Data Morphing: An Adaptive, Cache-Conscious Storage Technique

Richard A. Hankins Jignesh M. Patel

University of Michigan
1301 Beal Avenue; Ann Arbor, MI 48109-2122; USA
{hankinsr, jignesh}@eecs.umich.edu

Abstract

The number of processor cache misses has a critical impact on the performance of DBMSs running on servers with large main-memory configurations. In turn, the cache utilization of database systems is highly dependent on the physical organization of the records in main-memory. A recently proposed storage model, called PAX, was shown to greatly improve the performance of sequential file-scan operations when compared to the commonly implemented N-ary storage model. However, the PAX storage model can also demonstrate poor cache utilization for other common operations, such as index scans. Under a workload of heterogenous database operations, neither the PAX storage model nor the N-ary storage model is optimal.

In this paper, we propose a flexible data storage technique called Data Morphing. Using Data Morphing, a cache-efficient attribute layout, called a partition, is first determined through an analysis of the query workload. This partition is then used as a template for storing data in a cache-efficient way. We present two algorithms for computing partitions, and also present a versatile storage model that accommodates the dynamic reorganization of the attributes in a file. Finally, we experimentally demonstrate that the Data Morphing technique provides a significant performance improvement over both the traditional N-ary storage model and the PAX model.

**Proceedings of the 29th VLDB Conference,
Berlin, Germany, 2003**

1 Introduction

Database systems have traditionally focused on improving the overall system performance by minimizing the number of disk I/O operations. Disk I/O has traditionally been the most important component of the memory access hierarchy, but the current trend in Random Access Memory (RAM) cost and capacity now makes it necessary to re-evaluate this focus. The cost of RAM is decreasing rapidly, while the capacity of RAM is increasing exponentially. This trend was recognized by the authors of the Asilomar report where they predicted that, in the future, all but the largest data sets will reside entirely in main-memory [5]. As increasing amounts of data become main-memory resident, the performance bottleneck becomes the accesses to main-memory rather than the accesses to disk [7]. Because of this shift, intelligently storing and accessing the data in main-memory is becoming critical to the performance of database systems [2, 7, 17, 22].

Unfortunately, the main-memory bottleneck is becoming more severe due to the growing disparity between processor speeds and the latency in accessing the main-memory. To alleviate this disparity, modern processors employ a hierarchy of low-latency memory, called *caches*, that reside between the processor and the RAM. Each cache stores the most frequently accessed data blocks (including instructions), reducing the amount of data that must be retrieved directly from the main-memory. Starting at the first level of cache memory, each subsequent level of cache stores larger amounts of data, but each subsequent level of cache is also more expensive to access. Modern processors typically contain two, or even three, levels of processor cache, with each level storing both instructions and data. Research has shown that database systems incur a significant amount of second level (L2) data cache misses, and consequently, the L2 cache utilization is critical to overall performance [3]. In this paper we only focus on techniques for reducing the L2 cache misses, and simply refer to them as cache misses for the remainder of this paper.

Critical to system performance is the efficient retrieval and update of data records inside the database. Database systems typically store records of data in *files*, where each

file consists of a collection of *slotted-pages*, and each slotted-page contains a collection of records. A *page* is a fixed-size block of allocated memory. A *slotted-page* is a page that contains an array of byte-offsets that reference the start of each record allocated within the page. Typically, each attribute of the record is stored in consecutive memory addresses on the slotted page, in what is called the N-ary storage model. Common implementations of the N-ary storage model place variable length attributes at the end of the fixed-length attributes, and use fixed length placeholders to reference the relocated attributes.

A recently proposed storage model, called PAX, vertically decomposes each record, storing each attribute in subdivisions of a page, called *mini-pages* [2]. The PAX storage model improves the cache utilization for queries that access only a few attributes from a high percentage of records, such as during a sequential file scan. This improvement in cache utilization is a direct result of being able to pack attributes of many different records into the same cache line. However, for access plans where only a fraction of the records are accessed, such as during an index scan, the PAX storage model can result in a greater number of cache misses. In such cases, N-ary may actually perform better, especially as the number of attributes accessed per record increases.

As previously demonstrated, both the N-ary and the PAX storage models work well for different families of access plans. A choice between the two storage models is difficult as the query workload may be large or may vary over time. A better storage model should incorporate the best characteristics of both models; namely, vertical decomposition and groups of sequentially allocated attributes. This storage model should also provide the flexibility to adapt to changing workloads. In this paper, we propose a novel data storage technique, called *Data Morphing* (DM) that meets these goals.

This paper makes the following contributions:

- We present a flexible page architecture that is a generalization of the previously proposed PAX page architecture. In the Data Morphing page architecture, attributes of the same tuple can be stored in non-contiguous groups, which increases the spatial locality of memory accesses. As a result, fewer cache misses are incurred during query processing.
- We present two algorithms for calculating the attribute groups that are stored on each page. The Naive algorithm performs an exhaustive search of the possible attribute layouts, but the algorithm is expensive to compute for relations with a large number of attributes. The Hill-Climb algorithm performs a greedy search of the layout space, trading optimality for faster time complexity.
- We present an empirical evaluation of the Data Morphing technique. We show that the partitioning algorithms calculate attribute groupings that reduce the number of cache misses incurred during query execution. As a direct result of improving the cache utilization, our prototype DBMS implementing the DM technique can evaluate queries up to 45% faster than the N-ary storage model and up to 25% faster than the PAX model.

The remainder of this paper is organized as follows: Section 2 presents an example motivating the need for Data Morphing. Section 3 provides definitions for the common terms used in the subsequent sections. In Section 4, we present the Data Morphing technique. A detailed experimental evaluation is presented in Section 5. Related work is discussed in Section 6, and Section 7 contains our conclusions.

2 Motivating Example

As described in the introduction, neither the N-ary storage model nor the PAX storage model provide the optimal storage model for data. To illustrate this fact, we will use the relation and query shown in Figure 1 in the following example.

Client (`id`: Integer, `priority`: Integer, `name`: Varchar(32),
`usage`: Real, `address`: Varchar(32),
`location`: Integer) key (`id`);

SELECT `location`, `usage`
FROM Client WHERE `priority` < 12

Figure 1: Client Relation and Query

In Figure 1, the `Client` relation consists of six attributes of types Integer, Real, and Varchar. The Integer and Real data types are each four bytes in size. The Varchar(32) data type is a variable length string with a maximum length of 32 bytes. The selectivity of the predicate on `priority` is 12.5%, and the processor cache line size is 32 bytes. Figure 2 shows a single record in the `Client` relation as it is stored on a page using the N-ary storage model; as in typical implementations, the variable-length strings are stored at the end of the fixed-length attributes. To visualize the attributes that are read into the processor cache as the result of a cache miss, the width of the diagram is shown to be the same size as a cache line.

