Weaving Relations for Cache Performance

Anastassia Ailamaki [‡]
Carnegie Mellon University
natassa@cs.cmu.edu

David J. DeWitt
Univ. of Wisconsin-Madison
dewitt@cs.wisc.edu

Mark D. Hill
Univ. of Wisconsin-Madison
markhill@cs.wisc.edu

Marios Skounakis
Univ. of Wisconsin-Madison
marios@cs.wisc.edu

Abstract

Relational database systems have traditionally optimzed for I/O performance and organized records sequentially on disk pages using the N-ary Storage Model (NSM) (a.k.a., slotted pages). Recent research, however, indicates that cache utilization and performance is becoming increasingly important on modern platforms. In this paper, we first demonstrate that in-page data placement is the key to high cache performance and that NSM exhibits low cache utilization on modern platforms. Next, we propose a new data organization model called PAX (Partition Attributes Across), that significantly improves cache performance by grouping together all values of each attribute within each page. Because PAX only affects layout inside the pages, it incurs no storage penalty and does not affect I/O behavior. According to our experimental results, when compared to NSM (a) PAX exhibits superior cache and memory bandwidth utilization, saving at least 75% of NSM's stall time due to data cache accesses, (b) range selection queries and updates on memory-resident relations execute 17-25% faster, and (c) TPC-H queries involving I/O execute 11-48% faster.

1 Introduction

The communication between the CPU and the secondary storage (I/O) has been traditionally recognized as the major database performance bottleneck. To optimize data transfer to and from mass storage, relational DBMSs have long organized records in 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 [27].

Unfortunately, most queries use only a fraction of each record. To minimize unnecessary I/O, the Decomposition Storage Model (DSM) was proposed in 1985 [10]. DSM partitions an n-attribute relation vertically into n sub-relations, each of which is accessed only when the corresponding attribute is needed. Queries that involve multiple attributes from a relation, however, must spend tremendous additional time to join the participating sub-relations together. Except for Sybase-IQ [33], today's relational DBMSs use NSM for general-purpose data placement [20][29][32].

Recent research has demonstrated that modern database workloads, such as decision support systems and spatial applications, are often bound by delays related to the processor and the memory subsystem rather than I/O [20][5][26]. When running commercial database systems on a modern processor, data requests that miss in the cache hierarchy (i.e., requests for data that are not found in any of the caches and are transferred from main memory) are a key memory bottleneck [1]. In addition, only a fraction of the data transferred to the cache is useful to the query: the item that the query processing algorithm requests and the transfer unit between the memory and the processor are typically not the same size. Loading the cache with useless data (a) wastes bandwidth, (b) pollutes the cache, and (c) possibly forces replacement of information that may be needed in the future, incurring even more delays. The challenge is to repair NSM's cache behavior without compromising its advantages over DSM.

This paper introduces and evaluates **Partition Attributes Across (PAX)**, a new layout for data records that combines the best of the two worlds and exhibits performance superior to both placement schemes by eliminating unnecessary accesses to main memory. For a given relation, PAX stores the same data on each page as NSM. *Within* each page, however, PAX groups all the values of a particular attribute together on a minipage. During a sequential scan (e.g., to apply a predicate on a fraction of the record), PAX fully utilizes the cache resources, because on each miss a number of a single attribute's values are loaded into the cache together. At the same time, all parts of the record are on the same page. To reconstruct a record one needs to perform a *mini-join* among minipages, which incurs minimal cost because it does not have to look beyond the page.

We evaluated PAX against NSM and DSM using (a) predicate selection queries on numeric data and (b) a variety of queries on TPC-H datasets on top of the Shore storage manager [7]. We vary query parameters including selectivity, projectivity, number of predicates, distance between the projected attribute and the attribute in the predicate, and degree of the relation. The experimental results show that, when compared to NSM, PAX (a) incurs 50-75% fewer second-level cache misses due to data

[‡] Work done while author was at the University of Wisconsin-Madison.

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

Proceedings of the 27th VLDB Conference,
Roma, Italy, 2001

accesses when executing a main-memory workload, (b) executes range selection queries and updates in 17-25% less elapsed time, and (c) executes TPC-H queries involving I/O 11-42% faster than NSM on the platform we studied. When compared to DSM, PAX executes queries faster and its execution time remains stable as more attributes are involved in the query, while DSM's execution time increases due to the high record reconstruction cost.

Finally, PAX has several additional advantages. Implementing PAX on a DBMS that uses NSM requires only changes to the page-level data manipulation code. PAX can be used as an alternative data layout, and the storage manager can decide to use PAX or not when storing a relation, based solely on the number of attributes. Furthermore, research [13] has shown that compression algorithms work better with vertically partitioned relations and on a per-page basis, and PAX has both of these characteristics. Finally, PAX can be used orthogonally to other storage decisions such as affinity graph-based partitioning [11], because it operates at the page level.

The rest of this paper is organized as follows. Section 2 presents an overview of the related work, and discusses the strengths and weaknesses of the traditional NSM and DSM data placement schemes. Section 3 explains the design of PAX in detail and analyzes its storage requirements, while Section 4 describes the implementation of basic query processing and data manipulation algorithms. Section 5 analyzes the effects of PAX on cache performance on a simple numeric workload. Section 6 demonstrates PAX's efficiency on a subset of the TPC-H decision-support workload. Finally, Section 7 concludes with a summary of the advantages of PAX and discusses possibilities for further improvement.

2 Related work

Several recent workload characterization studies report that database systems suffer from high memory-related processor delays when running on modern platforms. A detailed survey of these studies is provided elsewhere [1][34]. All studies that we are aware of agree that stall time due to data cache misses accounts for 50-70% (OLTP [19]) to 90% (DSS [1]) of the total memory-related stall time, even on architectures where the instruction cache miss rate (i.e., the number of cache misses divided by the number of cache references) is typically higher when executing OLTP workloads [21].

Research in computer architecture, compilers, and database systems has focused on optimizing data placement for cache performance. A compiler-directed approach for cache-conscious data placement profiles a program and applies heuristic algorithms to find a placement solution that optimizes cache utilization [6]. Clustering, compression, and coloring are techniques that can be applied manually by programmers to improve cache performance of pointer-based data structures [8]. For database management systems, attribute clustering improves both compression [13] and the performance of relational query processing [29].

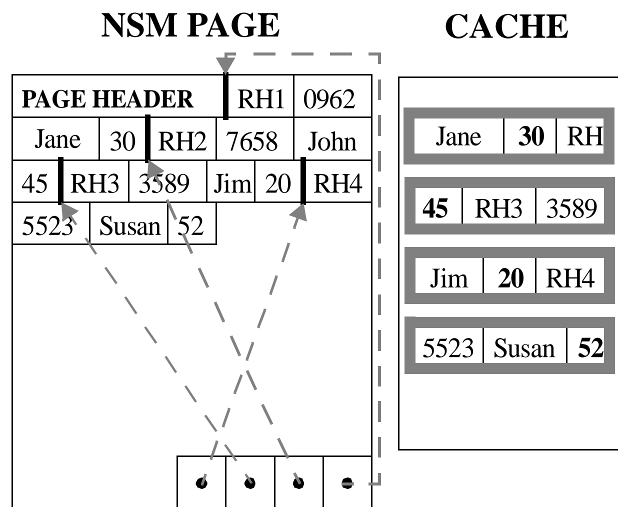

FIGURE 1: *The cache behavior of NSM.*

The remainder of this section describes the advantages and disadvantages of the dominant data placement scheme (NSM) and its alternative (DSM), and briefly outlines their variants.

