A hybrid page layout integrating PAX and NSM

Goetz Graefe, Ilia Petrov, Todor Ivanov, Veselin Marinov

HP Laboratories
HPL-2012-240

Keyword(s):

database; storage manager; PAX; NSM; Hybrid Page Layout (HPL); page layout

Abstract:

Prior work on in-page record formats has contrasted the "N-ary storage model" (NSM) and the "partition attributes across" (PAX) format. The former is the traditional standard page layout whereas the latter "exhibits superior cache and memory bandwidth utilization" [ADH 01], e.g., in data warehouse queries with large scans. Unfortunately, space management within each page is more complex due to the mini-pages in the PAX layout. "Borrowing" space from one mini-page for another requires moving an entire mini-page. In contrast, the NSM format simply grows a slot array and the data space from opposite ends of the page until all space is occupied. The present paper explores a hybrid page layout (HPL) that aims to combine the advantages of NSM and PAX. Predicate evaluation in large scan queries have the same number of cache faults as PAX, and space management uses two data areas growing towards each other. Moreover, the design defines a continuum between NSM and PAX in order to support both efficient scans and efficient insertions and updates. This design is equally applicable to cache lines within RAM memory (the original design goal of PAX) and to small pages on flash storage within large disk pages. Our experimental evaluation is based on an implementation in the former environment. It demonstrates that the HPL design scans almost as fast as the scan-optimized PAX layout and updates almost as fast as the update-optimized NSM layout, i.e., it is competitive with both in their best use cases.

External Posting Date: December 6, 2012 [Fulltext] Approved for External Publication
Internal Posting Date: December 6, 2012 [Fulltext]

A hybrid page layout integrating PAX and NSM

Goetz Graefe – Ilia Petrov, Todor Ivanov, Veselin Marinov
Hewlett-Packard Laboratories – Technical University Darmstadt

Abstract

Prior work on in-page record formats has contrasted the "N-ary storage model" (NSM) and the "partition attributes across" (PAX) format. The former is the traditional standard page layout whereas the latter "exhibits superior cache and memory bandwidth utilization" [ADH 01], e.g., in data warehouse queries with large scans. Unfortunately, space management within each page is more complex due to the mini-pages in the PAX layout. "Borrowing" space from one mini-page for another requires moving an entire mini-page. In contrast, the NSM format simply grows a slot array and the data space from opposite ends of the page until all space is occupied.

The present paper explores a hybrid page layout (HPL) that aims to combine the advantages of NSM and PAX. Predicate evaluation in large scan queries have the same number of cache faults as PAX, and space management uses two data areas growing towards each other. Moreover, the design defines a continuum between NSM and PAX in order to support both efficient scans and efficient insertions and updates.

This design is equally applicable to cache lines within RAM memory (the original design goal of PAX) and to small pages on flash storage within large disk pages. Our experimental evaluation is based on an implementation in the former environment. It demonstrates that the HPL design scans almost as fast as the scan-optimized PAX layout and updates almost as fast as the update-optimized NSM layout, i.e., it is competitive with both in their best use cases.

1 Introduction

The traditional page layout used in many database systems stores entire records contiguously. For variable-size records, a level of indirection (using an array with byte offsets within the page) is used. A record identifier includes a page identifier plus an index into this array. Each time a record is inserted into a page, the offset array grows towards the data area and the data area grows towards the offset array. The page is full when the two areas threaten to overlap.

This organization is optimal for fetching all fields of a single row based on a record identifier, which is a typical access pattern in online transaction processing. In relational data warehousing, however, many queries cannot be answered by fetching the result using indexes. Instead, many query execution plans require large scans, inspecting the same field in each record in order to find records satisfying the query predicate. For those query execution plans, the traditional in-page record organization is not optimal, due to the number of cache faults and associated execution stalls.

For example, assume records of 100 bytes and a predicate focusing on a single 4-byte field within each record. Assume also a modern processor with multiple levels of caches and with cache lines of 64 bytes. In the traditional page layout, each record forces at least one fault in the CPU cache. Ideally, with cache lines 16 times larger than the field in the query predicate, there should be only one cache fault per 16 records.

This consideration has led to the design of the PAX storage layout. Both the NSM and the PAX storage layouts attempt to put the same records into each page, and thus load the same set of record with each I/O, etc. Their difference is that NSM stores the information in each page one record at a time whereas PAX stores the same information one field at a time, aligned to cache lines. A large scan can perform very efficient predicate evaluation with very few cache faults.

The core design element in the PAX format is a mini-page per field. When a new record is inserted into the page, one value is inserted into each mini-page. *Null* values are indicated in a bit vector for fixed-size fields and, for variable-size fields, by a special value in the offset array pointing to the locations of variable-size values.

One disadvantage of the PAX format is the complexity of free space management within each page for records with one or more variable-size fields. In most cases, the average length of a variable-size field is not predictable. Even if the average size is known for a table, it probably is not predictable for each page. Thus, in order to fit as many records into each page as the NSM format permits, in-page optimizations may be required after the page is filled when one or more of the mini-pages run out of space. The required optimization decisions are a bit complex and may be revised repeatedly, implying expensive data movement.

The first purpose of this paper is to describe a third page layout that combines the advantages of NSM and PAX. From NSM, this design preserves the simplicity of two variable-size allocation spaces growing towards each other. From PAX, it preserves the cache efficiency during large scans. Multiple variants of this third page layout are introduced. For example, the first variant is simple but fairly rigid with possibly substantial fragmentation in each page, whereas the last one is more flexible with less fragmentation in each page.

The basic ideas of NSM and PAX apply not only to traditional pages but also to the organization of records and their fields in flash storage. A recent study demonstrates the advantages of filling small pages (that are appropriate for flash storage and its fast access latency) with values from the same field, and of filling large pages (that are appropriate for disks and their fast transfer bandwidth) with an appropriate collection of such small pages. These small and large pages are very similar to mini-pages and disk pages in the original PAX design.

The second purpose of this paper is to report on a performance evaluation comparing the three alternative designs for database pages in RAM. We found that both cache line access and address calculations are crucial components of scan and update performance.

The next section describes NSM and PAX in more detail. The following sections introduce the hybrid page layout. Both the rigid and the incremental variants of the hybrid page layout are described. After a section reporting an experimental performance evaluation follows a summary and a few conclusions from this effort.

2 Prior work

Our proposed in-page record format aims to combine the advantages of NSM and PAX. These two formats are described and illustrated very aptly and succinctly elsewhere [ADH 01]; rather than attempting to improve upon those descriptions, they are cited verbatim repeatedly here.