Assuming that a sequential file scan is used to answer this query, the records are retrieved as follows. The first `priority` attribute is requested from memory. This memory access incurs a cache miss to read the contiguous block of memory containing the attribute into the processor cache. If the value of the `priority` attribute is less than 12, the `location` and `usage` attributes are accessed. Because the `location` and `usage` attributes are contained in the processor cache, retrieving them incurs no additional cache misses. The select operator then continues with the next record, until all of the records have been read. The cost of answering this query totals approximately one cache miss per record.

The PAX storage model vertically decomposes the records into zones, called mini-pages, as shown in Figure 3; for simplicity, we do not show all of the details. In the figure, the attributes are presented as belonging

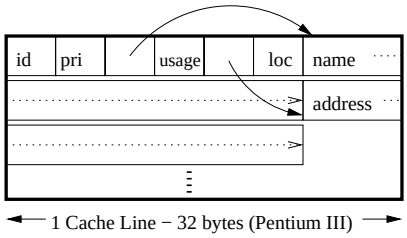

Figure 2: N-ary Storage Model

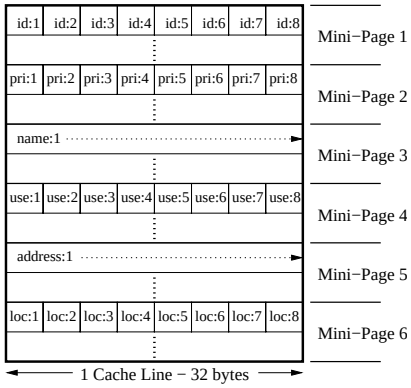

Figure 3: PAX (Vertical Decomp.)

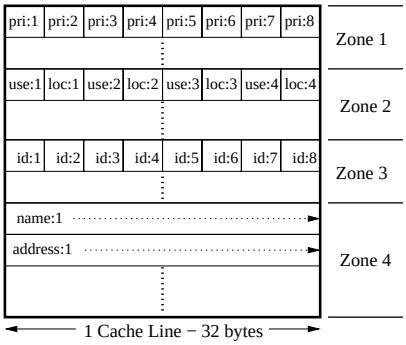

Figure 4: Attribute Grouping

to specific records by indicating the record number next to the attribute name. Using the PAX storage model, a cache miss is incurred upon reading the first `priority` attribute. After the first cache miss, the `priority` attributes of the next eight records are read without incurring any additional cache misses. Because the predicate on the `priority` attribute is true for one record out of every eight records accessed, reading the `location` and the `usage` attributes incurs two additional cache misses per eight records. The resulting cost of the select operator is approximately $1/8+2/8 = 0.375$ cache misses per record. This is a significant improvement over the traditional N-ary storage model.

While the PAX storage model performs better than the N-ary storage model, the number of cache misses can be reduced even further. Recognizing that the `usage` and `location` attributes are always accessed collectively, the record should be partitioned into four *zones*: the `priority` attribute in one zone, the `usage` and `location` attributes in a second zone, the `id` attribute in a third zone, and the remaining attributes in a fourth zone, as shown in Figure 4. Similar to the PAX layout, the `priority` attribute from eight consecutive records can be read while incurring only a single cache miss. For every value of the `priority` attribute that is less than 12, the `usage` and `location` attributes are read. Because the `usage` and `location` attributes were located in different cache blocks in the PAX layout, two cache misses were incurred to read both of the attributes. When using the group layout (Figure 4), however, a single cache miss is incurred to read the first `usage` attribute, and the `location` attribute is then read from the cache. Using this particular grouping, the number of cache misses per record shrinks to two cache misses per eight records, or 0.25 cache misses per record.

As the example demonstrates, partitioning the records' attributes into non-contiguous zones can significantly reduce the number of processor cache misses. Determining the attribute partition for a single query is not difficult; however, choosing a partition that reduces the total amount of cache misses for the entire query workload is much more complex. In addition, the query workload may change over time, so the optimal layout may change accordingly. The Data Morphing technique presented in this paper provides a method to calculate a cache-efficient partition for a given workload of queries, and also provides a method to reorganize data to dynamically adapt to a changing workload. Before the Data Morphing technique is presented, the following section provides the definitions to the terms used in the discussion.

Variable	Definition
A	The set of all attributes in relation $R = \{a_1, a_2, \cdots, a_n\}$
group	A subset of the set of attributes, $group \subseteq A$
partition	A collection of groups
zone	The area of a slotted page where all instances of a group are written
zone-record	An instance of the attributes in a particular group
G	The set of all possible groups
$\|G\|$	The number all possible groups
P	The set of all unique partitions
$\|P\|$	The number of all unique partitions

Table 1: Definitions

3 Definitions

For the presentation of the Data Morphing technique, the following definitions apply. A **group** represents a set of attributes that are written to consecutive memory addresses on a page. A **partition** is a set of groups that uniquely defines the position of every attribute in a relation. A **zone** defines the area of a page where all instances of a particular group are written. A **zone-record** defines an instance of the attributes in a particular group. These definitions are summarized in Table 1.

To further illustrate the concepts in this section succinctly, we will use a simplified version of the previous example. The new example relation and the corresponding query are shown in Figure 5. As before, the predicate on the `priority` attribute has a selectivity of 12.5%, and all of the attributes are four bytes in size.

R = (priority:Integer, location:Integer, usage:Integer)

SELECT location, usage
FROM R WHERE priority < 12

Figure 5: Example Relation and Query

Using the relation R, shown in Figure 5, a partition p representing the traditional N-ary Storage Model is

$$p = \{\{\texttt{priority}, \texttt{location}, \texttt{usage}\}\}.$$

Partition p has one group of attributes, and all three attributes are to be written consecutively in memory. A partition representing the PAX storage model is

$$p = \{\{\texttt{priority}\}, \{\texttt{location}\}, \{\texttt{usage}\}\}.$$

This partition has three groups, with each attribute belonging to a separate group.

Formalizing the discussion, consider the set A of all attributes in relation R. From the definition, a partition p of set A is a collection of subsets of A, called *groups*, such that each group is non-empty and is pairwise disjoint with all other groups, and $\cup p = A$. The set of all possible groups, G, is the power set of the attribute set A. The size of G is 2^n, where n is the number of attributes in the relation R. The set of all possible partitions is P. The number of possible partitions of set A, or $|P|$, is described by Bell numbers [4, 21].