2.1 The N-ary Storage Model

Traditionally, a relation's records are stored in slotted disk pages [27] obeying the N-ary Storage Model (NSM). NSM stores records sequentially on data pages. Figure 1 depicts an NSM page after inserting four records of a relation R with three attributes: SSN, name, and age. Each record has a record header (RH) containing a null bitmap, offsets to the variable-length values, and other implementation-specific information [20][29][32]. 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. One can access the nth record in a page by following the nth pointer from the end of the page.

During predicate evaluation, however, NSM exhibits poor cache performance. Consider the query:

select *name*
from *R*
where *age* < 40;

To evaluate the predicate, the query processor uses a scan operator [14] that retrieves the value of the attribute *age* from each record in the relation. Assuming that the NSM page in Figure 1 is already in main memory and that the cache block size is smaller than the record size, the scan operator will incur one cache miss per record. If *age* is a 4-byte integer, it is smaller than the typical cache block size

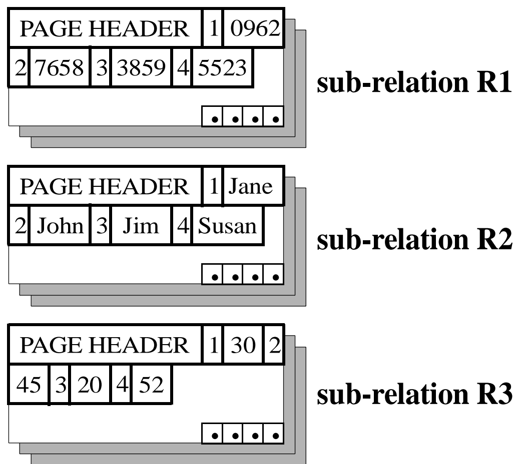

FIGURE 2: *The Decomposition Storage Model (DSM).*

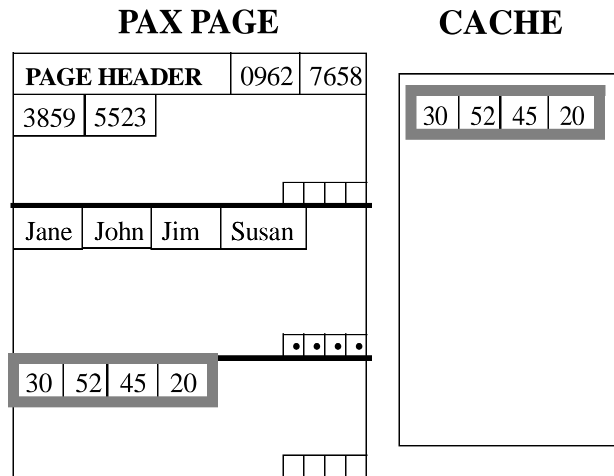

FIGURE 3: *The cache behavior of PAX.*

(32-128 bytes). Therefore, along with the needed value, each cache miss will bring into the cache the other values stored next to *age* (shown on the right in Figure 1), wasting useful cache space to store unreferenced data, and incurring unnecessary accesses to main memory.

2.2 The Decomposition Storage Model

Vertical partitioning is the process of striping a relation into sub-relations, each containing the values of a subset of the initial relation's attributes, in order to reduce I/O-related costs [24]. The fully decomposed form of vertical partitioning (one attribute per stripe) is called the Decomposition Storage Model (DSM) [10]. DSM partitions an *n*-attribute relation vertically into *n* sub-relations (for example, Figure 2 shows relation R stored in DSM format). Each sub-relation contains two attributes, a logical record id (surrogate) and an attribute value (essentially, a sub-relation is a clustered index on the attribute). Sub-relations are stored as regular relations in slotted pages, enabling each attribute to be scanned independently.

Unlike NSM, DSM offers a high degree of spatial locality when sequentially accessing the values of one attribute. During a single-attribute scan, DSM exhibits high I/O and cache performance. DSM performs well on DSS workloads known to utilize a small percentage of the attributes in a relation, as in Sybase-IQ which uses vertical partitioning combined with bitmap indices for warehousing applications [25]. In addition, DSM can improve cache performance of main-memory database systems, assuming that the record reconstruction cost is low [4].

Unfortunately, DSM's performance deteriorates significantly for queries that involve multiple attributes from each participating relation. The database system must join the participating sub-relations on the surrogate to reconstruct the partitioned records. The time spent joining sub-relations increases with the number of attributes in the result relation. An alternative algorithm [11] partitions each relation based on an attribute affinity graph, which connects pairs of attributes based on how often they appear together in queries. The performance of affinity-based vertical partitioning depends heavily on whether queries involve attributes within the same fragment, significantly limiting its capability.

3 PAX

In this section, we introduce a new strategy for placing records on a page called PAX (Partition Attributes Across). PAX (a) maximizes inter-record spatial locality within each column in the page, thereby eliminating unnecessary requests to main memory without a space penalty, (b) incurs a minimal record reconstruction cost, and (c) is orthogonal to other design decisions because it only affects the layout of data stored on a single page (e.g., one may decide to store one relation using NSM and another using PAX, or first use affinity-based vertical partitioning, and then use PAX for storing the 'thick' sub-relations). This section presents PAX's detailed design.

3.1 Overview

The motivation behind PAX is to keep the attribute values of each record on the same page as in NSM, while using a cache-friendly algorithm for placing them inside the page. PAX vertically partitions the records *within each page*, storing together the values of each attribute in minipages. Figure 3 depicts a PAX page that stores the same records as the NSM page in Figure 1 in a column-major fashion. When using PAX, each record resides on the same page as it would reside using NSM, but all *SSN* values, *name* values, and *age* values are grouped together on minipages respectively. PAX increases the inter-record spatial locality (because it groups values of the same attribute that belong to different records) with minimal impact on the intra-record spatial locality. Although PAX employs in-

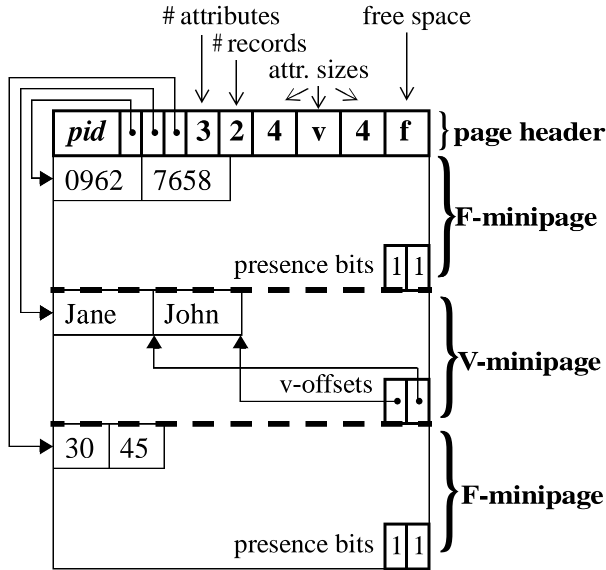

FIGURE 4: *An example PAX page.*

page vertical partitioning, it incurs minimal record reconstruction costs, because it does not need to perform a join to construct a result tuple.

