Dictionary-based order-preserving string compression for main memory column stores

Conference Paper · January 2009

DOI: 10.1145/1559845.1559877 · Source: DBLP

3 authors, including:

Carsten Binnig
Brown University
66 PUBLICATIONS **737** CITATIONS

SEE PROFILE

Franz Färber
SAP Research
41 PUBLICATIONS **1,776** CITATIONS

SEE PROFILE

Dictionary-based Order-preserving String Compression for Main Memory Column Stores

Carsten Binnig
ETH Zurich
binnigc@inf.ethz.ch

Stefan Hildenbrand
ETH Zurich
stefanhi@inf.ethz.ch

Franz Färber
SAP AG
franz.faerber@sap.com

ABSTRACT

Column-oriented database systems [19, 23] perform better than traditional row-oriented database systems on analytical workloads such as those found in decision support and business intelligence applications. Moreover, recent work [1, 24] has shown that light-weight compression schemes significantly improve the query processing performance of these systems. One such a lightweight compression scheme is to use a dictionary in order to replace long (variable-length) values of a certain domain with shorter (fixed-length) integer codes. In order to further improve expensive query operations such as sorting and searching, column-stores often use order-preserving compression schemes.

In contrast to the existing work, in this paper we argue that order-preserving dictionary compression does not only pay off for attributes with a small fixed domain size but also for long string attributes with a large domain size which might change over time. Consequently, we introduce new data structures that efficiently support an order-preserving dictionary compression for (variable-length) string attributes with a large domain size that is likely to change over time. The main idea is that we model a dictionary as a table that specifies a mapping from string-values to arbitrary integer codes (and vice versa) and we introduce a novel indexing approach that provides efficient access paths to such a dictionary while compressing the index data. Our experiments show that our data structures are as fast as (or in some cases even faster than) other state-of-the-art data structures for dictionaries while being less memory intensive.

Categories and Subject Descriptors

H.2.7 [**Database Management**]: Database Administration—*Data dictionary/directory*; E.4 [**Data**]: Coding and Information Theory—*Data compaction and compression*; H.3.1 [**Information Storage and Retrieval**]: Content Analysis and Indexing—*Dictionaries, Indexing methods*

General Terms

Algorithms, Design, Performance

SIGMOD'09, June 29–July 2, 2009, Providence, Rhode Island, USA.

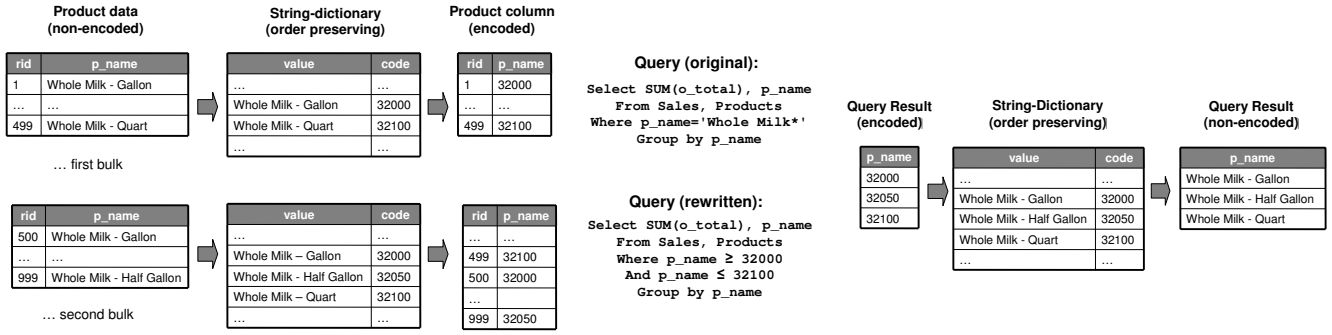
1. INTRODUCTION

Column-oriented database systems (such as Monet-DB [23] and C-Store [19]) perform better than traditional row-oriented database systems on analytical workloads such as those found in decision support and business intelligence applications. Recent work [1, 24] has shown that lightweight compression schemes for column-oriented database systems (called column stores further on) enable query processing on top of compressed data and thus lead to significant improvements of the query processing performance. Dictionary encoding is such a light-weight compression scheme that replaces long (variable-length) values of a certain domain with shorter (fixed-length) integer codes [1]. In order to compress the data of a certain column that is loaded into a data warehouse using such a compression scheme, existing column stores usually create an array of distinct values (i.e., the dictionary) and then store each attribute value of that column as an index into that array. Dictionaries are usually used in column stores if the size of the corresponding domain is small.

Bit packing is then used on top of dictionaries to further compress the data [12]. This compression scheme calculates the minimal number of bits that are necessary to represent the maximal index into the dictionary. Bit packing makes sense if the size of the domain is stable (or known a priori). However, in many practical data warehousing scenarios the domain size is not stable. As an example, think of a cube inside a data warehouse of a big supermarket chain which holds the sales of all products per category (e.g., whole milk, low fat milk, fat free milk) , While the total number of categories is not too large, it is likely that the categories will change over time (i.e., new products are added to the selection of the supermarket).

In order to deal with situations where the domain size is not known a priori, existing column stores usually analyze the first bulk of data that is loaded in order to find out the current domain size of a certain attribute (e.g., the total number of product categories) and then derive the minimal number of bits (for bit packing). However, if subsequent bulks of data contain new values that were not loaded previously, existing column stores usually have to decode all the previously loaded data (e.g., the data stored inside the sales cube) and then encode that data again together with the new bulk using more bits to represent the new domain size. This situation becomes even worse if different attributes (that are not known a priori) share the same global dictionary to enable join processing or union operations directly on top of the encoded data.

In addition, column stores often use order preserving compression schemes to further improve expensive query operations such as sorting and searching because these operations can then be executed directly on the encoded data. However, order-preserving compression schemes either generate variable-length codes (e.g.,



(a) Data Loading (2 bulks) (b) Query Compilation (c) Query Execution

Figure 1: Dictionary-based order-preserving string compression

[2]) that are known to be more expensive for query processing in column stores than fixed-length codes [12], or they generate fixed-length codes (e.g., by using indexes in a sorted array) that are more difficult to extend when new values should be encoded in an order-preserving way.

In contrast to the existing work, in this paper we argue that order-preserving dictionary compression does not only pay off for attributes with small domain sizes but also for long string attributes with a large domain size. For example, the sales cube mentioned before could contain product names of type `VARCHAR(100)`. If we encode one million different product names using a dictionary that generates fixed-length integer codes (e.g., 32-bit), we would get a very good average compression rate for that column.

However, using a sorted array and indexes into that array as fixed-length integer codes is too expensive for large dictionaries where the domain size is not known a priori. There are different reasons for this: First, using a sorted array and binary search as the only access path for encoding data is not efficient for large string dictionaries. Second, if the index into the sorted array is used as integer code, each time a new bulk of string data is loaded it is likely that the complete dictionary has to be rebuilt to generate order-preserving codes and all attributes that use that dictionary have to be re-encoded. For the same reasons, strict bit packing on top of an order-preserving dictionary compression scheme does not make sense either.

Motivated by these considerations, this paper introduces data structures that efficiently support an order-preserving dictionary-compression of (variable-length) string attributes where the domain size is not known a priori (e.g., when the dictionary is shared by different attributes). Furthermore, the integer codes that are generated have a fixed length to leverage efficient query processing techniques in column stores and we do not use bit-packing on top of these integer codes to efficiently be able to support updates. Consequently, in this paper we model the dictionary as a table that specifies a mapping from string-values to arbitrary integer codes and vice versa. Our goal is also to provide efficient access paths (i.e., index structures) to such a dictionary. More precisely, we identify index structures for a string dictionary that efficiently support the following tasks:

- **Data loading:** As discussed before, data is usually loaded bulk-wise into a data warehouse. This means that the dictionary must efficiently support the encoding of bulks of string values using integer codes. The encoding logically consists of two operations: the first operation is a bulk lookup of the integer codes for the string values that are already a part of the dictionary and the second operation is the bulk insertion of the new string values as well as the generation of order-preserving integer codes for those new values. As an example, in Figure 1 (a) we see how two bulks of product data (i.e., the column p_name) are loaded into the sales cube.
- **Query Compilation:** In order to execute analytical queries directly on top of encoded data, it is necessary to rewrite the query predicates. If an order preserving encoding scheme is used, this step is trivial: The string constants of equality- and range-predicates only have to be replaced by the corresponding integer codes. Moreover, prefix-predicates (e.g., `p_name='Whole Milk*'`) can be mapped to range predicates[1]. Consequently, a string-dictionary should enable efficient lookups to rewrite string constants as well as string prefixes. As an example, in Figure 1 (b), we see how the predicate of a query is rewritten.
- **Query Execution:** During query execution, the final query result (and sometimes intermediate query results) must be decoded using the dictionary (which can be seen as a semi-join of the encoded result with the dictionary). As most column stores use vectorized query operations (or sometimes even materialize intermediate query results) [12, 24], a string-dictionary should also support the efficient decoding of the query results for bulks (i.e., bulk-lookups of string values for a given list of integer codes). As an example, in Figure 1 (c), we see how the dictionary is used to decode the column p_name of the encoded query result.