From the example, the set of all attributes, A, the set of all possible groups, G, and the set of all possible partitions, P, are as follows:

$$\begin{aligned}
A = {} & \{\texttt{priority}, \texttt{location}, \texttt{usage}\} \\
G = {} & \{\{\texttt{priority}\}, \{\texttt{location}\}, \{\texttt{usage}\}, \\
& \{\texttt{priority}, \texttt{location}\}, \{\texttt{priority}, \texttt{usage}\}, \\
& \{\texttt{location}, \texttt{usage}\}, \\
& \{\texttt{priority}, \texttt{location}, \texttt{usage}\}\} \\
P = {} & \{\{\{\texttt{priority}\}, \{\texttt{location}\}, \{\texttt{usage}\}\}, \\
& \{\{\texttt{priority}, \texttt{location}\}, \{\texttt{usage}\}\}, \\
& \{\{\texttt{priority}, \texttt{usage}\}, \{\texttt{location}\}\}, \\
& \{\{\texttt{priority}\}, \{\texttt{location}, \texttt{usage}\}\}, \\
& \{\{\texttt{priority}, \texttt{location}, \texttt{usage}\}\}\}.
\end{aligned}$$

Now that the definitions have been presented, the next section introduces the Data Morphing process.

4 Data Morphing

Data Morphing consists of two phases: (a) calculating a cache-efficient storage template and (b) reorganizing the data into this cache-efficient organization. In this section, we first introduce the DM page architecture, which allows attributes to be grouped into non-contiguous zones. We then introduce two algorithms for calculating attribute partitions that improve the spatial locality of the data.

4.1 Page Structure

To support decomposition of the records' attributes into groups, a flexible page structure is required. The new page structure must allow the attributes of a record to be written in an arbitrary pattern while also retaining the link between the external record id and the internal record. The PAX page structure [2] has many of these properties; however, the PAX page structure stores each attribute individually. We need a more general page architecture that allows arbitrary grouping of attributes and also seamlessly accommodates the traditional N-ary representation (which is more efficient in some cases). In this section, we present the Data Morphing page structure. The DM page structure can be viewed as a generalization of the PAX page structure.

4.1.1 Page Description

The traditional slotted page includes meta-data at the top of the page along with a slot array located at the bottom of the page. The attributes of each record are written to consecutive memory addresses on the page, with the starting offset of a record stored in the slot array [20]. The record space typically grows downwards (increasing memory addresses) while the slot array grows upwards (decreasing memory addresses), and the free space on the page is determined by the unallocated memory between the two regions.

To support the partitioning of attributes into separate zones, the new DM slotted-page structure requires five additional arrays. The first array, ATT-ZON, records the zone number for each attribute. The second array, ATT-SZ, records the size of each attribute, with variable-length attributes marked accordingly. The third array, ATT-OFF, records the offset of the attribute into each zone-record. The fourth array, ZON-OFF, records the starting offset of each zone on the page. The fifth array, REC-SZ, records the size of each zone-record. Figure 6 illustrates the new slotted-page data structure.

A partition is local to each page. This allows the working set of pages for one workload to be organized differently than the working set of pages for a different workload. If the entire relation uses the same static layout, the partition information can be stored in the system catalogs.

The size of each DM array is implementation dependent. For our particular implementation, the number of entries in each table is equal to the number of attributes in the relation. We use 1-byte entries for the ATT-ZON table and 2-byte entries for the remaining tables. For a relation with 16 attributes, the total size of the meta-data would be 144 bytes, spanning five 32-byte cache lines. If only a few records on each page are accessed, the cost of accessing the meta-data can be quite expensive. For this reason, we are examining alternatives to allocating the meta-data on each page while still allowing the partitions to vary between pages.

It is important to note that the slotted-page data structures are located at the same byte-offsets on each page. If the records are accessed in a random order, the data structures on each page may be removed from the processor

cache due to conflict misses; therefore, a more efficient technique is to access all the necessary records on a page before moving to any other pages in the heap file. This problem is not unique to the DM technique, but is common to all heap files that use the slotted-page data structure.

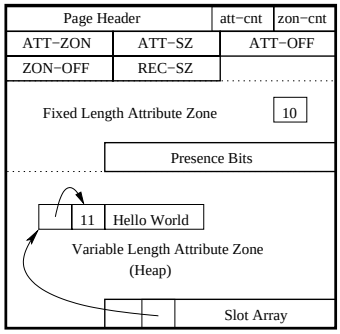

Figure 6: DM Slotted Page Layout

The DM slotted page consists of one or more zones. Fixed-length attributes and variable-length attributes can be grouped together into **variable-length zones**. Each variable-length zone has a slot-array that contains references to the position of all of that zone's records. All variable-length zone-records store the variable-length values at the end of each record, as is typically done in the N-ary storage model. If a group consists of only fixed-length attributes, the group can be stored in a **fixed-length zone**. Since the attributes in a fixed-length zone can be accessed using a simple byte-offset calculation, the fixed-length zones do not contain a slot-array. In place of the slot-array, a bit-array is used to indicate the presence or absence of a zone-record.

The fixed-length zone is not required but can be convenient. For example, fixed-length zones are easier to manage because there is no need to maintain a heap. Also, using a simple byte-offset calculation to access the attribute is more efficient than accessing a slot array.

4.1.2 Page Operations

For the DM slotted-page layout, the typical operations, including attribute retrieval, record insertion, deletion, and updates, are modified in a straight-forward way. The key changes are that these operations need to consult the additional meta-data to locate the attribute values. In the interest of space, we omit these details in this presentation and refer the interested reader to [15].

4.1.3 Dynamically Reorganizing Pages

Because the query workload may change over time, the Data Morphing process may periodically recommend a new attribute partition. The attribute layout will then need to be reorganized to match the new partition. The reorganization process is as follows. The Storage Manager is first provided a new attribute partition. When a page is accessed from the data file, the page's partition description is compared to the recommended partition. If the two partitions differ, the page is reorganized based on the recommended partition.

The structure of the DM slotted page allows pages to be lazily reorganized at access time. Therefore, only highly accessed pages will be reorganized. Some pages in the file may rarely be reorganized. Because the partition information is stored on each page, pages with different partitions may co-exist in the same file.

We expect page reorganization to be an expensive operation relative to the cost of scanning the records on a page, so a reorganization should be performed judiciously. Analysis of the reorganization strategy is left for future work.

4.2 Partitioning the Attributes

The Data Morphing technique consists of two phases: calculating a cache-efficient attribute partition, and reorganizing the data. The first phase of the DM process calculates a cache efficient layout for the attributes of a given relation $R(a_1, a_2, \cdots, a_n)$, and a given set of queries, Q. To calculate this layout, we present two algorithms. First, we present a naive algorithm which finds the *optimal* layout, but requires exponential space and time in the number of attributes. We then present a heuristic algorithm to reduce the computational complexity. The heuristic algorithm is based on a hill-climbing (steepest-descent) algorithm, and trades optimality for improved time complexity.