3.2 Design

To store a relation with degree n (i.e., with n attributes), PAX partitions each page into n minipages. It then stores values of the first attribute in the first minipage, values of the second attribute in the second minipage, and so on. The page header at the beginning of each page contains pointers to the beginning of each minipage. The record header information is distributed across the minipages. The structure of each minipage is determined as follows:

- Fixed-length attribute values are stored in F-minipages. At the end of each F-minipage there is a presence bit vector with one entry per record that denotes null values for nullable attributes.
- Variable-length attribute values are stored in V-minipages. V-minipages are slotted, with pointers to the end of each value. Null values are denoted by null pointers.

Each newly allocated page contains a page header and as many minipages as the degree of the relation. The page header contains the number of attributes, the attribute sizes (for fixed length attributes), offsets to the beginning of the minipages, the current number of records on the page and the total space available on the page. Figure 4 depicts an example PAX page in which two records have been inserted. There are two F-minipages, one for the *SSN* attribute and one for the *age* attribute. Because the *name* attribute is a variable-length string, it is stored in a V-minipage. At the end of each V-minipage there are offsets to the end of each variable-length value.

Records on a page are accessed either sequentially or in random order (e.g., through a non-clustered index). To sequentially access a subset of attributes, the algorithm accesses the values in the appropriate minipages. For instance, the sequential scan algorithm reads all values of a fixed-length attribute f or a variable-length attribute v from a newly accessed page, and the indexed scan reads a value of an attribute a, given the record id [34].

To store a relation, PAX requires the same amount of space as NSM. NSM stores the attributes of each record contiguously, and therefore it requires one offset (slot) per record and one additional offset for each variable-length attribute in each record. In contrast, PAX stores one offset for each variable-length value, plus one offset for each of the n minipages. Therefore, regardless of whether a relation is stored using NSM of PAX, it will occupy the same number of pages. As explained in Section 4.1, implementation-specific details may introduce slight differences which are insignificant to the overall performance.

3.3 Evaluation

The data placement scheme determines two factors that affect performance. First, the *inter-record spatial locality* minimizes data cache-related delays when executing iterators over a subset of fields in the record. DSM provides inter-record spatial locality, because it stores attributes contiguously, whereas NSM does not. Second, the *record reconstruction cost* minimizes the delays associated with retrieving multiple fields of the same record.

TABLE 1: *NSM, DSM, and PAX comparison.*

Characteristic	NSM	DSM	PAX
Inter-record spatial locality		✓	✓
Low record reconstruction cost	✓		✓

Table 1 summarizes the characteristics of NSM, DSM, and PAX, demonstrating the tradeoff between inter-record spatial locality and record reconstruction cost. NSM exhibits suboptimal cache behavior. In contrast, DSM requires costly joins that offset the benefit from the inter-record spatial locality. PAX offers the best of both worlds by combining the two critical characteristics: inter-record spatial locality, and minimal record reconstruction overhead, by keeping all parts of each record in the same page. As an additional advantage, implementing PAX on an existing DBMS requires only changes to the page-level data manipulation code.

4 System Implementation

We implemented NSM, PAX, and DSM in the Shore storage manager [7]. Shore provides all the features of a modern storage manager, namely B-trees and R-trees [16], ARIES-style recovery [22], hierarchical locking (record,

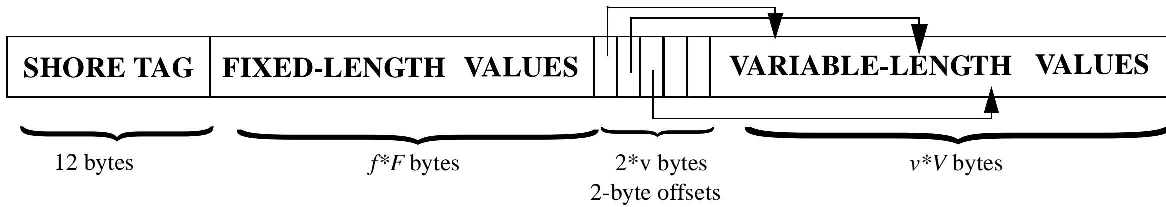

FIGURE 5: *An example NSM record structure in Shore.*

page, file), and clock-hand buffer management with hints. Typically, Shore stores records as contiguous byte sequences.

We implemented NSM, DSM and PAX as alternative data layouts in Shore. To implement NSM, we added attribute-level functionality on top of the existing Shore file manager (as explained later in this section). DSM was implemented on top of Shore, by decomposing the initial relation into *n* Shore files that are stored in slotted pages using NSM. Each sub-relation includes two columns, one with a logical record id and one with the attribute value. Finally, PAX was implemented as an alternative data page organization in Shore. All schemes use record-level locking and the ARIES-style logging and recovery mechanism provided by Shore. Only minor modifications to the recovery system were needed to handle PAX records. In the rest of this section, we discuss record implementation and the most relevant data manipulation and retrieval algorithms.

4.1 Record Implementation

Figure 5 illustrates how NSM records were implemented. The fixed-length attribute values are stored first, followed by an array of offsets and a mini-heap containing the variable-length attribute values. In front of each record, Shore adds a 12-byte tag with Shore-specific information such as serial number, record type, header length, and record length. In addition, it uses 4-byte slots at the end of the page (two bytes for the offset and another two for the amount of space allocated for each record). This adds up to a 16-byte overhead per record.

In the current implementation, TPC-H tables stored using PAX need 8% less space than those stored using NSM. Up to half of the space saved in PAX is due to eliminating the slot table at the end of a page. The rest is Shore-specific overhead resulting from the 12-byte record tag. Commercial DBMSs store a header in front of each record, that keeps information such as the NULL bitmap, space allocated for the record, true record size, fixed part size, and other flags. The record header's size varies with the number and type of columns in the table. For the TPC-H table *Lineitem*, the record header would be about 8 bytes. Therefore, the Shore tag adds a space overhead of 4 bytes per record. Due to this overhead, NSM takes 4% more storage than it would if the Shore tag were replaced with common NSM header information.