2.1 *N-ary storage model (NSM)*

The following diagram, taken from [ADH 01], illustrates four records within a page, including page header and record headers (RH1-RH4). The figure also shows the offset array ("slots") that enables variable-size records and the required space management as well as the free space between data space and offset array growing towards each other during record insertions.

In order to enable efficient insertions and size-changing updates, the page contains two allocation spaces. These allocation spaces, one for records and one for the offset array, grow towards each other from opposite ends of the page. The page is full when they meet. Compaction and reclamation of fragmented free space (after updates) delays that time as much as possible.

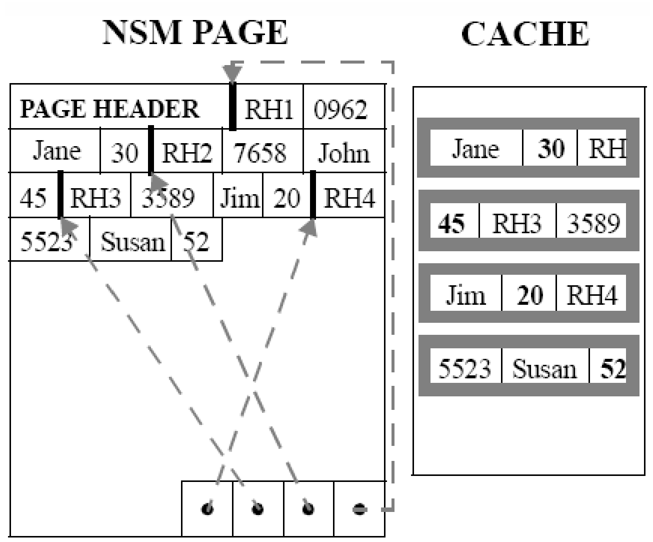

Figure 1. The cache behavior of NSM (from [ADH 01]).

A scan with a predicate on age incurs a lot of cache faults: "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 (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." [ADH 01]

2.2 *Partition attributes across (PAX)*

The following paragraphs and diagrams are the original description of the PAX format [ADH 01], with the numbering of figures adjusted:

"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.

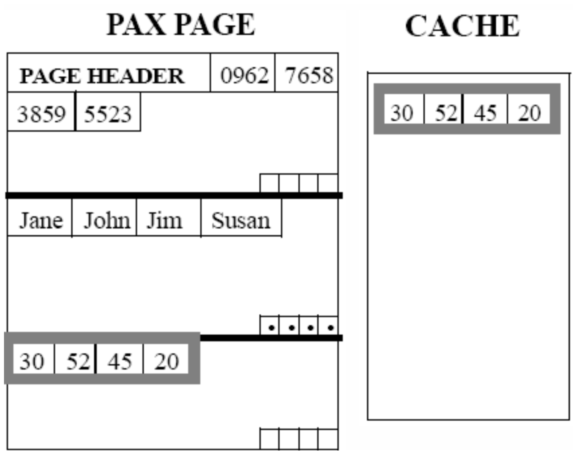

Figure 2. The cache behavior of PAX (from [ADH 01]).

"Figure 2 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."

Unfortunately, the details of the page layout are somewhat complex, as shown in Figure 3. The data structure requires multiple levels of indirection, e.g., for variable-size mini-pages in addition to variable-size field values. Modifications to the space management within a PAX page is also complex and might need to happen more often than in NSM pages. The main reason is that NSM pages have only one pool of empty space ready for allocation, located between the two areas growing towards each other.

PAX pages, on the other hand, have multiple pools of free space, one per mini-page. The fixed-size fields present little problem, but if the sizes of future values in the variable-size fields cannot be predicted accurately and reliably, the size of mini-pages must be adjusted repeatedly. The reorganization and movement effort is exaggerated if mini-pages for fixed-size fields and for variable-size fields are interleaved as shown in Figure 3.

In fact, Figure 3 does not show the full complexity of the design. Specifically, the variable-size field requires either a presence bit (*null* bit) or an additional size field. Otherwise, setting a single variable-size field to *null* cannot be represented without substantial effort or information loss for neighboring fields. If a *null* bit is added, there is no need to overwrite the offset value and the size of the neighboring record can be calculated as the difference between two offsets. If a *null* value in a variable-size field is indicated by a special offset value, then a neighboring field (and thus all variable-size values) requires an additional size indicator. Alternatively, an entire mini-page must be compacted immediately when a valid variable-size value is replaced by a *null*. After the compaction, a zero difference between offsets can represent a *null* value.

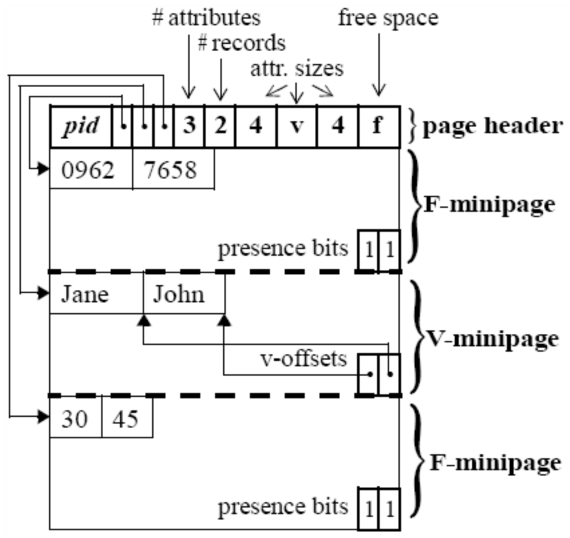

Figure 3. An example PAX page (from [ADH 01]).

It could be argued that a page in NSM format is affected twice by a variable-size field value, because both the field value and the record as a whole require size information and indirection for space management (offsets). A page in PAX format seems to be affected only once and, in fact, requires no size information because it is implied in the difference between two neighboring offsets. This argument is not correct, however. First, a PAX page requires space management for variable-size mini-pages in addition to space management for individual variable-size values. Second, differences between offsets can indicate sizes only if the sequence of offsets equals the sequence of values and if size-changing updates always imply immediate compaction. Neither of these conditions is acceptable if data changes are likely or even frequent. Incidentally, some NSM designs avoid sizes for individual variable-size fields within records using differences of offsets, e.g., IBM's Starburst [HCL 90].