4.2.1 Data Morphing Input

Each of the proposed algorithms relies on information about the queries that are executed. A query q in the query set Q is described by an ordered sequence of pairs, (x, y), where x represents the attribute accessed, and y represents the frequency at which attribute x is accessed. Using the example from Section 3 for illustration, a sequential file-scan will access 100% of the `priority` attributes. If the value of the `priority` attribute is less than 12, the remaining two attributes are accessed. The selectivity of the predicate on `priority` is 12.5%, so both of the remaining attributes will be accessed for 12.5% of the records. The resulting query sequence, q, will be:

$$q = ((\texttt{priority}, 100\%), \\ (\texttt{location}, 12.5\%), (\texttt{usage}, 12.5\%))$$

4.2.2 Cache Miss Model

The cache miss model computes the cost, in the number of processor cache misses, for a query to access the attributes in a group. For the cache miss model, we define the access rate of the group, *acc*, as the rate of the most accessed attribute in that group. Given a query q and an attribute group g, the number of cache misses incurred is calculated as follows: Accessing the first attribute a in the sequence q that is also a member of group g incurs a cache miss. The

number of zone-records available per cache line, τ, is given by Equation 1.

$$\tau = \frac{\text{cache line size}}{sizeof(g)} \quad (1)$$

The number of zone-records between successive record accesses, $itval$, is given in Equation 2

$$itval = \frac{1}{acc} \quad (2)$$

If the interval of zone-records, $itval$, is larger than the number of zone-records available in one cache line, each zone-record accessed incurs a cache miss for each cache line spanned by the group. The model returns the number of cache misses per record, as shown in Equation 3.

$$model(q,g) = \begin{cases} \frac{1}{\tau} & \text{if } itval < \tau \\ acc * \lceil \frac{1}{\tau} \rceil & \text{if } itval >= \tau \end{cases} \quad (3)$$

Our cache-miss model attempts to capture the first-order effects of accessing the records in a heap file, either through a sequential file-scan or as part of an index scan. This model relies on the assumption that the attributes are accessed in a uniformly random fashion, the cache line being accessed is not removed from the processor cache before the all of the attributes in that cache line have been processed, and that the data was removed from the processor cache between successive record scan operations. Our model may unfairly penalize group sizes larger than a cache line due to the assumption that a query on that group will incur a cache miss for each cache line spanned by that group.

For typical cache line sizes (32 and 64 bytes), we believe it is reasonable to assume that all of the data in a single cache line will be processed before that cache line is evicted. Also, when processing sequential scans of even moderate size heap files, and with many concurrently executing database operations, the processor cache is unlikely to contain the same records upon the next occurrence of a file scan operation. But, if the heap file is used in many queries or the heap file is not large relative to the processor cache, the data may reside in the processor cache between successive scans. For this scenario, incorporating Cardenas's formula [10], or possibly Yao's formula [23], into the model may provide a more cache-efficient attribute partition and is the subject of future work. We also plan to revisit the assumption that data is accessed uniformly by incorporating the distribution type of the data. Finally, we plan to more fairly incorporate group sizes larger than the cache line size in future versions of the cost model.

Even though our cache-miss model is very simple, the experimental evaluation shows that it still allows cache-efficient attribute partitions to be calculated. We expect that the performance of a more complex cost model will improve the quality of the attribute partitions but with an additional computational cost.

Using the example relation and query in Figure 5, the cost of grouping the `priority` and `usage` attributes is calculated as follows. The query, q, on the relation is:

$$q = ((\texttt{priority}, 100\%), (\texttt{location}, 12.5\%), (\texttt{usage}, 12.5\%))$$

The group, g, is:

$$g = \{\texttt{priority}, \texttt{usage}\}$$

The size of group g is 8 bytes. The size of a cache line is assumed to be 32 bytes. The access rate, acc, of group g is the access rate of the highest accessed attribute $a \in g$. In this example, $acc = 1$. The number of groups per cache line, $\tau = 32/8 = 4$, means that four zone-records of this group g can be accessed in one cache miss. Now, we need to calculate how often this group is accessed. $itval = 1$ states that every record is accessed. Because four zone-records can be read for every cache miss, the cost of this query on group g is 0.25 cache misses per record.

4.2.3 Naive Algorithm

The Naive algorithm calculates the optimal attribute partitions based on the cost of each possible partition. The optimal partitions are the candidates that result in the fewest number of overall cache misses for the query workload. The cost of executing a query on a relation that uses a particular partition p is the sum of the cost to access the attributes from each group in the partition, as shown in Equation 4.

$$cost(q,p) = \sum_{i=1}^{|p|} model(q, p_i) \quad (4)$$

Equation 4 calculates the cost of executing query q on the partition p. In Equation 4, $|p|$ represents the number of groups in the partition p, p_i represents group i in partition p, and $model(q, p_i)$ is the number of cache misses incurred for query q to access the attributes in group p_i (Equation 3).

The Naive algorithm calculates the cost of each possible partition and selects the partitions with the lowest overall cost for the set of queries Q. The method for calculating the lowest cost is shown in Equation 5.

$$cost_{min} = \min_{i=1...|P|} \left(\sum_{j=1}^{|Q|} cost(Q_j, P_i) \right) \quad (5)$$

In Equation 5, $|P|$ is the number of possible partitions of the set of attributes A, P_i is partition $i \in P$, $|Q|$ is the number of queries in the workload, and Q_j is query $j \in Q$.

Because a group may be used in several partitions, calculating the cost of a partition in Equation 5 repeats work that may have been previously computed. Instead of calculating the cost of each partition, we can calculate the cost of each possible group. The cost of a partition is the sum of the costs of each group in the partition. If a table is created

```
Naive(Q,P,G)
(* Q is the set of all queries *)
(* P is the set of all partitions *)
(* G is the set of all groups *)

    (* Compute each group cost *)
    for i ← 1 to length[G]
        for j ← 1 to length[Q]
            table[i] ← model(Q[j], G[i])
    (* Compute each possible partition's cost *)
    R ← ∅
    for i ← 1 to length[P]
        p ← P[i]
        for j ← 1 to length[p]
            (* g is a group in partition p *)
            g ← p_j
            sum ← sum + table[g]
        (* Record all min cost partitions *)
        if sum = mincost
            R ← R ∪ p
        if sum < mincost
            R ← p
            mincost ← sum
    return R
```

Figure 7: Naive Algorithm

to record the cost of each group $g \in G$, the cost of the partition p can be calculated by a series of table lookups. The $cost_{min}$ formula in Equation 5 can then be expressed using Equation 8.

$$table[g] = \sum_{i=1}^{|Q|} model(Q_i, g) \quad (6)$$

$$cost(p) = \sum_{i=i}^{|p|} table[p_i] \quad (7)$$

$$cost_{min} = \min_{i=1...|P|} (cost(P_i)) \quad (8)$$

In Equation 6, a table of the costs of each group g is first computed for the query workload Q. In Equation 7, the cost of a partition is then computed by summing the costs of the individual groups contained in the partition. Equation 8 finds the minimum cost of all the partitions in P. The partitions that correspond to the minimum number of cache misses is the solution. Pseudo-code for the Naive algorithm is shown in Figure 7.

