M multiplied by the number of pages required from each super-block.

Let q_i equal the number of pages in a super-block referenced by the ith query in the workload. If one attribute's values are stored in more than one page, all the pages should be counted. Let p_i equal the probability of the ith query in the workload. The score function is defined as:

$$Score = M * \sum_i (q_i * p_i)$$

Note that both M and q_i vary as the placement scheme varies. Our revised algorithm is given below:

- We define a affinity threshold. Any pair of attributes having affinity larger than the threshold are merged into a larger attribute with the size of the sum of the two attributes' sizes. More than two attributes may be merged in this way.
- Use the 4/3-approximation method to place attributes for each p (the range of p is discussed in Section 4.2).
- Split large attributes as described in Section 3.3. However, if an attribute actually consists of several attributes merged using the affinity criterion, and it is too big, undo one step of the merge procedure for this attribute, and rerun the placement algorithm.
- Choose the plan with minimal score.

4 Mega-Blocks

How do we organize the super-blocks on disk? Our aim is to support the claim that performance can be proportional to the number of pages needed for scan operations.

Suppose we have a single-attribute scan query and the referenced attribute values are all in the first page of a super-block. If we place super-blocks contiguously, the database needs to read a page in one super-block, skip another $p - 1$ pages, then read a page and so on. What is the disk performance for this kind of access method, compared with sequential access?

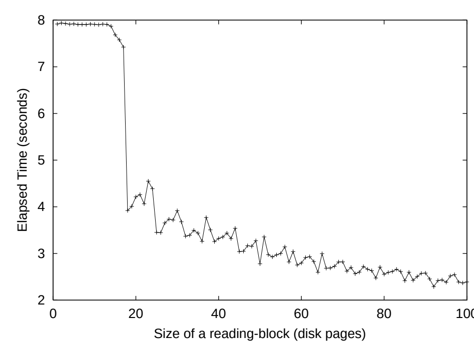


Figure 3. Simulation of Interleaved Reading

We simulate this kind of interleaved reading on the Seagate Cheetah X15 disk[1] when $p = 4$. We define a reading-block as the number of blocks read per sequential access. The simulation reads a reading-block, skips another 3 reading-blocks, then reads a reading-block and so on. The simulation varies the reading-block size from 1 disk page to 100 disk pages, but the total number of pages read is kept the same. Figure 3 shows the results for reading 10,000 blocks. For reference, it takes 2 seconds to sequentially read 10,000 blocks.

[1]Similar results were obtained for the Quantum disk.

When the reading-block size is one disk page, we are simulating the case of a single-attribute scan over a 4-page super-block. The time taken is only a little less than sequentially reading all of the super-blocks. The time drops significantly when the reading-block is larger than one track (about 20 pages in this case). This is because when the reading-block is smaller than one track, the dominant cost is rotational latency; we have lost the benefit of sequential I/O optimization. As the reading-block size increases, the number of skip operations decreases and there is more sequential reading *within* each reading-block, so the time drops.

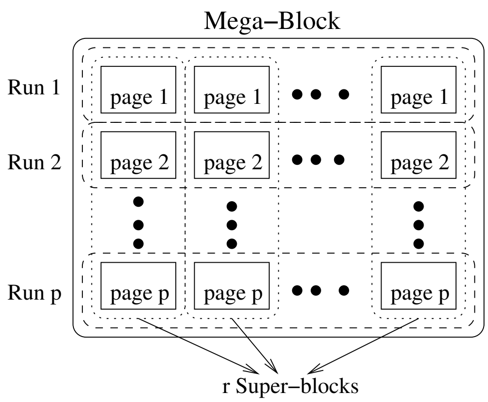


Figure 4. Mega-Block Structure

As we see, it is not a good idea to place super-blocks contiguously on disk. Instead, we group super-blocks into a *mega-block*. Corresponding pages are stored in one contiguous *run*. Figure 4 shows the structure of a mega-block. The size of a run is r disk pages. As we see in Figure 3 a lower bound on the run length should be the track size. The total size of a mega-block is $p * r$ disk pages.

