Page As You Go: Piecewise Columnar Access In SAP HANA

Reza Sherkat, Colin Florendo, Mihnea Andrei, Anil K. Goel, Anisoara Nica, Peter Bumbulis
Ivan Schreter, Günter Radestock, Christian Bensberg, Daniel Booss, Heiko Gerwens
SAP SE
<*firstname.lastname*>@sap.com

ABSTRACT

In-memory columnar databases such as SAP HANA achieve extreme performance by means of vector processing over logical units of main memory resident columns. The core in-memory algorithms can be challenged when the working set of an application does not fit into main memory. To deal with memory pressure, most in-memory columnar databases evict candidate columns (or tables) using a set of heuristics gleaned from recent workload. As an alternative approach, we propose to reduce the unit of load and eviction from column to a contiguous portion of the in-memory columnar representation, which we call a *page*. In this paper, we adapt the core algorithms to be able to operate with partially loaded columns while preserving the performance benefits of vector processing. Our approach has two key advantages. First, partial column loading reduces the mandatory memory footprint for each column, making more memory available for other purposes. Second, partial eviction extends the in-memory lifetime of partially loaded columns. We present a new in-memory columnar implementation for our approach, that we term *page loadable column*. We design a new persistency layout and access algorithms for the encoded data vector of the column, the order-preserving dictionary, and the inverted index. We compare the performance attributes of page loadable columns with those of regular in-memory columns and present a use-case for page loadable columns for cold data in *data aging* scenarios. Page loadable columns are completely integrated in SAP HANA, and we present extensive experimental results that quantify the performance overhead and the resource consumption when these columns are deployed.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Query processing*

Keywords

In-Memory columnar databases; Data aging

SIGMOD'16, June 26-July 01, 2016, San Francisco, CA, USA

DOI: http://dx.doi.org/10.1145/2882903.2903729

1. INTRODUCTION

Modern in-memory database systems achieve extreme performance for analytical workloads by taking advantage of recent trends in hardware technology, including affordable large dynamic random-access memory (DRAM) and the instruction level parallelism of register vector processing, e.g. Advanced Vector Extensions (AVX) processor capabilities [7, 23, 27]. For in-memory columnar databases, dictionary based compression schemes produce uniform representation and in-memory layout for each column. The memory layout can be optimized for extremely fast basic database operations, such as scan and search [8, 17, 26].

With the rapid growth in the variety of applications that want to combine business data with the emerging Internet of Things (IoT) and/or social media data, there is an important need to also provide very high performance for very large datasets while optimizing the use of precious main memory for low value or low density data. To reduce overall systems memory pressure and to achieve optimal total cost of ownership (TCO), an application may intelligently analyze its workload, and keep only an active subset of data in memory at a given time. As Plattner indicated in [21], SAP HANA optimally serves business applications by being aware of the lifecycle of business data. As data ages, it is rarely accessed and can be moved in the cold store. Unlike the hot store containing the current data, which offers extreme performance, the cold store has a performance−TCO tradeoff biased towards TCO reduction. Another example is an Enterprise Resource Planning (ERP) system which may have two extreme data subsets. On one side lie the frequently accessed columns needed to generate high value reports for decision making. On the other extreme are rarely accessed columns needed for infrequent or low value queries and activities.

In order to achieve optimal memory utilization, an adaptive workload-aware strategy may keep an active subset of columns in memory, and may decide to evict some columns under critical memory pressure. However, loading and unloading columns completely in this way poses two important performance challenges. First, each on-demand access to a column needs a fully memory resident column, even if the access needs to read only a portion of the column (e.g. a single row). Second, the loading of infrequently accessed columns can trigger the eviction of another column, which may be needed again soon. The common root cause of these challenges is the *granularity* of column loading and eviction. Ideally, the memory overhead of each column must be correlated with the portion of the column which is accessed during a workload while limiting the resulting loss of performance

as compared to highly optimized in-memory access methods. Therefore, for infrequently accessed columns, we advocate for piecewise columnar presence in memory. To support this, we implemented a new type of column in SAP HANA which we termed *page loadable column*.

A page loadable column is logically treated as a regular column, with one key physical difference. A regular column fully resides in memory during access while one or more fractions of a page loadable column may be evicted from (or never be loaded into) memory. This physical difference is, however, invisible to applications and is reminiscent of pagination in classic database buffer pool implementations. In SAP HANA, each column is associated with three data structures, namely an encoded data vector, a dictionary, and an optional inverted index. We carefully designed page loadable columns in a way that every data structure pertaining to a column becomes page loadable. This design provides for some unique advantages for page loadable columns. For instance, instead of the complete dictionary, only sections of it may be loaded during late materialization [11] of the result of a query on a page loadable column. As another advantage, only a section of a paged inverted index may get loaded corresponding to the search criteria of a scan on a page loadable column. In summary, page loadable columns extend the properties of in-memory column-store databases [3, 25] with support from classic paged buffer pool concepts. The preferred loading behavior of a column is specified at creation time as being loaded either fully or piecewise. Once created, page loadable columns can be manipulated by the same SQL statements as regular columns in SAP HANA. The access layer implements algorithms for piecewise loading of column data structures. The resource manager evicts portions of data structures of a page loadable column.

Our main contributions are:

- We study fine granular columnar accesses for a modern in-memory database system. Our solution provides adaptive flexibility for an extreme performant in-memory column store database with selective and judicious techniques from traditional page-based database system without losing the core advantages of modern in-memory vector processing for scans.
- We propose physical layouts and on-demand access algorithms for data structures of a page loadable column, i.e. its data vector, its order preserving dictionary, and its inverted index for a hybrid implementation.
- We present an overview of our memory management component, which balances system memory allocation by reactive and proactive piecewise column eviction mechanisms.
- We present a use case for page loadable columns in a data aging scenario, and demonstrate the benefits of our solution.
- We present an experimental evaluation of page loadable columns in a large database, and report on runtime overhead and memory footprint reduction.

Roadmap: Section 2 is the background. Section 3 presents page loadable data structures. Section 4 demonstrates data aging as a use case of page loadable columns. Section 5 presents an overview of our memory manager. Section 6 reports our experimental study. Related work is surveyed in Section 7 followed by future research directions in Section 8. Finally, we conclude in Section 9.

2. COLUMN STORE IN SAP HANA

We present an overview of SAP HANA's column store. The architecture of SAP HANA is described in [7, 23]. Each columnar table is composed of a set of columns. Every column has a read optimized section, referred to as its *main* fragment and a write optimized section known as its *delta* fragment. Inserting new rows or updating existing rows are both regarded as changes. Changes do not physically modify existing rows (i.e. no in-place update) but append new rows into delta fragments. During the delta merge operation, all committed row changes from the delta fragment are moved into a newly constructed main fragment. Each query on a column is evaluated independently on the main and on the delta fragment of the column. The two result sets are joined and returned, with some rows removed after applying proper row visibility rules.

As for any in-memory database, both the main and the delta fragments of each column are kept in memory during query processing. To achieve a compact in-memory representation which allows for efficient CPU access patterns when processing a column, each data fragment of a column is encoded using dictionary encoding which assigns a unique value identifier to each value. The value identifier vector for each column fragment is termed its *data vector*. The value identifiers in each data vector are assigned using the value dictionary of the corresponding fragment. Each fragment may also have a memory resident inverted index, to support efficiently finding of all row positions given a value identifier. Every loaded fragment has its data vector, its dictionary, and its inverted index fully loaded in memory. The memory consumption of each column is the total sum of the memory footprints of its two fragments. Because delta merge regularly moves most of the data in a column from the delta fragment to the main fragment, in general, the main fragment of a column consumes more memory than the corresponding delta fragment.

The construction phase of value dictionaries for delta and main are inherently different. The former is built while inserts arrive and identifiers are being appended at the end of the data vector of the delta. To be optimized for write operations, it would be costly to keep the dictionary for delta fragment in a way that the order between values remain consistent with the order between value identifiers. The main dictionary, however, is created during delta merge. During delta merge, the set of possible values for the new main fragment is known. Therefore, it is possible to assign value identifiers in the same sorted order as that of keys in main dictionary. The data vector, the order−preserving dictionary, and the inverted index for main fragment remain unchanged between two consecutive delta merge operations.

