Submission to HPTS 2001

A Storage Model to Bridge the Processor/Memory Speed Gap

Anastassia Ailamaki
Carnegie Mellon University

(joint work with David DeWitt and Mark Hill
at the University of Wisconsin-Madison)

1 Introduction

Memory speeds in today’s computers have fundamentally lagged behind processor speeds [7]. Today's memory systems incur access latencies that are up to three orders of magnitude larger than the latency of a single arithmetic operation. To alleviate the processor/memory performance gap, computer designers employ a hierarchy of cache memories (e.g., three levels in the recently announced IBM Power 4 processors), in which each level trades off higher capacity for faster access times. As database applications become increasingly memory-intensive, high performance database systems must maximize cache utilization by keeping data that are likely to be referenced in the cache hierarchy. Ideally, the database application should run under the illusion that the database is *cache-resident*, i.e., the processor should never be idle due to main memory latency.

When optimizing cache utilization, data placement is extremely important. According to earlier studies [1][3], when running commercial DBMSs on a modern processor, data misses in the cache hierarchy are a key memory bottleneck and a major reason for processor stalls. Choosing a data placement scheme that follows the workload’s memory access patterns improves spatial locality, which in turn improves cache space utilization and performance. More specifically, the *inter-record locality* eliminates delays due to unnecessary memory accesses when evaluating a predicate using a fraction of each record. The *intra-record locality* minimizes the record-reconstruction cost (the delays associated with retrieving multiple fields of the same record).

The traditional data placement scheme used in DBMSs, the N-ary Storage Model (NSM, a.k.a., *slotted pages*), 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. NSM offers high intra-record locality. Query operators, however, often access only a small fraction of each record causing suboptimal cache behavior, that (a) wastes bandwidth, (b) pollutes the cache with useless data, and (c) possibly forces replacement of information that may be needed in the future, incurring even more delays.

As an alternative, the decomposition storage model (DSM) [5] partitions an *n*-attribute relation vertically into *n* sub-relations, each of which is accessed only when the corresponding attribute is needed. DSM maximizes inter-record locality; however, the DBMS must join the participating sub-relations together to reconstruct a record. When running queries that involve few attributes from each relation, DSM saves I/O and improves main memory utilization. As the number of participating attributes per relation increases, however, the record reconstruction cost dominates the query execution. For this reason, *all* of today’s commercial database systems that we are aware of still use the traditional NSM algorithm for data placement

[9][12][13][14]. The challenge is to repair NSM's cache behavior, without compromising its advantages over DSM.

This paper introduces **Partition Attributes Across (PAX)**, a new layout for data records that combines the best of the two worlds and exhibits superior performance 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 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 the same 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.

Section 2 briefly discusses why NSM, the currently used storage model, hurts cache performance. Section 3 presents the design of PAX, while Section 4 discusses PAX/NSM speedup when running decision-support queries on a prototype system. Additional results and further improvements are summarized in Section 5.

2 The N-ary Storage Model

Traditionally, the records of a relation are stored in slotted disk pages [11] obeying an n-ary storage model (NSM). NSM stores records sequentially on data pages. Figure 1 depicts an NSM page after having inserted four records of a relation with three attributes: *SSN*, *name*, and *age* (left). Each record has a record header (RH) that contains a null bitmap, offsets to the variable-length values, and other implementation-specific information [9][12][13]. Each new record is typically inserted into the first available free space starting at the beginning of the page. Records may have variable length, therefore a pointer (offset) to the beginning of the new record is stored in the next available slot from the end of the page. The n^{th} record in a page is accessed by following the n^{th} pointer from the end of the page.

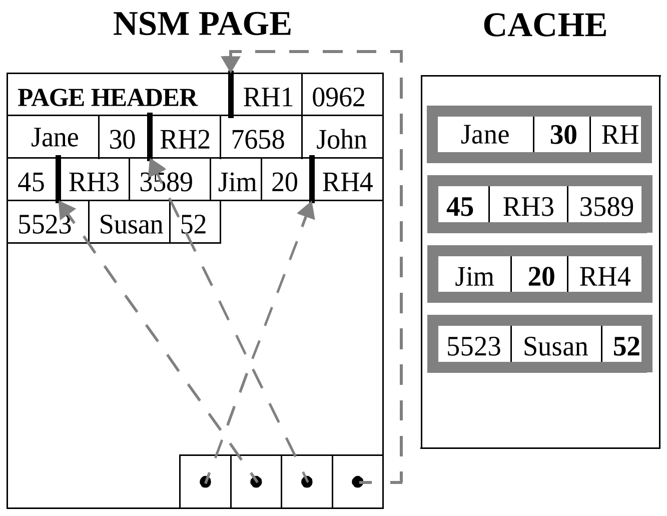

FIGURE 1: The cache behavior of NSM.