If one attribute is stored in more than one page in a super-block, we try to arrange these pages in adjacent runs so that a single-attribute scan query can be answered by sequentially reading one or more runs in each mega-block. We also define the affinity of any two distinct pages in a super-block as the largest affinity of any two attributes, one from each of the two pages. A pair of pages with higher affinity are more likely to be accessed in the same query. Thus, we also try to arrange such pages into adjacent runs.

A subtle point about this ordering of runs is that we can sometimes get I/O transfer units larger than r. For example, if p is large, an attribute might be split across several (say k) pages within a super-block, and be the only attribute on all (or all but one) of those pages. A scan of the relation to get that attribute can perform a sequential I/O request for k contiguous runs, meaning that the I/O unit is actually $k * r$.

4.1 Mega-Block Performance Issues

Disk Prefetching. Consider a query which involves a few (more than one) attributes. Database systems typically require that all the referenced attributes for the same record are available before performing any operations. A naive scan operator would have some drawbacks. For example, a two-attribute scan operator might first get the first block in the first run which stores the first required attribute, then seek to the first block in the second run which stores the second required attribute. After that, the scan operator goes back to the first run and reads the second block there. The disk head *thrashing* damages the performance.

Disk prefetching in buffer pool management can be useful in this case. During a sequential read, when we come to the first super-block of a mega-block, all blocks in the current run, instead of just one block, are read each time and buffered in memory. Disk heads move to the other run (or somewhere else) only after reading one run of blocks.

Updates and Lookups. Modifying a single attribute under MBSM requires one *random* disk write for the block containing the attribute value. Inserting or deleting a record requires $min(u, p)$ block writes, where u is the number of attributes in the relation. Those blocks belong to a single super-block and are stored on different runs of a mega-block. Fortunately, runs on a mega-block are close to each other, within a few tracks. Thus the update cost is 1 *random* disk write plus $min(u, p) - 1$ *nearby* disk writes. Under DSM, inserting or deleting a record requires u *random* disk writes because different sub-relations are not necessarily stored in an interleaved fashion. Note that MSBM handles bulk appends efficiently, since many records can be placed into a super-block, and data may be written one run at a time.

Lookups of a single record under MBSM require a number of page reads that depends on the number of attributes required. Lookups for a single attribute require one page of I/O. Lookups for v attributes require at most $min(v, p)$ pages, and even fewer if several of the requested attributes reside on the same page within a super-block.

4.2 Choosing p and r

We defined the size of a mega-block as $r * p$ (r is the size of a run and p is the size of a super-block). It is tempting to make mega-blocks large in order to get better performance. However, we will be forced to allocate disk space in mega-block sized units. The allocation unit on disk cannot be arbitrarily large, since disk fragmentation will result. Further, a large mega-block means that more disk pages have to be prefetched during sequential data access, which increases the memory requirements and decreases the efficiency of database buffer pool management. Different

	NSM	DSM	PAX	MBSM
I/O volume for scan	full table	read columns + surrogate overhead	full table	read columns + overhead due to fragmentation
Space overhead	small internal fragmentation	surrogates	small internal fragmentation	small internal fragmentation plus overhead for variable-sized attributes
Update a single attribute	one random I/O	one random I/O	one random I/O	one random I/O
Insert/delete a record	one random I/O	many random I/Os	one random I/O	one random I/O, many "nearby" I/Os
Cache behavior	poor: useless data in cache	moderate: surrogate overhead	good: single-col values in cache	good: single-col values in cache
Record reconstruction	none	potentially expensive	cheap	cheap

Table 2. NSM, DSM, PAX and MBSM Comparison

systems may have different mega-block sizes, according to their resources. Within a single system, the mega-block size may be different for different tables. In our experiments, we set 4 MBytes as the upper bound for mega-block size.