4.2 Data Manipulation Algorithms

BULK-LOADING AND INSERTIONS. The algorithm to bulk-load records from a data file starts by allocating each minipage on the page based on attribute value size. In the case of variable-length attributes, it initially uses a hint (either from the DBMS or the average length from the first few records) as an indication of the average value size, and then uses feedback from the actual size of inserted values to adjust the average value size. PAX inserts records by copying each value into the appropriate minipage. When variable-length values are present, minipage boundaries may need to be adjusted to accommodate records as they are inserted in the page. If a record fits in the page but its individual attribute values do not, the algorithm recalculates minipage sizes based on the average value sizes in the page so far and the new record size, and reorganizes the page structure by moving minipage boundaries appropriately to accommodate new records. When the page is full, it allocates a new page with the initial minipage sizes equal to the ones in the previously populated page (so the initial hints are quickly readjusted to the true per-page average value sizes).

Shore supports inserting a record into the first page that can accomodate it (i.e., not necessarily appending records at the end of the relation). The NSM insertion algorithm concatenates record values into the byte sequence presented in Figure 5. Shore then adds a tag and stores the record. PAX calculates the position of each attribute value on the page, stores the value, and updates the presence bitmaps and offset arrays accordingly.

UPDATES. NSM uses the underlying infrustructure provided by Shore, and updates attribute values within the record. Updates on variable-length attributes may stretch or shrink the record; page reorganizations may be needed to accomodate the change and the slot table must be updated. If the updated record grows beyond the free space available in the page, the record is moved to another page. PAX updates an attribute value by computing the offset of the attribute in the corresponding minipage. Variable-length attribute updates require only V-minipage-level reorganizations, to move values of the updated attribute only. If the new value is longer than the space available in the V-minipage, the V-minipage borrows space from a neighboring minipage. If the neighboring

minipages do not have sufficient space, the record is moved to a different page.

DELETIONS. The NSM deletion algorithm uses the slot array to mark deleted records, and the free space can be filled upon future insertions. PAX keeps track of deleted records using a bitmap at the beginning of the page, and determines whether a record has been deleted using fast bitwise calculations. Upon record deletion, PAX reorganizes minipage contents to fill the gaps to minimize fragmentation that could affect PAX's optimal cache utilization. As discussed in Section 6.2, minipage reorganization does not affect PAX's performance because it incurs minimal overhead. Attribute value offsets are calculated by converting the record id to the record index within the page, which takes into account deleted records.

For deletion-intensive workloads, an alternative approach is to mark records as deleted but defer reorganizations, leaving gaps in the minipages. As a result, cache utilization during file scan is suboptimal, because deleted record data are occasionally brought into the cache. To maintain cache performance to acceptable levels, the system schedules periodic file reorganizations. In the near future we plan to implement this algorithm and provide the option to use on a per-file basis the one best suited for each application. For the purposes of this study, however, we have implemented the first "exhaustive" alternative.

4.3 Query Operators

SCAN OPERATOR. A scan operator that supports sargable predicates [28] was implemented on top of Shore. When running a query using NSM, one scan operator is invoked that reads each record and extracts the attributes involved in the predicate from it. PAX invokes one scan operator for each attribute involved in the query. Each operator sequentially reads values from the corresponding minipage. The projected attribute values for qualifying records are retrieved from the corresponding minipages using computed offsets. With DSM, as many operators as there are attributes in the predicate are invoked, each on a sub-relation. The algorithm makes a list of the qualifying record ids, and retrieves the projected attribute values from the corresponding sub-relations through a B-tree index on record id.

JOIN OPERATOR. The adaptive dynamic hash join algorithm [23], which is also used in DB2 [20], was implemented on top of Shore. The algorithm partitions the left table into main-memory hash tables on the join attribute. When all available main memory has been consumed, all buckets but one are stored on the disk. Then it partitions the right table into hash tables in a similar fashion, probing dynamically the main-memory portion of the left table with the right join attribute values. Using only those attributes required by the query, it then builds hash tables with the resulting sub-records. The join operator receives its input from two scan operators, each reading one relation. The output can be filtered through a function that is passed as a parameter to the operator.

5 Analysis of the impact of data placement

To evaluate PAX's impact on cache performance, we first ran plain range selection queries on a memory-resident relation that consists of fixed-length numeric attributes. Such a controlled workload helped us understand the facts and perform sensitivity analysis. This section analyzes cache performance and execution time when running simple queries, and discusses the three schemes' limitations.

5.1 Setup and methodology

We conducted experiments on a Dell 6400 PII Xeon/MT system running Windows NT 4.0. This computer features a Pentium II Xeon processor running at 400MHz, 512MB of main memory, and a 100MHz system bus. The processor has split 16-KB first-level (L1) data and instruction caches and a unified 512-KB second-level (L2) cache. Caches at both levels are non-blocking (they can service new requests while earlier ones are still pending) with 32-byte cache blocks.[1] We obtained experimental results using the Xeon's hardware counters and the methodology described in previous work [1].

The workload consists of one relation and variations of the following range selection query:

select avg(a_p)
from R
where a_q > Lo and a_q < Hi (1)

where a_p, a_q are attributes in R. This query is sufficient to examine the net effect of each data layout when accessing records sequentially or randomly (given their record id). Unless otherwise stated, R contains eight 8-byte numeric attributes, and is populated with 1.2 million records. For predictability and easy correctness verification of experimental results, we chose the attribute size so that exactly four values fit in the 32-byte cache line, and record sizes so that record boundaries coincide with cache line boundaries. We varied the projectivity, the number of attributes in the selection predicate, their relative position, and the number of attributes in the record. The values of the attribute(s) used in the predicate are the same as in the *l_partkey* attribute of the *Lineitem* table in the TPC decision-support benchmarks [15], with the same data skew and uniform distribution.

PAX primarily targets optimizing data cache behavior, and does not affect I/O performance in any way. In

1. Other systems employ larger cache blocks (e.g., 64-128 bytes), especially in L2. In such systems, PAX's spatial locality will result in even higher cache and bandwidth utilization. Results with larger cache blocks are shown in [3].

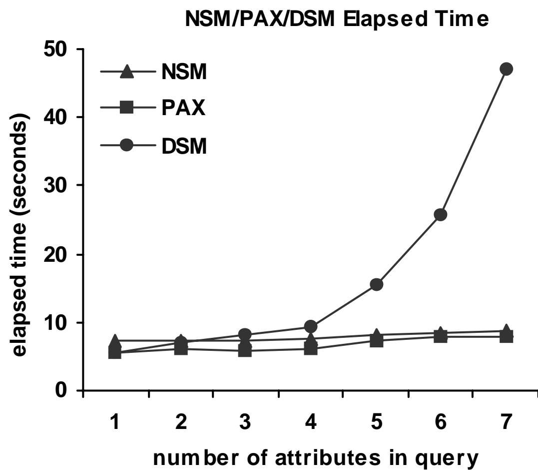

FIGURE 6: *Elapsed time comparison as a function of the number of attributes in the query.*

workloads where I/O latency dominates execution time, the performance of PAX and NSM eventually converge. PAX is designed to ensure high data cache performance *once the data page is available from the disk,* and is orthogonal to any additional optimizations to enhance I/O performance. In the rest of this section, we study the effects of the data placement scheme when running queries on memory-resident relations.