Unfortunately, the Naive algorithm is exponential in both time and space. For each query $q \in Q$, n possible attributes are examined. The number of groups to examine is 2^n, where n is the number of attributes in a relation. The total cost to record the group cost table is $\mathcal{O}(mn2^n)$, where m is the number of queries in the query set. The number of possible attribute partitions is described by Bell numbers. Several authors have studied the asymptotic limit of the Bell numbers [13, 19]. Approximating the findings of de Bruijn, the Bell numbers have an asymptotic limit of $\mathcal{O}(e^{n \ln(n)})$. The time complexity of the algorithm is therefore $\mathcal{O}(e^{n \ln(n)} + mn2^n)$. The space complexity is $\Theta(2^n)$ to store the group-cost table.

Group	Cost
{priority}	0.125
{location}	0.125
{usage}	0.125
{priority, location}	0.25
{priority, usage}	0.25
{location, usage}	0.125
{priority, location, usage}	0.375

Table 2: Example: Group Costs

Partition	Cost
{{priority}, {location}, {usage}}	0.375
{{priority, location}, {usage}}	0.375
{{priority, usage}, {location}}	0.375
{{priority}, {location, usage}}	0.25
{{priority, location, usage}}	0.375

Table 3: Example: Partition Costs

We can demonstrate the Naive algorithm on the example query in Figure 5. First, the cost of each group is precomputed and stored in a cost table, as shown in Table 2. After precomputing the cost of each group, we compute the cost of each partition, as shown in Table 3. The partition that results in the minimum cost for this query workload is {{priority}, {location, usage}}. In other words, the priority attributes should be in one zone, and the location and usage attributes should be in a separate zone, with the location and usage attributes of each record written contiguously on the page.

The time complexity of the Naive algorithm is exponential due to the number of possible partitions that must be examined. The Hill-Climb algorithm, presented next, trades the guarantee of finding the optimal partition for an improvement in time complexity.

4.2.4 Hill-Climb Algorithm

The Naive algorithm computes the set of optimal partitions but the time complexity is exponential due to the number of possible partitions that must be examined. The Hill-Climb algorithm trades the guarantee of optimality for faster computation time. The Hill-Climb algorithm is computed as follows. First, the cost of each attribute grouping is calculated as in Equation 6 of the Naive algorithm. Each of the n attributes are then partitioned into separate groups. The algorithm then begins an iterative process to select a partition. In the first iteration, the algorithm considers the effect of "merging" any two partitions. The cost of each of these combinations is computed. (Each partition has exactly $n - 2$ groups with one attribute, and one group with two attributes.) The partition that has the cheapest cost is then picked for the next iteration. The next iteration proceeds in a similar fashion and considers all possible pairings of the remaining groups. Thus, in each iteration, the number of groups is reduced by one. This process is re-

```
Hill-Climb(Q,G)
(* Q is the set of all queries *)
(* G is the set of all groups *)

    (* Compute the cost, table[i], of each group i*)
    for i ← 1 to length(G)
        for j ← 1 to length(Q)
            table[i] ← model(Q[j], G[i])
    (* Compute the cost for each partition *)
    Cand ← {{1}, {2}, ··· , {n}}
    candcost ← cost(Cand) (* Equation 7 *)
    R ← ∅
    do
        R ← Cand
        mincost ← candcost
        Cand ← ∅
        for i ← 1 to length(R)
            for j ← i + 1 to length(R)
                s ← {{R_1, ··· , R_i ∪ R_j, ···}}
                Cand ← Cand ∪ s
        (* Compute lowest cost partition *)
        candcost ← min_{i=1...|Cand|}(cost(Cand_i))
        Cand ← min_{i=1...|Cand|}(Cand_i)
    while candcost < mincost
    return R
```

Figure 8: Hill-Climb Algorithm

peated until the total cost of the partition does not improve. Ties among candidate partitions with the same cost are broken by randomly selecting a partition from the set. The pseudo-code for this algorithm is shown in Figure 8.

The time complexity is $\mathcal{O}(mn2^n)$ to calculate the group cost table and $\mathcal{O}(n^3)$ to calculate the best partition. The resulting time complexity is therefore $\mathcal{O}(mn2^n)$. Again, the space complexity is $\Theta(2^n)$ to store the group-cost table. If space is not an issue, and if the query information arrives at a relatively low rate, then this algorithm may perform well, even for a large number of attributes.

The following describes the use of the Hill-Climb algorithm on the example in Figure 5. The candidate partition is initialized to

$$cand = \{\{\texttt{priority}\}, \{\texttt{location}\}, \{\texttt{usage}\}\},$$

with a cost of $candcost = 0.375$ cache misses per record. The result of combining each group would result in a candidate set of:

$$cand = \{\{\{\texttt{priority}, \texttt{location}\}, \{\texttt{usage}\}\}, \\ \{\{\texttt{priority}, \texttt{usage}\}, \{\texttt{location}\}\}, \\ \{\{\texttt{priority}\}, \{\texttt{location}, \texttt{usage}\}\}\}$$

The respective costs for each partition in the candidate set is $(0.375, 0.375, 0.25)$ cache misses per record. The best partition from the candidate set is

$$p = \{\{\texttt{priority}\}, \{\texttt{location}, \texttt{usage}\}\}$$

with a cost of 0.25 cache misses per record, as calculated from Table 2. The second iteration creates the candidate set

$$cand = \{\{\{\texttt{priority}, \texttt{location}, \texttt{usage}\}\}\}$$

with a cost of 0.375. Because the cost is greater than the current best partition cost of 0.25 cache misses per record, the algorithm terminates and the partition

$$p = \{\{\texttt{priority}\}, \{\texttt{location}, \texttt{usage}\}\}$$

is returned.

5 Experimental Evaluation

In this section, we present the results of an experimental evaluation of the Data Morphing technique.

5.1 Experimental Setup

Data Morphing was implemented within an experimental database system that we are developing, called Quickstep. The Quickstep DBMS uses fixed-size pages for storing and retrieving data from the disk. The DBMS allocates memory in pages that can then be saved to disk, and uses a buffer manager to manage page caching in main memory. If the database size is less than the buffer pool size, then the entire database image is mapped to a contiguous space in memory. In this mode, all disk pointers are *swizzled* to direct memory pointers. In the experiments presented in this section, the entire data set is always pinned in main memory, so there is no disk I/O from swapping pages out of the buffer. We expect that the performance improvement of the DM technique to decrease as the amount of disk I/O increases, but the DM technique is still applicable for use on datasets that primarily reside in the main-memory.

Currently, Quickstep only supports file-level locks, and updates are not logged. The code path in Quickstep is optimized for high-performance when the database primarily resides in the main-memory. Quickstep executes fewer instructions per query and incurs fewer L1 instruction-cache misses than commercial databases are expected to experience [3]. Quickstep uses cache-conscious hash and B+tree index structures. Joins are evaluated using either nested-loops or hash-join algorithms, and aggregates are evaluated using a hash-based aggregate algorithm.

