# Efficient columnar storage in B-trees

Goetz Graefe
Hewlett-Packard Laboratories
Palo Alto, CA

## Abstract

Column-oriented storage formats have been proposed for query processing in relational data warehouses, specifically for fast scans over non-indexed columns. This short note proposes a data compression method that reuses traditional on-disk B-tree structures with only minor changes yet achieves storage density and scan performance comparable to specialized columnar designs. The advantage of the proposed method over alternative storage structures is that traditional algorithms can be reused, e.g., for assembling rows with multiple columns, bulk insertion and deletion, logging and recovery, consistency checking, etc.

## 1 Introduction

Columnar storage has been proposed as a performance enhancement for large scans and therefore for relational data warehouses where ad-hoc queries and data mining might not find appropriate indexes. The basic idea for columnar storage is to store a relational table not in rows but in columns, such that scanning a single column can fully benefit from all the data bytes in a page fetched from disk or in a cache line fetched from memory.

The purpose here is to introduce a compression method that permits reuse of traditional storage structures with minimal change yet with storage efficiency comparable to specialized structures. Specifically, columns are stored separately and each value is tagged with the row to which it belongs, yet storage requirements for the tags are practically zero.

The goal is to replicate the performance effects of vertical partitioning. The mechanisms do not prescribe a specific policy, e.g., that all columns must be stored individually such that there are as many partitions as there are columns. Similarly, the mechanisms do not prescribe policies by type, e.g., that fixed-length columns or all columns of type “date” must be stored together. Finally, the mechanisms do not prescribe that all columns must be stored using the mechanisms proposed. Instead, the aim is to add another storage mechanism to the options available in physical database design, and to maximize storage efficiency to the extent possible. Policy decisions about usage of these mechanisms should be left to physical design for each specific database.

An alternative to vertical partitioning is to store each column in its own index. If there is no single clustered index that contains all columns for a particular table, it seems moot to distinguish between clustered and non-clustered indexes. In a way, it is multiple non-clustered indexes together that hold entire rows. Assembling entire rows, e.g., queries like “select * from …”, requires joining these indexes.

However, if each of these indexes is sorted by the column it contains, these join operation may be rather slow and expensive. In order to avoid this expense, indexes must be stored all in the same order. This order might be called the order of the rows in the table, since no one index determines it, and it is this order that the present proposal captures in tags with practically zero additional storage.

These tags are in many ways similar to row identifiers, but there is an important difference between these tags and traditional row identifiers: tag values are not physical but logical. In other words, they do not capture or represent a physical address such as a page identifier, and there is no way to calculate a page identifier from a tag value. If a calculation exists that maps tag values to row address and back, this calculation must assume maximal length of variable-length columns. Thus, storage space would be wasted in some or all of the vertical partitions, which would contradict the goal of columnar storage, namely very fast scans.

Moreover, the design aims to reuse code and functionality already available in most database systems as much as possible. Since most database management systems rely on B-trees for most of their indexes, reuse and adaptation of traditional storage structures means primarily adaptation of B-trees, including their space management and their reliance on search keys. In order to ensure that rows and their columns appear in the same sequence in all B-trees, the search key in all indexes must be the same. Moreover, in order to achieve the objectives, the storage requirement for search keys must be practically zero, which seems rather counter-intuitive.

The proposed technique affects the tag column only. The compression of actual column values is an orthogonal topic, and the validity and effectiveness of traditional compression techniques are not affected.

One particular usage patterns that is common in relational data warehouses is bulk insertion and bulk deletion, typically both in the order of time. The storage structures considered here accommodate those operations rather well, because data is not organized by their value but by a row tag or row identifier that is assigned during initial insertion of each row in the table or database.

## 2 Related work

The need for the proposed compression method was inspired by [SAB 05]. Inasmuch as columnar storage and columnar database management systems promise to be broadly viable technically and economically for relational database management systems supporting online analytical processing and data mining, database compression of columnar storage is critically important for performance and scalability. Moreover, code reuse and adaptation is more cost-efficient than invention of new storage structures followed by their design, implementation, testing, maintenance, etc.; therefore, our focus is on implementing columnar storage with, essentially, very traditional B-tree indexes.

The Decomposition Storage Model [CK 85], for example, vertically partitions a table and stores each column individually. Each resulting single-field record is augmented with a tag or surrogate that indicates the row within the logical table. Thus, a fair fraction of storage is occupied by tags, in particular if the table's columns are small. Elimination of this space overhead is the goal of the proposed compression technique.

The proposed compression technique is reminiscent of the simple compression proposed in [GRS 98], and in fact is similarly simple. However, their method was designed primarily for "measures" rather than "keys", to employ two terms common in online analytical processing. Measures are descriptive columns, e.g., size and weight, and often amenable to compression using arithmetic difference and variable-length integers [GRS 98]. In the specific application targeted here, the desired method needs to compress keys or identifier columns. Nonetheless, our method achieves compression comparable to that for constant columns in [GRS 98], i.e., practically zero bits per value.