5.2 Experimental results

Figure 6 shows the elapsed time when using NSM, PAX, or DSM to run query (1), as the number of attributes involved in the query is varied from 1 to 7. The graph shows that, while the performance of the NSM and PAX schemes are relatively insensitive to the changes, DSM's performance is very sensitive to the number of attributes used in the query. When the query involves one or two attributes from R, DSM actually performs well because the record reconstruction cost is low. However, as the number of attributes involved increases, DSM's performance deteriorates rapidly because it joins more sub-relations together. On the contrary, NSM and PAX maintain stable performance, because all the attributes of each record reside on the same page, eliminating the need for an expensive join operation to reconstruct the record.

The rest of this section analyzes the significance of the data placement scheme to query execution performance. The first part discusses in detail the cache behavior of NSM and PAX, while the second part presents a performance sensitivity analysis for NSM and PAX as the query projectivity and the number of attributes in the predicate and the relation vary.

5.2.1 NSM vs. PAX Impact on Cache Behavior

Figure 7 illustrates that the cache behavior of PAX is significantly superior to that of NSM. As the predicate is applied to a_q, NSM suffers one cache miss per record. Because PAX groups attribute values together on a page, it only incurs a miss every n records, where n is the cache block size divided by the attribute size. In our experiments, PAX takes a miss every four records (i.e., 32 bytes per cache block divided by 8 bytes per attribute). Consequently, PAX saves about 75% of the data misses NSM incurs in the L2 cache.

PAX reduces cache delays related to data accesses, and therefore runs queries faster. The graphs on the left and center of Figure 7 show the processor stall time per record due to data misses at both cache levels for NSM and PAX, respectively (processing time is 100% during all of the experiments, and therefore processor cycles are proportional to elapsed time). Due to the higher spatial locality, PAX reduces the data-related penalty at both cache levels. The L1 data cache penalty does not affect the overall execution time significantly, because the penalty associated with one L1 data cache miss is small (10 processor cycles). Each L2 cache miss, however, costs 70-80

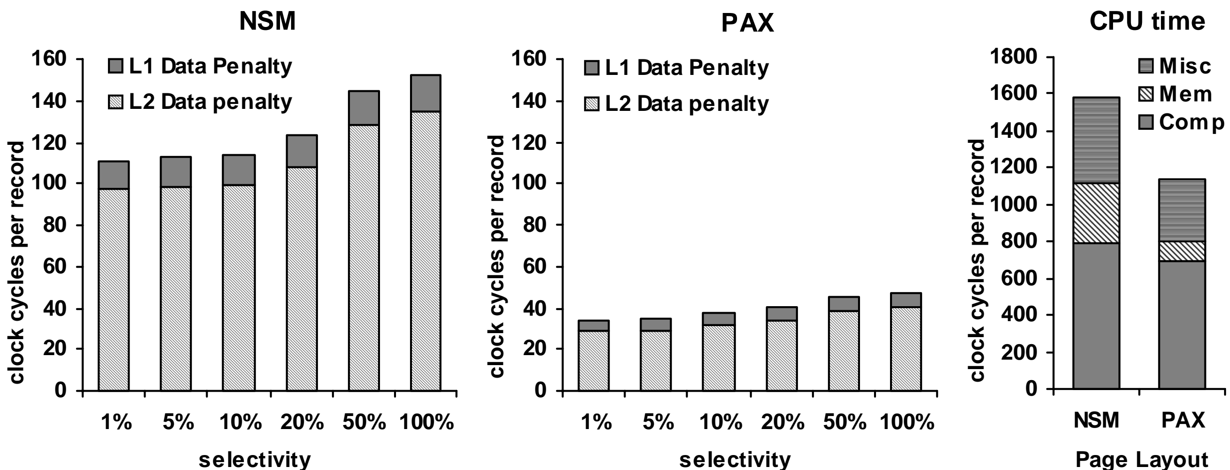

FIGURE 7: *PAX impact on memory stalls*

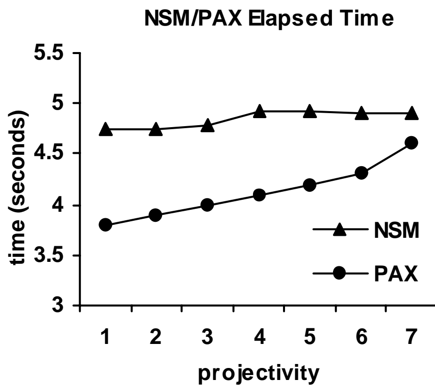

FIGURE 8: *PAX/NSM sensitivity to projectivity.*

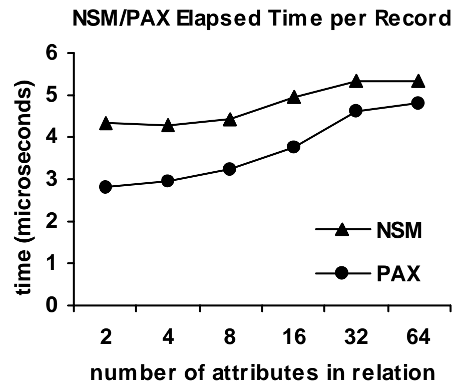

FIGURE 9: *PAX/NSM sensitivity to the number of attributes in the relation.*

cycles.[2] PAX reduces the overall L2 cache data miss penalty by 70%. Therefore, as shown in the graph on the right of Figure 7, the overall processor stall time is 75% less when using PAX, because the processor does not need to wait as long for data to arrive from main memory. The memory-related penalty contributes 22% to the execution time when using NSM, and only 10% when using PAX.

Reducing the data cache misses at both cache levels and minimizing the data stall time also reduces the number of instruction misses on the L2 cache, and minimizes the associated delays. L2 is organized as a unified cache and contains both data and instructions, potentially replacing each other as needed to accomodate new requests. To evaluate the predicate, NSM loads the cache with unreferenced data, which are likely to replace potentially needed instructions and incur extra instruction misses in the future. PAX only brings useful data into the cache, occupying less space and minimizing the probability to replace an instruction that will be needed in the future.

Although PAX and NSM have comparable instruction footprints, the rightmost graph of Figure 7 shows that PAX incurs less computation time. This is primarily due to reducing memory-related delays. Modern processors can retire[3] multiple instructions per cycle; the Xeon can retire up to three. When there are memory-related delays, the processor cannot operate at its peak retirement bandwidth. With NSM, only 30% of the total cycles retire three instructions, with 60% retiring either zero or one instruction. As reported by previous studies [1][19], database systems suffer high data dependencies and the majority of their computation cycles retire significantly fewer instructions than the processor's peak retirement bandwidth. PAX partially alleviates this problem by reducing the stall time, and the queries execute faster because they exploit better the processor's superscalar ability.

2. The corresponding latency on a Pentium 4 processor is more than 300 cycles (source: Intel Corporation).
3. Xeon's pipeline is capable of simultaneously issuing (reading) three instructions, executing five instructions (that were issued in previous cycles) and retiring (writing results onto registers and cache) three reviously issued instructions.