A larger run size r improves sequential I/O performance. The minimum r should be larger than a disk track size, as suggested by Figure 3. A larger super-block size p gives more opportunities for attribute placement, potentially improving query time and/or fragmentation. But increasing p increases the cost of insertion or deletion (until p exceeds the number of attributes, at which point the cost of insertions and deletions remains constant). Given the upper bound on $r * p$, there is a trade-off in choosing r and p.

In our experiments, we are considering a decision-support workload, which would have relatively few single-record insertions and deletions. Considering that different disk zones have different track sizes, we choose r as 30 disk pages, a little more than the average track size (23 disk pages). This gives an upper bound on p of 17 pages. We vary p from 1 to 17, and choose the plan of placement with minimum score. Ties are broken by choosing the plan with smaller p.

Table 2 summarizes and compares the characteristics of NSM, DSM, PAX and MBSM.

5 Database Implementation

Conventional database systems use a layout based on NSM. We describe in this section how components of such a database would need to be modified to use MBSM. The changes are more extensive than those required by PAX. Nevertheless, we believe that the changes needed are relatively manageable.

Under MBSM, a record is identified by using the pair $<$ *superblock id, offset* $>$, instead of using the pair $<$ *page id, offset* $>$ in conventional database systems. A super-block is identified by the page id of its first block. Given the super-block size p and the run size r of the mega-block which stores it, the p blocks in a super-block are addressed by $pageid_{first\ block} + i * r$, $(0 \leq i \leq p-1)$. These numbers p and r are stored in the catalog for each table that uses the MBSM format. Since the super-block id itself is a page-id, the query execution engine doesn't have to change record representations. All the address mapping happens in a layer just above the buffer manager. Layers above this address mapping are essentially unchanged; they simply have the illusion of a larger "page" size equal to the super-block size. The buffer manager itself still deals with pages. Indexes can also be built using MBSM addresses without knowing the mapping.

When an operator opens an MBSM-organized table for subsequent access, it is provided with a *descriptor*. In addition to the conventional arguments to `open`, the operator supplies a list of attributes to be referenced, and a flag to indicate whether the operator is performing a scan (or scan-like, sequential) operation. The descriptor remembers the flag. The descriptor also records the set of pages within each super-block that are needed to include all of the requested attributes. (This information is derived from the catalog.)

Subsequent operator requests include the record identifier (pg, offset) together with the descriptor d. If d indicates a sequential operation, and pg is the first super-block in the mega-block, then all runs for the pages identified in d are read into the buffer sequentially. (If those pages are already in the buffer, no actual I/O happens.) Otherwise, just the pages for the single super-block corresponding to the required pages specified in d are read into the buffer.

A subtle point in this latter operation is that there may be several pages in the super-block containing attribute values specified in the `open` operation. In that case, d records the offset boundaries corresponding to switches from one page to the next. The value of *offset* is compared against these boundaries to locate the single page containing the attribute for the requested record. Note how important the assumption of fixed-size attributes is for this operation. With variable-sized attributes, there would be no simple way to uniquely determine which of several pages contained the right record.

Many database operators expect records to be stored in memory as (nonpartitioned) records. To use such operators, a mapping function that takes the required attributes and creates a contiguous record would be required. A similar mapping is needed for PAX [1].

Unlike the improved version of DSM [14], our method can support query plans involving the intersection of physical pointers (ie. $<$ *superblock id, offset* $>$). Such plans are often useful when multiple indexes are available.

We now describe additional changes in behavior of other components of a database system.

Buffer Manager. There is little change in the buffer manager. The request to the buffer manager is still the page id. Under MBSM, the buffer manager needs to be able to allocate at least one mega-block to the scanned table to avoid thrashing. A buffer replacement policy that was aware of sequential MBSM access patterns would be desirable.

Disk Space Manager. The disk space manager needs to respond to requests for allocations of mega-blocks at a time. Note that not all tables need to use MBSM, so the disk space manager may be allocating space at several granularities. Mega-blocks should be allocated in a contiguous sequence where possible, but this is not required. MBSM can adapt itself to data striping (such as RAID) by increasing the run size. Operations on each data disk in the array still maintain good sequential I/O performance.