Achieving practically zero storage overhead for row identifiers is quite similar to truncation of prefixes common among keys within a B-tree node [BU 77]. Without doubt very effective where it applies, prefix truncation also aids search performance, in terms of both instruction path and cache faults [L 01]. Thus, our method strives to achieve the same effect as prefix truncation, although the key column in our application is not constant but varies from record to record.

Columnar storage has been proposed for both on-disk data structures and for in-page data organization, the latter to improve the access patterns and performance in CPU caches [ADH 99]. In this note, let us ignore the in-page data organization and focus instead on B-trees in general, aware of the general concepts of applying traditional B-tree structure not only to disk pages but also to cache lines [HP 03, L 01, RR 00] and to contiguous extents on disk [O 92].

## 3 Tag column compression

The essence of our technique is quite simple. Rows are assigned tag values in the order in which they are added to the table. Note that tag values identify rows in a table, not records in an individual partition or in an individual index. Each tag value appears precisely once in each index. All vertical partitions are stored in B-tree format with the tag value as the leading key. The important novel aspect is how storage of this leading key is reduced to practically zero.

The essence of our technique is that in each B-tree page, the page header stores the lowest tag value among all B-tree entries on that page, and the actual tag value for each individual B-tree entry is calculated by adding this value and the slot number of the entry within the page. There is no need to store the tag value in the individual B-tree entries; only a single tag value is required per page. If a page contains tens, hundreds, or even thousands of B-tree entries, the overhead for storing the minimal tag value is practically zero for each individual record. If the size of the row identifier is 4 or 8 bytes and the size of a B-tree node is 8 KB, the per-page row identifier imposes an overhead of 0.1% or less.

If all the records in a page have consecutive tag values, this method not only solves the storage problem but also reduces "search" for a particular key value in the index to a little bit of arithmetic followed by a direct access to the desired B-tree entry. Thus, the access performance in leaf pages of these B-trees can be even better than that achieved with interpolation search or in hash indexes.

If records in a page do not have consecutive tag values, the proposed method does not work, at least not immediately. There are multiple ways to design for possible gaps in the sequence of tags. One way is to prohibit and avoid gaps, e.g., by means of a strict requirement that rows in the table are only appended at the end or they are deleted only in the order in which they were added. Gaps in the tag sequence within one index usually imply that the rows in the table lack consecutive tag values and that the same problem exists in all B-tree representing vertical partitions. Alternatively, gaps may occur due to missing values or *Null* values.

A second way for dealing with gaps in the sequence of tags is to retain ghost records in the B-tree pages. During deletion of a single row in the table and the corresponding records in all the table's indexes, the B-tree entries are only pseudo-deleted, i.e., marked as "ghosts" [JS 89] yet retained in the storage structure. This is already a standard way of ensuring successful transaction rollback without possibility of failure due to problems in space allocation, and queries already employ an implicit predicate to ignore ghost records. The new requirement due to the proposed compression scheme is that ghost clean-up does not erase ghost records but instead cuts their size to retain merely the B-tree key with no additional columns. Note that shortening ghost records to their key is very space efficient due to our compression scheme, because these key values are not stored for each individual B-tree entry. Thus, only an entry in the page's indirection vector is wasted, but no space in the page's area for data bytes.

The considerations so far have covered only the B-tree's leaf pages. Of course, the upper index pages also need to be considered. Fortunately, they introduce only moderate additional storage needs. Storage needs in interior nodes is determined by the key size, the pointer size, and any overhead for variable-length entries. In this case, the key size is equal to that of row identifiers, typically 4 or 8 bytes. The pointer size is equal to a page identifier, also typically 4 or 8 bytes. The overhead for managing variable-length entries, although not strictly needed for the B-tree indexes under consideration, is typically 4 bytes for a byte offset and a length indicator. Thus, the storage needs for each separator entry is 8 to 20 bytes. If the node size is, for example, 8 KB, and average utilization is 70%, the average B-tree fan-out is 280 to 700. Thus, all upper B-tree pages together require disk space less than or equal to 0.3% of the disk space for all the leaf pages, which is a negligible in practice.

Compared to other schemes for storing vertical partitions, the proposed method permits very efficient storage of variable-length values in the same order across multiple partitions. Thus, assembly of an entire table is very efficient using a multi-way merge join. In addition, assembly of an individual row is also quite efficient, because each partition is indexed on the row identifier. All traditional optimizations of B-tree indexing apply, e.g., very large B-tree nodes and interpolation search. Note that interpolation search among a uniform data distribution in practically instant.