While all the tasks above are time-critical in today's data warehouses, query processing is the most time-critical one. Consequently, the data structures that we present in this paper should be fast for encoding but the main optimization goal is the performance of decoding integer codes during query execution. In that respect, in this paper we identify efficient (cache-conscious) indexes that support the encoding and decoding of string-values using a dictionary. While there has already been a lot of work to optimize index structures for data warehouses on modern hardware platforms (i.e., multi-core-systems with different cache levels), much of this work concentrated on cache-conscious indexes for numerical data (e.g., the CSS-Tree [16] and the CSB$^+$-Tree [17]). However, there has been almost no work on indexes that enable (cache-)efficient bulk lookups and insertions of string values. Consequently, we focus on indexes for encoding string data.

[1]In this paper we do not support wildcards at arbitrary positions.

In addition, the amount of main memory as well as the size of the CPU caches of today's hardware are constantly growing. Consequently, data warehouses start to hold all the critical data in main memory (e.g., the data that is used for online analytical reporting). In that respect, in this paper we address the question of how to compress the dictionary in order to keep as much data as possible in the different levels of the memory hierarchy. Again, while there has been a lot of work on compressing indexes for numerical data [10], almost no work exists for string data [5].

Thus, the contributions of this paper are:

(1) In Section 2, we introduce a new approach for indexing a dictionary of string values (called *shared leaves*) that leverages an order-preserving encoding scheme efficiently. In the shared leaves approach, indexes on different attributes (that can be clustered the same way) can share the same leaves in order to reduce the memory consumption while still providing efficient access paths.
(2) In Section 3, we introduce a concrete *leaf structure* for the shared-leaves approach that can be used by the indexes of a dictionary for efficiently encoding and decoding string values while the leaf structure itself is compressed. We also discuss the most important operations on this leaf structure (i.e., lookup and update) and analyze their costs.
(3) As another contribution, in Section 4, we present two new *cache-conscious string indexes* that can be used on top of our leaf structure to efficiently support the encoding of string data in the dictionary. For the decoding of integer codes we argue why the CSS-Tree [16] is optimal in our case.
(4) Finally, in Section 5, our experiments evaluate the new leaf structure and the new cache-conscious indexes under different types of workloads and show a detailed analysis of their performance and their memory behavior. As one result, the experiments show that in terms of performance our leaf structure is as efficient as other read-optimized indexes while using less memory (due to compression).

2. OVERVIEW

In this section, we first discuss the operations that an order-preserving string-dictionary must support. Afterwards, we present a new idea for indexing such a dictionary (called *shared-leaves*) which is not bound to particular index structures and we show how the operations above can be implemented using this approach. Finally, we discuss requirements and design decisions for index structures that can be efficiently used for a dictionary together with the shared-leaves approach.

2.1 Dictionary Operations

As mentioned in Section 1, in this paper we model a string dictionary as a table T with two attributes: $T = (value, code)$. Thus, table T defines a mapping of variable-length string values (defined by the attribute $value$) to fixed-length integer codes (defined by the attribute $code$) and vice versa. In order to support the data loading as well as the query processing task inside a column store, the interface of the dictionary should support the following two bulk operations for encoding and decoding string values:

- **encode: values → codes:** This bulk operation is used during data loading in order to encode data of a string column (i.e., the $values$) with corresponding integer codes (i.e., the $codes$). This operation involves (1) the lookup of codes for those strings that are already in the dictionary and (2) the insertion of new string values as well as the generation of order-preserving codes for those new values.
- **decode: codes → values:** This bulk operation is used during query processing in order to decode (intermediate) query results (i.e., a bulk of integer $codes$) using the corresponding string values (i.e., the $values$).

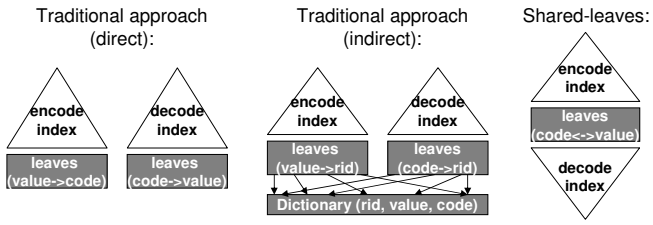


Figure 2: Traditional approaches vs. Shared-leaves

Moreover, the interface of the dictionary should also support the following two operations in order to enable the rewrite of the query predicates (for query processing):

- **lookup: (value, type) → code:** This operation is used during query compilation in order to rewrite a string constant (i.e., the $value$) in an equality-predicate (e.g., `p_name = 'Whole Milk - Gallon'`) or in a range-predicate (e.g., `p_name` $\geq$ `'Whole Milk - Gallon'`) with the corresponding integer code (i.e., $code$). The parameter $type$ specifies whether the dictionary should execute an exact-match lookup (as it is necessary for string constants in equality-predicates) or return the integer code for the next smaller (or larger) string value (as it is necessary for string constants in range-predicates). An example will be given in the following subsection.
- **lookup: prefix → (mincode, maxcode):** This operation is used during query compilation to rewrite the $prefix$ of a prefix-predicate (e.g., `p_name = 'Whole Milk*'`) with the corresponding integer ranges (i.e., the $mincode$ and the $maxcode$). Again, an example will be given in the following subsection.

In order to use table T to efficiently support all these operations, we propose to build indexes for both attributes of table T ($value$ and $code$) because encoding and decoding usually access only a subset of the dictionary. Moreover, indexing the dictionary is especially important if the dictionary is shared between different attributes, because then it is more likely that only a subset of the dictionary is touched (if the domains of the individual attributes are not completely overlapping). Consequently, we believe that in many cases a sequential scan of the complete dictionary does not pay off. The choice of whether to use a sequential scan or an index to access the dictionary, however has to be done as a part of the cost-based query optimization (because it strongly depends on the particular workload). However, this discussion is out of the scope of this paper.

2.2 Shared-leaves Indexing

Traditional approaches for indexing can be classified into two general categories: *direct* and *indirect* indexes. Using these approaches for indexing the two attributes ($value$ and $code$) of table T would result in the following situations (see Figure 2):

(1) In the direct indexing approach, two indexes for encoding and decoding are created that hold the data of table T directly in their leaves. In this case, the table T itself does not need to be explicitly kept in main memory since the data of T is stored in the indexes.
(2) In the indirect indexing approach, two indexes for encoding and decoding are created that hold only references to the data inside table T (i.e., a row identifier rid). In this case, the table T itself needs to be explicitly kept in main memory.

While direct indexing (1) has the disadvantage of holding the data of table T redundantly in the two indexes which is not optimal if the indexes should be main memory resident, indirect indexing (2) (which is standard in main-memory databases [9, 7]) requires one level of indirection more than the direct indexing approach (i.e., pointers into the table T). Thus, (2) results in higher cache miss rates on modern CPUs. Another alternative to index the dictionary data is to extend a standard index (e.g., a B^+-Tree) in order to support two key attributes instead of one (i.e., in our case for $value$ and $code$) . However, in that case both access paths of the index need to read the two key attributes during lookup which increases the cache miss rates (especially when decoding the integer $codes$).

The new idea of this paper is that the two indexes for encoding and decoding share the same leaves (see *shared-leaves* approach in Figure 2) where both indexes directly hold the data of table T in their leaves but avoid the redundancy of the direct indexing approach. Thus, the shared leaves also avoid the additional indirection level of the indirect indexing approach.

As the string dictionary uses an order-preserving encoding scheme, the string values and the integer codes in table T follow the same sort order (i.e., we can have clustered indexes on both columns and thus can share the leaves between two direct indexes). Consequently, as the attribute values $value$ and $code$ of table T can both be kept in sort order inside the leaves, the leaves can provide efficient access paths for both lookup directions (i.e., for the encoding and decoding) using a standard search method for sorted data (e.g., binary search or interpolation search). Moreover, using the shared-leaves for indexing the dictionary means that T does not have to be kept explicitly in main memory because the leaves hold all the data of table T (as for direct indexes).

In the following, we discuss how the shared-leaves approach can support the bulk operations mentioned at the beginning of this section to support the data loading and query processing inside a column store:

Figure 3 shows an example of how the shared-leaves approach can be used to efficiently support the bulk operations for encoding and decoding string values. In order to encode a list of string values (e.g., the list shown at the top of Figure 3), the encode-index is used to propagate these values to the corresponding leaves. Once the leaves are reached, a standard search algorithm can be used inside a leaf to lookup the integer code for each single string value. The decoding operation of a list of integer codes works similar. The only difference is that the decode index is used to propagate the integer codes down to the corresponding leaves (e.g., the list of integer codes shown on the bottom of Figure 3).

If some integer codes for string values are not found by the lookup operation on the encode-index (e.g., the string values 'aac' and 'aad' in our example), these string values must be inserted into the dictionary (i.e., the shared-leaves) and new integer codes must be generated for those values (see the right side of Figure 3). The new codes for these string values have to be added to the result (i.e., the list of $codes$) that are returned by the encoding operation.

