Adaptive String Dictionary Compression in In-Memory Column-Store Database Systems

Ingo Müller [*#], Cornelius Ratsch [#], Franz Faerber [#]

ingo.mueller@kit.edu, cornelius.ratsch@sap.com, franz.faerber@sap.com

[#] *SAP AG, Walldorf, Germany*

[*] *Karlsruhe Institute of Technology, Karlsruhe, Germany*

ABSTRACT

Domain encoding is a common technique to compress the columns of a column store and to accelerate many types of queries at the same time. It is based on the assumption that most columns contain a relatively small set of distinct values, in particular string columns. In this paper, we argue that domain encoding is not the end of the story. In real world systems, we observe that a substantial amount of the columns are of string types. Moreover, most of the memory space is consumed by only a small fraction of these columns.

To address this issue, we make three main contributions: First we survey several approaches and variants for *dictionary compression*, i. e., data structures that store the dictionary of domain encoding in a compressed way. As expected, there is a trade-off between size of the data structure and its access performance. This observation can be used to compress rarely accessed data more than frequently accessed data. Furthermore the question which approach has the best compression ratio for a certain column heavily depends on specific characteristics of its content. Consequently, as a second contribution, we present non-trivial sampling schemes for all our dictionary formats, enabling us to estimate their size for a given column. This way it is possible to identify compression schemes specialized for the content of a specific column.

Third, we draft how to fully automate the decision of the dictionary format. We sketch a compression manager that selects the most appropriate dictionary format based on column access and update patterns, characteristics of the underlying data, and costs for set-up and access of the different data structures. We evaluate an off-line prototype of a compression manager using a variation of the TPC-H benchmark [15]. The compression manager can configure the database system to be anywhere in a large range of the space / time trade-off with a fine granularity, providing significantly better trade-offs than any fixed dictionary format.

1. INTRODUCTION

In columns stores, *domain encoding* is a widely used technique for compression and query acceleration, especially for string columns [40, 1, 44, 29]. It consists of replacing the values of a column by a unique integer *value ID* and storing the mapping between val-

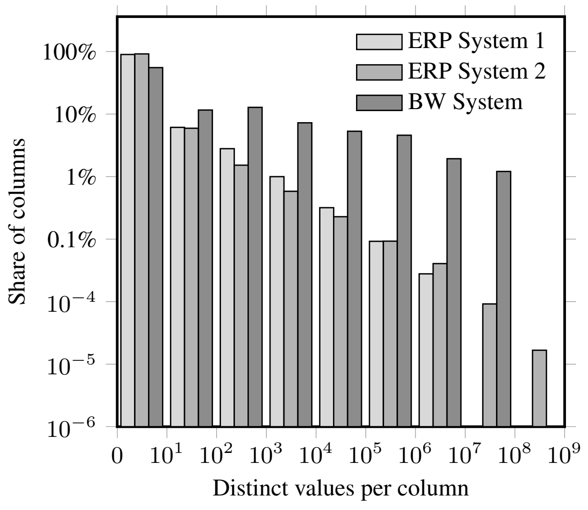

Figure 1: Distribution of the number of values per column.

ues and IDs in a separate data structure, the *dictionary*[1]. Under the assumption that many values occur multiple times in a column, domain encoding leads to a compression of the data, since every value has to be stored only once and the codes are typically much smaller than the original values. The resulting list of codes can be compressed further using integer compression schemes [1, 29]. Furthermore, most queries can be processed on the codes directly, which can be done faster thanks to the smaller and fixed size data type [1, 42, 29, 41].

In this paper, we argue that it is necessary to go beyond domain encoding. During the development of the SAP HANA database [18, 19], we gained several insights about the before-mentioned assumption and about other characteristics of the usage of string dictionaries in real-world business applications. To share these insights, we show some statistics of two enterprise resource planning (ERP) and one business intelligence warehouse (BW) system that the SAP HANA database department uses for testing: *ERP System 1* is an anonymized SAP ERP base system (the core set of tables that every SAP ERP system has), *ERP System 2* is a snapshot of a productive customer ERP system, *BW System* is a snapshot of a productive customer BW system. Our first insight concerns the distribution of the number of distinct values per string column, i. e., the distribution of dictionary sizes in terms of number of entries.

[1]Note that sometimes, domain encoding is also called *dictionary compression*. However when we talk about dictionary compression in this paper, we mean *compressing the dictionary*.

10.5441/002/edbt.2014.27

Figure 1 shows the distribution in all three systems: by far most dictionaries are very small and only few are very big. For every order of magnitude of smaller size, there is half an order of magnitude less dictionaries of that size. This means that the dictionary sizes roughly follow a Zipf distribution. The high number of small dictionaries matches the motivation of domain encoding.