During predicate evaluation, however, NSM exhibits poor cache performance (right). For instance, to evaluate a predicate, the query processor uses a scan operator that retrieves the value of the corresponding attribute from each record in the relation. Assuming that the cache block size (typically 32-128 bytes) is smaller than the record size, the scan operator will incur one cache miss per record (as illustrated in Figure 1). In addition, along with the needed value, each cache miss will bring into the cache unreferenced data (values stored next to that of interest), wasting useful cache space and incurring unnecessary accesses to main memory.

3 Partition Attributes Across

We introduce a new strategy for placing records on a page called PAX (Partition Attributes Across). When using PAX, each record resides on the same page as it would reside if NSM were used; however, all *SSN* values, all *name* values, and all *age* values are grouped together on minipages (for example, the PAX page in Figure 2 stores the same records as the NSM page in Figure 1). 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. Therefore, only useful information is transferred into the cache.

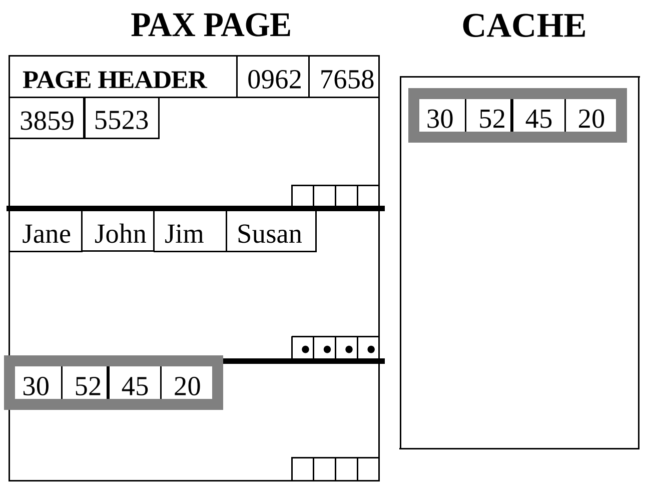

FIGURE 2: The cache behavior of PAX.

Figure 3 illustrates the design details of an example PAX page in which two records have been inserted. At the beginning of each page there is a page header that contains offsets 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 (e.g., *SSN* and *age*) 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 (e.g., *name*) 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 a number of minipages equal to 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 still available.

To store a relation, PAX requires the same amount of space as NSM. NSM stores the attributes of each record contiguously, therefore it requires one offset (slot) per record and one additional offset for each variable-length attribute in each record. PAX, on the other hand, stores one offset for each variable-length value, plus one offset for each of the *n* minipages. Therefore, regardless of whether a rela-

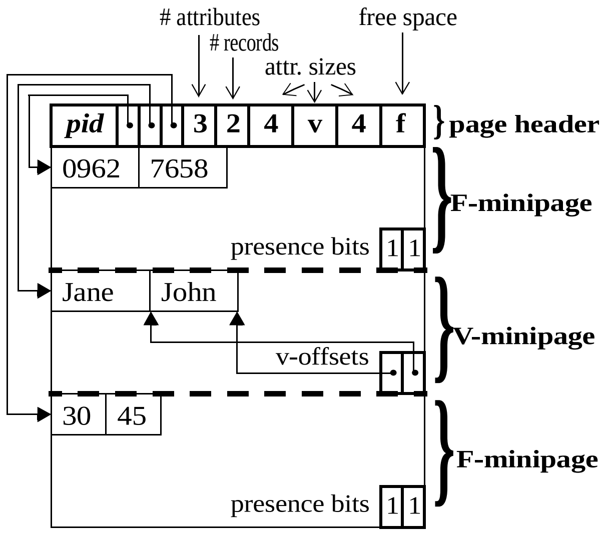

FIGURE 3: An example PAX page.

tion is stored using NSM of PAX, it will occupy the same number of pages. Implementation-specific details may introduce slight differences which are insignificant to the overall performance.