2.3 Discussion

The principal strength of the PAX format, clearly demonstrated in the original research [ADH 01], is a reduction of cache faults and execution stalls during large scans with predicate evaluation. Cache faults and execution stalls were not reported for single-record insertions, deletions, and full-record retrievals based on record identifiers. The cache faults during single-record operations may be expected to be linear with the number of attributes and mini-pages. Bulk loading performs similarly for NSM and PAX because it touches every cache line within a page in either format.

The principal weakness of the PAX format, in our assessment, is the complexity of variable-size updates. A single size-changing update might require that the sizes and placements of several mini-pages change within the page. While physiological logging avoids the need to copy the page contents to the recovery log, it nonetheless is an expensive and complex operation with many cache faults. If the relative size of actual attribute values cannot be estimated reliably a priori, many invocations of this operation may be required over time.

In contrast, the NSM format has a single pool of free space between the offset array and the data records. If record deletions or record-shrinking updates create additional fragmentation, a very simple compaction algorithm can be used. This compaction algorithm may sort all record offsets and process records in this order. A simple optimization avoids copying effort where relocation is not required. Alternatively, it may copy all records to a new page frame. The new page frame then replaces the old one in the buffer pool.

For use with flash storage, both NSM and PAX require slight modifications. A large disk page contains the same set of records with either NSM or PAX. With NSM, entire records are assigned to individual small flash pages. With PAX, each small flash page contains values for one field only, with an appropriate number of small flash pages allocated for each field in the record format.

PAX pages are very efficient for large scans with predicates that require only a few fields from each record. NSM pages are efficient in single-record operations, both retrieval and update. Moreover, the NSM in-page organization with a single pool of free space is particular flexible for records and field values with unpredictable sizes. As both scans and updates are important database operations, it is extremely desirable to employ a page layout that is efficient for both.

3 A simplistic hybrid page layout

The goal of the proposed hybrid page layout is to combine the advantages of the PAX format and of the NSM format, namely efficient predicate evaluation during large scans and efficient space management during insertions, deletions, and size-changing updates. Like PAX, the hybrid page layout fills entire aligned cache lines with values of a single field; obviously, these values belong to multiple records. Like NSM, the hybrid page layout employs only two variable-size allocation spaces and grows them towards each other starting from the opposite ends of the page.

Three variants of the hybrid page layout are introduced. The simplistic variant is meant merely to convey and illustrate the principal idea. The rigid variant adds bits for *null* values and for ghost records. It also adds considerations for read-ahead of cache lines. Finally, the incremental variant of the hybrid page layout reduces fragmentation and enables the hybrid page layout for small pages.

The example of [ADH 01] is used throughout, with each record including a 4-byte *identifier*, a 1-byte *age* field, and a variable-size *name* string requiring a 2-byte in-page offset and a 2-byte size.

3.1 Simple segments

The simplest variant of the hybrid page layout assumes that each field is at least one byte in size (no bit fields) and that all field sizes are multiple bytes. It ignores *null* values and ghost records (also known as pseudo-deleted records).

The essence of the new design is quite simple. An allocation space for fixed-size fields grows from one end of the page and an allocation space for variable-size fields grows from the other end. Offsets for variable-size fields are considered fixed-size fields in their own right. Size information for variable-size fields is treated as another fixed-size field. Alternatively, pairs of offset and size can be treated as a single fixed-size field or the size information can be integrated into the string values.

For fixed-size fields, the set of records is divided into segments. The number of records per segment is equal to the number of bytes per cache line. The number of cache lines per fixed-size field is equal to the number of bytes in each field value.

In the running example from [ADH 01], the fixed-size fields total 4+1+2+2=13 bytes. Thus, each segment contains 13 cache lines. The first 4 cache lines contain *identifier* values, the next one *age* values, the next two cache lines contain offsets, and the last two cache lines in each segment contain size information for the variable-size *name* field.

Segments are numbered starting with 0. Assuming all records within a page are numbered starting with 0, an integer division determines the segment number, another simple calculation determines both the location of the appropriate segment, and a third calculation (division with remainder) finds any field's byte offset within the segment. In the example, with cache lines of 64 bytes, each segment contains field values of 64 records and occupies 13 cache lines or 13×64=832 bytes. Record 145, for example, belongs to segment

145÷64=2 starting at byte offset 2×832=1,664 after the page header. Within the segment, the values of record 145 are found in position 145%64=17. The 4-byte *identifier* value of this record, for example, is located at byte offset 1,664+17×4=1,732.

For variable-size fields, the space management policy treats all values without regard to the record or the field to which the value belongs. Variations to this basic scheme will be considered below.

Space management for the entire page has fixed-size fields and variable-size fields grow towards each other. The space for fixed-size fields grows one segment at a time, i.e., by 13 cache lines at a time in the example. The incremental hybrid page layout will reduce this minimal unit of growth.

3.2 Illustration

Figure 4 illustrates the hybrid page layout and its cache behavior. The page header is omitted in order to focus on the essence of the page layout. The figure shows the same data as the PAX layout illustrated in Figure 3. For example, employee 0962 is named Jane and is 30 years old. The figure assumes 4 bytes per cache line and thus 4 values of each field per segment. In order to better illustrate segments, 1 record has been added compared to the prior figures. The offset values are shown as pointers and the length values are omitted for simplicity in the diagram.

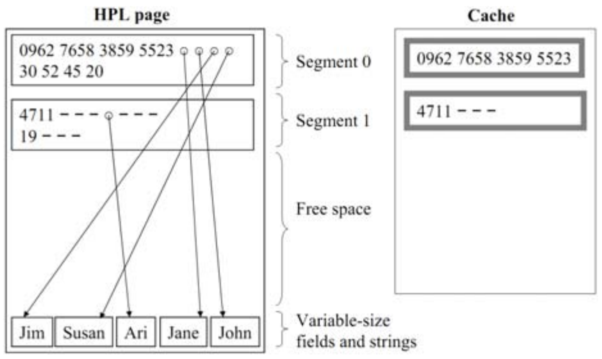

Figure 4. Hybrid page layout – simple variant.

Most prominent in Figure 4 is the single pool of free space between the two allocation spaces for fixed-size values and for variable-size values. This feature is very similar to traditional NSM pages and very different from PAX pages.

The second obvious aspect is that the variable-size fields are all allocated from the same pool of free space. This particular example uses only a single variable-size field; if it had multiple variable-size fields, values of different fields would be mixed freely. In the simplest variant of the hybrid page layout, cache lines are not considered in space allocation for variable-size fields.