3. PAGE LOADABLE COLUMNS

We present our solutions to partially load data vector (Section 3.1), dictionary (Section 3.2), and inverted index (Section 3.3) for the main fragment of a column. Table 1 summarizes our notations.

3.1 Paged Data Vector

In SAP HANA, the data portion of main fragments are stored in memory as value identifier vectors. For large columns, the memory footprint of these vectors can be significant even after compression with algorithms such as sparse encoding [15]. In this section, we introduce partially loadable

Table 1: List of Notations

Notations	Description
$pDataVector$	paged data vector containing, for each row, its encoded value identifier
$pDataVector[p, rpos]$	$O(1)$ operation on $pDataVector[p, \_]$ to retrieve the vid for a given row position stored on page p
$ipDataVector$	paged inverted index on data vector
$ipDataVector[vid]$	find in $ipDataVector$ all row positions having vid as value identifier
$pDict$	the paged dictionary of a column
$pDict[p, vid]$	$O(1)$ operation on $pDict[p, \_]$ to retrieve the value encoded with vid from page p
$ipDict_{ValueId}$	the sparse helper index on $pDict$ with one entry (vid, p) per page p of $pDict$: vid is the last id stored on page p
$ipDict_{ValueId}[vid]$	binary search operation on $ipDict_{ValueId}$ to find the page p where vid is stored
$ipDict_{Value}$	the sparse helper index on $pDict$ with one entry $(value, p)$ per page p of $pDict$: $value$ is the last stored on page p
$ipDict_{Value}[value]$	binary search operation on $ipDict_{Value}$ to find the page p where vid encoding $value$ is stored

value identifier vectors. We first present the physical layout for a paged data vector (Section 3.1.1) and algorithms that use a stateful iterator to read from a paged data vector (Section 3.1.2) followed by performance enhancements (Section 3.1.3).

3.1.1 Physical layout

The value identifiers in a paged data vector $pDataVector$ are encoded using uniform n-bit compression where n is equal to the number of bits needed to encode the largest value identifier in the entire vector. Paged vectors are split into chunks of exactly 64 identifiers and stored as a chain of disk resident pages. Each page contains an integral number of chunks. We denote the vector of value identifiers stored on page p with $pDataVector$ $[p, \_]$. Storing 64 identifiers per chunk ensures that each chunk consists of an integral number of bytes[1] independent of n. Our choice of uniform compression supports efficient mapping from row position to logical page number in $pDataVector$'s page chain (Section 3.1.2). Also, it enables vectorized implementations of the primitive iterator operators (Section 3.1.3).

3.1.2 Algorithms

One type of logical access operation on paged data vector is to find all row positions having a given value identifier vid. The general algorithm (Alg. 1) loads one page at a time, decodes each value identifier on the page, and searches each vector $pDataVector[p, \_]$ for specified vid[2]. To implement the basic operations of decoding a value identifier for a given row position and searching in a range of row positions given a set of value identifiers, we use an iterator based access to paged data vector, in order to perform paging as part of the iterator's functionality. The iterator implements two decode methods and a set of search methods: $get(rpos)$ decodes the value identifier in $pDataVector$ at the given row position $rpos$, and $mget(rpos_{from}, rpos_{to})$ decodes the set of value identifiers that appear in the given row position range.

There are four varieties of the method *search(set of row positions, set of value identifiers*) which takes a set of row positions (e.g., a range, a bitmap) and a search predicate specified as a set of value identifiers, and returns a set of row positions for which the predicate is true. For all iterator methods, the iterator is provided with a range of row positions which is used to selectively load pages from $pDataVector$'s page chain. Finding the right set of pages to load can be achieved because of the uniform value identifier encoding. More specifically, the iterator can identify the chunk(s) that contain the value identifiers in the specified row position range. In order to read the chunks, the iterator must load $pDataVector$'s pages that contain the desired chunks. The iterator can use the constant chunk size (in bytes) and the page size, to find the logical page number of pages where the desired chunks belong to. The iterator loads the pages associated with the logical page number(s) (e.g., a page p) and executes the desired functionality (e.g., $mget(rpos)$) on page level (e.g. on $pDataVector$ $[p, \_]$). The iterator may need more than one chunk from the same page, in which case it pins the page in memory to make sure the page does not get evicted by the resource manager when it is being read. On the other hand, the iterator may need to read more than one page. In this case, it pins each new page after releasing the handle to the previous page during page reposition. Our design choice of *preventing each value identifier from spanning more than one chunk* guarantees that the iterator has stable access to each chunk, and that the page is never evicted by the resource manager while being accessed by an iterator. Using uniform n-bit packing simplifies associating row position to corresponding encoded value identifier.

Algorithm 1 $findByValueID(pDataVector, vid)$

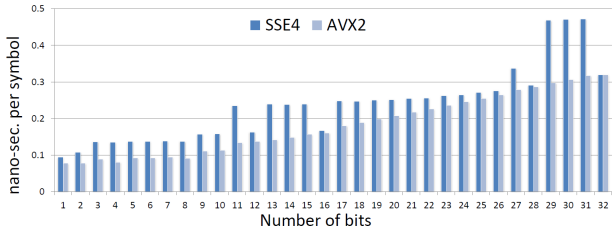
```
1: rposs ← ∅
2: for all p a page of pDataVector do
3:     load page p of pDataVector if not loaded
4:     for all row position rpos in page p do
5:         if pDataVector[p, rpos] == vid then
6:             rposs ← rposs ∪ {rpos}
7: return rposs
```

3.1.3 Optimizations

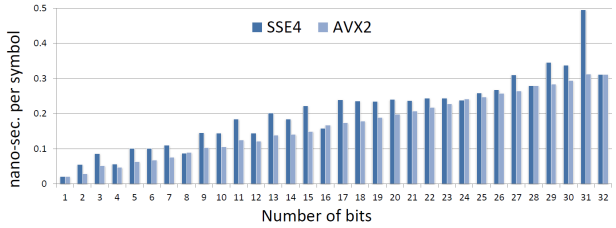
Having chunks of uniformly encoded value identifiers enables us to vectorize most of the primitive operators used to implement the paged data vector's iterator [26]. We take advantage of the platform provided by Single Instruction Multiple Data (SIMD) instruction set (AltiVec on IBM Power, SSE4 and AVX2 on Intel 64) and attempt to minimize both CPU cache misses and branch misprediction. Fig. 1 shows micro benchmark results (average time per symbol) for the *mget* and *search* operators on n-bit encoded data vectors. In particular, on current Intel server processors, the search primitive is bottlenecked by the memory read bandwidth available to a single core; improving search performance is not possible without reducing the amount of data scanned. For example, performance could be improved for some queries by using a representation like vertical bitweaving with early pruning [17] but only at the expense of noticeably worse performance for the remaining queries. Optimizations higher up in the iterator call stack include cache awareness: both ensuring that intermediate results do not exceed the Last Level Cache (LLC) size and processing data in cache line sized units.

[1] More precisely, 64-bit words.

[2] An optimization of Alg. 1 may prune loading and examining pages using column-level metadata information.



(a) *mget*



(b) *search*

Figure 1: Average time (nano-second) per symbol: varying number of bits n used for n-bit compression. Micro benchmarks conducted on a server with Intel(R) Xeon(R) CPU E5-2697 v3 @ 2.60GHz

3.2 Paged Dictionary

Even with paging less frequently accessed data vectors, the memory footprint can be large due to tables with a large number of columns with memory resident dictionaries, specifically for columns with large strings. Often an active subset of a dictionary needs to be accessed for page loadable columns. The subset could be expressed explicitly in the query predicate[3]. Alternatively, dictionary access might be needed to retrieve the values, given a set of value identifiers for the result set of a query. To reduce the memory footprint of dictionaries in page loadable columns, we propose a hybrid representation for a dictionary (Section 3.2.1) which has an in memory data structure, always loaded, and a physical page loadable representation. Our goal is to reduce the memory footprint with minimum overhead to the read performance. As for the paged data vector, we apply our solution to the main fragment of page loadable columns (i.e. the read only section of a column) as opposed to the delta fragment. This is mainly because performing regular delta to main merge guarantees that the memory footprint of the main dominates that for the delta section of a column. We hide the details of loading and unloading dictionary pages via paged dictionary iterators as described in Section 3.2.2. Iterators take advantage of transient data structures and physical layout of dictionary pages.