5.2.2 NSM/PAX Sensitivity Analysis

As the number of attributes involved in the query increases, the elapsed execution times of NSM and PAX converge. Figure 8 depicts the query execution time when the projectivity varies from 1 to 7 attributes, and the predicate is applied to the eighth attribute (selectivity is 50%). In the experiments shown, PAX is faster even when the resulting relation includes all the attributes. The reason is that the selectivity is maintained at 50%, and PAX exploits spatial locality in both the predicates and the projected attributes. In contrast, NSM brings useless information into the cache 50% of the time. Varying the number of attributes in the selection predicate has a similar effect [34]. In these experiments, DSM's performance is about a factor of nine slower than NSM and PAX. As the number of attributes involved in the query increases, DSM must join the corresponding number of sub-relations.

Figure 9 shows the elapsed time PAX and NSM need to process each record, as a function of the number of attributes in the record. More attributes per record implies fewer records per page. To keep the relation memory-resident, doubling the record size implies halving R's cardinality. Therefore, the execution times are normalized by R's cardinality. The graph illustrates that, while PAX still suffers fewer misses than NSM, the execution time is dominated by factors other than data stall time, such as the buffer manager overhead associated with getting the next page in the relation after completing the scan of the current page. Therefore, as the degree of the relation increases, the times PAX and NSM need to process a record converge.

6 Evaluation Using DSS Workloads

This section compares PAX and NSM when running a TPC-H decision-support workload. Decision-support applications are typically memory and computation intensive [20]. The relations are not generally memory-resident,

and the queries execute projections, selections, aggregates, and joins. The results show that PAX outperforms NSM on all TPC-H queries in this workload.

6.1 Setup and methodology

We conducted experiments on the system described in Section 5.1, using a 128-MB buffer pool and a 64-MB hash join heap. The workload consists of the TPC-H database and a variety of queries. The database and the TPC-H queries were generated using the *dbgen* and *qgen* software distributed by the TPC. The database includes attributes of type integer, floating point, date, fixed-length string, and variable-length string. We present results for bulk-loading, range selections, four TPC-H queries, and updates:

Bulk-loading. When populating tables from data files, NSM performs one memory-to-memory copy per record inserted, and stores records sequentially. PAX inserts records by performing as many copy operations as the number of values in the tuple, as described in Section 3.2. DSM creates and populates as many relations as the number of attributes. We compare the elapsed time to store a full TPC-H dataset when using each of the three schemes for variable database sizes. Full recovery was turned on for all three implementations.

Range selections. This query group consists of queries similar to those presented in Section 5.1 but without the aggregate function. Instead, the projected attribute value(s) were written to an output relation. To stress the system to the maximum possible extent, the range selections are on *Lineitem*, the largest table in the database. Lineitem contains 16 attributes having a variety of types, including three variable-length attributes. There are no indices on any of the tables, as most implementations of TPC-H in commercial systems include mostly clustered indices, which have a similar access behavior to sequential scan [2]. As in Section 5, we vary the projectivity and the number of attributes involved in the predicate.

TPC-H queries. We implemented four TPC-H queries, Q1, Q6, Q12, and Q14, on top of Shore. Queries 1 and 6 are range selections with multiple aggregates and predicates. The implementation pushes the predicates into Shore, computes the aggregates on the qualifying records, groups and orders the results. Queries 12 and 14 are equijoins with additional predicates and compute conditional aggregates. Their implementation uses the adaptive dynamic hash join operator described in Section 4.

Updates. We evaluate update transactions on Lineitem's attributes. Each transaction implements the following statement:

update R
set $a_p=a_p + b$
where $a_q > Lo$ and $a_q < Hi$ (2)

where a_p, a_q, and b are numeric attributes in Lineitem. We vary the number of updated fields, the number of fields in the predicate, and the selectivity. For each query, we measure the transaction execution time (i.e., including flushing the updated pages to the disk), and the actual main-memory update times, to determine the memory-hierarchy impact of the update algorithm.

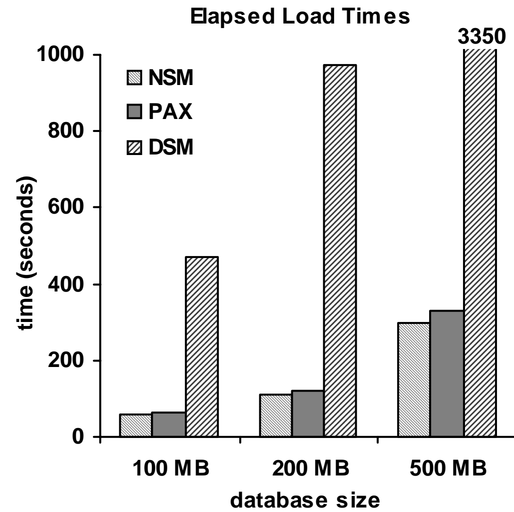

FIGURE 10: *Elapsed bulk-loading times.*

6.2 Bulk-loading

Figure 10 compares the elapsed time required to load a 100-MB, 200-MB, and 500-MB TPC-H database with each of the three storage organizations. DSM load times are much higher than those of NSM and PAX, because DSM creates one relation per attribute and stores one NSM-like record per value, along with the value's record id. In Section 5.2 we demonstrated that, when executing queries that involve multiple attributes, DSM never outperforms either of the other two schemes. Therefore, the rest of Section 6 will focus on comparing NSM and PAX.

Although the two schemes copy the same total amount of data to the page, NSM merely appends records. In contrast, PAX may need to perform additional page reorganizations if the relation contains variable-length attributes. PAX allocates V-minipage space in a new page based on average attribute sizes, and may occasionally over- or underestimate the expected minipage size. As a consequence, a record that would fit in an NSM page does not fit into the corresponding PAX page unless the current minipage boundaries are moved. In such cases, PAX needs additional page reorganizations to move minipage boundaries and accommodate new records.

The bulk-loading algorithm for PAX estimates initial minipage sizes on a page to be equal to the average attribute value sizes on the previously populated page. With the TPC-H database, this technique allows PAX to use about 80% of the page without any reorganization. Attempting to fill the rest of the 20% of each page results in an average of 2.5 reorganizations per page, half of

which only to accomodate one last record on the current page before allocating the next one. Figure 10 shows this worst case scenario: using an exhaustive algorithm that attempts to fill each page by 100%, PAX incurs 2-10% performance penalty when compared to NSM.

As an alternative, we implemented a smarter algorithm that reorganizes minipages only if the free space on the page is more than 5%. As was expected, the number of reorganizations per page dropped to half the number incurred by the exhaustive algorithm. A more conservative algorithm with a 10% "reorganization-worthy" free-space threshold dropped the average number of reorganizations to 0.8 per page. Table 2 shows that, as the average number

TABLE 2: *Effect of the "reorganization worthy" threshold on PAX bulk-loading performance.*

"Reorganization-worthy" threshold	Avg. # reorg. / page	Penalty wrt. NSM
0% *(always reorganize)*	2.25	10.1%
5% *(reorg. if < 95% full)*	1.14	4.9%
10% *(reorg. if < 90% full)*	0.85	0.8%

of reorganizations falls, the performance penalty of PAX versus NSM during bulk-loading becomes minimal (and is independent of the database size).