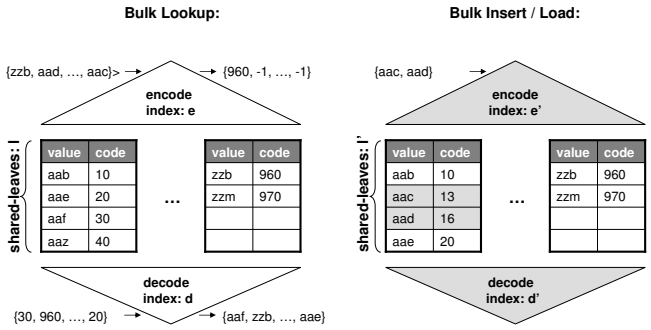


Figure 3: Operations on indexes with shared-leaves

Moreover, the encoding and decoding indexes must be updated if necessary.

In this paper, we do not focus on how to generate new order-preserving integer codes for new string values that are inserted into the dictionary. In order to generate the codes, we simply partition the code range where new string values are inserted in into equi-distant intervals (e.g., in our example the two strings 'aac' and 'aab' are inserted into the code range between 10 and 20). The limits of these intervals represent the new codes (e.g., 13 for 'aac' and 17 for 'aad' in our example). In case that the range is smaller than the number of new string values that have to be inserted in the dictionary, re-encoding of some string values as well as updating the data (i.e., the columns of a table) that use these string values becomes necessary. Analyzing more sophisticated order-preserving encoding schemes that are optimal (i.e., that require minimal re-encoding) under certain workloads as well as strategies for re-encoding the dictionary data are an avenue of our future research work.

In the following, we discuss how the shared-leaves approach can support the two lookup operations mentioned at the beginning of this section that support the predicate rewrite a column store:

The lookup operation which is necessary to rewrite the equality- and range-predicates is similar to the bulk lookup explained before: the encoding index propagates the string constant to the corresponding leaf and then a standard search algorithm can be used on the leaf to return the corresponding integer code. For example, in order to rewrite the predicate `value` $\geq$ `'zzc'` using the encode index in Figure 3 (left side), the encode index propagates the string value 'zzc' to the rightmost leaf and this leaf is used to lookup the next integer code for that string value that is equal or greater than the given value (i.e., the integer code 970 for the string value 'zzm'). The rewritten predicate thus would be `code` $\geq$ `970`.

In order to support the other lookup operation that is necessary to rewrite a prefix-predicate, the encoding index needs to propagate the string prefix to those leaves which contain the minimum and the maximum string value that matches this prefix. For example, in order to rewrite the predicate `value = 'aa*'` using the encode index in Figure 3 (left side), the encode index has to propagate the prefix to the first leaf which contains the minimum and the maximum string value that matches this prefix. Afterwards, those leaves are used to map the strings that represent the boundaries for the given prefix to the corresponding codes (e.g., in our example we retrieve the codes for 'aab' and 'aaz' and rewrite the predicate as `10` $\leq$ `code` $\leq$ `40`).

2.3 Requirements and Design Decisions

The main requirement is that the dictionary should be fast for encoding (i.e., the bulk lookup/insert of integer codes for a list of string values) but the optimization goal is the performance for decoding (i.e., the bulk lookup of string values for a given list of integer codes). Thus, the data structures of the dictionary (i.e., leaves and indexes) should also be optimized for encoding/decoding bulks instead of single values. Moreover, the data structures should be optimized for modern CPUs (i.e., they should be cache-conscious and the operations should be easy to parallelize). In the following we discuss further requirements and design decision for the leaf structure and the indexes of the dictionary.

Leaf structure: The most important requirement for the leaf structure is that it must be able to hold the string values as well as the integer codes in sort order to enable efficient lookup operations (e.g., binary search) for both encoding and decoding (while the leaf structure should be optimized for decoding).

As the dictionary must be memory resident, the memory footprint of the dictionary should be small (i.e., it might make sense to apply a lightweight compression scheme such as incremental encoding to the leaf data). Moreover, the leaf should support the encoding of variable-length string values. While the lookup operations on a leaf are trivial for fixed-length string-values that are not compressed, the lookup operations get more complex if the leaves should also support variable-length string values and compression.

When encoding a bulk of string values, new string values might be inserted into the dictionary which involves updating the shared-leaves. Consequently, the leaf should also enable efficient bulk loads and bulk updates.

Finally, note that the leaf structure can be totally different from the data structure that is used for the index nodes on top. A concrete leaf structure that satisfies these requirements is discussed in detail in the next section.

Encode/Decode index structure: Same as the leaf structure, the indexes for encoding and decoding should keep their keys in sort order to enable efficient lookup operations over the sorted leaves. Another requirement is that the encode index must also support the propagation not only of string constants but also of string-prefixes to the corresponding leaves in order to support the predicate-rewrite task. Moreover, the indexes should also be memory resident and thus have a small memory footprint.

When bulk encoding a list of string values using the encoding index, in addition to the lookup of the integer codes for string values that are already a part of the dictionary, it might be necessary to insert new string values into the dictionary (i.e., update the leaves as well as the both indexes for encoding and decoding) and generate new order-preserving codes for those values. We propose to combine these two bulk operations (lookup and insert) into one operation. In order to support this, we see different strategies:

(1) **All-Bulked:** First, propagate the string values that need to be encoded to the corresponding leaves using the encode index and lookup the codes for those strings that are already in the leaves. Afterwards, insert the new values that were not found by the lookup into the leaves and if appropriate reorganize the updated leaf level (e.g., create a leaf level where all leaves are filled up to the maximal leaf size). Afterwards generate integer codes for the new string values and bulk load a new encode and a new decode index from the updated leaf level (in a bottom-up way).

(2) **Hybrid:** First, propagate the string values that need to be encoded to the corresponding leaves using the encoding index and update the encoding index directly (i.e., do updates in-place during propagation). Then, lookup the codes for those strings that are already in the leaves. Afterwards, insert the new values that were not found by the lookup into the leaves and generate integer codes for all new string values. Finally, bulk load a new decode index from the updated leaf level (bottom-up).

(3) **All-In-Place:** First, propagate the string values that need to be encoded to the corresponding leaves using the encoding index and update the encoding index directly (i.e., do updates in-place during propagation). Then, lookup the codes for those strings that are already in the leaves. Afterwards, insert the new values that were not found by the lookup into the leaves and generate integer codes for all new string values. Propagate each update on the leaf level that causes an update of the decode index (e.g., a split of a leaf) directly to the decode index and apply the update.

In the first two strategies above, the decode index is bulk loaded from the updated leaf level, which means that it should provide a better search performance for decoding which is our main optimization goal. Consequently, in this paper we focus on the first two strategies.

In order to guarantee consistency of the data dictionary, for simplicity we decide to lock the complete indexes as well as the leaves during data loading (i.e., the encoding of string values) because this usually happens only at predefined points in time in data warehousing (i.e., once a day). Thus, no concurrent updates and reads are possible during data loading. However, during query processing (i.e., for decoding query results), we allow concurrency because these are read-only operations.

For persisting the dictionary, currently we only write the updates leaves sequentially to disk as a part of data loading. More sophisticated persistence strategies are a part of our future work.

3. LEAF STRUCTURE

In this section, we present a leaf structure that can be used in the shared-leaves approach (see Section 2) for efficiently encoding and decoding variable-length string values on a particular platform. The general idea of this leaf structure is to keep as much string values as well as the corresponding fixed-length integer codes sorted and compressed together in one chunk of memory in order to increase the cache locality during data loading and lookup operations[2].

3.1 Memory Layout

Figure 4 shows an example of the memory layout of one concrete instance of a leaf structure that represents the dictionary shown in Figure 1 (c). The leaf structure compresses the string values using incremental-encoding [21] while each n-th string (e.g., each 16-th string value) is not compressed to enable an efficient lookup of strings without having to decompress the complete leaf data: In the example, *value 16* 'Whole Milk - Gallon' is not compressed and *value 17* 'Whole Milk - Half Gallon' is compressed using incremental-encoding; i.e., the length of the common prefix compared to the previous value (e.g., 11) is stored together with the suffix that is different (e.g., 'Half Gallon')

In order to enable an efficient lookup using this leaf structure, an offset vector is stored at the end of the leaf that holds references (i.e., offsets) and the integer codes of all uncompressed strings of

[2]In this paper we assume that string values are encoded in ASCII using one byte per character.

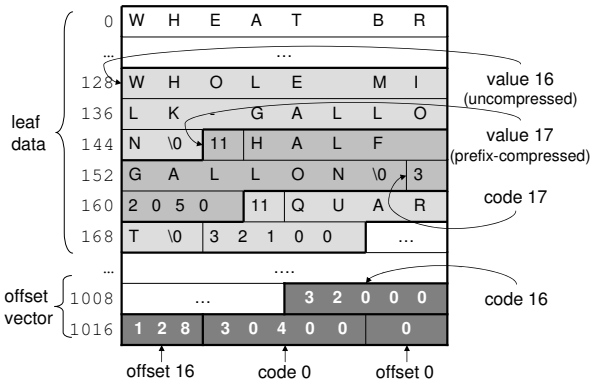


Figure 4: Leaf for variable-length string values

a leaf also in a sorted way. For example, the offset 128 and the code 32000 are stored in the offset vector for *value 16* (see Figure 4). The integer codes of the compressed string-values are stored together with the compressed string values in the data section and not in the offset vector (e.g., the code 32050 for *value 17*). The offset vector is stored in a reverse way to enable efficient data loading while having no negative effect on the lookup operations.