3.2.1 Physical layout and transient structure

This section presents the physical and the transient data structures for paged dictionary, denoted by *pDict*. We focus on data types for which the memory footprint for dictionary is noticeable (CHAR and VARCHAR), as opposed to basic data types such as INTEGER and FLOAT. Therefore, in this section, we use the term *string* to refer to a value encoded in a paged dictionary. The dictionary for the main fragment is

[3]E.g., in SELECT * FROM T WHERE Name='Mike', we first find the value identifier *vid* for the value 'Mike' by consulting the dictionary of column Name; then the data vector is used to find all row positions for *vid*.

prefix length	block length	string suffix (the new piece in this block)			*on-page* ← *section*
# of logical pointers	logical pointer 1	...	logical pointer n	total length	*off-page* ← *section*

Figure 2: Prefix-encoded string

created during the delta merge. The strings in main dictionary are sorted and its value identifiers are assigned using the same order of strings. Often consecutive strings in main dictionary have a prefix in common. We take advantage of this redundancy and use prefix encoding to reduce space for paged dictionary page. We group every 16 consecutive strings in a single string value block. We use the format shown in Fig. 2 to store each string in a value block. First, we store the length of the common prefix with the preceding string in the same value block, or zero for the first string in a value block. Then we store the block length, followed by the string suffix (i.e. the string minus the common prefix). For large strings, the size of each string can exceed the size of a dictionary page. Therefore, we store each string in two sections; an *on-page* piece (stored literally in the block) and an *off-page* piece. The off-page section contains references to sections of the large string, each stored on a separate dictionary page. We keep a set of logical pointers per string, where each logical pointer refers to a piece of the string, in a different dictionary page. This enables re-constructing large strings. Meanwhile it reduces the memory footprint of the dictionary pages with large strings, for instance when the page is loaded to materialize only one large string during single dictionary lookup. We store value blocks (along with their offsets on page) on each dictionary page.

A paged dictionary is stored as a chain of dictionary pages, with two helper dictionaries to support search. The first helper dictionary $ipDict_{ValueId}$ is the value identifier directory, a sparse index having one entry of the form (vid, p) for each dictionary page, where *vid* is the *last* value identifier stored on the page p of *pDict*. This index is organized into a set of directory blocks. Each directory block stores a list of value identifiers. Each identifier in a directory block corresponds to the identifier of the *last* string of a dictionary page. The second helper dictionary $ipDict_{Value}$ is the value separator directory, indexing entries of the form $(value, p)$, one entry per page p of the dictionary *pDict*, where *value* is the *last* value stored on page p. This index contains a set of separator blocks, each such block stores the last string in each dictionary page. If the string has an off-page component, we store the logical pointers to the off-page components in a separator block for large strings.

Besides the physical structures of *pDict*, we construct a transient data when a dictionary page gets loaded. The transient data contains a vector of pointers to string value blocks stored on the *in-memory* representation of the page. The transient data is registered to the corresponding dictionary page during load, and is unloaded (i.e. destroyed) when either the column is unloaded or the dictionary page is unloaded by the resource manager (see Section 5).

3.2.2 Algorithms

We implemented an iterator based access to paged dictionary, to hide the paging of the dictionary from the consumers of the dictionary functionality. The paged dictionary iterator implements two methods, $findByValue$ and

Algorithm 2 $findByValue(pDict, ipDict_{Value}, value)$

1: $p \leftarrow binary\_search(ipDict_{Value}, value)$
2: load page p of $pDict$ if not loaded
3: $vid \leftarrow binary\_search(pDict[p, \_], value)$
4: return vid

Algorithm 3 $findByValueID(pDict, ipDict_{ValueId}, vid)$

1: $p \leftarrow binary\_search(ipDict_{ValueId}, vid)$
2: load page p of $pDict$ if not loaded
3: $value \leftarrow pDict[p, vid]$
4: return $value$

$findByValueID$[4]. The former method finds the value identifier encoding a given value, whereas the latter method finds the value corresponding to the given value identifier.

For a fully loaded dictionary, $findByValue$ can perform a binary search as strings in main dictionary are always sorted and available in memory (even the large strings). For a paged dictionary, $findByValue$ (Alg. 2) first needs to load the right dictionary page. For this purpose, the separator helper dictionary (i.e. $ipDict_{Value}$) is used to find the logical page number of dictionary page, given the search string value. Because the strings in the separator dictionary are sorted, the probe is done by performing a binary search. Once it is determined that the query string maybe in the dictionary, the logical page number p is retrieved from the separator dictionary. If page p is not in memory, it is loaded and its transient data is constructed for the iterator. A second binary search is performed on the transient data, to identify the value block which may contain the search string. For small strings, only the on-page content is compared against the search string. For large strings, however, the off-page contents are compared against the search string by incrementally loading each off-page piece. $findByValue$ terminates in a value block when either a match with query string is found or there is no match.

Similarly, $findByValueID$ (Alg. 3) takes a two phase approach. It first uses the directory helper dictionary to find the logical dictionary page given a *vid*. Because the value identifiers are assigned the same order as dictionary strings, the logical page number on directory can be found using binary search. It loads the dictionary page into memory, and from the block header of the loaded dictionary page and the value identifier, it extracts the offset for the the value block corresponding to the query value identifier. The sought after string is constructed incrementally by scanning the value block, and loading the off-page sections of at most one string.

Apart from the helper dictionaries, the memory footprint of paged dictionary for the invocation of $findByValue$ and $findByValueID$ methods is one dictionary page (which contains the target value block) and, for large strings, a set of dictionary pages containing the off-page contents. We typically choose the pages in dictionary page chain to be 1MB, except the last dictionary page, which is large enough to hold the value blocks and offsets in the last dictionary page. In practice, the number of pages in helper dictionaries is small, and the memory footprint of the helper dictionaries is negligible.

3.2.3 Optimizations

In the lifetime of a paged dictionary iterator, we may need to load the same page multiple times, for different string values and value identifiers. This, for instance, can happen during a batch lookup for a set of ordered strings or value identifiers. In this scenario, it makes sense to keep pages loaded during the lifetime of the iterator, and prevent the resource manager to unload the pages, until the iterator gets out of scope. For this purpose, we create a handle cache for each iterator as follows. For each logical page number that we are requested to load after probing helper dictionaries, we update the handle cache if the handle for the corresponding logical page is invalid (i.e. not set yet), then load the page spanning the dictionary block and update the handle. Otherwise, we reuse the page handle. The iterator cache is destroyed and its handles are released when the iterator gets out of scope. Keeping handles to dictionary pages prevents the resource manager from unloading these pages during the lifetime of the iterator. Each of the helper dictionaries has a single value per dictionary page. Because a search on helper dictionaries is the first step in both $findByValue$ and $findByValueID$, we pre-load the page chains of the helper dictionaries on the first access to a paged dictionary. This improves the performance of probing helper dictionaries to find logical page numbers for paged dictionaries.

3.3 Paged Inverted Index

Retrieving all rows which satisfy a search predicate is a common query task performed on a column. This involves dictionary lookups to find the encoding value identifiers for the predicate, and then find all the row positions in the data vector (Alg. 4). Unfortunately, a full scan of paged data vector to find row positions that fulfill search predicate would trigger the load of *all* data vector pages, even those that do not have any eligible row. One may keep a summary of each data vector page in a transient data structure. The summary can be used to filter some $pDataVector$ pages without loading them. An example summary may keep the minimum and the maximum of the encoded values per page [2]. The summary can be used to determine whether a page contains value identifiers within a range without actually loading the $pDataVector$ page. However, if the search value identifier range overlaps with the summary, page load is required to prune false positives. This approach is effective when the range per summary is compact.

To alleviate the large memory overhead and CPU cost, we use a secondary *paged* data structure (its inverted index) which is built from $pDataVector$. A classic inverted index provides a mapping from each *vid* to the set of row positions observing the *vid*, we call this set the *postinglist* for *vid*). We denote such index by $ipDataVector$. The inverted index can efficiently answer *which row positions of the data vector have a given vid*. However, the size of the inverted index is linearly proportional to the size of the corresponding data vector. The size factor makes a fully loaded inverted index a prohibitive solution for in-memory database systems. We optimize the physical layout (Section 3.3.1) and iterator based access algorithms (Section 3.3.2) to support on-demand load of fractions of inverted index with a memory footprint proportional to query result size.