Lock and Recovery Manager. Attributes of a record are stored in different pages in a super-block. Traditional page-level locking could become super-block-level locking under MBSM. Generally, under MBSM, updates on records touch more pages because attributes are stored across pages. Thus, there might be less concurrency under page-level locking. (This issue is relatively unimportant for data warehousing workloads.)

Query Optimizer. The cost model should be revised, considering the efficiency of sequential scans under MBSM. Catalog information can be used during optimization. The I/O cost is (essentially) the cost of retrieving just the referenced pages.

6 Experimental Evaluation

In this section, we evaluate and compare four data placement schemes: MBSM, NSM, DSM and PAX on both synthetic and TPC-H based workloads. We conduct the experiments on a Dell Precision 330 PC, with a 1.8Mhz Pentium 4 CPU and 1G Rambus Memory. This computer is running the RedHat Linux 7.1 operating system. We treat the two SCSI disks as raw devices. Our experiments read or write directly from or directly to the devices independent of the filesystem. Before operations, we clear both the disk cache and the operating system buffer caches. The size of a memory page and a disk page is 8KB.

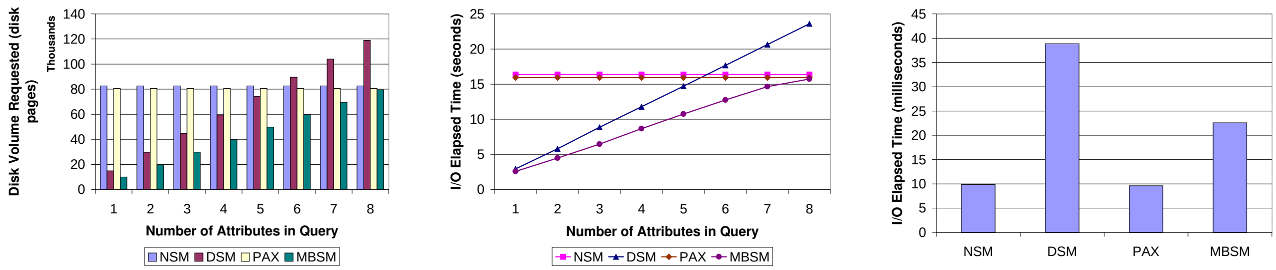
MBSM stores attributes in a vertically partitioned manner similar to PAX, and thus shares PAX's good CPU cache behavior (which is thoroughly studied in [1]). PAX saves at least 75% of NSM's cache stall time, and it is also better than DSM. Due to space limitations, we do not show cache-related experiments. [1] also demonstrates that DSM incurs high reconstruction cost. Our experiments directly simulate the I/O behavior by generating I/O operations on raw disk devices; we do not employ a database system.

6.1 Projectivity Analysis

To demonstrate how MBSM works for different query projectivity, we create a workload consisting of one relation R and variations of queries which involve different numbers of attributes. Relation R contains eight 8-byte attributes and is populated with 10 million records. The super-block in MBSM has eight pages. Values for one attribute go to one page in a super-block. This is a synthetic "best-case" for MBSM because there is no fragmentation. This example is used for illustration of the potential performance benefits. More realistic workloads are considered in Section 6.2.

We vary the query projectivity from one attribute to all eight attributes. Figure 5(a) shows the disk volume requested to answer different queries. MBSM has the least disk volume requested for each query. For NSM and PAX, all the disk pages must be requested for all the queries. DSM has the most disk volume request when the query involves more than six out of eight attributes. Note that the final columns of Figure 5(a) show the disk space used for each scheme. DSM uses the most space because there is a 4-byte logical record ID for each record in every sub-relation. Our scheme used slightly less space than PAX because we save the space of pointers to minipages within PAX pages.