The experiments were performed on a 600 MHz, Intel Pentium III processor, with 768MB of main memory. This processor includes a two level cache hierarchy. There are two first level caches, named L1-I and L1-D, that cache instructions and data respectively. There is also a single L2 cache that stores both instructions and data. The L1 caches are 16KB, 4-way, set-associative caches with a 32 byte line size. The L2 cache is a 512KB, 4-way, set-associative cache, also with a 32 byte line size. The operating system was Linux, kernel version 2.4.18.

The Pentium III processor includes two event counters that are available for recording events, such as the number of instructions executed. To access the event counters, the PAPI library was used [8]. The events measured include: the number of cycles executed, the number of instructions executed, and the number of L2 cache misses incurred. In the experimental results that follow, only the execution time and the number of cache misses are reported.

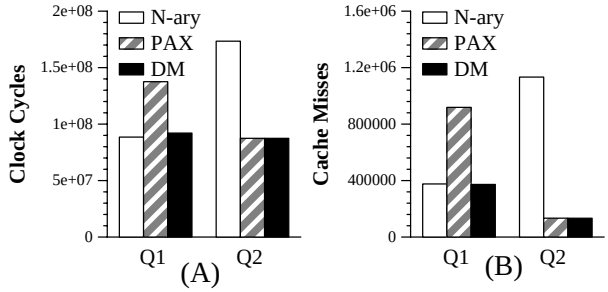

Figure 9: Baseline Performance

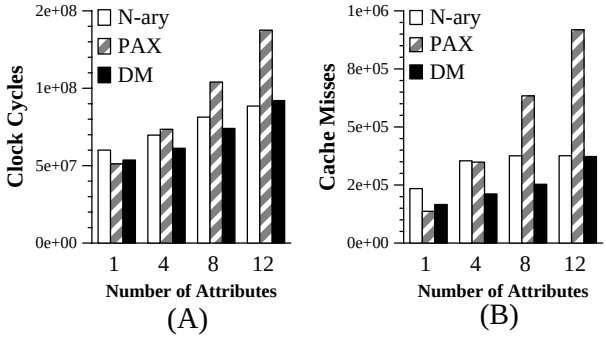

Figure 10: Q1: Attribute Scaleup

The PAX storage model was implemented using the Data Morphing slotted-pages, where each attribute is assigned to a separate zone. We chose to replace variable-length attributes with fixed-length attributes, if the values in the variable-length attribute are 32 bytes, or smaller, in size. It is possible to concurrently prefetch cache-lines but we do not implement this optimization. Adding cache-line prefetching *may* improve the performance of both the PAX storage model and the DM model, but prefetching cache lines can hurt performance if done incorrectly [1]; therefore, this optimization is left for future work.

For all the experiments presented here, the Hill-Climb algorithm produced the same partitions as the Naive algorithm.

5.2 Queries

The following experiments used query Q1 and query Q2, shown in Table 4. These queries select records from the the Wisconsin Benchmark's [14] `TENK1` relation, scaled to one million records. A non-clustered, B+-tree index is constructed on the `unique1` attribute of this relation.

To answer a given query, the database system's query optimizer selects an index scan operator over a sequential file scan operator if (1) there exists a predicate on the `unique1` attribute, and (2) the selectivity is estimated to be 10%, or less. The index scan operator uses the B+-tree that is constructed on the `unique1` attribute.

#	Query	Access Plan
Q1	SELECT [varies] FROM Tenk1 WHERE `unique1` $< 100,000$	Non-clustered index scan
Q2	SELECT [varies] FROM Tenk1 WHERE `unique1` $< 200,000$	Sequential file scan

Table 4: Queries

Query Q1 selects 10% of the records in the *TENK1* relation and accesses a variable number of attributes from each selected record. Query Q2 selects 20% of the records in the `TENK1` relation. Using the criteria specified for the query optimizer, Q1 is executed using an index scan operator and Q2 is executed using a sequential file scan.

5.3 Experiment 1: Baseline Performance

In this experiment, we executed queries Q1 and Q2, shown in Table 4. Query Q1 selects the first twelve attributes from the relation `TENK1`, and query Q2 selects only the first attribute from relation `TENK1`.

Figure 9 shows the execution time and cache misses experienced for queries Q1 and Q2. Query Q1 typifies a query that can be most efficiently executed when the data is stored in the N-ary storage model. Query Q1 is executed 36% faster with the N-ary model than with the PAX model. In addition, the N-ary storage model incurred 59% fewer cache misses than PAX. By allocating the attributes in a single zone, the DM storage model is similar to the N-ary storage model; therefore, the performance of the query on DM was very close to the performance on the N-ary model.

The PAX storage model provides cache efficient data storage for plans that access a small number of attributes from a high percentage of the records in a file. Query Q2 typifies such a query. As shown in Figure 9, query Q2 was 50% faster using PAX versus the N-ary storage model. In addition, with PAX, this query experienced 88% fewer cache misses than when using the N-ary storage model. By allocating each attribute in its own zone, the DM storage model is identical to the PAX storage model and shows similar performance.

5.4 Experiment 2: Attribute Scale-up

We now examine the performance sensitivity of each storage model when accessing an increasing number of attributes from each selected record. For this analysis, we again used queries Q1 and Q2, and changed the project list to include an increasing number of randomly chosen attributes.

Figure 10 shows the execution time and cache misses for query Q1, as the number of attributes accessed increased. When projecting eight attributes, the DM storage model resulted in 29% faster evaluation as compared to PAX, and incurred 60% fewer cache misses. Compared to the N-ary model, DM performed 9% faster, with 33% fewer cache misses.

The results shown in Figure 10 demonstrate that, for an index scan operation, partitioning the attributes into sequential groups is more efficient than vertically decompos-

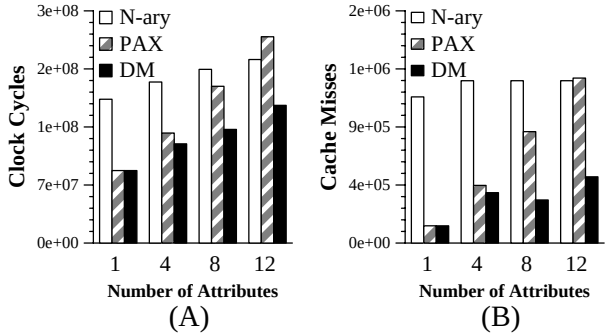

Figure 11: Q2: Attribute Scaleup

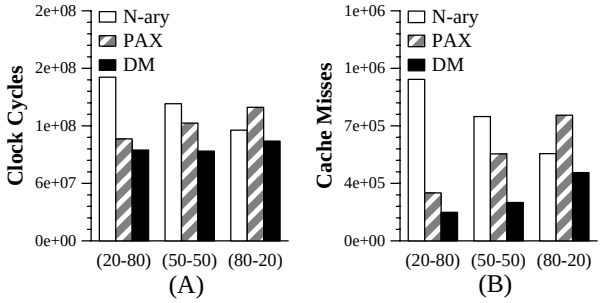

Figure 12: Workload Performance

ing each attribute into separate groups. The N-ary storage model performs worse than the DM storage model because non-sequential attributes have a higher probability of being allocated to memory addresses that map to different cache lines. The Data Morphing process improves the spatial locality of those non-sequential attributes by placing them in a group where the attributes are stored in consecutive memory addresses.