3.3.1 Physical layout

An inverted index of a dictionary encoded data vector is composed of two data structures; a directory of offsets (directory) and a vector of row positions (postinglist). The

[4]For brevity, we only discuss primitive search types, and do not discuss high level search operations, e.g. LIKE.

Algorithm 4 $findByValue(pDataVector, ipDataVector, pDict, ipDict_{Value}, value)$

1: $vid \leftarrow findByValue(pDict, ipDict_{Value}, value)$
2: $rposs \leftarrow findByValueID(pDataVector, ipDataVector, vid)$
3: return $rposs$

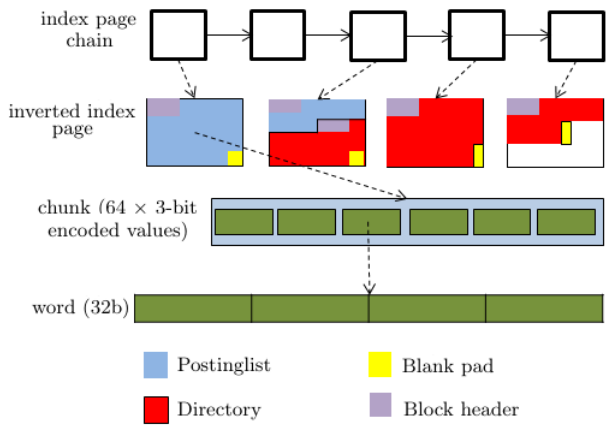


Figure 3: Layout of paged inverted index

postinglist is a rearrangement of the data vector such that row positions are ordered based on the value identifier they contain. This property ensures that row positions with the same value identifiers are grouped together in the posting list. The directory provides the offset of the first occurrence of a value identifier in the postinglist.

For in-memory columns, both directory and postinglist are kept in memory as compressed vectors. For page loadable columns, we store postinglist and directory in *one* chain of *index* pages (Fig. 3). The chain contains a set of index pages for postinglist, at most one mixed page (described shortly), and a set of index pages to persist the directory vector. Each index page is composed of one or more index blocks. Each index block can represent a section of either a postinglist or a directory. Each index block is composed of an index block header, followed by a set of one or more index block chunks. An index block chunk contains 64 n-bit encoded values. In a chunk of postinglist index block, each value is a row position. In a chunk of a directory index block, each value is an offset to the postinglist. Naturally, the number of bits required to encode a row position can be different from that of an offset in directory block. The mixed index page contains a postinglist index block, followed by a directory index block. For unique columns, each value appears in a single row position, meaning that the directory vector becomes an identity vector. Because of this, for unique indexes we save the space for storing directory.

3.3.2 Algorithms

We implemented an iterator based access to paged inverted index, to hide the paging details for postinglist and directory from the consumers of iterator's functionality. This iterator is used, for example, for the logical operation $rposs \leftarrow ipDataVector[vid]$ in Alg. 5. The paged index iterator implements $getFirstRowPos$(vid) and $getNextRowPos$() methods. These methods are in charge of loading and pinning the right pages from the index page chain, and returning the correct row position, given a value identifier which

Algorithm 5 $findByValueID(pDataVector, ipDataVector, vid)$

1: **if** $ipDataVector$ exists **then**
2: $rposs \leftarrow ipDataVector[vid]$
3: **else**
4: /* Sequential scan of $pDataVector$ data vector */
5: $rposs \leftarrow findByValueID(pDataVector, vid)$
6: return $rposs$

is in the main dictionary. $getFirstRowPos$ finds the index page containing the directory section corresponding to vid. This page is loaded, if it is not already in memory. Then, $getFirstRowPos$ reads the first offset of vid's first row position in the postinglist. Let b be the logical page number of the mixed index page in the index page chain. If there is no mixed page, then b is the page number of the first directory page. Let v_{first} and v_{page} be, respectively, the number of encoded offsets on the first directory index page (which can be a mixed page) and a full directory page. Because offsets in the directory are all encoded using the same number of bits, the logical page number of the directory page which contains the offset of vid is

$$p = \begin{cases} b & \text{if } vid \leq v_{first} \\ b + \lfloor \frac{vid - v_{first}}{v_{page}} \rfloor & \text{otherwise.} \end{cases} \quad (1)$$

The offset of vid in the postinglist is the k^{th} n-bit encoded offset on the directory index block on page p, where

$$k = \begin{cases} vid & \text{if } vid \leq v_{first} \\ (vid - v_{first}) \bmod v_{page} & \text{otherwise.} \end{cases} \quad (2)$$

After reading the offset, one more page access is required to read the first row position in $getFirstRowPos$. To achieve this, the logical page number and the location of the encoded row position on its postinglist block needs to be computed. This computation can be performed in a similar way as in Eq. 1 and Eq. 2. Next is to load an index page (with the computed logical page number) and decode the row position (with the computed row position location). Invoking $getFirstRowPos$ is often followed by several calls to $getNextRowPos$. The index iterator keeps the offset to postinglist as its *state*, as well as the offset of the last row position to read. Therefore, each call to $getNextRowPos$ only needs to read from a postinglist block. However, the postinglist block to read from might be already cached in memory, by a previous call to $getNextRowPos$ which has loaded and pinned the required index page. The iterator keeps a handle to the postinglist page, which prevents the unload of the page, until either the iterator gets out of scope, or a reposition of the iterator brings in a new postinglist page and unloads the previously loaded index page. Having handle to the postinglist block prevents the unload of the corresponding index page being used by the iterator. Thus, in consecutive calls of $getNextRowPos$ for the same vid, there is high chance that the corresponding postinglist page is already loaded with a chunk cached. The memory footprint for the index of a paged data vector is bounded by the number of pages that are loaded by corresponding index iterator. At each call of $getFirstRowPos$ and $getNextRowPos$, the iterator brings in at least one page (if both row position and offset are in one mixed index page) and at most two index pages. If the index iterator is created for many value identifiers, then without proper memory management many directory and postinglist pages may stay in memory. In Section 5, we explain how we can effectively limit this issue.

4. DATA AGING USING PAGE LOADABLE COLUMNS

Business data is typically accessed frequently in the beginning of its life-cycle, when the corresponding business processes are still active. After some time, dependent on its status, the data is not accessed as part of the regular working set any more. However, it may happen that the old data becomes interesting again, for example when running analysis over multiple years, when accessing the order history of a customer, or during an audit. This observation is often characterized with the data temperature metaphor. Operationally relevant data is called *hot*, and data that is no longer accessed during normal operation is called *cold*. The data temperature can be used to horizontally partition the application data for optimizing resource consumption and performance. The process of moving data between the different partitions (i.e. from hot to cold) is called *data aging*.

In SAP HANA, aging is different from archiving in the sense that cold data is still kept within the SAP HANA database and remains accessible via SQL in the very same table as the hot data (yet in another partition). Aging does not necessarily replace archiving, although it is enables the system to keep data longer in the database before it is being archived. The goal of aging is to both reduce the main memory footprint and speed up database queries by only keeping operationally relevant (hot) data in main memory. In contrast to this, cold data may be stored, loaded and accessed differently, but remaining accessible via SQL. Business objects often span multiple tables, for example *header* and *lineitem* tables. These objects have statuses, e.g an order's fulfillment. However, such a status might only be set in one of the tables. Using a status as partitioning column is therefore usually neither sufficient nor always possible as partitioning always relates to the columns of a single table where the tuple is stored in. To maintain a brace for the tables that make up a business object and to be backwards compatible, for applications including customer code, we have introduced an artificial temperature column for aging-aware tables. The application actively sets values in this column to a date to indicate that the object is closed and the row shall be moved to the cold partition(s). It does so consistently for all related tables of the same business object, or even for a group of related business objects.