To gain better I/O performance, we assume both NSM and PAX pages are stored contiguously on disk. For DSM, each sub-relation is stored as a contiguous big file and sub-relations are stored one after another. This disk organization actually favors DSM. In practice, DSM sub-relations can be stored at different locations on disk. MBSM stores the records in mega-blocks of size 4 MB, which translates into a run length r of 64 pages. Figure 5(b) compares the speed of four schemes as a function of the number of attributes in the query for an implementation on the Seagate disk. Although MBSM requests the fewest disk pages for all the queries, it is about the same as DSM at first. This is because for



(a) Disk Volume (b) Query Performance (c) Single-record Insert/Delete

Figure 5. Synthetic Workloads

the first query, all the pages requested for DSM are from one sub-relation and that sub-relation is stored contiguously and is accessed sequentially. In MBSM, records are stored in mega-blocks. It reads one run in a mega-block, skips the remaining runs and goes to the next mega-block and so on. This slows the throughput of the disk slightly. As the number of attributes increases, MBSM reads more tracks in a mega-block and the speed overhead is less important than the fewer disk pages transferred. (Similar results were observed on the Quantum disk.) We don't include the record reconstruction cost for DSM in the comparison. As the number of attributes increases, we expect the overall cost of DSM to be much higher [1].

Figure 5(c) compares the performance of inserting or deleting a record in the four schemes. We assume that before the update, the disk head is located at the beginning of the corresponding disk file. Under NSM and PAX, only one random disk write for the block containing the new or old record is required. Both MBSM and DSM require eight disk writes. However, the performance is different in that MBSM has one random disk write and the remaining writes are in adjacent or nearby tracks. The results shown are favorable to DSM since sub-relation files are stored together. If sub-relation files are located at different places on the disk, the cost can be much higher. A random write on this disk takes about 10 milliseconds.

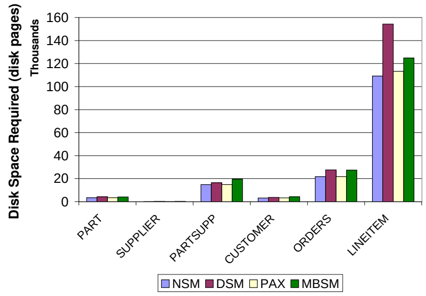
6.2 TPC-H Workloads

This section compares the different storage models when running a decision-support workload based on TPC-H. We conducted experiments on a factor 1 TPC-H database, generated using the *dbgen* software distributed by the TPC [18]. The database includes attributes of type integer, floating point, date, fixed-length string and variable-length string. We convert variable-length attributes to fixed-length attributes by allocating space equal to their maximum size. The workload consists of all 22 TPC-H queries and we assume each of them has the same probability of execution.

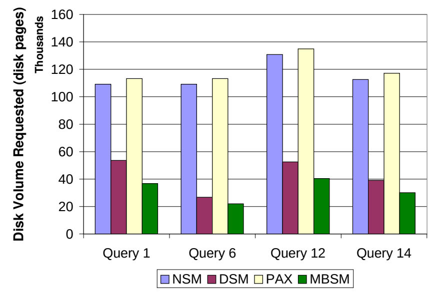
We do not actually execute the 22 queries. Instead, we reformulate the actual queries into "abstract" queries. This reformulation allows us to focus on the aspects of the query most relevant to MBSM, namely which relations are scanned, and which attributes are accessed in the same query. An abstract query is a set of entries of the form "table-name.column-name". (We omit the table name when the table is clear.) For a TPC-H query, every column syntactically mentioned in the query is part of this set. For example, Query Q14 from TPC-H (given in Appendix A) is reformulated as {p_type, p_partkey, l_partkey, l_extendedprice, l_discount, l_shipdate}.

We interpret an abstract query as requiring a single scan through all records in the referenced tables, accessing at least the attributes mentioned in the set. That the queries require scans is reasonable since we are assuming a decision-support workload in which a large fraction of all records are touched by most queries. Index-based access is usually not competitive due to random I/O. That the queries require *single* scans is an approximation. It is conceivable that the data is sufficiently large that multiple scans would be required for hash joins, for example. Nevertheless, we believe that abstract queries are sufficiently descriptive to capture important aspects of the workload.