The ability to store variable-length column values very densely is another important characteristic of our design, because it guarantees maximal scan performance. Like any B-tree structure, space management is "built-in" with guaranteed minimal space utilization [BM 70, BM 72], and efficient mechanisms for reorganization and defragmentation are well known and implemented in commercial products. Thus, there is no cost or complexity for new storage structures. Without doubt, this can be a decisive argument in a commercial environment where all aspects of on-disk storage must be considered, including concurrency control and recovery, bulk insertions and deletions, online index creation, index creation with allocation-only logging, verification to guard against corruption due to hardware or software faults, etc.

## 4 Additional applications

In the discussions above, a single table was partitioned vertically and tags assigned per table. Alternatively, a table may be partitioned in multiple steps. The first step groups columns into subsets and sort order defined for each subset. Tags are assigned based on this sort order. The second step partitions each subset into storage structures, e.g., B-trees on tag columns with the compression feature described earlier. Thus, in this storage architecture, the earlier discussions apply not to a traditional logical table or view but to each vertical partition of such a table or view.

In more traditional database settings, there are some real-world business processes in which sequential numbers or identifiers are common, important, or even legally required. For example, orders, invoices, cheques, etc. fall into this category. For databases that describe these real-world objects, indexes that map real-world identifiers to additional information can benefit from the compression method described. Even if there are large gaps in the overall sequence, e.g., in vehicle identification numbers, data in many index pages will be short coherent sequences that may benefit. For small gaps, ghost slots as described above might be sufficient.

In other words to maximize effective scan bandwidth, column stores should be 100% full. Thus, changes should be initially retained elsewhere using techniques like differential files [SL 76] and then applied in bulk. For efficient capture, e.g., during bulk loading, the changes should likely be in row format rather than column format [SAB 05].

A recent study of master-detail clustering within B-tree indexes [G 07] found that the proposed compression method applies even there. In other words, multiple columns can be stored in a single B-tree in a column-oriented format rather than the traditional row-oriented form. Each column is assigned to key range within the B-tree such that the individual columns are concatenated. Each record's key consists of column identifier and row identifier. After the column identifier has been truncated using prefix truncation [BU 77], the row identifier can be truncated using the presented design. Even recent insertions and deletions in row format can be represented in the same B-tree in yet another key range.

Finally, independent of the scan performance in relational data warehousing environments, vertical partitioning and columnar storage using B-trees as described automatically turns row-level locking into column-level locking. Compared to prior designs for adapting row-level locking to column-level locking [P 01], columnar storage in B-tree indexes requires fewer software modifications.

## 5 Summary

In summary, a very simple method exists that permits storing vertical partitions in traditional B-tree indexes with practically zero overhead for storage. Thus, code reuse is maximized and cost of code development and maintenance are minimized. Storage format and record ordering permit very efficient assembly of many rows using merge join and of individual rows using index nested loops join. In conclusion, it seems that the described storage format is very promising for commercial implementations of columnar storage for relational tables and views.

## References

[ADH 99] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, David A. Wood: DBMSs on a Modern Processor: Where Does Time Go? VLDB 1999: 266-277.

[BM 70] Rudolf Bayer, Edward M. McCreight: Organization and Maintenance of Large Ordered Indexes. SIGFIDET Workshop 1970: 107-141.

[BM 72] Rudolf Bayer, Edward M. McCreight: Organization and Maintenance of Large Ordered Indices. Acta Inf. 1: 173-189 (1972).

[BU 77] Rudolf Bayer, Karl Unterauer: Prefix B-Trees. ACM TODS 2(1): 11-26 (1977).

[CK 85] George P. Copeland, Setrag Khoshafian: A Decomposition Storage Model. SIGMOD 1985: 268-279.

[G 07] Goetz Graefe. Master-detail clustering using merged indexes. To appear in Informatik Forschung und Entwicklung.

[GRS 98] Jonathan Goldstein, Raghu Ramakrishnan, Uri Shaft: Compressing Relations and Indexes. IEEE ICDE 1998: 370-379.

[HP 03] Richard A. Hankins, Jignesh M. Patel: Effect of Node Size on the Performance of cache-conscious B+-trees. SIGMETRICS 2003: 283-294.

[JS 89] Theodore Johnson, Dennis Shasha: Utilization of B-trees with Inserts, Deletes and Modifies. PODS 1989: 235-246.

[L 01] David B. Lomet: The Evolution of Effective B-tree: Page Organization and Techniques: A Personal Account. SIGMOD Record 30(3): 64-69 (2001).

[O 92] Patrick E. O'Neil: The SB-Tree: An Index-Sequential Structure for High-Performance Sequential Access. Acta Inf. 29(3): 241-265 (1992).

[P 01] Nagavamsi Ponnekanti: Pseudo Column Level Locking. ICDE 2001: 545-550.

[RR 00] Jun Rao, Kenneth A. Ross: Making B+-Trees Cache Conscious in Main Memory. SIGMOD 2000: 475-486.

[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.

[SL 76] Dennis G. Severance, Guy M. Lohman: Differential Files: Their Application to the Maintenance of Large Databases. ACM TODS 1(3): 256-267 (1976).