Each segment represents 4 records in this example diagram. Thus, 5 records require 2 segments. The pointers (offsets) for variable-size fields are represented just like user-defined fixed-size fields. Note that the 5 values of each field (e.g., employee identifiers) are not contiguous, very different from the PAX design. In the proposed format, each cache line contains values from only one field (as in PAX) but the cache lines holding the same field are distributed over all segments within a page.

3.3 Basic efficiency considerations

The two cache lines shown on the right of Figure 4 are equivalent to the behavior of PAX shown in Figure 2. In a large scan with predicate evaluation touching only a single field of each record, e.g., *age*, the number of cache faults is minimized, and each line fetched from memory into the cache is full with values of the required field. Thus, the number of cache faults in a large scan of the new hybrid format should mirror that of the PAX format.

At the same time, space management within a page is quite similar to that of the NSM format. Two allocation spaces grow towards each other. The difference to the traditional NSM format is that the unit of growth in the fixed-size area is not a single record but a segment.

Thus, in the worst case with only a single record in the last segment, as shown in Figure 4, the space lost to fragmentation is almost equal to one segment. On average for all possible page and record sizes, about half a segment is lost to fragmentation. Some of the variants below reduce the fragmentation.

3.4 Summary: simplistic hybrid page layout

The simplest hybrid page layout suffices to illustrate the principal technique in the proposed hybrid page layouts: Like PAX, it dedicates entire cache lines to individual fields. This technique enables large scans with few cache faults. Like NSM, it employs only two allocation spaces within each page and lets them grow towards each other. This technique promises that size-changing updates can be captured as efficiently as in NSM.

4 A rigid hybrid page layout

The simplistic hybrid page layout is not proposed for implementation; various refinements are required in order to satisfy the requirements of a real system. The present section adds three of these refinements. Each of these adds required functionality but increases the segment size. The resulting segment sizes are not practical for common page sizes. Therefore, the subsequent section introduces partial or incremental segments.

4.1 Read-ahead of cache lines

In many systems, multiple contiguous cache lines can be fetched from memory more efficiently as a single block than in individual cache faults. In order to take advantage of this hardware optimization, the basis for cache line optimizations should focus on a block of cache lines, not a single cache line.

Instead of a single cache line, a small number of cache lines might be employed in this calculation in order to exploit the read-ahead capability of modern caching hardware. For example, if the true size of a hardware cache line is 64 bytes and if the hardware favors blocks of 4 contiguous cache lines, the calculation might employ 256 bytes in place of the cache line size.

In the example from [ADH 01] with 13 bytes in fixed-size fields per record, each segment size contains 256 records and occupies 256×13=3,328 bytes.

4.2 Poor man's normalized keys

Strings can be stored in many forms. For efficient comparisons, normalized keys represent strings such that a simple binary comparison suffices, at least in the default collation sequence and locale (for international strings). For even more efficient search, 'poor man's normalized keys' turn the first few bytes (e.g., 2 or 4 bytes) of a normalized key into a fixed-size field (e.g., an unsigned integer).

In a B-tree or similar index using an NSM page layout, the offset array in each page may include the poor man's normalized keys such that many comparison may avoid cache faults for the main record as well as invocation of a comparison function. In PAX and other scan-optimized page layouts, a poor man's normalized key is a fixed-size field permitting the full set of optimizations for space management and scan performance.

4.3 Ghost records

Many databases employ bits and bitmaps as status indicators for entire records, e.g., "pseudo-deleted" or "ghost" records that are physically present but logically not valid. Many database systems use ghost records in order to simplify transaction rollback after deletion of a record and for other purposes.

In order to optimize cache alignment of such bits, the segment definition can focus on bits instead of bytes. With a cache line of 64 bytes or 512 bits, for example, the segment size is 512 records rather than 64 records. In this variant of the hybrid page layout, the first cache line in each segment is filled with 512 ghost bits. The following cache lines contain 512 field values of the 1st fixed-size field, then 512 values of the 2nd fixed-size field, etc.

In the example of [ADH 01], 512 records with 13 bytes plus a 512 ghost bits result in a segment size of 512÷8+512×13=6,720 bytes.

If the page layout is optimized for blocks of cache lines, e.g., blocks with 4 cache lines of 64 bytes, then each segment is even larger. For example, a block of 4×64=256 bytes or 2,048 bits requires segments of 2,048 records. In the example, each segment thus occupies 2,048÷8+2,048×13=26,880 bytes. Larger records with more fixed-size fields lead to even larger segment sizes. Such segment sizes might be possible in very large disk pages (e.g., 1 MB) but are unrealistic for traditional disk page sizes (e.g., 8 KB) or page sizes optimized for flash storage (e.g., 2 KB).

4.4 Bit fields for null values

For those fields not covered by a "not null" integrity constraint, databases typically have *null* bitmaps. In NSM implementations, there is one such bitmap per record, covering both fixed-size and variable-size fields. Alternatively, variable-size fields with no value can be expressed by equal field offsets, i.e., zero length (although SQL defines an empty string as different from a *null* value). PAX has a "presence bitmap" within each mini-page, as shown in Figure 3.

If presence bitmaps are required to indicate *null* values, they can be additional fixed-size fields. Alternatively, they could be integrated into the cache lines. The former uses a bitmap per record with a bit per field; the latter uses a bitmap per field with a bit per record. We only pursue the former alternative here.

Null bitmaps may force large segments even in designs that do not employ ghost records and ghost bits.

4.5 Summary: rigid segments

The rigid variant adds bits for *null* values and for ghost records. It also adds considerations for read-ahead of cache lines as well as poor man's normalized keys for efficient search on string fields. Both bitmaps and blocks of cache lines increase segments to sizes beyond those of traditional page sizes. In other words, with this design, a single page could not even hold a single segment. The following design of incremental segments addresses this issue.

5 An incremental hybrid page layout

Simplistic segments fail to support *Null* values and efficient deletion of records using ghost bits; rigid segments with all optimizations are too large; and incremental segments attempt to remedy both of those shortcomings at the expense of some additional addresses calculations.

In this design, each full segment has the same size as a rigid segment. However, partially full segments require substantially less space. Thus, incremental segments are suitable for traditional pages sizes (e.g., 8 KB). For large page sizes (e.g., 1 MB), incremental segments leave much less unused space than rigid segments.