Figure 11 shows the execution time and cache misses for query Q2, as the number of attributes selected increased. From the figure, when selecting eight attributes, DM was 35% faster than N-ary and incurred 73% fewer cache misses. DM was also faster than PAX by 28% and incurred 61% fewer cache misses.

Figure 11 illustrates that vertically decomposing the attributes into separate groups improves the data locality of the attributes that are accessed during a sequential scan of the file. But, as the number of projected attributes increases, the vertical decomposition becomes less cache-efficient. Data Morphing performs well as the number of projected attributes increases because, unlike the PAX model, the attributes can be stored in sequential memory, and, unlike the N-ary model, the attributes can be grouped into a single cache line.

5.5 Experiment 3: Query Workload

In this experiment, we examined the benefits of using the Data Morphing process for a workload of queries. Workload W1 consists of two queries, Q1 and Q2 (Table 4). Each query in this workload was executed a number of times that is determined by the ratio of Q1 to Q2. The ratios used in this experiment are (Q1%-Q2%): 20%-80%, 50%-50%, and 80%-20%. These ratios describe the number of times query Q1 and query Q2 were executed as a percentage of the total number of queries executed.

Figure 12 shows the execution time and cache misses for the database system when executing these two queries. From the figures, the workload performance when using DM was 25% faster than PAX, with 46% fewer cache misses. Compared to the N-ary storage model, the performance when using DM was 45% faster, with 82% fewer cache misses. This experiment demonstrates the DM storage technique's ability to adapt to a dynamic query workload. By analyzing the workload, the Data Morphing technique found attribute partitions that increased the overall spatial locality of the data and, as a result, improved the performance of the system.

5.6 TPC-H Benchmark Queries

To further substantiate the performance results, we examined two queries from the TPC-H benchmark: query 6 and query 12. Both queries were evaluated using a sequential file scan. Query 12 also required a join operation between two tables. The join operation was executed using a hash join.

Figure 13 shows the execution time and the number of cache misses incurred during the evaluation of each query. From the figure, for query 6, using DM was 9% faster than PAX and incurred 18% fewer cache misses. Compared to N-ary, the DM storage resulted in a 41% improvement in query response time and incurred 80% fewer cache misses. For query 12, the DM model resulted in a 6% improvement in response time over PAX and incurred 5% fewer cache misses. For this same query, DM was 29% faster than N-ary and incurred 64% fewer cache misses.

The results of both query 6 and query 12 are similar to the results of executing query Q2 in Experiment 2 (Figure 11). In Experiment 2, we show that PAX and Data Morphing perform similarly when executing a sequential file scan and projecting only a few attributes from every selected record.

5.7 Bulkload Time

We also measured the effect of the various storage models on the time it takes to bulkload the `TENK1` relation. The bulkloading time for DM pages never exceeded 10% of the time to bulkload that same relation into the N-ary format. As our page structure is very similar to the PAX page structure, these results are similar to the bulkloading performance of PAX [3]; in the interest of space, we omit this graph.

5.8 Algorithm Performance

We verified the execution time of the two partitioning algorithms by calculating an attribute partition for a relation, named `TENKX`, that is similar to the `TENK1` relation but contains an increasing number of attributes. For input to

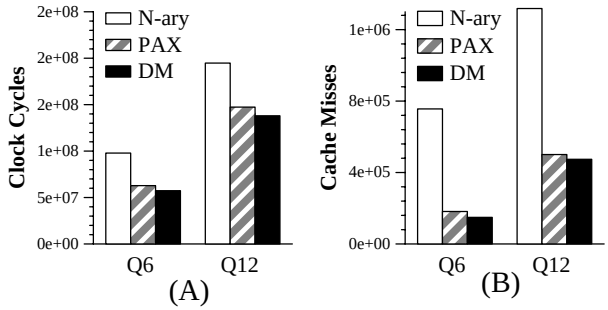

Figure 13: TPC-H Performance

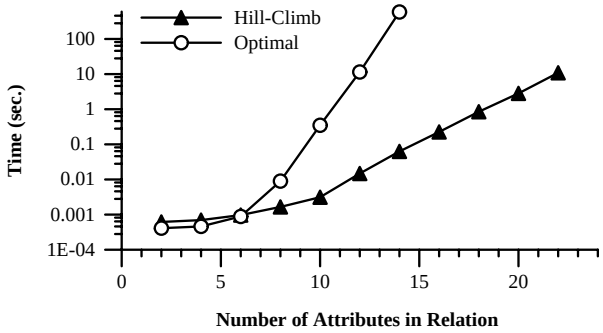

Figure 14: Algorithm Performance

the algorithms, we modeled queries Q1 and Q2 operating on the TENKX relation. Query Q1 projects all attributes from 10% of the records in TENKX. Query Q2 projects only the first attribute from 100% of the records in TENKX. We varied the number of attributes in relation TENKX from 2–22 attributes. Figure 14 shows the resulting execution time for both algorithms on a logarithmic scale. As expected, the Naive algorithm was much more expensive to compute than the Hill-Climb algorithm. When the relation contained more than 16 attributes, the Naive algorithm became prohibitively expensive. While the Hill-Climb algorithm was also exponential in time, the algorithm performed well for a much larger relation size.

5.9 Summary

In this section, we have experimentally evaluated the effects of using the N-ary, PAX, and DM storage models. We have demonstrated the effectiveness of the DM storage model over the N-ary and the PAX storage models. We have also shown that, for a heterogeneous workload of queries, neither the PAX nor the N-ary storage models provide the most cache-efficient storage model. The DM storage model is more efficient in such cases. Finally, we note that the cache-miss latency is expected to increase over the next several years, so the performance improvement from using Data Morphing will increase relative to PAX and the N-ary storage model.

6 Related Work

The Decomposition Storage Model (DSM) was proposed as an alternative to the N-ary Storage Model in [12]. DSM decomposes the attributes of a relation into sub-relations, with one attribute per sub-relation. DSM requires expensive sub-relation joins to access attributes that are contained in different sub-relations. Monet is a main-memory database system that utilizes DSM to reduce the need for main-memory bandwidth [6, 7].