In order to adapt the leaf structure to a certain platform (e.g., to the cache sizes of a CPU) different parameters are available:

- **Leaf-size** l**:** Defines the memory size in bytes that is initially allocated for a leaf. In our example, we used $l = 1024$.
- **Offset-size** o**:** Defines the number of bytes that are used as offset inside the leaf. This parameter has influence on the maximal leaf size. If we use $o = 2$ bytes, as in the example in Figure 4, then the maximal leaf size is 2^{16} bytes (64kB).
- **Code-size** c**:** Defines the number of bytes that are used to represent an integer codeword. If we use $c = 4$, we can generate 2^{32} different codewords.
- **Prefix-size** p**:** Defines the number of bytes that are used to encode the length of the common prefix for incremental encoding. This parameter is determined by the maximal string length. For example, if we use strings with a maximal length of 255, then we can use $p = 1$ because using one byte allows us to encode a common prefix length from 0 to 255.
- **Decode-interval** d**:** Defines the interval that is used to store uncompressed strings (16 in our example). This parameter has influence on the size of the offset vector (i.e., the smaller this interval is, the more space the offset vector will use).

We do not allow the definition of these parameters on the granularity of bits because most CPUs have a better performance on data that is byte-aligned (or even word- or double-word aligned on modern CPUs). We decide to use byte-aligned data (and not word- or double-word aligned data) because the other variants might lead to a dramatic increase in the memory consumption of the leaves.

As an example, assume that we want to tune the leaf structure for a platform using one CPU (with one core) having an L2 cache of 3MB and an L1 cache of 32kB. Moreover, assume that the leaf should only store strings with an average length of 50 characters and a maximum length of 255 characters (which means that $p = 1$ can be used). In that case it would make sense to use a leaf size not bigger than the L2 cache size, say max. 2MB. Consequently, an offset-size $o = 3$ is sufficient to address all values inside such a leaf. The code-size depends only on the overall number of strings that should be stored in a dictionary. We assume that $c = 4$ is an appropriate choice for this example.

Defining the value for the decode interval d is more difficult: in general, we want to make sure that the offset vector once loaded remains in the L1 cache (e.g., having a max. size of 32kB) such that an efficient binary search is possible for a bulk of strings. With the given settings of the example above, we assume that a leaf will be able to store approx. 42000 strings (which results from the max. leaf size of 2MB and the average string length of 50). Moreover, each uncompressed string utilizes 7 bytes of the offset vector (for the offset and the code). Consequently, the offset vector can store max. $32k/7 \approx 4681$ entries (i.e., offsets and codes) for uncompressed strings which means that we could store each $d \approx 42000/4681 \approx 9$th string uncompressed in the example.

3.2 Leaf Operations

The most important operations on the leaf structure are the lookup operations for encoding string values with their integer codes and decoding the integer codes with their string values. Moreover, the leaf also supports updates (i.e., inserting new strings). In the following paragraphs, we examine these operations in detail.

Bulk Lookup: In this paragraph, we first explain how the lookup works for a single value and then discuss some optimizations for bulks. The leaf supports the following lookup operations: one to lookup the code for a given string value (i.e., value $v \rightarrow$ code c) and another one that supports the lookup vice versa (i.e., code $c \rightarrow$ value v). In order to search the code c for a given value v, the procedure is as follows:

1. Use the offset vector to execute a binary search over the uncompressed strings of a leaf in order to find an uncompressed string value v' that satisfies $v' \leq v$ and no other uncompressed value $\bar{v}$ exists with $v' < \bar{v} < v$.
2. If $v' = v$ return the corresponding code c for v that is stored in the offset vector.
3. Otherwise, sequentially search value v from value v' on until v is found or the next uncompressed value appears. In the first case return the code c, in the second case indicate that the value v was not found.

Note that for sequentially searching over the incrementally encoded string values no decompression of the leaf data is necessary. Algorithm 1 shows a search function that enables the sequential search over the compressed leaf data. The parameters of this function are: the leaf data (i.e., $leaf$), the offset where to start and end the sequential search (i.e., $start$ and end), as well as the value that we search for (i.e., v). The return value is the corresponding code c if the value is found, otherwise the algorithm returns -1. The basic idea of the algorithm is that it keeps track of the common prefix length (i.e., $prefix\_len$) of the current string (at the offset $start$) and the search string v. If this common prefix length is the same as the length of the search string then the correct value is found and the code can be returned. The variables p and c are constants that represent the prefix-size and the code-size to increment the offset value $start$.

The lookup operation to find a string value v for a given code c works similar as the lookup operation mentioned before using the offset vector. The differences are that the first step (i.e., the search over the offset vector) can be executed without jumping into the leaf data section because the codes of the uncompressed strings are stored together with the offset vector. In contrast to the other lookup operation, for the search over the offset vector we theoretically expect to get only one L1 cache miss using a simplified cache

Algorithm 1 Sequential search of string v on compressed leaf

```
function SEQUENTIALSEARCH(leaf, start, end, v)
    v' ← leaf[start]                          ▷ read string v' at offset start
    start ← start + size(v')                  ▷ increment offest by string size
    prefix_len ← prefix_len(v, v')            ▷ calculate common prefix len
    while start ≤ end and prefix_len < |v| do
        curr_prefix_len ← leaf[start]         ▷ get curr. prefix len
        start ← start + p                     ▷ increment offest by prefix-size p = 1
        v' ← leaf[start]
        start ← start + size(v')
        if curr_prefix_len <> prefix_len then
            continue                          ▷ prefix of curr. value v' too short/long
        else if compare(v', v) > 0 then
            return -1                         ▷ curr. value v' comes after search value v
        end if
        prefix_len ← prefix_len + prefix_len(v, v')
        start ← start + c                     ▷ increment offest by code-size c = 4
    end while
    if prefix_len = |v| then
        return leaf[start − c]                ▷ string v found: return code
    else
        return -1                             ▷ string v not found
    end if
end function
```

model (if the offset vector fits into the L1 cache). Another difference of this lookup operation is that during the sequential search the string values have to be incrementally decompressed.

When executing both lookup operations, we expect to theoretically get one L2 cache miss in a simplified cache model if we assume that loading the complete leaf causes one miss and the leaf fits into the L2 cache[3]. The average costs for these two lookup operations are as follows (where n is the number of strings/codes stored in a leaf and d is the decode interval):

$$O(log(n/d)) + O(d)$$

In order to optimize the lookup operation for bulks (i.e., a list of string values or a list of integer codes), we can sort the lookup probe. By doing this, we can avoid some search overhead by minimizing the search space after each search of a single lookup probe (i.e., we do not have to look at that part of the offset vector/leaf data that we already analyzed).

Bulk Update: In this paragraph, we explain how to insert new strings into the leaf structure. As we assume that data is loaded in bulks, we explain the initial bulk load and the bulk insert of strings into an existing leaf.

In order to initially bulk load a leaf with a list of string values, we first have to sort the string values. Afterwards, the leaf data can be written sequentially from the beginning of the leaf while the offset vector is written reversely from the end. If the string values do not utilize the complete memory allocated for a leaf (because we do not analyze the compression rate before bulk loading) then the offset vector can be moved to the end of the compressed data section and the unused memory can be released.

In order to insert a list of new string values into an existing leaf, we again have to sort these string values first. Afterwards, we can do a sort merge of these new string values and the existing leaf in order to create a new leaf. The sort merge is cheaper if we can reuse as much of the compressed data of the existing leaf as possible and thus do not have to decode and compress the leaf data again. Ideally, the new string values start after the last value of the existing leaf. In that case, we only have to compress the new string values without decoding the leaf. If the list of string values and the existing leaf data do not fit into one leaf anymore, the data has to be split. However, as the split strategy depends on the index structure that we build on top of the leaves, we discuss this in the next section.

[3]A more fine grained cache model would respect cache lines and cache associativity. However, as we focus on bulk lookups our simplified model is sufficient to estimate the costs of cache misses.

4. CACHE-CONSCIOUS INDEXES

In this section, we present new cache-conscious index structures that can be used on top of the leaf structure presented in the previous section. These indexes support one of the first two update strategies (*All-Bulked* and *Hybrid*) discussed in Section 2. For the encoding index, we present a new cache sensitive version of the patricia trie (called *CS-Array-Trie*) that supports the *Hybrid* update strategy and a cache sensitive version of the Prefix-B-Tree [5] (called *CS-Prefix-Tree*) that supports the *All-Bulked* update strategy.

As decoding index, we reuse the CSS-Tree [16] which is known to be optimized for read-only workloads. We create a CSS-Tree over the leaves of the dictionary using the minimal integer codes of each leaf as keys of the index (i.e., the CSS-Tree is only used to propagate the integer values that are to be decoded to the corresponding leaves). As the CSS-Tree can be bulk loaded efficiently in a bottom-up way using the leaves of the dictionary, it satisfies the requirements for both update strategies (*Hybrid* and *All-Bulked*).

4.1 CS-Array-Trie

As a first cache-conscious index structure that can be used as an encode index to propagate string lookup probes and updates to the corresponding leaf in a shared leaf approach, we present the *CS-Array-Trie*. Compared to existing trie implementations the CS-Array-Trie uses read-optimized cache-conscious data structures for the index nodes and does not decompose the strings completely.