5.1 *Incremental segments*

If each cache line contains 64 bytes, if read-ahead of cache lines suggests blocks of 4 cache lines, and if bit-size fields and the number of bits per block determine the segment size, then each segment contains 2K records. Clearly this does not work with pages of 8 KB. Even for large pages, say 1 MB, 2K records per segment seem a lot. For example, if each record contains 150 bytes of fixed-size fields, each segment requires 300 KB. A lot of space would be wasted due to fragmentation within each page.

While it make sense to allocate 2K bits at a time, it might not be as sensible to allocate 2K 8-byte values at a time. The incremental version of the hybrid page layout therefore allocates space in units of equal size independent of the field size. If bits are allocated 2K at a time, fields of 1 byte are allocated 256 at a time, fields of 2 bytes are allocated 128 at a time, etc. Thus, each segment starts with 2K bits for each field, which supports some records with all fields. The number of records is 2K divided by the size of the largest field in bits. This is followed by allocation units for the largest fields in order to increase the number of supported records. The goal is to support the largest number of records for any segment size.

In other words, the smallest fixed-size field size determines the record count per segment. This field may be a bit-size field, e.g., the ghost bit. In this case, the record count per segment is equal to the bit count per cache line. The largest fixed-size field determines the precision with which the record count per page can be adjusted. Contiguous cache lines are allocated to the fields one at a time. The allocation algorithm loops over the fields (including, and starting with, the ghost bits and the *null* bits).

This scheme wastes some space in fragmentation, but never more than a cache line per fixed-size field. On average, half a cache line per field is wasted. This matches the best case for PAX. The hybrid page layout achieve this fragmentation loss quite naturally as the two allocation spaces grow towards each other from opposite ends of the page; PAX, on the other hand, might require multiple reorganization operations unless the sizes of variable-size fields is known and considered a priori.

5.2 *Illustration*

In order to illustrate the full power of incremental segments, the following example relation adds *Null* values.

Identifier	*Name*	*Age*
0962	Jane	30
7658	John	52
3859	Jim	45
5523	Susan	20
4711	Ari	
8334		52
6455	Jason	
3822	Jill	39
3340	Cathy	42

Figure 5. Table with *Null* values.

Figure 5 shows a slightly larger instance of the example table, including some *Null* values in non-key columns. It is a design choice to prepare the physical representation for *Null* values in key columns in case the logical integrity constraint is subsequently dropped. In the following, there is no allowance for *Null* values in the *Identifier* column.

1111 1111 1000 0…0	64 ghost bits
0000 0100 0000 0…0	64 *Null* bits for *Name*
0000 1010 0000 0…0	64 *Null* bits for *Age*
0962 7658	2 4-bytes *Id* values
Offsets of Jane…Susan	4 2-byte *Name* offsets
Sizes of Jane…Susan	4 2-byte *Name* sizes
30 52 45 20 – 52 – 39	8 1-byte *Age* values 2
3859 5523	2 4-byte *Id* values 4
4711 8334	2 4-byte *Id* values
Offsets of Ari…Jill	4 2-byte *Name* offsets
Sizes of Ari…Jill	4 2-byte *Name* sizes 6
6455 3822	2 4-byte *Id* values 8
3340 –	2 4-byte *Id* values
Offsets for Cathy …	4 2-byte *Name* offsets
Offsets for Cathy …	4 2-byte *Name* sizes
42 …	8 1-byte *Age* values 10

Figure 6. Data arrangement in incremental segments.

Figure 6 shows an incremental segment for the table in **Figure 5**. Each line in the box represents one cache line. For a concise illustration, the size of a cache line is assumed to be 8 bytes or 64 bits. The segment starts with a bitmap indicates valid records and ghost records, indicating 9 valid records followed by 55 ghosts or missing records. Next are bitmaps for *Null* values. The column values form groups of 2, 4, or 8 values as required to fill a cache line. Column values are ordered as in the table definition and in **Figure 5**; a different sequence may simplify the implementation of address calculations. Variable-size string columns require 2 fixed-size fields, offset and size. Our implementation adds a third fixed-size field, a prefix of the normalized key called a poor man's normalized key elsewhere [GL 01]. Segments for fixed-size fields and the space for variable-size values grow towards each other until the page is full.

The red numbers trailing the field names indicate how many records are completely covered by the cache lines up to that point. For example, 8 records require 12 cache lines, 3 for bitmaps and 9 for field values. The smallest possible segment requires 7 cache lines and holds 2 records. A full segment of 64 records requires 579 cache lines, 3 for bitmaps and 9×8×8=576 for field values.

5.3 An example size calculation

While the size of a full segment is the same for rigid segments and incremental segments, the interesting case is a segment less than full, e.g., half full. A rigid segment still requires the same size, whereas an incremental segment can be much smaller and fit into any practical page size.

It is fairly straightforward to map a record count to the size of an incremental segment. It requires identifying all fields, including bit fields for ghost and *Null* information as well as pointers and sizes for variable-size fields, determining for each the count of values per cache line and rounding those counts up, and then adding up the number of required cache lines.

For example, assume that the goal is to allocate (and later scan) 2K bits at a time. Bit-size attributes such as ghost and *Null* bits require one allocation for 2K records or a fraction thereof. Byte-size attributes require one allocation for 256 records or a fraction thereof. Two-byte values such as offsets and sizes of variable-size fields require one allocation for 128 records or a fraction thereof. Four-byte values such as many integers poor man's normalized keys require one allocation for 64 records or a fraction thereof.

For the running example shown in earlier diagrams, **Figure 6** shows specific values based on cache lines of 8 bytes. For cache lines of 2K bits or 256 bytes, the increments must be 32 times larger, e.g., 64 *Identifier* values and 128 offsets and sizes for variable-size *Name* values.

Mapping in the opposite direction is a bit more complex and may require an iterative process, i.e., repeatedly testing possible record counts and calculating whether or not those records will fit in the available space. If the record format includes variable-size values, such a calculation is typically moot, since incremental segments and variable-size fields grow towards each other to fill the page similar to the NSM page layout.

5.4 Summary: an incremental hybrid page layout

In summary, incremental segments exploit very large segments where the space is available. For small pages or for the remainder in a large space, incremental segments support the maximal number of records. They waste no space for the largest fixed-size fields and, on average, a half cache line for the other fixed-size fields. Thus, incremental segments avoid the limitations of simplistic segments and the fragmentation problems of rigid segments.

6 Performance evaluation

The design goal of HPL is to combine the advantages of PAX for data warehouse query execution, in particular large scans, and of NSM for transaction processing, i.e., updates of individual rows and records. In order to evaluate HPL, we compared it with both NSM and PAX for queries and updates in TPC-B, -C, and -H.