6.3 Queries

In Section 5.2 we explained why the performance improvement provided by PAX is reduced as the projectivity increases and the query accesses a larger portion of the record. As shown in Section 5, PAX's performance is superior to NSM when running range selections, especially when the query uses only a fraction of the record. The leftmost bar group (labeled 'RS') in Figure 11 shows the average speedup[4] obtained by a variety of range selection queries on Lineitem (described in Section 6.1). When using a 100-MB, a 200-MB, and a 500-MB dataset the speedup is 14%, 13%, and 10%, respectively.

Figure 11 also depicts PAX/NSM speedups when running four TPC-H queries against a 100, 200, and 500-MB TPC-H database. PAX outperforms NSM for all these experiments. The speedups obtained, however, are not constant across the experiments due to a combination of differing amounts of I/O and interactions between the hardware and the algorithms being used.

Queries 1 and 6 are essentially range queries that access roughly one third of each record in Lineitem and calculate aggregates. The difference between these TPC-H queries and the plain range selections (RS) discussed in

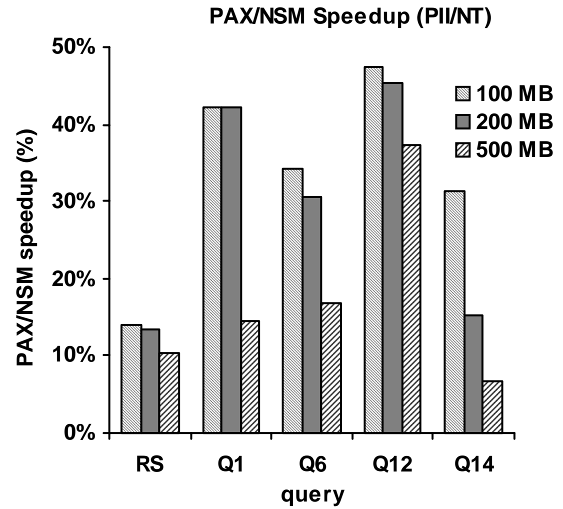

FIGURE 11: *PAX/NSM speedup on read-only queries.*

the previous paragraph is that TPC-H queries exploit further the spatial locality, because they access projected data multiple times to calculate aggregate values. Therefore, PAX speedup is higher due to the increased cache utilization and varies from 15% (in the 500-MB database) to 42% (in the smaller databases).

Queries 12 and 14 are more complicated and involve two joined tables, as well as range predicates. The join is performed by the adaptive dynamic hash join algorithm, as was explained in Section 4. Although both the NSM and the PAX implementation of the hash-join algorithm only copy the useful portion of the records, PAX still outperforms NSM because (a) with PAX, the useful attribute values are naturally isolated, and (b) the PAX buckets are stored on disk using the PAX format, maintaining the locality advantage as they are accessed for the second phase of the join. PAX executes query 12 in 37-48% less time than NSM. Since query 14 accesses fewer attributes and requires less computation than query 12, PAX outperforms NSM by only 6-32% when running this query.

6.4 Updates

The update algorithms implemented for NSM and PAX (discussed in Section 4.2) are based on the same philosophy: update attribute values in place, and perform reorganizations as needed. The difference lies in updating variable-length attributes. When replacing a variable-length attribute value with a larger one, PAX only needs to shift half the data of the V-minipage on average to accomodate the new requirements. NSM, on average, must move half the data of the *page* (because it moves records that include "innocent" unreferenced attributes). Less frequently, variable-length updates will result in a page reorganization for both schemes. Overall, massive updates on variable-length fields are rare; TPC-C's only update query is on the client credit history in the Payment transaction,

4. $Speedup = \left(\frac{ExecutionTime(NSM)}{ExecutionTime(PAX)} - 1\right) \times 100\%$ [17].

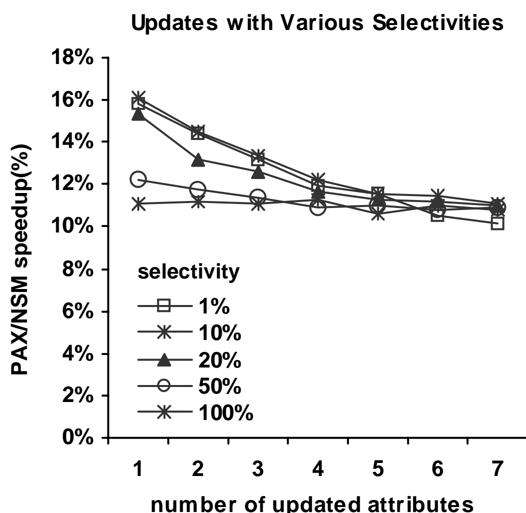

FIGURE 12: *PAX/NSM speedup on updates.*

and only affects one record (plus, the history field has a maximum length of 500 bytes).

When executing updates PAX is always faster than NSM, providing a speedup of 10-16%. The speedup is a function of the fraction of the record accessed as well as the selectivity. As discussed in Section 5.2, the execution times of NSM and PAX converge as we increase the projectivity. Similarly, the PAX/NSM speedup decreases as we increase the number of updated attributes. Figure 12 illustrates the speedup as a function of the number of updated attributes, when running updates on fixed-length attributes in a 100-MB database for various selectivities. For low selectivities, PAX provides a speedup of 10-16%, and its execution is dominated by read requests. As the selectivity increases, the probability that the updated attributes are in the cache is higher. However, increasing the selectivity also increases the probability that every data request will cause replacement of "dirty" cache blocks, which must be written back to memory. For this reason, as the selectivity increases from 20%-100%, the speedup is dominated by the write-back requests and becomes oblivious to the change in the number of updated attributes.

7 Summary

Data accesses to the cache hierarchy are a major performance bottleneck for modern database workloads [1]. Commercial DBMSs use NSM (N-ary Storage Modelary Storage Model) instead of DSM (Decomposition Storage Model) as the general data placement method, because the latter often incurs high record reconstruction costs. This paper shows that NSM has a negative effect on data cache performance, and introduces PAX (Partition Attributes Across), a new data page layout for relational DBMSs. PAX groups values for the same attribute together in minipages, combining inter-record spatial locality and high data cache performance with minimal record reconstruction cost at no extra storage overhead.

We compared PAX to both NSM and DSM:

- When compared to NSM, PAX incurs 75% less data cache stall time, while range selection queries and updates on main-memory tables execute in 17-25% less elapsed time. When running TPC-H queries that perform calculations on the data retrieved and require I/O, PAX incurs a 11-48% speedup over NSM.
- When compared to DSM, PAX cache performance is better and queries execute consistently faster because PAX does not require a join to reconstruct the records.