4.1 Storage for Cold Data

The main goal of aging is to remove huge volumes of historical data from main memory and to place it on less expensive storage. There is a direct trade-off between performance and cost. Requirements for access to cold partitions may vary, so do access patterns. Some applications may do OLAP-intensive operations on them, others may access only individual rows using a primary key lookup. A proper sizing and gauging has to take place to find ideal settings in terms of what data is classified as cold and how to store data to accommodate for a given scenario. Cold partitions may reside *externally* as a whole in a separate database, e.g. a federated disk-based database that belongs to the same database image. An orthogonal option is to store cold partition within the same database. For this, there are two possible options:

Default columns. The default behavior of SAP HANA is to load columns entirely into memory upon first access. In case of partitioned tables, only the columns of relevant partitions are touched. A built-in least recently used (LRU) algorithm unloads columns upon memory pressure. A higher unload priority may be used as a weight in the LRU algorithm so that cold partitions are evicted from memory earlier than hot partitions or other memory artifacts of the same age. The drawback of this storage, in the context of aging, is that it causes a long wait time for the first access to cold partitions caused by the load of complete columns. This is not ideal from user perspective. Moreover, huge quantities of memory get loaded that may come into conflict with hot partitions despite a higher unload priority. In an extreme case, a cold partition is read with a fully specified primary key, but the entire column gets loaded into memory.

Page Loadable columns. With cold partitions stored as page loadable columns, only requested pages are being loaded into memory, allowing a lower overall memory consumption. Page loadable columns also facilitate a finer loading granularity. This results in a quicker response to many types of OLTP-like queries, especially for the first access to cold partitions. The pages of columns in cold partitions are loaded into a page pool separate from the pool for other database objects. As described in Section 5, the page pool is associated with a tunable upper and lower boundary. When the upper boundary is reached, pages are unloaded until the lower boundary is reached. Query performance on cold partitions is better when more pages of a column are already loaded. Therefore, the configurable size of the page pool is again a direct trade-off between performance and cost.

The advantage of using page loadable columns rather than any federated database are: 1) it works seamlessly in the existing engine, using the same operators and engines on top without losing any engine features; and 2) the management of memory resources, compared with the default columns, can be done in finer granularity.

4.2 Working with Aging-aware Tables

Aging is an optional feature that may be switched on in the application on business object level, thus effecting the underlying tables. When doing so, typically a non-partitioned table is re-interpreted as a partitioned table with only one partition, which is the hot partition. Another option is that the table is already partitioned, for example with Hash partitioning. Then a logical second level of partitioning is added; Range partitioning with just one defined range for the hot partition. Further, cold partitions will be added when needed with an explicit ADD PARTITION. The cold partitions will typically have page loadable columns enabled from the very beginning.

In summary, enabling aging for a table can be conducted in all straightforward cases without a downtime; all relevant partitioning operations run in constant time, without reorganizing the data layout of the already existing data. The process which moves data from the hot partition to the cold partition(s) is implemented as an update on the partitioning column which triggers a move of the corresponding rows. The rows are being inserted into the delta fragment of the cold partitions as updates. During the asynchronous delta merge step, the data is eventually written into the main fragment of the cold partition, as page loadable columns. Because new data is always written first into the delta fragment, there is no performance impact when using page loadable columns. As the data movement from hot to cold is inherently an ordinary DML operation, it does not block any other operations on the same table.

5. MEMORY MANAGEMENT

Memory management in SAP HANA differs from traditional database buffer manager implementations, as SAP HANA manages memory used by logical resources rather than just physical pages. As an in-memory database, SAP HANA generally requires an entire logical structure, such as a column or index, to be present in contiguous memory. Although the contiguous memory may be allocated as traditional memory pages, the cache policy for these pages is controlled as part of the entire logical structure rather than the pages themselves.

In SAP HANA, the fully-in-memory structures are typically not disk-compatible, meaning the in-memory representation of the logical structure is different than the on-disk representation. In most cases, the in-memory representation must be built and managed as a separate memory resource from the on-disk pages themselves. With page-wise accessible structures described here, each page of the logical structure is registered as a separate resource, whereas completely memory resident structures are registered as a single resource. Unlike buffer caches in traditional database systems, SAP HANA manages page caching for not only pageable data structures, but also the memory resources for fully memory resident structures. These are typically managed by a separate heap manager in most traditional database systems.

SAP HANA uses a weighted LRU (Least Recently Used) strategy to evict unused resources, e.g. columns, persistent pages, and many other cached but currently unused memory objects. To solve a low memory situation, SAP HANA unloads currently unused resources in descending order of $\frac{t}{w}$, where t is the time since the last touch of the resource and w is the weight associated with the *disposition* of the corresponding in-memory page. A resource disposition categorizes the cache eviction policy for each resource. For instance, the objects loaded with *temporary* disposition are expected to be unloaded by the resource manager as soon as they are not needed any more, whereas resources loaded with *non-swappable* disposition cannot be unloaded at all. Every resource of a page loadable column is loaded with *paged attribute* disposition. The resource might be a column page, a dictionary page, or an inverted index page. These pages are unloaded by either the *reactive* or the *proactive* mechanism. The LRU strategy gives higher priority for eviction to resources that are not touched recently and have a smaller disposition weight, compared with other resources.

In case of low memory situation, paged attribute resources are unloaded if they consume more memory than specified by the *lower limit* threshold; we call this the reactive unload. If paged resources consume more memory than specified by the *upper limit* threshold, then these resources are evicted by the LRU strategy until the lower limit is reached – even if plenty of memory is still available. We call this mechanism the proactive unload. The weight of resources does not play any role neither in the reactive approach nor in the proactive approach of unloading paged attribute resources. The proactive unload is executed asynchronously, meaning that it does not block the creation of new paged attribute resources if needed (e.g. loading a piece of a column). As a consequence, the total size of paged attributes resources may exceed the upper limit threshold until the proactive unload strategy is completely executed.

Table 2: Notations used for experiments

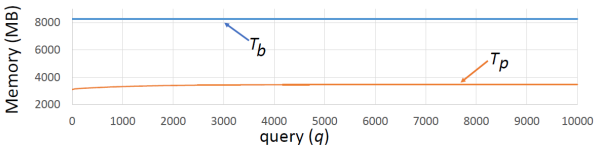
T_b	The base table
T_p	Same schema and data as T_b, with all non-primary key columns in T_p designated to be PAGE LOADABLE
T_{pp}	Same schema and data as T_b, with only the primary key column in T_{pp} designated to be PAGE LOADABLE
T_b^i	T_b with one inverted index per column
T_p^i	T_p with one inverted index per column
Q_{pk}^{num}	Select a numeric column, for a random row: SELECT C_{num} FROM T WHERE C_{pk}=value
Q_{pk}^{str}	Select a string column, for a random row: SELECT C_{str} FROM T WHERE C_{pk}=value
Q_{pk}^{*}	Select all columns, for a random row: SELECT * FROM T WHERE C_{pk}=value
Q_{num}^{count}	Select total count, for random rows where numeric column is equal to a value: SELECT COUNT(*) FROM T WHERE C_{num}=value
Q_{str}^{count}	Select total count, for random rows where string column is equal to a value: SELECT COUNT(*) FROM T WHERE C_{str}=value
Q_{pk}^{rid}	Select row identifier for a random row: SELECT ROWID() FROM T WHERE C_{pk}=value
$Q_{\sigma_{pk}}^{*}$	Select all columns for a range query on the primary key column with a given selectivity SELECT * FROM T WHERE $v_1 \leq C_{pk} \leq v_2$
$Q_{\sigma_{pk}}^{sum}$	Select sum of a numeric column for a range query on the primary key column with given selectivity SELECT SUM(C_{num}) FROM T WHERE $v_1 \leq C_{pk} \leq v_2$
$t(q, T)$	runtime to evaluate query q on table T

6. EXPERIMENTAL EVALUATIONS

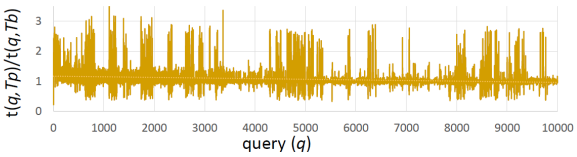
We report the performance and the memory footprint for SAP HANA's default columns, which are always fully loaded vs. the page loadable columns. We first inspect the impact of paging on each data structure individually (Section 6.2) followed by an end-to-end evaluation (Section 6.3).