We generate the attribute affinity for each relation from the workload. The attribute affinity threshold is set empirically as 4: if any two attributes from the same relation appear together in more than 4 out of the 22 queries, we try to assign them into one page in the super-block. For variable-length attributes, we use their maximum size as the attribute size. Table 3 shows the super-block size p chosen by the optimization algorithm for the six largest TPC-H tables. (The tables NATION and REGION are too small to be partitioned further.) Recall that we varied p from 1 to 17, as discussed in Section 4.2. Details of some of the assignments can be found in Appendix A. The fragmentation represents the total proportion of wasted space, not including wasted space due to variable-length attributes being smaller than their maximum size.



(a) Disk Space Used



(b) Disk Volume Requested Per Query

Figure 6. Storage Performance for TPC-H Workloads

Table	Columns	Pages per Super-Block	Frag. Overhead
PART	9	17	3.53%
SUPPLIER	7	17	4.62%
PARTSUPP	5	11	0.45%
CUSTOMER	8	13	4.7%
ORDERS	9	15	4.67%
LINEITEM	16	17	5.27%

Table 3. TPC-H Table Partitions

Figure 6(a) shows the disk space required for the six largest TPC-H tables in the different storage models. DSM uses significantly more space than the others for table LINEITEM because the table consists of 16 relatively small attributes, so the 4-byte surrogate overhead cannot be ignored. Our scheme uses a little more space than NSM and PAX for two reasons. First, there could be some fragmentation within the super-blocks because we cannot guarantee each page within a super-block is exactly full. Second, we use the maximum possible size for variable-length attributes. The actual average size is smaller than the maximum size, and all of PAX, DSM and NSM use only the space required by the actual attribute value.

The layouts of Table 3 are optimized for the overall workload of all 22 queries. We choose four TPC-H queries, Q1, Q6, Q12 and Q14, as examples to demonstrate individual query performance on this layout. Queries 1 and 6 are range queries on the LINEITEM table, with multiple aggregates and predicates. Queries 12 and 14 are equijoins of LINEITEM with another table; they involve additional selection predicates, and they compute conditional aggregates.

Figure 6(b) shows the disk volume requested for the four queries in the different schemes. Figure 7(a) shows the I/O elapsed time for the four queries. Figure 7(c) shows the average I/O elapsed time for the 22 queries. Both DSM and MBSM read fewer disk pages and perform faster, compared to NSM and PAX. While DSM looks competitive, remember that we have excluded the record reconstruction cost. If this cost is included, the overall cost of DSM is much higher than MBSM. Figure 7(b) shows the performance of single-record insert/delete operation on the biggest table LINEITEM. Both NSM and PAX requires only one block write, while DSM and MBSM each require for 16 block writes. MBSM is faster than DSM due to the proximity of the block that need to be written.

Figure 7(d) shows how the quality of the layout varies as we adjust p and r, keeping $p * r$ (i.e., the mega-block size) bounded by 4MB. Given a run size, the y-axis shows the workload average I/O elapsed time for the best table layouts. With larger r, the potential I/O speed increases. But smaller p means that more attributes could be stored in one page in a super-block. Queries which involve only a few attributes may end up requiring more disk pages than necessary. As we can see, $r = 30$ seems to balance the competing aims; this is the value used for the previous experiments.[2]