4 Evaluation using a DSS workload

Figure 4 depicts PAX/NSM speedups when running range selections and four TPC-H queries against a 100, 200, and 500-MB TPC-H database on top of the Shore storage manager [4]. The detailed experimental setup and methodology are described elsewhere [2]. Decision-support systems are especially processor- and memory-bound, and PAX outperforms NSM for all these experiments. The speedups obtained 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.

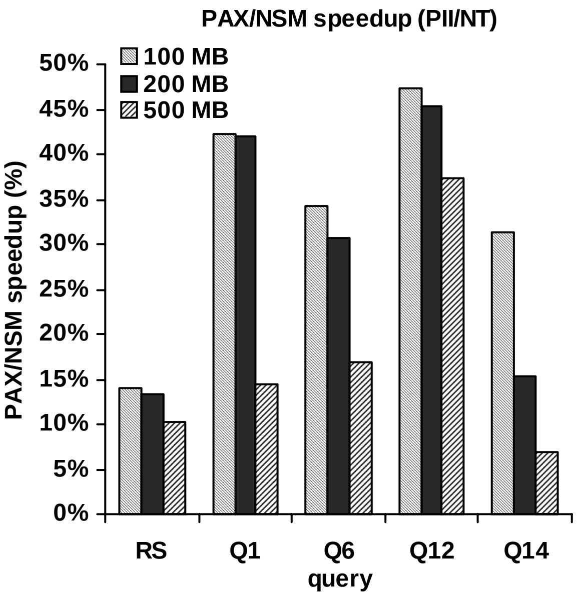

FIGURE 4: PAX/NSM speedup for DSS queries.

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 the previous paragraph is that TPC-H queries exploit further the spatial locality, because they access projected data multiple times in order 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 (using the adaptive dynamic hash join algorithm [10]), as well as range predicates. 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. A more complete study [2] shows results and a sensitivity analysis when we vary the degree of the relation and several other query parameners, and when running insertions and updates.

5 Summary

The traditional N-ary storage model's poor spatial locality has a negative impact on data cache performance. Alternatively, the decomposition storage model (DSM) partitions relations vertically, creating one sub-relation per attribute. DSM exhibits better cache locality, but incurs a high record reconstruction cost. For this reason, most commercial DBMSs use NSM to store relations on the disk.

This paper introduces PAX (Partition Attributes Across), a new layout for data records on pages that combines the advantages of NSM and DSM. For a given relation, PAX stores the same data on each page as NSM. The difference is that within each page, PAX groups values for the same attribute together in minipages. PAX balances the tradeoff between cache space utilization and record reconstruction cost by improving inter-record spatial locality while keeping all parts of each record in the same page at no extra storage overhead. The results show that PAX outperforms NSM on all TPC-H queries in this workload:

- When compared to NSM, PAX reduces L2 cache data misses and stall time by a factor of four. 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 retains all the advantages of NSM and executes queries faster because it combines high cache performance with minimal reconstruction cost.

PAX has several additional advantages. Implementing PAX on a DBMS that uses NSM requires only changes on the page-level data manipulation code. As a low-overhead solution with a high impact on performance, PAX is particularly attractive to existing large DBMSs. In addition, 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. PAX is orthogonal to other design decisions, because it only affects the layout of data stored on a single page (e.g., one may first use affinity-based vertical partitioning, and then use PAX for storing the 'thick' sub-relations). Finally, research [6] has shown that compression algorithms work better with vertically partitioned relations and on a per-page basis, and PAX has both of these characteristics.

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, D. J. DeWitt, M. D. Hill, and M. Skounakis. Weaving Relations for Cache Performance. Submitted for review to the *27th International Conference on Very Large Data Bases (VLDB),* to be held in Rome, Italy, September 2001.

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

[4] 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 th_e ACM SIGMOD Conference on Management of Data,* Minneapolis, MN, May 1994.

[5] 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.

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

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

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

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

[10] 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.

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

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

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

[14] http://www.ncr.com/sorters/software/teradata_or.asp