4.1.1 General Idea

Many existing trie implementations are using nodes that hold an array of pointers to the nodes of the next level with the size of the alphabet (e.g., 128 for ASCII). While such an implementation allows efficient updates and lookups on each node, it is not memory-efficient because the array trie allocates space for a pointers per node, where a is the size of the alphabet (while a pointer uses 8 bytes on a 64-bit system).

Other trie implementations avoid the memory overhead and use a sorted linked list [13] to hold only the characters of the indexed strings together with a pointer to the next level of the trie. While this implementation still offers efficient node updates, the lookup must execute a search over the characters stored in the linked list. However, linked lists are known to be not very cache efficient on modern CPUs because of pointer-chasing [20]. Moreover, most existing trie implementations decompose the indexed strings completely (i.e., each letter of a string results in a node of the trie).

Compared to the implementations mentioned above, a node of the *CS-Array-Trie* uses an array instead of a linked list to store the characters of the indexed string values. Compared to a linked list an array is not efficient when sequentially inserting single values into a trie. However, when bulk inserting new values into a CS-Array-Trie, we need grow the array of a node only once for each bulk. In order to lookup a string value of a CS-Array-Trie, the search over the characters of a node (i.e., the array) is more efficient than the sequential search on a linked list because the array supports binary search and all characters are stored clustered in memory.

The second key idea of the *CS-Array-Trie* is that it does not decompose the string values completely. The *CS-Array-Trie* stores

a set of strings that have the same prefix together using the leaf structure that we discussed in the previous section. A leaf of the *CS-Array-Trie* stores the complete strings and not only their suffixes (without the common prefix) in order to enable efficient decoding of integer codes using the same leaves (as described in our shared-leaves approach). Moreover, storing the complete strings in a leaf (i.e., repeating the same prefix) is still space efficient because we use incremental encoding to compress the strings. Finally, for those trie nodes that only hold a pointer to one child node, we use the path compression used in patricia tries [15].

Figure 5 (left side) shows an example of a CS-Array-Trie that indexes nine different strings. If we follow the path 'aa' in the trie, we reach a leaf that holds only strings that start with the prefix 'aa'. The leaves are shown as uncompressed tables for readability. The physical memory layout is as discussed in Section 3.

4.1.2 Index Operations

In order to encode a bulk of string values during data loading using that trie, we implement the *Hybrid* update strategy for the CS-Array-Trie which means that new string values are inserted into the trie when the strings that should be encoded are propagated through the encoding index (i.e, the trie). In order to leverage the fact of having bulk inserts, we propagate the string values (in preorder) to the leaves using variable buffers at each node of the trie to increase the cache locality during lookup as described in [22]. Moreover, when using buffers at each node we can grow the array of characters stored inside a node only once per bulk. The work in [22] showed that this effectively reduces data cache misses for tree-based indexes and results in a better overall lookup performance.

Figure 5 (right side) shows an example for encoding a bulk of six strings using the existing encode index (i.e., the trie shown on the left side). First, all strings that need to be encoded are propagated from the root node of the trie to the first level of nodes creating three buffers ((1), (2), and (3)). In order to keep the order of strings in the lookup probe for creating the encoded result (at the bottom of Figure 5), a sequence number is added for each value in the buffers (as suggested in [22]). The strings that are propagated from the root to the first level of buffers are analyzed and the missing character 'm' for the string 'mzb' is added to the root node[4].

Afterwards, the buffer (1) is propagated to the next level creating two new buffers (4) and (5) and buffer(1) is returned to a buffer pool (to avoid expensive memory allocation for buffer pages). Next, buffers (4) and (5) are processed, which means that values for the codes for existing string values are looked up and new strings are inserted into the existing leaves with a placeholder as their integer code (e.g., -1): While buffer (4) contains two new strings 'aax' and 'aay' that are inserted to the leftmost leaf, buffer (5) contains only one new string 'amc' that is inserted the next leaf. Note, that the new values in buffers (4) and (5) are not yet deleted and kept to lookup the integer codes for those string values (that are not yet generated).

A question that arises here is, whether the new strings fit into the existing leaf (i.e., the new leaf size is expected to be less than the maximal leaf size) or whether the leaf must be split up into several leaves. In order to estimate the expected leaf size, we add the size (uncompressed) of all new strings in a buffer page as well as the size of their new codes (without eliminating duplicates) to the current leaf size. If the bulk is heavily skewed and contains many duplicates it is likely that we decide to split a leaf even if it is not necessary.

When all string values are propagated to their corresponding leaves (i.e., the new strings are inserted into the leaves), new integer codes are generated for the new string values. This is done by analyzing the number of strings that are inserted in between two existing string values. For example, in order to generate codes for the three new string values that are inserted into the first two leaves between 'aam' and 'amd' in Figure 5 (right side), we have to generate three new codes that must fit into the range between 40 and 50. Using our equi-distance approach (discussed in Section 2), we generate 43, 45, and 48 as new codes. Therefore, the trie must allow us to sequentially analyze all leaves in sort order (i.e., each leaf has a pointer to the next leaf). Finally, after generating the integer codes for the new string values of the trie, we use the buffer pages that we kept on the leaf level (e.g., the new strings in the buffers (2), (4), and (5) in the example) to lookup the integer codes for the new string values and use the sequence number to put the integer code at the right position in the answer (see bottom of Figure 5).

Another task that can efficiently be supported using the CS-Array-Trie, is the predicate rewrite: for equality- and range-predicates the constants are simply propagated through the trie without buffering. For prefix-predicates, the prefix is used to find the minimal and maximal string value that matches this prefix (which is also trivial when using a trie).

4.1.3 Cost Analysis

For propagating a bulk of string values from the root of a CS-Array-Trie to the leaves, we theoretically expect to get one L2 data cache miss for each node in our simplified cache model. In addition, for leaf that has to be processed during the lookup, we expect to get another L2 data cache miss using our simplified cache model.

Moreover, the input and output buffers of a node should be designed to fit into the L2 cache together with one node (to allow efficient copying from input to output buffers). In that case, we expect to get one cache miss for each buffer page that needs to be loaded. Moreover, generating the new integer codes will cause one L2 data cache miss for each leaf of the trie. Finally, executing the lookup of the new integer codes, will also cause one cache miss for each buffer page that has to be loaded plus the L2 cache misses for executing the lookup operations on the leaves (as discussed in the section before).

The costs for encoding a bulk that contains no new string values (i.e., a pure lookup) are composed of the costs for propagating the strings through the trie plus the costs for the lookup of the codes for all strings using the leaves. If we assume that all strings in the lookup probe have the same length m and are distributed uniformly, the height of the trie is $log_a(s/l)$ in the best case and equal to the length of the string m in the worst case (where s is the total number of strings, l is the number of strings that fit in one leaf, and a the size of the alphabet). The lookup costs on each node are $O(log(a))$. Thus, the average costs for propagation are:

$$O(s * ((log_a(s/l) + m)/2) * log(a))$$

4.1.4 Parallelization

The propagation of string values from the root of the trie to the leaves (including the update of the nodes) can be easily parallelized for different sub-tries because sub-tries share no data.

Moreover, the generation of new integer codes can be done in parallel as well (without locking any data structures). For this we need to find out which leaves hold contiguous new string values (i.e., sometimes a contiguous list of new string values might span more than one leaf as shown in Figure 5 on the right side for the first two leaves).

Finally, the lookup operation of the new string values can also be

[4]All updates on the trie and the leaves are shown in light gray in Figure 5 (right side).

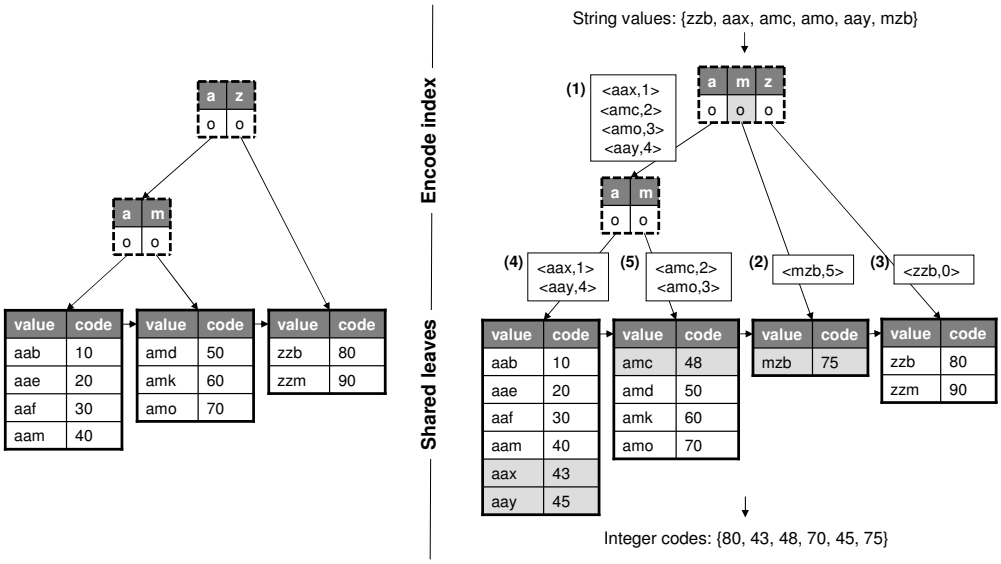


Figure 5: Encode data using the CS-Array-Trie

parallelized without locking any data structures. In Figure 5 (right side), the new values in the buffer pages (2), (4), and (5) can be processed by individual threads for example. We can see that each thread writes the resulting integer codes to different index positions of the result (shown at the bottom) using the sequence number of the buffer pages. Thus, no locks on the result vector are necessary.