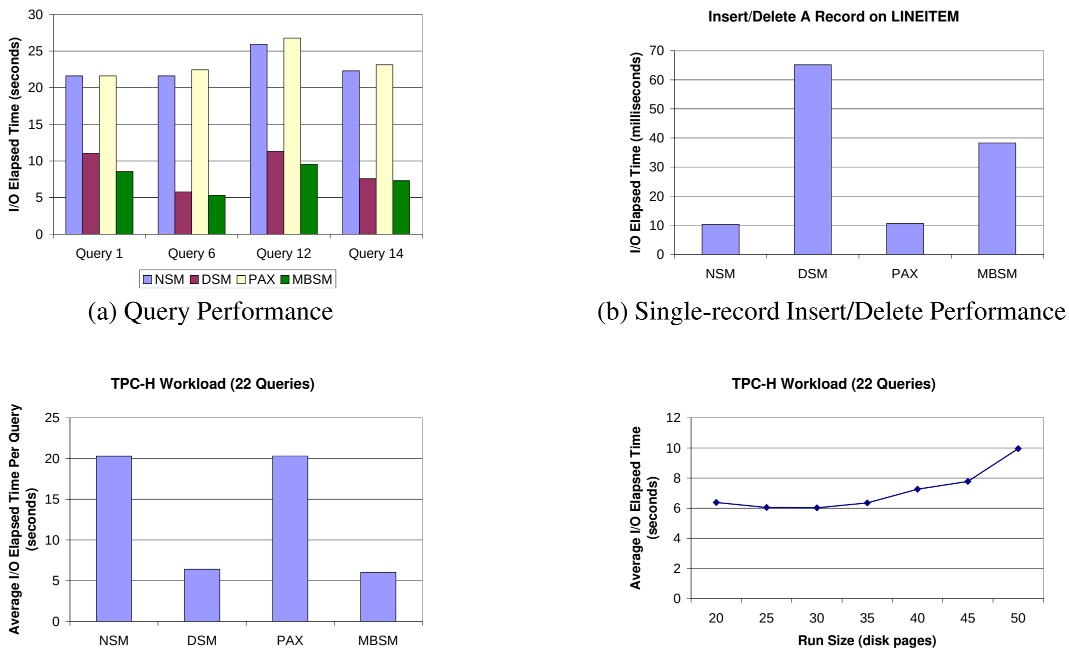
As a final note, we observed that for large p, the use of attribute affinity for designing a super-block becomes relatively unimportant, because most of the pages store at most one attribute.

7 Conclusion and Future Work

We have proposed a new storage model called MBSM, which stores records in a partitioned way in a super-blocks, and then organizes super-blocks on disk into mega-blocks. MBSM is most suitable for decision-support workloads that frequently execute table scans.

- Compared to NSM and PAX, MBSM requests fewer disk pages and uses 70% less I/O processing time for a decision-support workload involving table scans. MBSM shares PAX's good CPU cache behavior.
- Compared to DSM, MBSM's scan performance is comparable. However, MBSM's cache performance is better because no surrogates are involved and MBSM

[2]The results were qualitatively similar for the Quantum disk, but a larger value of r was optimal.



(a) Query Performance

(b) Single-record Insert/Delete Performance

(c) Average Query Performance

(d) Mega-Block Design Trade-off

Figure 7. Performance for TPC-H Workloads

has better insert/update I/O performance. Further, MBSM doesn't require a join to reconstruct the records while DSM has high reconstruction cost [1].

We plan to investigate several directions in future research. Queries seldom use large variable-length attributes, such as "comment" etc. It could be a better idea to store these attributes separately in DSM, and to use MBSM for the fixed-sized attributes (with one surrogate). Alternatively, one could store variable-length attributes in a pointer-based way using a heap to avoid wasted space. For this second option, one needs a method to handle page overflows, and one may not be able to uniquely identify the physical page containing the required attribute. It is conceivable that a probabilistic analysis of attribute size and query reference patterns could do a reasonable job of ensuring good performance without wasting space.

Another direction for future work is disk page compression. Systems such as Sybase IQ actively use compression to reduce I/O in a DSM-like setting. We could apply similar techniques, compressing at the run level. Dealing with variable-sized mega-blocks would then become an issue.

Acknowledgement We thank Cliff Stein for references to the parallel scheduling problem and its approximation algorithms.

References

[1] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. Weaving relations for cache performance. In *Proceedings of VLDB Conference*, 2001.

[2] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. DBMSs on a modern processor: Where does time go? In *Proceedings of VLDB conference*, 1999.