6.1 Dataset and Metrics

We used an in-house data generator to create a realistic dataset for our study. Our generator produces tables that are very similar to those in a real ERP system. Our base dataset is a columnar table T_b with 100 million records and 128 columns of types INTEGER, DECIMAL, DOUBLE, CHAR, and VARCHAR. Column cardinalities ranged from 1 to 10M. In T_b, 112 columns have less than 100 distinct values and 14 columns have more than 1,000 distinct values. For each experiment, we created a modified table with the same schema and data as T_b, but with a subset of the columns designated as being page loadable and possibly some inverted indexes. Table 2 summarizes the notations for modified tables and query workloads. We compared paged and fully resident columns along two key metrics. First, we monitored system memory when queries were executed. A smaller memory footprint is more desirable. Second, for each test, we measured the time ratio of running a query against the modified table with page loadable columns, over the processing time of the same query on the reference table. A smaller time ratio is more desirable. Every experiment was conducted after a cold system start, to avoid having already cached pages. We set the page loadable limits to default values. All experiments were performed on a server with two CPU sockets, eight cores per socket, and two threads per core, with 256GB of RAM. The machine had SUSE Linux Enterprise Server 11 with a single instance of SAP HANA SPS11.

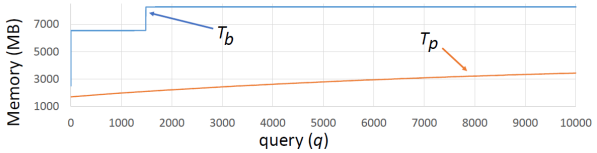


(a) memory footprint varying q

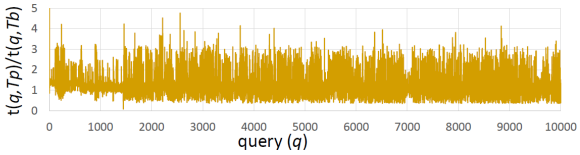


(b) run-time ratio varying q

Figure 4: Single read of C_{num} paged data vector: $q = Q_{pk}^{num}$

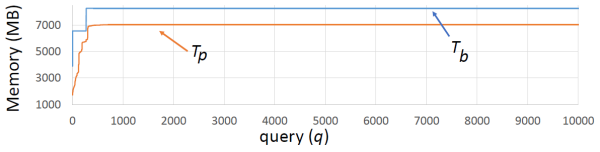


(a) memory footprint varying q

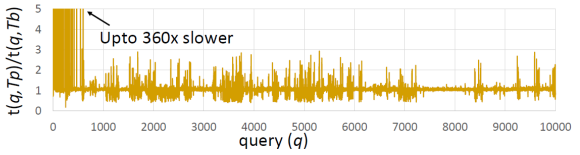


(b) run-time ratio varying q

Figure 5: Single read of C_{str} paged dictionary and paged data vector: $q = Q_{nk}^{str}$



(a) memory footprint varying q



(b) run-time ratio varying q

Figure 6: Multiple reads of C_{str} paged dictionary: $q = Q_{str}^{count}$

6.2 The Impact of Paging Data Structures

6.2.1 *Paging Data Vector*

We created 10,000 random queries of type Q_{pk}^{num} and measured the system memory footprint and the query run-time ratio on T_p over T_b. We picked the query type Q_{pk}^{num} in order to exercise the code path for paging data vector only. In this experiment, the numeric column doesn't have a paged dictionary and there is no access to inverted index of any result column. As Fig. 4 shows, the system memory footprint reduced significantly (from 8.2GB to 3.6GB) when page loadable column was used. As more random queries were processed, more pieces of data vector were pulled into memory. Therefore, the memory footprint for T_p grew while processing more queries. With loading new pieces of the data vector, spikes appear in query run-time ratio (Fig. 4b). The memory resident columns suffer from this performance drop only once, during their complete load. The load time for a full column is significantly larger than for a piece of a page loadable column (43.5 seconds vs. 9.6 seconds). The average query run-time ratio is 1.07, with 90% confidence interval of 0.29. This means accessing paged data vector does not cause a large performance impact for single point queries that do not touch the paged dictionary or the paged inverted index.

6.2.2 *Paging Dictionary*

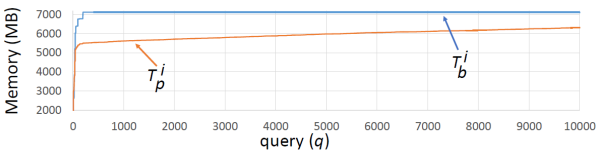
We created 10,000 random queries of type Q_{pk}^{str} on T_b and T_p. Each query in Q_{pk}^{str} returns a (random) string column value for one random row. There is one access to paged data vector, to read the value identifier for the random row, one access to dictionary directory, and one access to one dictionary page. Except for the primary key column, no other inverted index was defined on T_b nor T_p. As Fig. 5a demonstrates, the system memory footprint is again smaller in the page loadable scenario. Notice that at the 1621th query, we observe a 1.2GB increase in the memory footprint for T_b. The query touches one particular column for the first time. Such an access could trigger a column eviction, if the system is under memory pressure. However, we do not observe a memory jump of this magnitude for T_p. From the performance point of view (Fig. 5b), we observe a run-time degradation for T_p whenever either a required paged data vector or a needed dictionary piece – either a directory or a dictionary page – is not in memory. Loading a page has an I/O cost which is much higher than accessing an in-memory data structure. The degradation is often larger than that for a paged data vector[5] because here there is demand for paged data vector and paged dictionary. However, the gap between the run-times of T_p and T_b reduces when the requested dictionary pages are already loaded.

We next created 10,000 random queries of type Q_{str}^{count} on T_b and T_p. Each query of type Q_{str}^{count} exercises search by value and accesses pages from the helper separator index dictionary on value, and the paged dictionary. Searching paged dictionary, in general, requires more pages to be loaded from the helper dictionaries, as well as the paged dictionary itself. Therefore, as depicted in Fig. 6a, the memory footprint for T_p increased very fast for the first 400 queries. The burst in pulling pages is characterized as significant performance degredation (up to 360X) in the run-time ratio of Fig. 6b. After this point, the memory footprint becomes stable in T_p. As in earlier experiments, we see spikes in query run-time ratio when pages are loaded. As the helper dictionaries are key interfaces needed for both $findByValue$ and $findByValueID$, it would be more effective to have these auxiliary dictionaries always loaded in memory, or in a very fast storage (e.g. Non-Volatile Memory).

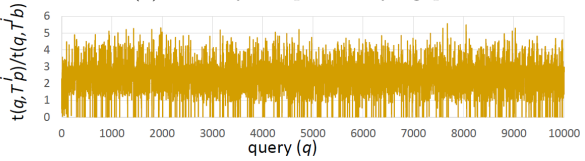
6.2.3 *Paging Inverted Index*

We created 10,000 random queries of type Q_{num}^{count} and evaluated on T_b^i and T_p^i, to only exercise the code path for paged inverted index. As shown in Fig. 7a, the paged inverted index had smaller memory footprint than the loaded inverted index (note, our test here does not include the primary key

[5]Average ratio is 1.24, with 90% confidence interval of 0.69.