4.2 CS-Prefix-Tree

As a second cache-conscious index structure that can be used as an encode index to propagate string lookup probes and updates to the corresponding leaf in a shared leaf approach, we present the *CS-Prefix-Tree*. This index structure combines ideas from the Prefix-B-Tree [5] and the CSS-Tree [16].

4.2.1 General Idea

Same as the Prefix-B-Tree, a node of a CS-Prefix-Tree contains the shortest prefixes that enable the propagation of string values to the corresponding child nodes. However, instead of storing a pointer to each child, the *CS-Prefix-Tree* uses a contiguous block of memory for all nodes and offsets to navigate through this block (as the CSS-Tree). This effectively reduces memory consumption and avoids negative effects on the performance due to pointer chasing. In order to further reduce the memory footprint of the CS-Prefix-Tree, we only store the offset to the first child node explicitly. Since the nodes have a fixed size s, we can calculate the offset to a child node using offset arithmetics (i.e., the i-th child of a node can be found at offset $o = offset(first\_child) + (i * s)$).

In order to enable fast search over the variable-length keys of a node (e.g., binary search), we store the offsets to the keys (i.e., the string prefixes) in an offset vector at the beginning of each node. Moreover, the node size has to be fixed in order to use offset arithmetics for computing the index to the child nodes. Thus, the number of children of a node is variable because we store variable-length keys inside a node.

4.2.2 Index Operations

The CS-Prefix-Tree (as the CSS-Tree) can only be bulk loaded in a bottom-up way which means that it is only suitable for the *All-bulked* update strategy discussed in Section 2. Using the *All-bulked* update strategy for the encoding a list of string values means that the new string values (that are not yet a part of the dictionary) must first be inserted into the leaves (in sort order) and then the index can be created.

Thus, if the first bulk of string values should be encoded using the *All-bulked* update strategy, the complete leaf level has to be built using these string values. Therefore, we reuse the idea in [18] and create a trie (more precisely a CS-Array-Trie) to partition the string values into buckets that can be sorted efficiently using multi-key quicksort [6]. The sorted string values can then be used to create leaves that are filled up to the maximum leaf size. From these leaves, a new encode index (i.e., a CS-Prefix-Tree) is bulk loaded in a bottom-up way.

Figure 6 shows an example of a CS-Prefix-Tree. In the following, we describe the bulk load procedure of a CS-Prefix-Tree from a given leaf level:

1. In order to bulk load the CS-Prefix-Tree, we process the leaf level in sort order: We start with the first two leaves and calculate the shortest prefix to distinguish the largest value of the first leaf and the smallest value of the second leaf. In our example it is sufficient to store the prefix 'am' in order to distinguish the first two leaves. This prefix is stored in a node of the CS-Prefix-Tree. Note, a node does not store a pointer to each child since we can derive an offset into the leaf level (i.e., the sorted list of leaves). Since we do not know the size of the offset vector in advance, we write the offset vector from left to right and store the keys from right to left. The example assumes a fixed node size of 32 bytes and therefore we store offset 29 in the offset vector and write the prefix at the corresponding position.
2. Next, we calculate the shortest prefix to distinguish the largest value of the second leaf and the smallest value of the third leaf and so on until all leaves are processed. If a node is full, we start a new node and store the index to the first leaf that will be a child of this new node as an anchor. In our example the first node covers the first four leaves and therefore the index of the first child in the second node is 4. Note that the nodes are stored continuously in memory.
3. As long as more than one node is created for a certain level of the CS-Prefix-Tree, we add another level on top with nodes

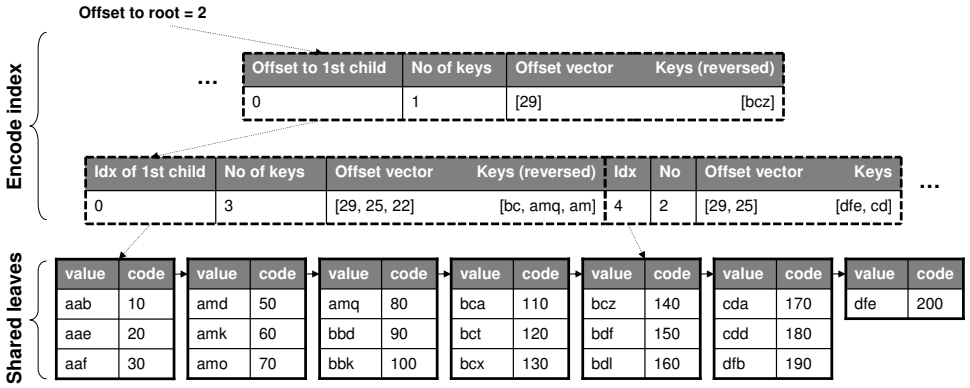


Figure 6: Example of a CS-Prefix-Tree

that store prefixes that distinguish their child nodes. In our example we have one node on top of the lowest level of the CS-Prefix-Tree. This node is the root of the tree and we store the offset to this node in the contiguous memory block (2 in our example). Since the tree is built bottom up, the nodes have to be stored in that sequence in memory.

For subsequent bulks, we use the existing CS-Prefix-Tree to propagate the string values that are to be encoded to the corresponding leaves. In our current implementation we buffer the string values only on the leaf level (and not within the CS-Prefix-Tree). Afterwards, we do a sort-merge of the existing leaf with the new string values stored in the buffers. If the new string values in the buffers and the values of the existing leaf do not fit into one leaf, we simply create another leaf. This very simple sort-merge strategy can be improved in different ways: for example, one could try to create equally sized leaves to avoid degenerated leaves that contain only a small number of strings. Moreover, after encoding several bulks it might make sense to reorganize the complete leaf level by filling all leaves to their maximum. Once all updates are processed we bulk load a new CS-Prefix-Tree from the merged leaf level.

For example, in order to propagate a bulk of strings to the leaves using the CS-Prefix-Tree, we proceed as follows: assume we are looking for value 'amk' in our example in Figure 6. We start at the root node and see that 'amk' is smaller than 'bcz', thus we proceed at the first child of this node at offset 0. We find that 'amk' is between 'am' and 'amq' and thus proceed at the second child of this node at index 1 (calculated from the index of the first child). The CS-Prefix-Tree stores the information that nodes below a certain offset point to leaves instead of internal nodes.

To rewrite query predicates using the CS-Prefix-Tree, we do a simple lookup with the string constants if it is an equality-predicate or a range-predicate. If it is a prefix-predicate, the prefix is used to find the minimal string value that matches the prefix. The lookup for the prefix will end up at leaf that contains this value even if the value itself is not in the dictionary. From that leaf on, we execute a sequential search for the maximum string value that matches the prefix. We could save some effort compared to a sequential search, if we also use the index to find the leaf that holds the maximum value directly (similar to a skip list).

One problem of using a contiguous block of memory and offsets is that the memory has to be allocated in advance. We calculate the maximum amount of memory that all nodes of the CS-Prefix-Tree need by introducing an artificial limit on the maximum length of the keys in the tree. We then can calculate the minimum number of keys that fit into one node and thus can estimate the maximal number of nodes that we need to store the data. One possibility to overcome this problem is to leverage the idea of a CSB-Tree [17] to use a mix of pointers and offset arithmetics (e.g., one pointer per node) to identify the correct child and thus allow multiple blocks of memory instead of one single block.

4.2.3 Cost Analysis

We suggest to set the size of a node of a CS-Prefix-Tree at most to the size of the L2 cache. Thus for propagating a bulk of string values from the root of a CS-Prefix-Tree to the leaves, we theoretically expect to get one L2 data cache miss (in our simplified cache model) for each node that is traversed during the lookup for each string value and another L2 data cache miss when the value is written to the corresponding buffer of the leaf (which should also be designed to fit into the cache). Moreover, the generation of new codes and the encoding itself will each cause one L2 data cache miss for each leaf in the leaf level (if the leaf and the output buffer fit in the L2 cache together).

The costs for encoding a bulk that contains no new string values (i.e., a pure lookup) is composed of the costs for propagating the strings through the tree plus the costs of the lookup in the leaf. If we assume that all strings in the lookup probe have the same length and are distributed uniformly, the height of the tree is $log_k(s/l)$ (where s is the total number of strings, l is the number of strings that fit in one leaf, and k the number of keys that fit into one node). The lookup costs on each node are $O(log(k))$. Thus, the average costs for propagation are:

$$O(s * log_k(s/l) * log(k))$$

Compared to the CS-Array-Trie, it is more expensive to build a CS-Prefix-Tree because the data has to be sorted first and then the CS-Prefix-Tree is loaded bottom-up as opposed to the CS-Array-Trie that is loaded top-down and implicitly sorts the data during that process. We show in our experiments that the CS-Prefix-Tree performs slightly better than the CS-Array-Trie for a pure lookup workloads (i.e., encoding a bulk of strings that does not contain new values) since on average the tree is expected to be less high than the trie and the leaves are organized more compact.