System configuration: The experimental evaluation was performed on two different machines, based on different CPU architectures and standing for different technological generations. Machine A is an Intel Xeon server with two quad-core Intel Xeon 5630 2.5 GHz processors (256 KB L1 cache, 1 MB L2 cache and 12 MB L3 cache) and 48 GB RAM and newer generation QPI bus architecture. Machine B is a Sun Fire x4440 server, with 64 GB RAM and four quad-core AMD Opteron 8356 2.3 GHz processors (4× 512 KB L1 cache, and 2 MB L2 and L3 cache) and front-side-bus architecture. Both processor types have a cache line size of 64 bytes. Both systems run Debian Linux.

Benchmark implementation: For PAX and NSM, we used the available benchmark implementation in the respective Shore-MT kits. We implemented and validated ourselves the TPC-B, -C, -H kits for HPL. All experiments were performed on memory resident databases (using RAM disk). We experimented with different data sets and database sizes; detailed information on the benchmark instrumentation is provided in the respective section.

6.1 Query execution: TPC-H

We configured TPC-H with two different data set sizes, namely scale factors 0.5 and 1. Since the differences are negligible, results are reported only for SF 1.

Within the TPC-H workload, we focused on queries Q1, Q6, Q12 and Q14, because they represent different access patterns. Query Q1 scans a large table with almost all record satisfying the predicate and forms intermediate results records for the next operation, an aggregation with a very small result. The scan in query Q6 disqualifies most records based on a single field, the ideal case for PAX. Queries Q12 and Q14 are also large queries but join efficiency, not scan efficiency, determines their performance. Below we report the average sequential execution times of 50 test runs and the respective standard deviations.

HPL, even with incremental segments, suffers some fragmentation, which can be minimized by pages sizes optimized for analytical query processing. Unfortunately, Shore-MT limits the page size of 32 KB. In order to emulate the effect of larger pages and the reduced loss to fragmentation, we scaled the number of records per page in all page layouts to 204 records for the 'line item' table, 256 records for the 'orders' table, and 199 records for the 'part' table.

Table 1. TPC-H performance results.

Machine A/Intel – Scale Factor 1								
query	NSM			PAX			HPL	
	Time [s]	Stdv [s]	Perf.Δ [%]	Time [s]	Stdv [s]	Perf.Δ [%]	Time [s]	Stdv [s]
Q1	0.58	0.02	8.4	0.49	0.01	-10.9	0.55	0.01
Q6	0.46	0.01	6.9	0.41	0.01	-5.1	0.43	0.01
Q12	3.59	0.12	6.6	3.40	0.09	0.8	3.36	0.11
Q14	3.39	0.09	-6.6	3.38	0.11	-7.1	3.63	0.13

Machine B/AMD – Scale Factor 1								
query	NSM			PAX			HPL	
	Time [s]	Stdv [s]	Perf.Δ [%]	Time [s]	Stdv [s]	Perf.Δ [%]	Time [s]	Stdv [s]
Q1	1.16	0.01	-3.5	1.06	0.01	-11.5	1.20	0.01
Q6	0.97	0.01	1.3	0.94	0.01	-1.6	0.96	0.01
Q12	7.17	0.10	3.9	6.81	0.09	-1.3	6.90	0.09
Q14	8.53	0.21	-8.3	8.24	0.13	-11.4	9.30	0.61

Table 1 shows the measured query execution times. Values in the Δ columns indicates the difference of the prior techniques to HPL; a positive value means the HPL is faster than the prior technique. The results demonstrate that the performance of HPL falls between PAX and NSM under analytical loads. HPL is consistently faster than an NSM except for Q14, and Q1 on machine B. HPL is consistently slower than PAX (by 5-11%). Query Q1 and Q6 represent large scans, but of different types. In Q1, most records are selected and multi-field records are extracted for the next operation in the query execution plan. Assembly of records requires repeated address calculations. Even with optimizations and caching of prior addresses, these calculations incur many L1 cache misses. This is clearly the Achilles heel of HPL. In Q6, on the other hand, scans primarily one field, where the majority of HPL's cache efficiency optimizations come to effect. Therefore, the performance difference between HPL and PAX is fairly small – 5% on machine A and 1.6% on machine B.

Q12 and Q14 stress the performance of different join algorithms and management of temporary intermediate query results. HPL has good performance on Q12 due to the use of poor man's normalized keys.

In summary, field scan operations are marginally slower on HPL than on PAX and much faster than on NSM due to better cache efficiency. Record scan operations are slower on HPL compared to both PAX and NSM, because of the high record cost of record construction due to scattered reads.

6.2 Database updates: TPC-B

To evaluate the update performance of HPL, in particular with NSM, we implemented and performed TPC-B and TPC-C experiments. TPC-B was instrumented with two different data set sizes, scale 1 and scale 10. For brevity, we only report results for scale 10.

Table 2 summarizes the results. Clearly, in update-intensive environments such as TPC-B, HPL achieves very good performance. The average transactions throughput is comparable to that of NSM on both machine A and machine B. The performance difference is within one standard deviation.

Table 2. TPC-B performance results

Machine A/Intel – Scale 10				
NSM [tps]	NSM stdv [tps]	Perf.Δ [%]	HPL[tps]	HPL stdv [tps]
18394	1%	-2%	18033	2%

Machine B/AMD – Scale 10				
NSM [tps]	NSM stdv [tps]	Perf.Δ [%]	HPL [tps]	HPL stdv [tps]
11580	1%	3%	11903	2%

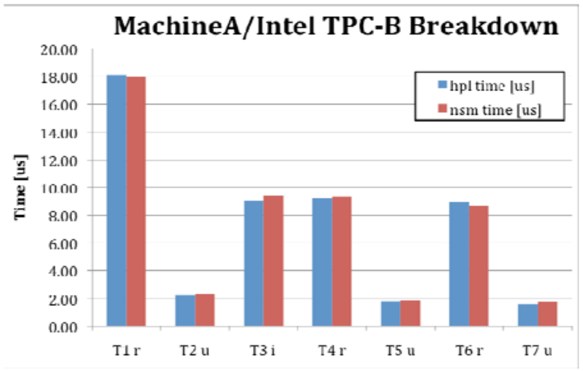

Figure 7. TPC-B transaction breakdown (machine A).

Figure 7 breaks down the performance of each statement in one of the TPC-B transaction, specifically the "account update" transaction. Tj stands for the j^{th} statement within the transaction; r, i, and u stand for retrieval, insertion, and update. Figure 8 shows the same information for machine B.