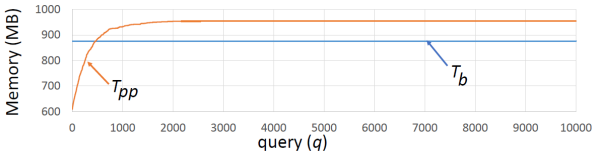


(a) memory footprint varying q

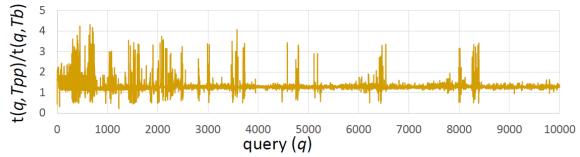


(b) run-time ratio varying q

Figure 7: Multiple reads of C_{num} paged inverted index: $q = Q_{num}^{count}$



(a) memory footprint varying q



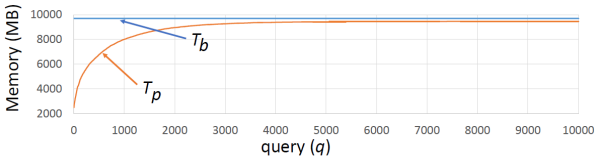
(b) run-time ratio varying q

Figure 8: Single read of C_{pk} paged unique inverted index: $q = Q_{pk}^{rid}$

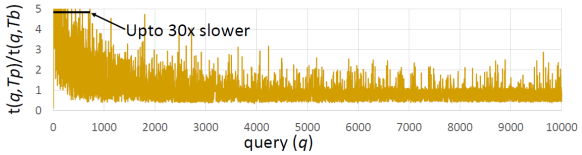
column which is VARCHAR). As most of the columns in T_b^i and T_p^i are sparse, each paged inverted index has a mixed page of postinglist and directory for each column, and no index page which contains just the directory. In the mixed page, searching inverted index is not as catastrophic as the early phase of searching paged dictionary (i.e. Fig. 6b). However, because each search on the paged index requires at most two page accesses (if the index page is not loaded), the overall performance for the paged inverted index is between that of the paged data vector (Fig. 4b) and paged dictionary (Fig. 6b). To evaluate the unique paged inverted index, we created 10,000 Q_{pk}^{rid} queries on T_b and T_{pp}. Search by primary key requires to decode only one value in the postinglist, thus the run-time gets closer to the non-paged index (Fig. 8b) with fewer spikes. Access to paged inverted index is on average 29% slower than non-paged inverted index. However, as shown in Fig. 8a, the memory consumption of a paged inverted index is larger than that of a non-paged inverted index; both indexes store only the postinglist vector but the minimum size for the paged inverted index is one page.

6.3 Putting It All Together

We generated 10,000 random Q_{pk}^* queries on T_p^i and T_b^i. This resembles random single point queries on aged data (e.g. data auditing). Each query does a single read of the paged inverted index of the primary key column, and, to construct the result set, a single read of each column's paged dictionary and paged data vector. As shown in Fig. 9a, the memory footprint is smaller when page loadable columns were used. For the first 1,000 queries, the run-time ratio is



(a) memory footprint varying q



(b) run-time ratio varying q

Figure 9: Single read of a random row, $q = Q_{pk}^*$

Table 3: Memory saving and average run-time ratio, after 10 times repeating 1,000 queries of types $Q_{\sigma_{pk}}^*$, $Q_{\sigma_{pk}}^{sum}$

		rows accessed in T			
		1 row	0.01%	0.1%	1%
Memory reduction (GB)	$Q_{\sigma_{pk}}^*$	5.1	4.9	4.3	2.3
	$Q_{\sigma_{pk}}^{sum}$	4.8	4.6	4.6	4.6
Average run-time ratio	$Q_{\sigma_{pk}}^*$	1.29	1.82	1.62	1.65
	$Q_{\sigma_{pk}}^{sum}$	1.01	1.33	1.15	1.11

large (Fig. 9b); it was rare that all pages required for the first 1,000 queries are loaded in memory. However, as the queries are random, the run-time degradation was not as severe as that for searching the paged dictionary (Fig. 6b). The run-time ratio, after processing 2,000 queries gets very close to one, with average 1.09 and 90% confidence interval of 1.25. However, as exactly same query is almost never re-run, these results do not show run-time if all pages needed to evaluate a query on page loadable columns were already loaded. We designed another experiment which runs queries of types $Q_{\sigma_{pk}}^*$ and $Q_{\sigma_{pk}}^{sum}$ on T_b^i and T_p^i, once after a database restart (cold run) followed by 10 runs of the exactly same workload (hot run). This evaluates the impact of paging on query time when data is already loaded. As Table 3 reports, page loadable columns offer significant reduction in memory footprint, especially for very selective queries. The overhead of using iterators to read from page loadable columns degrades performance up to 82% when 0.01% of rows are selected in $Q_{\sigma_{pk}}^*$. However, when only a single row of a page loadable column is accessed, the performance remains very close to fully loaded column. On the other hand, while page loadable columns often come with a performance overhead, they can be used to keep performance critical columns fully loaded and avoid their eviction.

7. RELATED WORK

There is an extensive history of research on in-memory databases [27] and a growing interest in column oriented DBMS products, e.g. VectorWise [29], MonetDB [18], Vertica [14], IBM DB2 BLU [22], and ParAccel [1], to name a few. The first approach, when the dataset size is slightly larger than the available memory, is to further compress data (i.e. on top of dictionary encoding). Lemke et al. [15] propose sparse encoding of columns along with a number of size specific optimizations. Column reordering has been suggested [16] to be applied on top of existing optimizations to further reduce memory footprint when columns have clus-

ters of adjacent value identifiers. Funke et al. [9] propose to compact the immutable data and optimize it for efficient, memory-consumption friendly snapshotting. This is to free up as much as possible space for having mutable data loaded, thus having minimum impact on mission critical OLTP throughput. They assign a *temperature* to data at Virtual Memory (VM) page granularity, and optimize the layout of the frozen pages for OLAP.

There has been research to improve the performance of cold data handling, mostly for in-memory databases optimized for OLTP. HyPer [13] takes advantage of processor-inherent support for VM management such as address translation, caching, and copy on update. Stoica et al. [24] log data access at tuple level, and compute scores for operating system's VM pages, and evict cold pages. DeBrabant et al. [5] propose *anti-caching* to move cold data from memory to disk in a transactionally safe manner. Under memory pressure, anti-caching gathers the coldest tuples and flush them to disk. Fine grained eviction (at tuple level) is the key advantage of anti-caching, over techniques that rely on OS level VM paging (e.g. [13]). As empirically evaluated recently by Zhang et al. [28], most application semantic is available to user-space approaches that offer finer granularity (e.g. at tuple level) eviction. However, kernel-space approaches (e.g. VM based) can integrate better with the operating system and directly take advantage of CPU, hardware features, and low-level system measurements. Zhang et al. [28] propose a hybrid method to exploit both kernel-level and application level anti-caching. Our approach is similar to anti-caching, in the sense that our internal memory manager keeps track of the resource usage at the granularity of paged column pieces, and decides eviction in cooperation with statistics collected from page usage and operating system. Eldawy et al. [6] propose Siberia as a framework to manage cold data in Microsoft Hekaton main memory engine, optimized for OLTP. For cold data, they propose compact in-memory data structures (bloom filter and range filters) to reduce the number of accesses to cold data for point and range queries. The key difference from our approach with Siberia is that our focus is the main fragment i.e. the read only piece of page loadable column. Siberia guarantees transactional consistency for updates to hot and cold data. However, in SAP HANA, updates are only applied to the delta fragment and applied to cold data during delta merge operation.

Graefe et al. [10] adapt the buffer pool design by replacing page identifiers with specialized pointers. This saves one level of redirection to convert from page identifier to page address. Using this technique, they report in-memory performance for disk based databases, when the working dataset fits completely in the database buffer pool. A number of in-memory database systems avoid the overhead of disk based systems by providing the best performance for memory resident data [19, 18]. There are other commercial systems that do not make this assumption. For instance, the second generation of IBM DB2 BLU [22, 4] provides a buffer pool supported access to relational data. Each DB2 BLU table is partitioned into column groups. Column groups are persisted in pages. Each page contains a header, a section with page-specific compression dictionaries and compressed columns. Fine-grain in-memory synopses and a specialized probabilistic buffer pool protocol for scans were implemented to reduce disk access with aggressive prefetching. SAP HANA is architecturally different from IBM DB2 BLU as SAP HANA applies changes to delta fragments only. Furthermore, SAP HANA uses uniform encoding of n-bit packed columns to efficiently find, prefetch, and load the pieces of each column on demand for page loadable columns.

8. FUTURE RESEARCH DIRECTION

Our current implementation of page loadable columns is based on the classical in-memory columnar database architecture where all the data structures for the main fragment of a logical column are rebuilt during delta-merge operation which brings delta fragment into the main fragment, practically rebuilding the whole main fragment and resetting the delta fragment to an empty state. For page loadable columns, the order-preserving paged dictionary, its helper sparse indexes on the paged dictionary, the paged data vector, and its paged inverted index are all fully rebuilt and persisted during delta-merge operation. A promising new extension to page loadable design is to selectively rebuild and persist, during delta-merge, only the critical data of the main fragment, and employ adaptive and partial rebuilding for non-critical data, all driven by the current workload.