4.2.4 Parallelization

Our current implementation supports multiple threads at the leaf level. Once the bulk is completely buffered at the leaves, the lookup can be executed in parallel as described for the CS-Array-Trie. Since we want to support variable length keys we cannot parallelize the bottom-up bulk loading of the tree.

Currently, the bulk lookup on the index is single threaded since we are not buffering the requests within the tree and thus have no point to easily distribute the workload to multiple threads. One way to parallelize the lookup would be to partition the bulk before accessing the index and process these partitions using multiple threads. Since the buffers at the leaf level would then be shared by several threads, either locking would be needed or the partitioning has to be done according to the sort order, which is expensive.

5. PERFORMANCE EXPERIMENTS

This section shows the results of our performance experiments with the prototype of our string-dictionary (i.e., the leaf structure discussed in Section 3 and the cache conscious indexes discussed in Section 4). We executed three experiments: the first experiment (Section 5.1) shows the efficiency of our leaf structure and compares it to other read-optimized indexes[5], the second experiment (Section 5.2) examines the two new cache conscious indexes using different workloads, and finally the last experiment (Section 5.3) shows the overall performance and scalability of our approach.

We implemented the data structures in C++ and optimized them for a 64-bit Suse Linux Server (kernel 2.6.18) with two Intel Xeon 5450 CPUs (each having four cores) and 16 GB of main memory. Each CPU has two L2 caches of 6MB (where two cores share one L2 cache) and one L1 cache for each core with 32kB for data as well as 32kB for instructions.

In order to generate meaningful workloads, we implemented a string data generator that allows us to tune different parameters like the number of strings, string length, alphabet size, the distribution, and others. We did not use the TPC-H data generator *dbgen*, for example, because most string attributes either follow a certain pattern (e.g., the customer name is composed of the prefix 'Customer' and a unique number) or the domain size of such attributes is too low (e.g., the name of a country). Thus the data generated by *dbgen* does not allow us to generate workloads that let us analyze our data structures with workloads that have certain interesting properties. We will show the properties of the workloads that we generated for each experiment individually.

5.1 Efficiency of Leaf Structure

In this experiment, we analyze the efficiency of the leaf structure discussed in Section 3. The general idea of this experiment is to show the performance and memory consumption of the leaf operations for two different workloads. We used the parameters in the following table to configure the leaf structure (first part) and to generate our workloads (second part).

Parameter	Value
Leaf-size l	64kB - 16MB
Offset-size o	4 bytes
Code-size c	8 bytes
Prefix-size p	1 byte
Decode-interval d	16
String-length	(1) 25, (2) 100
String-number	(1) $\sim$ 450000, (2) $\sim$ 150000
Alphabet-size	128
Distribution	Distinct (unsorted)

The two different workloads ((1) and (2)) were designed that each of these workloads fits into a leaf with a size of 16 MB while each workload uses a different fixed string-length and thus represents a different number of strings. We only used distinct unsorted workloads (i.e., no skew) because these workloads represent the worst case for all lookup operations (i.e., each string value/integer code of the leaf is encoded/decoded once for each workload). We used each of these workloads to load a set of leaves that hold the workload in a sorted way while for each workload we used different leaf sizes varying from 64kB to 16MB (resulting in a different set of leaves for each combination of workload and leaf size).

The first goal of this experiment is to show the costs (of the bulk loading and bulk lookup operations) caused by the different workloads (of approximately the same size in memory) using leaves with different sizes without the overhead of an encoding and decoding index on top (by simulating the pure lookup on the leaves). We measured the time as well as the L2 cache misses that resulted from executing the bulk loading of the leaves and executing the lookup operations for encoding as well as decoding.

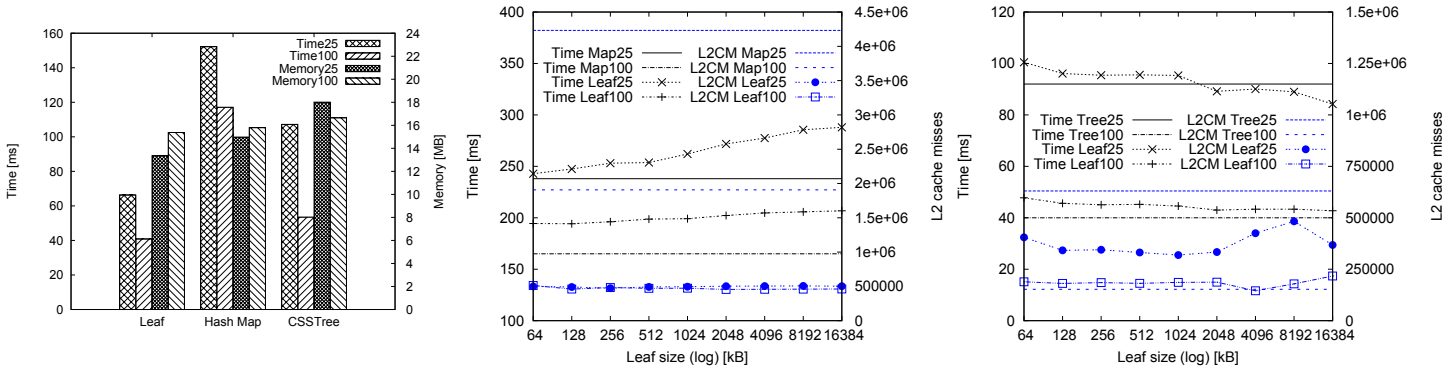
In order to load the leaves, we first sorted the workloads and then bulk loaded each leaf up to its maximal size (see Section 3). Afterwards, we generated the integer codes for these leaves. As shown in the table before, we use 8 bytes for the integer code in this experiment to show the memory consumption expected for encoding attributes with a large domain size. In order to measure the pure lookup performance of the leaf structures, we assigned each string value of the workloads mentioned above to the corresponding leaf using a buffer and then we looked up the code for each string in the individual buffers (i.e., we created a corresponding encoded workload for each leaf). Finally, we used the encoded workload to execute the lookup operation (in the same way as described before) on the leaves to decode the integer codes again.

A second goal of this experiment is to show a comparison of the 16MB leaf structure (which can be used for encoding as well as for decoding) and two cache-conscious read-optimized index structures using the workloads (1) and (2): for encoding the string values we compare the leaf structure to the compact-chain hash table [4] and for decoding integer codes we compare the leaf structure to the CSS-tree [16].

The main results of this experiment are that (1) the leaf structure is optimal when having a medium size ($\sim$ 512kB) and (2) the performance of our leaf structure is comparable to the read-optimized index structures mentioned above while using less memory:

- Figure 7 (a) shows the time and memory that is needed to bulk load the leaf structure (of 16MB size) compared to bulk loading the compact-chain hash table and the CSS-tree. As a result, we can see that bulk loading the leaf structure is faster for both workloads (i.e., string length 25 and 100) and uses less memory compared to the compact-chain hash table and the CSS-tree.
- Figure 7 (b) shows that the time and the L2 data cache misses (L2CM) for encoding the two workloads of this experiment (using the bulk loaded leaves) are increasing for large leaf sizes. Moreover, in terms of performance we can see that the 16MB leaf structure is comparable to the compact-chain hash map (Map) while offering sorted access (i.e., the leaf structure is a little slower).
- Finally, in Figure 7 (c) we can see that our leaf structure is optimized for decoding: While decoding $450k$ strings of length 100 using the smallest leaf size takes about 50ms, it takes 200ms to encode them. Moreover, the L2 cache misses for encoding are almost twice as high as for decoding these

[5]We used PAPI to measure the performance counters: http://icl.cs.utk.edu/papi/.



(a) Bulk Loading (b) Bulk Lookup (Encoding) (c) Bulk Lookup (Decoding)

Figure 7: Performance and memory overhead of the leaf structure

strings. Finally, compared to the CSS-Tree (Tree) our 16MB leaf structure is again only a little slower when used for decoding.

5.2 Efficiency of Encoding Indexes

This experiment shows the efficiency of our new encoding indexes compared to an implementation of the list-trie that decomposes the strings completely (i.e., it does not use our leaf structure). The idea of this experiment is to show the costs for encoding workloads that produce different update patterns on the indexes (which cause different costs). Therefore, we first bulk load a dictionary with 10m strings and afterwards encode another bulk of 10m strings that represents a certain update pattern (details later). All workloads consist of string values with a fixed length of 20 characters (while the other parameters to generate the workload are the same as in the experiment before). For the leaves we fixed the maximum leaf size to be 512kB (while the other leaf parameters are the same as in the experiment before). As update patterns we used:

- **No-updates:** The second bulk contains no new strings.
- **Interleaved** 10%**:** The second bulk contains 10% new string values where each 10th string in sort order is new.
- **Interleaved** 50%**:** The second bulk contains 50% new string values where every other string in sort order is new.
- **Interleaved** 100%**:** The second bulk contains 100% new string values whereas each new string is inserted between two string values of the first bulk.
- **Append:** The second bulk contains 100% new string values whereas all string values are inserted after the last string of the first bulk.

The interleaved patterns cause higher costs (independent from the index on top) because all leaves must be decompressed and compressed again to apply the inserts of the new strings. Figure 8 shows the time to first bulk load the dictionary with 10m strings and afterwards encode the other 10m strings for the different update patterns (when we either use the CS-Array-Trie or the CS-Prefix-Tree as the encoding index).