PAX incurs no storage penalty when compared with either NSM or DSM. PAX reorganizes the records *within* each page, and therefore can be used orthogonally to other storage schemes (such as NSM or affinity-based vertical partitioning) and transparently to the rest of the DBMS. Finally, page-level compression algorithms are likely to perform better using PAX due to the type uniformity within each minipage and the smaller in-page value domain [13].

Acknowledgements

We would like to thank Ashraf Aboulnaga, Ras Bodik, Christos Faloutsos, Babak Falsafi, Jim Gray, James Hamilton, Paul Larson, Bruce Lindsay, and Michael Parkes for their valuable comments on earlier drafts of this paper. Mark Hill is supported in part by the National Science Foundation under grant EIA-9971256 and through donations from Intel Corporation and Sun Microsystems.

References

[1] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. DBMSs on a modern processor: Where does time go?. In *proceedings of the 25th International Conference on Very Large Data Bases (VLDB),* pp. 54-65, Edinburgh, UK, September 1999.

[2] A. Ailamaki and D. Slutz. Processor Performance of Selection Queries, *Microsoft Research Technical Report MSR-TR-99-94*, August 1999.

[3] A. Ailamaki, D. J. DeWitt, and M.D. Hill. Walking Four Machines By The Shore. In *Proceedings of the Fourth Workshop on Computer Architecture Evaluation using Commercial Workloads,* January 2001.

[4] P. Boncz, S. Manegold, and M. Kersten. Database Architecture Optimized for the New Bottleneck: Memory Access. In *proceedings of the 25th International Conference on Very Large Data Bases (VLDB),* pp. 266-277, Edinburgh, UK, September 1999.

[5] T. Brinkhoff, H.-P. Kriegel, R. Schneider, and B. Seeger. Multi-Step Processing of Spatial Joins. In *proceedings of the ACM SIGMOD International Conference on Management of Data,* pp. 197--208, Minneapolis, MN, May 1994.

[6] B. Calder, C. Krintz, S. John, and T. Austin. Cache-Conscious Data-Placement. In *Proceedings of the 8th Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VIII)*, pages 139-149, October 1998.

[7] M. Carey, D. J. DeWitt, M. Franklin, N. Hall, M. McAuliffe, J. Naughton, D. Schuh, M. Solomon, C. Tan, O. Tsatalos, S. White, and M. Zwilling, Shoring Up Persistent Applications. In *proceedings of the ACM SIGMOD Conference on Management of Data,* Minneapolis, MN, May 1994.

[8] T. M. Chilimbi, J. R. Larus and M. D. Hill. Making Pointer-Based Data Structures Cache Conscious. IEEE Computer, December 2000.

[9] Compaq Corporation. 21164 Alpha Microprocessor Reference Manual. Online Compaq reference library at *http://www.support.compaq.com/alpha-tools/documentation/current/chip-docs.html*. Doc. No. EC-QP99C-TE, December 1998.

[10] G. P. Copeland and S. F. Khoshafian. A Decomposition Storage Model. In *proceedings of the ACM SIGMOD International Conference on Management of Data,* pp. 268-279, May 1985.

[11] D. W. Cornell and P. S. Yu. An Effective Approach to Vertical Partitioning for Physical Design of Relational Databases. In *IEEE Transactions on Software engineering,* 16(2), February 1990.

[12] D.J. DeWitt, N. Kabra, J. Luo, J. Patel, and J. Yu. Client-Server Paradise. In *Proceedings of the 20th VLDB International Conference*, Santiago, Chile, September 1994.

[13] J. Goldstein, R. Ramakrishnan, and U. Shaft. Compressing Relations and Indexes. In proceedings of *IEEE International Conference on Data Engineering,* 1998.

[14] G. Graefe. Iterators, Schedulers, and Distributed-memory Parallelism. In software, Practice and Experience, 26(4), pp. 427-452, April 1996.

[15] Jim Gray. *The benchmark handbook for transaction-processing systems.* Morgan-Kaufmann Publishers, 2nd edition, 1993.

[16] A. Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching, In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1984.

[17] J. L. Hennessy and D. A. Patterson. Computer Architecture: A Quantitative Approach. Morgan Kaufmann Publishers, 2nd edition, 1996.

[18] Intel Corporation. Pentium® II processor developer's manual. Intel Corporation, Order number 243502-001, October 1997.

[19] K. Keeton, D. A. Patterson, Y. Q. He, R. C. Raphael, and W. E. Baker. Performance characterization of a quad Pentium pro SMP using OLTP workloads. In *Proceedings of the 25th International Symposium on Computer Architecture*, Barcelona, Spain, June 1998.

[20] Bruce Lindsay. Personal Communication, February / July 2000.

[21] J. L. Lo, L. A. Barroso, S. J. Eggers, K. Gharachorloo, H. M. Levy, and S. S. Parekh. An analysis of database workload performance on simultaneous multithreaded processors. In *Proceedings of the 25th International Symposium on Computer Architecture*, June 1998.

[22] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh and P. Schwarz. ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. In *ACM Transactions on Database Systems* 17, 1 (March 1992), pp. 94 - 162.

[23] M. Nakayama, M. Kitsuregawa, and M. Takagi: Hash-Partitioned Join Method Using Dynamic Destaging Strategy. In *Proceedings of the 14th VLDB International Conference*, September 1988.

[24] S. Navathe, S. Ceri, G. Wiederhold, and J. Dou. Vertical partitioning algorithms for database design. *ACM Transactions on Database Systems,* 9(4), pp/ 680-710, December 1984.

[25] P. O'Neil and D. Quass. Improved Query Performance With Variant Indexes. In *proceedings of the ACM SIGMOD International Conference on Management of Data,* Tucson, Arizona, May 1997.

[26] J. M. Patel and D. J. DeWitt. Partition Based Spatial-Merge Join. In *proceedings of the ACM SIGMOD International Conference on Management of Data,* pp. 259-270, Montreal, Canada, June 1996.

[27] R. Ramakrishnan and J. Gehrke. *Database Management Systems.* WCB/McGraw-Hill, 2nd edition, 2000.

[28] P. G. Selinger, M. M. Astrahan, D. D. Chamberlain, R.A. Lorie, and T. G. Price. Access Path Selection In A Relational Database Management System. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, 1979.

[29] A. Shatdal, C. Kant, and J. Naughton. Cache Conscious Algorithms for Relational Query Processing. In *proceedings of the 20th International Conference on Very Large Data Bases (VLDB),* pp. 510-512, September 1994.

[30] R. Soukup and K. Delaney. Inside SQL Server 7.0. Microsoft Press, 1999.

[31] Sun Microelectronics. UltraSparc™ Reference Manual. Online Sun reference library at *http://www.sun.com/microelectronics/manuals/ultrasparc/802-7220-02.pdf*, July 1997.

[32] http://technet.oracle.com/docs/products/oracle8i/doc_index.htm

[33] http://www.sybase.com/products/archivedproducts/sybaseiq

[34] http://www.cs.wisc.edu/~natassa/papers/PAX_full.pdf