[3] P. Boncz, S. Manegold, and M. Kersten. Database architecture optimized for the new bottleneck: Memory access. In *Proceedings of VLDB Conference*, 1999.

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

[5] 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), 1990.

[6] R. Graham. Bounds for certain multiprocessor anomalies. *Bell System Technical Journal*, 45:1563–1581, 1966.

[7] R. Graham, E. Lawler, J. Lenstra, and A. Kan. Optimization and approximation in deterministic sequencing and scheduling: a survey. *Annals of Discrete Mathematics*, 5:287–326, 1979.

[8] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal of Applied Mathematics*, 17(2):416–429, 1969.

[9] C. M. Kozierok. The PC guide, 2002. Available via http://www.pcguide.com/.

[10] S. Manegold, P. Boncz, and M. Kersten. What happens during a join? Dissecting CPU and memory optimization effects. In *Proceedings of VLDB Conference*, 2000.

[11] Maxtor Corp. Atlas 10K II overview, 2002. Available from http://www.maxtor.com/.

[12] P. O'Neil and D. Quass. Improved query performance with variant indexes. In *Proceedings of ACM SIGMOD Conference*, 1997.

[13] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill, 2 edition, 2000.

[14] R. Ramamurthy, D. J. DeWitt, and Q. Su. A case for fractured mirrors. In *Proceedings of VLDB conference*, 2002.

[15] C. Ruemmler and J. Wilkes. An introdurction to disk drive modeling. *IEEE Computer*, 27(3), 1994.

[16] Seagate Corp. Cheetah X15 product manual, volume 1, 2002. Available via http://www.seagate.com/.

[17] A. Shatdal, C. Kant, and J. F. Naughton. Cache conscious algorithms for relational query processing. In *Proceedings of VLDB Conference*, pages 510–521, 1994.

[18] Transaction Processing Performance Council. TPC Benchmark H. Available via http://www.tpc.com/tpch/.

A TPC-H Layout under MBSM

For reference, TPC-H query Q14 is given below.

```
select 100.00 * sum(case
      when p_type like 'PROMO%'
      then l_extendedprice*(1-l_discount)
      else 0 end) /
      sum(l_extendedprice * (1 - l_discount))
      as promo_revenue
from lineitem, part
where l_partkey = p_partkey
      and l_shipdate >= date '[DATE]'
      and l_shipdate < date '[DATE]' +
          interval '1' month;
```

Table 4 shows how the LINEITEM table was divided among 17 pages. The rows are the attributes, and the columns are page numbers within a super-block. For an attribute of size s, an entry of b means that b/s of the attribute values are stored in that page. (The rows should add to s.)

Table 5 shows the same LINEITEM table divided instead among 9 pages. This scheme corresponds to the best allocation for the point $r = 50$ in Figure 7(d).

Page Number	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
l_orderkey				4													
l_partkey			4														
l_suppkey				4													
l_linenumber																	4
l_quantity							8										
l_extendedprice					8												
l_discount						8											
l_tax												8					
l_returnflag						1											
l_linestatus							1										
l_shipdate								10									
l_commitdate									10								
l_receiptdate										10							
l_shipinstruct	10	10	5														
l_shipmode											10						
l_comment													10	10	10	10	4
total	10	10	9	8	8	9	9	10	10	10	10	8	10	10	10	10	8

Table 4. LINEITEM Attribute Placement (17-Page Super-Block)

Page Number	1	2	3	4	5	6	7	8	9
l_orderkey	4								
l_partkey		4							
l_suppkey	4								
l_linenumber		4							
l_quantity							8		
l_extendedprice						8			
l_discount						8			
l_tax							8		
l_returnflag						1			
l_linestatus						1			
l_shipdate	10								
l_commitdate		10							
l_receiptdate			10						
l_shipinstruct								18	7
l_shipmode									10
l_comment			8	18	18				
total	18	18	18	18	18	18	16	18	17

Table 5. LINEITEM Attribute Placement (9-Page Super-Block)