For page loadable columns, the *critical data* is the data stored in the paged dictionary and the data stored in the paged data vector. All other data structures are considered *non-critical data* as they can be recovered and rebuilt from critical data. Based on the workload immediately before a delta-merge operation, it can be decided if the helper indexes on dictionary should be built based on their usage in the past. The same decisions are made for inverted indexes. After the delta-merge operation, driven by the current workload, these structures can be dynamically rebuilt and refined as more queries are executed which access a page loadable column. For example, for a column which is always accessed just by full scan or by set of row identifiers, only the helper index dictionary on value identifier $ipDict_{ValueId}$ is needed. However, if point queries are the norm, all non-critical data structures are used and they are needed to achieve maximum performance. For instance, the usual search on data vector ($findByValueID$) can dynamically rebuild $ipDataVector$ entries driven by the queries being executed.

Recent developments, such as the announcement by Intel and Micron of the new non-volatile memory technology and the addition of persistent memory support to the Intel 64 instruction set [12], hint that Storage Class Memory (SCM) might become a reality in the not too distant future. SCM promises byte addressable persistence with memory-like performance and storage-like capacity, endurance, and cost. The expectation is that SCM will have read and write latencies only within an order of magnitude of DRAM [20], low enough to be masked by prefetching for sequential access patterns. One line of research that we are investigating is improving performance by moving latency sensitive disk-based data structures to SCM, accessing them directly rather than via a DRAM cached copy. In particular, we are considering to move to SCM those non-critical data structures that can be rebuilt from other primary data structures. This mitigates the concern that the contents of SCM can become unavailable if the server hosting the SCM suffers a hardware failure. For paged loadable columns, the most promising candidate data structures are the inverted indexes. and the helper sparse dictionary indexes as their performance determines the performance of paged dictionaries.

9. CONCLUSION

We proposed piecewise implementation of the columnar data structures and access methods for SAP HANA. Instead of having columns fully loaded in main memory, we partially load pieces as they are needed. Our experiments confirm that page loadable columns reduce the memory footprint of a system, especially when only a small fraction of each column is accessed. The small loss of performance may be acceptable for accessing small portion of cold data (e.g. auditing aged data). There are still additional scenarios particularly when dealing with huge datasets where additional online and offline storage hierarchies are still warranted beyond the use of page loadable columns. Page loadable columns are the main driver for reducing the memory footprint in data aging scenarios; they not only reduce the total cost of ownership of large ERP systems, but also leave more memory available to keep performance-critical columns always in memory. Thus, indirectly contribute to the overall database performance.

Acknowledgments

We thank the anonymous reviewers for the comments and suggestions, which contributed to improving significantly the presentation of this paper. We give special thanks to all colleagues and contributors to the paged loadable column project, including Manbeen Kohli, Carsten Thiel, Chaitanya Gottipati, Steffen Geißinger, Christian Lemke, Sebastian Seifert, Robert Schulze, and Mohammed Junaid Azad.

10. REFERENCES

[1] The ParAccel analytic database: A technical overview. http://www.paraccel.com/ Whitepaper, 2009.

[2] [RFC] Minmax indexes. A. Herrera's email to Pg Hackers, 2013.

[3] D. Abadi, P. A. Boncz, S. Harizopoulos, S. Idreos, and S. Madden. The design and implementation of modern column-oriented database systems. *Foundations and Trends in Databases*, 5(3):197–280, 2013.

[4] R. Barber, G. M. Lohman, V. Raman, R. Sidle, S. Lightstone, and B. Schiefer. In-memory BLU acceleration in ibm's DB2 and dashdb: Optimized for modern workloads and hardware architectures. In *ICDE*, 2015.

[5] J. DeBrabant, A. Pavlo, S. Tu, M. Stonebraker, and S. B. Zdonik. Anti-caching: A new approach to database management system architecture. *PVLDB*, 6(14):1942–1953, 2013.

[6] A. Eldawy, J. Levandoski, and P. Larson. Trekking through siberia: Managing cold data in a memory-optimized database. *PVLDB*, 7(11), 2014.

[7] F. Färber, N. May, W. Lehner, P. Große, I. Müller, H. Rauhe, and J. Dees. The SAP HANA database – an architecture overview. *IEEE Data Engineering Bulletin*, 35(1):28–33, 2012.

[8] Z. Feng, E. Lo, B. Kao, and W. Xu. Byteslice: Pushing the envelop of main memory data processing with a new storage layout. In *ACM SIGMOD*, 2015.

[9] F. Funke, , A. Kemper, and T. Neumann. Compacting transactional data in hybrid oltp&olap databases. *PVLDB*, 5(11), 2012.

[10] G. Graefe, H. Volos, H. Kimura, H. Kuno, J. Tucek, M. Lillibridge, and A. Veitch. In-memory performance for big data. *PVLDB*, 8(1), 2014.

[11] S. Idreos, M. L. Kersten, and S. Manegold. Self-organizing tuple reconstruction in column-stores. In *ACM SIGMOD*, 2009.

[12] Intel 64-bit instruction set. https://software.intel.com/sites/default/files/managed/07/b7/319433-023.pdf.

[13] A. Kemper and T. Neumann. Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots. In *ICDE*, 2011.

[14] A. Lamb, M. Fuller, R. Varadarajan, N. Tran, B. Vandier, L. Doshi, and C. Bear. The vertica analytic database: C-store 7 years later. *PVLDB*, 5(12), 2012.

[15] C. Lemke, K. Sattler, F. Faerber, and A. Zeier. Speeding up queries in column stores - A case for compression. In *DAWAK*, 2010.

[16] C. Lemke, K. Sattler, and F. Färber. Kompressionstechniken für spaltenorientierte bi-accelerator-lösungen. In *BTW*, 2009.

[17] Y. Li and J. M. Patel. BitWeaving: Fast Scans for Main Memory Data Processing. In *SIGMOD*, 2013.

[18] Monetdb. https://www.monetdb.org/.

[19] Oracle timesten. http://www.oracle.com/technetwork/products/timesten.

[20] I. Oukid, W. Lehner, T. Kissinger, T. Willhalm, and P. Bumbulis. Instant recovery for main memory databases. In *CIDR*, 2015.

[21] H. Plattner. The impact of columnar in-memory databases on enterprise systems. *PVLDB*, 7(13):1722–1729, 2014.

[22] V. Raman, G. K. Attaluri, R. Barber, N. Chainani, D. Kalmuk, V. KulandaiSamy, J. Leenstra, S. Lightstone, S. Liu, G. M. Lohman, T. Malkemus, R. Müller, I. Pandis, B. Schiefer, D. Sharpe, R. Sidle, A. J. Storm, and L. Zhang. DB2 with BLU acceleration: So much more than just a column store. *PVLDB*, 6(11):1080–1091, 2013.

[23] V. Sikka, F. Färber, W. Lehner, S. K. Cha, T. Peh, and C. Bornhövd. Efficient transaction processing in SAP HANA database: the end of a column store myth. In *ACM SIGMOD*, 2012.

[24] R. Stoica and A. Ailamaki. Enabling efficient os paging for main-memory oltp databases. In *DaMon*, 2013.

[25] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. J. O'Neil, P. E. O'Neil, A. Rasin, N. Tran, and S. B. Zdonik. C-store: A column-oriented DBMS. In *VLDB*, 2005.

[26] T. Willhalm, N. Popovici, Y. Boshmaf, H. Plattner, A. Zeier, and J. Schaffner. Simd-scan: Ultra fast in-memory table scan using on-chip vector processing units. *PVLDB*, 2(1):385–394, 2009.

[27] H. Zhang, G. Chen, B. C. Ooi, K. Tan, and M. Zhang. In-memory big data management and processing: A survey. *IEEE TKDE*, 27(7):1920–1948, 2015.

[28] H. Zhang, G. Chen, B. C. Ooi, W. Wong, S. Wu, and Y. Xia. "anti-caching"-based elastic memory management for big data. In *ICDE*, 2015.

[29] M. Zukowski and P. A. Boncz. Vectorwise: Beyond column stores. *IEEE Data Eng. Bull.*, 35(1), 2012.