An alternative to the N-ary storage model and DSM was introduced by Ailamaki, et al., called *PAX* for Partition Attributes Across [2]. PAX is a page level decomposition model where each attribute is stored in sub-divided regions of a page, called *mini-pages*. Unlike DSM, PAX does not require expensive reconstruction joins to access multiple attributes. The PAX storage model was found to significantly improve query execution time for sequential file scans because of the improved data locality. Data Morphing uses a page structure similar to PAX for decomposing attributes, but Data Morphing provides a more general storage model by allowing records to be decomposed into groups of attributes.

Decomposing a single relation into multiple relations to increase system performance has been studied in [18]. The proposed algorithm requires the computation of an attribute affinity matrix that records the pair-wise frequency of accessing the attributes. After decomposing the relation into multiple relations, an expensive join operation must be used to retrieve the record. We are proposing a page-level decomposition of a relation that does not require any additional join operation for retrieving the decomposed records.

Data locality techniques for improving cache utilization has been studied in [9, 11, 22]. Data Morphing takes advantage of data locality by analyzing the access pattern of the attributes in a relation and then grouping attributes with similar access patterns. Data Morphing is the only system that we are aware of that dynamically adjusts the storage model to account for access locality.

7 Conclusions and Future Work

Processor cache performance is critical to DBMSs in which the data is primarily main memory resident. In this paper, we have presented a technique, called Data Morphing, for improving the utilization of the processor cache by dynamically reorganizing the attributes of a record in memory. Through this reorganization, attributes that are accessed together are collocated in the same cache line, improving performance through a reduction in the number of cache misses.

Through experimental analysis, we have shown that the Data Morphing technique reduces the number of cache misses incurred during query execution; as a direct result, the database system experiences better overall performance. We have also shown that the partitioning algorithms provide cache-efficient organizations for the data, performing up to 45% faster than the N-ary storage model and up to 25% faster than the PAX storage model when executing a workload of queries.

Our experimental analysis was based on querying datasets that reside entirely in the main memory. In future work, we will examine the system performance when executing queries on much larger datasets. Increasing the size of the dataset will increase the amount of disk accesses required in executing the query workload. Similar to PAX, we expect the performance benefits of Data Morphing to reduce as the cost of servicing disk I/O becomes the dominating cost.

In addition, we have only examined Data Morphing as applied to the traditional slotted page of records. One possible direction for this work is to incorporate it into native XML databases, such as Natix [16]. In Natix, the trees that represent XML documents are decomposed into clusters of nodes, and each cluster is treated as a record. A decomposition algorithm is used to calculate the composition of each cluster, as this composition is critical to the performance of the query workload. Since accessing a node in a cluster is analogous to accessing the attribute of a record, we expect that the Data Morphing technique can be used to provide cache-efficient layouts for the storage of these clusters.

8 Acknowledgements

This research was supported by the National Science Foundation under grant IIS-0093059. We would like to thank Murali Annavaram and James Mickens for their valuable comments on earlier drafts of this paper.

References

[1] *The IA-32 Intel Architecture Software Developer's Manual, Volume 3: System Programming Guide*. Intel Corporation, 2002.

[2] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. Weaving Relations for Cache Performance. In *Proceedings of the 27th International Conference on Very Large Databases (VLDB)*, pages 169–180, Sept. 2001.

[3] 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 Databases (VLDB)*, pages 266–277, Sept. 1999.

[4] E. T. Bell. Exponential Numbers. *American Mathematical Monthly*, 41(7):411–419, Aug. - Sept. 1934.

[5] P. A. Bernstein, M. L. Brodie, S. Ceri, D. J. DeWitt, M. J. Franklin, H. Garcia-Molina, J. Gray, G. Held, J. M. Hellerstein, H. V. Jagadish, M. Lesk, D. Maier, J. F. Naughton, H. Pirahesh, M. Stonebraker, and J. D. Ullman. The Asilomar Report on Database Research. *SIGMOD Record*, 27(4):74–80, Dec. 1998.

[6] P. Boncz and M. Kersten. Monet: An Impressionist Sketch of an Advanced Database System. In *In Proceedings of Basque Int. Workshop on Information Technology (BIWIT), San Sebastian, Spain*, July 1995.

[7] P. A. Boncz, S. Manegold, and M. L. Kersten. Database Architecture Optimized for the New Bottleneck: Memory Access. In *Proceedings of the 25th International Conference on Very Large Databases (VLDB)*, pages 54–65, Sept. 1999.

[8] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A Portable Programming Interface for Performance Evaluation on Modern Processors. *The Int'l Journal of High Performance Computing Applications*, 14(3):189–204, Fall 2000.

[9] B. Calder, K. Chandra, S. John, and T. Austin. Cache-conscious data placement. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 139–149, Oct. 1998.

[10] A. Cardenas. Analysis and Performance of Inverted Data Base Structures. *Communications of the ACM*, 18(5):253–263, May 1975.

[11] T. Chilimbi, M. D. Hill, and J. R. Larus. Cache-Conscious Structure Layout. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 1–12, May 1999.

[12] G. P. Copeland and S. F. Khoshafian. A Decomposition Storage Model. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 268–279, May 1985.

[13] N. de Bruijn. *Asymptotic Methods in Analysis*. Dover, 1981.

[14] D. J. DeWitt. The Wisconsin Benchmark: Past, Present, and Future. In J. Gray, editor, *The Benchmark Handbook for Database and Transaction Systems (2nd Edition)*. Morgan Kaufmann, 1993.

[15] R. A. Hankins and J. M. Patel. Data Morphing: An Adaptive, Cache-Conscious Storage Technique. *Technical Report, http://www.eecs.umich.edu/quickstep/publ/dm.pdf*.

[16] C.-C. Kanne and G. Moerkotte. Efficient storage of XML data. In *Proceedings of the International Conference On Data Engineering (ICDE)*, page 198, 2000.

[17] S.-W. Kim, W. Choi, and B.-H. Kim. Design and Implementation of the Concurrency Control Manager in the Main-Memory DBMS Tachyon. In *26th Annual International Computer Software and Applications Conference*, pages 635–644, Aug. 2002.

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

[19] A. M. Odlyzko. Asymptotic Enumeration Methods. In R. L. Graham, M. Grötschel, and L. Lovsz, editors, *Handbook of Combinatorics*, volume 2, pages 1063–1229. North-Holland, Amsterdam, 1995.

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

[21] G.-C. Rota. The Number of Partitions of a Set. *American Mathematical Monthly*, 71(5):498–504, May 1964.

[22] A. Shatdal, C. Kant, and J. F. Naughton. Cache Conscious Algorithms for Relational Query Processing. In *Proceedings of 20th International Conference on Very Large Data Bases (VLDB)*, pages 510–521, Sept. 1994.

[23] S. Yao. An Attribute Based Model for Database Access Cost Analysis. *Communications of the ACM*, 20(4):260–261, Apr. 1977.