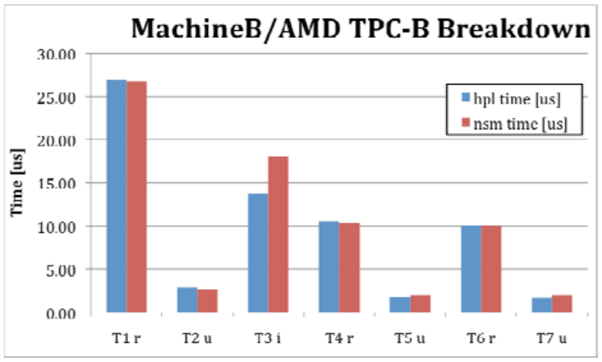

Figure 8. TPC-B transaction breakdown (machine B).

The numbers substantiate several rules that flow from the HPL design hypotheses (Section 5): (i) updates, especially single-field updates, are about 5% faster on HPL since they can be performed in place whereas Shore-MT always replaces entire records (due to its prefix truncation techniques); (ii) reads, especially those that construct a whole record, are faster on NSM, since HPL performs multiple address calculations and multiple scattered read operations; (iii) single-field reads (e.g., T1) exhibit comparable performance.

In brief, field update operations, are 5% faster on HPL. HPL and NSM perform very similarly under write intensive loads.

6.3 *OLTP Database Load: TPC-C*

Finally, TPC-C experiments stress the insert, random read, and delete operations of HPL as well as record-to-record and index-to-index navigation, e.g., searching a secondary index and fetching additional columns from a primary index [HSY 01, LD 93]. TPC-C is more read-intensive than TPC-B; the read operations are mostly random or record-to-record navigation. Record construction is common. In addition, the "delivery" transactions stresses delete mechanisms and the "new order" and "payment" transactions also test the insert functionality.

Table 3 shows the transaction throughput for individual transactions on HPL and NSM as well as the entire TPC-C transaction mix. Clearly, updating transactions such as "new order" and "payment" are about 11% slower on HPL than on NSM. This issue is addressed later in this section.

Table 3. TPC-C performance results

Machine A/Intel – Scale 10 Warehouses			
Tx. Name	NSM [tps]	Perf.Δ [%]	HPL [tps]
New Order	1649	-11%	1465
Payment	11855	-11%	10523
Order Status	9575	-9%	8674
Delivery	13282	3%	13648
Stock Level	518	1%	523
Mix	2198	-5%	2097

Machine B/AMD – Scale 10 Warehouses			
Tx. Name	NSM [tps]	Perf.Δ [%]	HPL [tps]
New Order	1149	-11%	1020
Payment	7494	-11%	6654
Order Status	6330	-11%	5605
Delivery	9731	6%	10353
Stock Level	340	3%	349
Mix	1516	-6%	1421

The "delivery" transaction performs equally well on both page formats. The delete operations on HPL perform about 15% better on than on NSM (Table 3, Figure 9, Figure 10). This is due to the ghost bit vector. Single field updates such as those measured by T4 ru or T5 ru are about 5% faster on HPL, which confirms the results in Section 6.2.

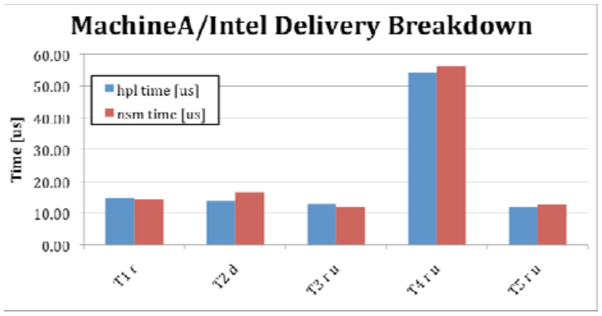

Figure 9. TPC-C "delivery" transaction (machine A).

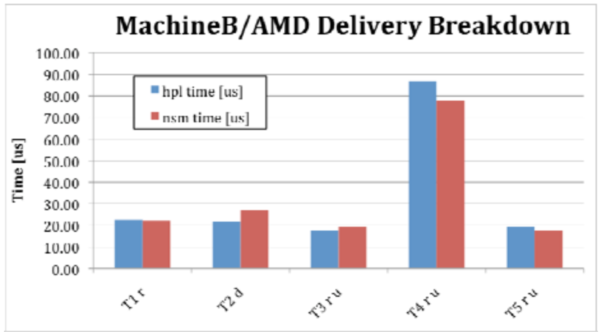

Figure 10. TPC-C "delivery" transaction (machine B).

In addition, we performed a detailed breakdown analysis of the "new order" and "payment" transactions (analogously to the TPC-B "account update" transaction) to investigate the performance effects of inserts, record construction and record-to-record navigation.

For brevity, we only discuss the "new order" breakdown; the "payment" breakdown yields similar findings.

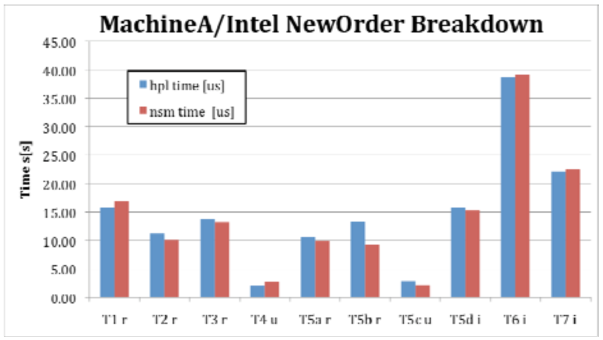

Figure 11: TPC-C NewOrder Transaction (machine A).

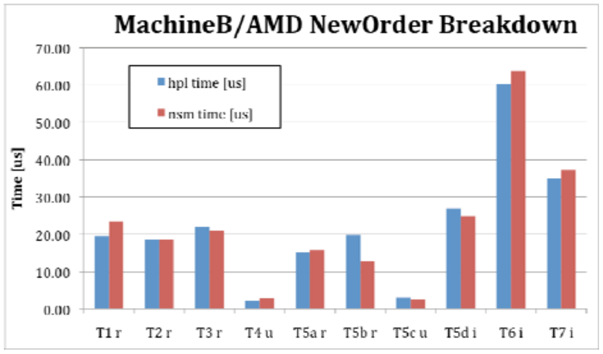

Figure 12. TPC-C NewOrder Transaction (machine B).