While the CS-Prefix-Tree is more expensive to bulk load (because we need to sort the complete input before bulk loading the index) than the CS-Array-Trie (which supports efficient top-down bulk loading), the workload which represents the *no-update* pattern is faster on the CS-Prefix-Tree because the CS-Prefix-Tree is (1) read-optimized and (2) not as high a the CS-Array-Trie. Furthermore, in general a workload that is more update-intensive (i.e., which has more new string values) has a better performance on the CS-Array-Trie. Comparing the costs of our CS-Array-Trie to the pure list-trie (that also uses buffers to speed-up the lookup of bulks), we can see that our index performs much better.

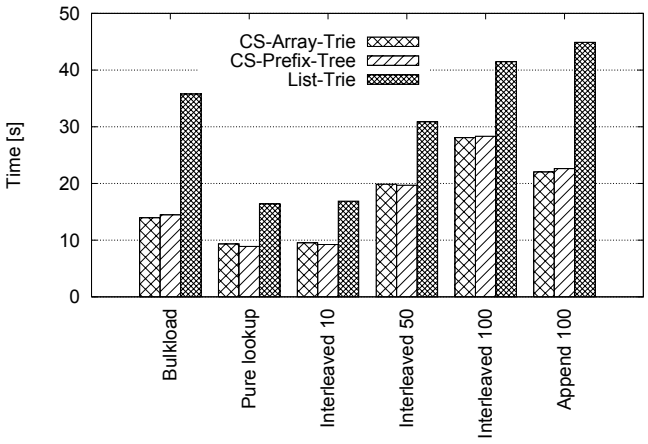


Figure 8: Lookup/update costs of encoding indexes

5.3 Scalability of the Dictionary

This experiment shows the performance of our data structures for unsorted workloads of different sizes with 1m up to 64m distinct strings (with a fixed string length of 10). The remaining configuration for data generation was the same as in Section 5.1. For the leaves, we used the same configuration as in the experiment before.

Figure 9 shows the time for encoding these workloads using different encoding indexes for the dictionary (starting with an empty dictionary). After encoding, we bulk loaded a decode index (i.e., a CSS-Tree) from the leaves of the encoding indexes and used the encoded workloads that we created before for decoding. For example, encoding 8m strings with the CS-Array-Trie takes 4.8s and the decoding takes 1.9s (while using one thread for both operations). We also scaled-up the number of threads (up to 16) to show the effects of parallelization on the CS-Array-Trie. For example, using 8 threads reduces time for encoding 8m strings from 4.8s to 2s. Figure 9 does not show the results for 16 threads because the performance slightly decreased compared to 8 threads due to the

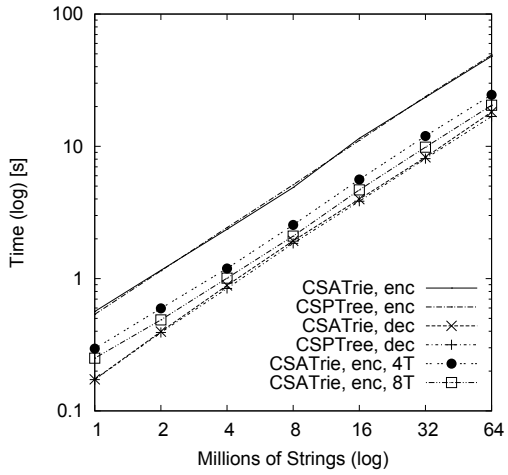


Figure 9: Scalability of the Dictionary

overhead for thread synchronization. Moreover, pinning individual threads to a single CPU to avoid thread migration did not improve the performance as well.

6. RELATED WORK

There has been recent work on dictionary compression in column-stores [1, 12, 24]. As mentioned before, this work focuses on small dictionaries for attributes with a stable domain size. Other work on dictionary compression of strings [2, 8] generates variable-length integer keys to support the order-preserving encoding of attributes with a variable domain size. However, none of this work addresses the problem of efficiently encoding a huge set of variable-length string values using fixed-length integer keys and how to efficiently support updates on such a dictionary without having to re-encode all existing data. Furthermore, to the best of our knowledge, there exists no work that exploits the idea of different indexes sharing the same leaves.

Moreover, there exists a lot of work on cache-conscious indexes and index compression for numerical data [16, 17, 10, 14, 11]. However, not much work is focused on cache-conscious indexing of string values. Compared to our indexes, [3] presents a cache-conscious trie that holds the string values in a hash table as its leaf structure and thus does not hold the strings in sort order. [5] presents a string index that holds the keys in sort order but is not designed to be cache-conscious.

An area that is also generally related to our work is the work on indexing in main memory databases [7, 9]. Finally, we applied the ideas of [22] for buffering index lookups to increase cache locality to our indexes and showed the benefits that buffering can also help for bulk loading these indexes.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we have shown an approach for efficiently using dictionaries for compressing a large set of variable-length string values with fixed-length integer keys in column stores. The dictionary supports updates (i.e., inserts of new string values) without changing codes for existing values in many cases. Furthermore, we have presented a new approach for indexing such a dictionary (called shared-leaves) that compresses the dictionary itself while offering efficient access paths for encoding and decoding. We also discussed a concrete leaf structure and two new cache-conscious indexes that can leverage the shared-leaves indexing approach.

As a part of our future work we plan to investigate different order preserving encoding schemes that are optimal for certain update patterns. Moreover, we want to analyze how to efficiently persist the dictionary data. Finally, the distribution of the dictionary over different nodes is very important for efficiently supporting the scale-up of the data in column stores.

8. REFERENCES

[1] D. J. Abadi, S. Madden, and M. Ferreira. Integrating Compression and Execution in Column-Oriented Database Systems. In *SIGMOD Conference*, pages 671–682, 2006.

[2] G. Antoshenkov, D. B. Lomet, and J. Murray. Order Preserving String Compression. In *ICDE*, pages 655–663, 1996.

[3] N. Askitis and R. Sinha. HAT-Trie: A Cache-Conscious Trie-Based Data Structure For Strings. In *ACSC*, pages 97–105, 2007.

[4] N. Askitis and J. Zobel. Cache-Conscious Collision Resolution in String Hash Tables. In *SPIRE*, pages 91–102, 2005.

[5] R. Bayer and K. Unterauer. Prefix B-Trees. *ACM Trans. Database Syst.*, 2(1):11–26, 1977.

[6] J. L. Bentley and R. Sedgewick. Fast Algorithms for Sorting and Searching Strings. In *SODA*, pages 360–369, 1997.

[7] P. Bohannon, P. McIlroy, and R. Rastogi. Main-Memory Index Structures with Fixed-Size Partial Keys. In *SIGMOD Conference*, pages 163–174, 2001.

[8] Z. Chen, J. Gehrke, and F. Korn. Query Optimization In Compressed Database Systems. In *SIGMOD Conference*, pages 271–282, 2001.

[9] H. Garcia-Molina and K. Salem. Main Memory Database Systems: An Overview. *IEEE Trans. Knowl. Data Eng.*, 4(6):509–516, 1992.

[10] J. Goldstein, R. Ramakrishnan, and U. Shaft. Compressing Relations and Indexes. In *ICDE*, pages 370–379, 1998.

[11] G. Graefe and P.-Å. Larson. B-Tree Indexes and CPU Caches. In *ICDE*, pages 349–358, 2001.

[12] S. Harizopoulos, V. Liang, and D. J. Abadi. Performance Tradeoffs in Read-Optimized Databases. In *VLDB*, pages 487–498, 2006.

[13] D. E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, 1973.

[14] D. B. Lomet. The Evolution of Effective B-tree: Page Organization and Techniques. *SIGMOD Record*, 30(3):64–69, 2001.

[15] D. R. Morrison. PATRICIA - Practical Algorithm To Retrieve Information Coded in Alphanumeric. *J. ACM*, 15(4):514–534, 1968.

[16] J. Rao and K. A. Ross. Cache Conscious Indexing for Decision-Support in Main Memory. In *VLDB*, pages 78–89, 1999.

[17] J. Rao and K. A. Ross. Making B^+-Trees Cache Conscious in Main Memory. In *SIGMOD Conference*, pages 475–486, 2000.

[18] R. Sinha, J. Zobel, and D. Ring. Cache-Efficient String Sorting Using Copying. *ACM Journal of Experimental Algorithms*, 11, 2006.

[19] 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*, pages 553–564, 2005.

[20] S. P. Vanderwiel and D. J. Lilja. Data Prefetch Mechanisms. *ACM Comput. Surv.*, 32(2):174–199, 2000.

[21] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes (2nd ed.): Compressing and Indexing Documents and Images*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.

[22] J. Zhou and K. A. Ross. Buffering Accesses to Memory-Resident Index Structures. In *VLDB*, pages 405–416, 2003.

[23] M. Zukowski, P. A. Boncz, N. Nes, and S. Héman. MonetDB/X100 - A DBMS In The CPU Cache. *IEEE Data Eng. Bull.*, 28(2):17–22, 2005.

[24] M. Zukowski, S. Héman, N. Nes, and P. A. Boncz. Super-Scalar RAM-CPU Cache Compression. In *ICDE*, page 59, 2006.