Field reads combined with single field scan operations, e.g. T1 (Figure 11, Figure 12) perform well on HPL. Read operations on HPL yielding record construction, e.g. T2, T3 or T5b are 5% to 10% slower than on NSM due to scattered reads. Since record construction also results from index-to-record transitions such operations also perform 5% to 10% slower on HPL compared to NSM. Single field update operations like T4 are faster on HPL, whereas multi-field updates are faster on NSM, due the lack of offset calculation and scattered accesses. In the general case, inserts on HPL are equally fast (T5d) or slower (T7) than on NSM.

In summary, read operations resorting to record construction are about 10% slower on HPL. Inserts on HPL are as fast as or slower than on NSM. Deletes are about 15% faster on HPL than on NSM.

6.4 *Summary: performance evaluation*

In summary, in our performance comparisons of HPL versus PAX and NSM using standard TPC benchmarks, we found that field scan operations are marginally slower on HPL than on PAX but much faster than on NSM due to better cache efficiency and optimized scanners. Record scan operations are about 11% slower on HPL compared to both PAX and NSM, due to the high cost for record construction due to scattered reads. Field update operations are about 5% faster on HPL compared to NSM. HPL and NSM perform equally well under write intensive loads. Inserts on HPL are marginally slower than on NSM. Deletes are about 15% faster on HPL compared to NSM.

In general, multi-field operations are slowest on HPL. HPL single-field operations exhibit performance that is equal to or marginally slower than that on NSM or PAX.

7 Conclusions

In summary, NSM is a strict record-at-a-time format, PAX is a strict field-at-a-time format, and the new hybrid page layout HPL is a combination aimed to capture the advantages of both. NSM is optimized for simple management of variable-size records and of free space, PAX is optimized for efficiency of large scans and their predicate evaluation, and HPL is optimized for both.

As in NSM, each HPL page contains a fixed-size page header and two variable-size areas that grow towards each other until the page is full. The difference to NSM is that the fixed-size area contains more than the record offsets. As in PAX, predicate evaluation against a single field incurs minimal cache faults in HPL because each cache line is filled entirely with a single field. The difference to PAX is that the cache lines holding a single field are not contiguous but are organized in repeating segments.

All three formats apply to records within traditional on-disk data pages and to small pages appropriate for flash storage within large pages appropriate for modern disk drives. NSM assigns entire records to small pages, PAX divides a large page into "mini-pages" each consisting of one or more small pages and then assigns fields to these "mini-pages," and the hybrid format assigns as many small pages to each field according to its size and then repeats this pattern. Thus, after a large page has been lifted from disk into flash storage, a large scan and its predicate evaluation require only a few of the small page in memory.

If not only query performance but also update performance, in particular insertion performance, are important, one might try to adopt an idea that seems very promising for column stores, namely a delta store in row format [SAB 05]. In other words, new insertions are retained in NSM format on the page and are not distributed over multiple cache lines as required in PAX and the hybrid format. Queries must inspect not only the full cache lines but also the records stored in the traditional way. A page compaction or reorganization will later optimize the page for large scan performance. The reorganization might be triggered by a query, by the passage of time, etc. The hybrid page format may be more suitable to this technique than the PAX format because of the single block of free space between the two areas for fixed-size and variable-size fields.

The performance measurements demonstrate that the new hybrid page layout fills the gap between NSM and PAX, recommending them to single-purpose systems whereas HPL is a compromise suitable to a wide variety of database systems and access patterns.

In conclusion, the hybrid format represents a blend and an improvement of both prior formats. All three formats are similar in terms of capacity, e.g., they fit similar record counts in each page. They differ in the complexity of space management and in their cache efficiency. The advantages apply both to records within a page and to data organization optimized for both flash storage and disk, and may be applicable to additional levels in a complex memory hierarchy.

Both PAX and the hybrid format could be generalized to fit multiple fields at-a-time, just as vertical partitioning is a generalization of a strict column-at-a-time storage format. In fact, it is often merely a matter of taste or convenience whether something is a single field or two separate fields. Consider, for example, telephone numbers: is the area code a separate field from the local number? Again, PAX and the hybrid format offer very similar facilities to model such choices as desired.

Additional research and design work is required to integrate PAX and the hybrid format with the various forms of data compression, e.g., prefix and suffix truncation [BU 77], duplicate value elimination [PP 03], run-length encoding (both for equal values such as “8, 8, 8, …” and for successive values such as “1, 2, 3, …”), order-preserving Huffman and Lempel-Ziv compression, etc., and to apply the resulting designs to multiple levels in a deep memory hierarchy including CPU caches, local and remote memory, NAND and NOR flash memory, “performance-optimized” “enterprise” disks and “capacity-optimized” “consumer” disks, mirrored and RAID storage, etc. Finally, we plan on investigating how formats, memory hierarchy, and query processing algorithm interact not only with respect to performance but also with respect to the robustness of performance.

Acknowledgements

Anastassia Ailamaki’s and Harumi Kuno’s comments on an early draft of this paper as well as their encouragement are highly appreciated. We thank Alex Buchmann for support and computing resources at TU Darmstadt.

References

[ADH 01] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, Marios Skounakis: Weaving relations for cache performance. VLDB 2001: 169-180.

[GL 01] Goetz Graefe, Per-Åke Larson: B-tree indexes and CPU caches. ICDE 2001: 349-358.

[HCL 90] Laura M. Haas, Walter Chang, Guy M. Lohman, John McPherson, Paul F. Wilms, George Lapis, Bruce G. Lindsay, Hamid Pirahesh, Michael J. Carey, Eugene J. Shekita: Starburst mid-flight: as the dust clears. IEEE TKDE 2(1): 143-160 (1990).

[HSY 01] Windsor W. Hsu, Alan Jay Smith, Honesty C. Young: I/O reference behavior of production database workloads and the TPC benchmarks – an analysis at the logical level. ACM TODS 26(1): 96-143 (2001).

[LD 93] Scott T. Leutenegger, Daniel M. Dias: A modeling study of the TPC-C benchmark. ACM SIGMOD 1993: 22-31.

[PP 03] Meikel Pöss, Dmitry Potapov: Data compression in Oracle. VLDB 2003: 937-947.

[SAB 05] Michael Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Samuel Madden, Elizabeth J. O'Neil, Patrick E. O'Neil, Alex Rasin, Nga Tran, Stanley B. Zdonik: C-store: a column-oriented DBMS. VLDB 2005: 553-564.