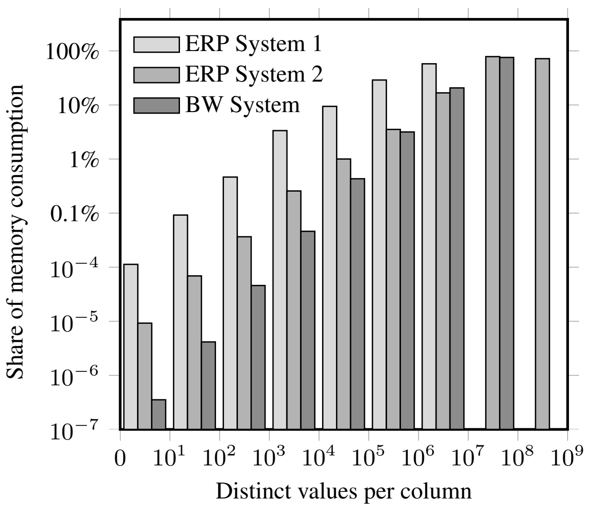

Figure 2: Distribution of memory consumption of all dictionaries depending on their number of entries.

Our other insights are more surprising and seem to be ignored by the prior work so far:

- The vast majority of columns in our systems are string columns, especially in the ERP systems (73%, 77% and 54% in above systems respectively).
- Large dictionaries have the largest share in memory consumption, even though there are not many of them. Figure 2 shows the memory consumption of the dictionaries of columns with different numbers of distinct values in our example systems. It is remarkable that in *ERP System 1*, 87% of the memory is consumed by the dictionaries with more than 10^5 entries, which only represents 0.1% of all dictionaries. The skew is even more extreme in *ERP System 2*, where the same memory share is consumed by just 0.01% of the dictionaries, while it is 3% in the *BW System* scenario.
- Many of the string columns are of a specific format, sometimes because they actually represent different types such as dates, but often just because they represent a specific domain such as hashes, UUIDs, URLs, product codes, etc. Some of these could be modelled by a more appropriate data type today, but many legacy applications still use strings.

We propose to address these issues by going beyond domain encoding and applying compression directly on the dictionaries. We concentrate on the dictionaries of the read-optimized store of the typical column-store architecture since they are both read-only and usually larger.

Dictionary compression naturally fits into the architecture of in-memory column-stores: First, memory is a valuable resource. Higher compression increases the amount of useful data a single system is capable to handle and makes more space for indices and query processing intermediates. Second, dictionaries reside in RAM at all times, so explicit dictionary compression is even more important for fast access to single entries. In contrast the page level compression commonly used in disk-based systems, where entire pages are transparently compressed with a general purpose compression scheme, would have a considerably higher cost [34]. Third, there are several points in the life cycle of columns in a typical column store where it makes sense to invest the time to compress the dictionary: When the write-optimized store is periodically merged into the read-optimized store or when aged data is moved into separate partitions, the dictionary needs to be reconstructed anyways. At this time the format can be changed without unnecessary reconstruction costs.

The contributions of this paper consists in proposing answers to open questions in how to integrate dictionary compression. For one, there is a lot of work on string dictionary compression from other research communities, so the first question is which compression format has attractive characteristics for in-memory column-stores. Consequently, in the first part of this paper, we implement and compare several interesting dictionary formats. As expected, different data structures provide different trade-offs between compression and access performance: higher compression usually means slower access. But there are also compression formats that can benefit from specific properties of certain columns, such as a fixed size or very restricted character set. The choice of the best dictionary format therefore heavily depends on the data. We contribute a sampling method to answer this question: for every dictionary variant we survey, we build a compression model that is capable to accurately predict its size by only looking at a small fraction of the data. With these estimated sizes and the performance characteristics of our survey, we have all the information at hand to manually pick a dictionary format for a given situation.

For easier set-up and maintenance, we also contribute first steps towards taking the decision of the dictionary format automatically. This decision is complex since it involves at least the two dimensions space and time. Our compression models provide knowledge about the size dimension, but this information should be complemented by information such as global memory pressure and the size of other data structures of the column. The time dimension is more complicated, since it is composed of the construction time of the data structure, its construction frequency, the access time of the forward and reverse look-up operations, and the respective access frequencies. To keep the decision reasonably cheap, we map all available local information onto one of the two axes, size and time, and let a global compression manager decide what kind of trade-off to choose depending on global information such as memory pressure or CPU usage. Since we only change the compression scheme when a dictionary is rebuilt anyways, the overhead of the automatic selection stays at a minimum.

To validate our approach, we evaluate an off-line prototype of our compression manager. Since it uses mostly local information, we believe that it is easily possible to apply the same principle for online decisions. We use workload and data of a modified TPC-H benchmark [15] and compare the performance and memory consumption of an automatic selection to the default dictionary format of the SAP HANA database. By applying an automatic selection strategy, we can reduce the over memory consumption to 60% while maintaining the performance of the benchmarks.

The rest of the paper is organized as follows. We first discuss related work in Section 2. Then we present a survey of dictionary formats in Section 3 and show how to estimate their size in Section 4. This forms the base for our automatic dictionary selection framework presented in Section 5, which is evaluated in Section 6. Section 7 concludes the paper.

2. RELATED WORK

Many column stores employ domain encoding as compression scheme [19, 40, 1, 44, 38]. Most of them also apply additional compression on the resulting codes. However to the best of our knowledge there is no work about dictionary compression in this context.

The idea to specialize compression schemes for a certain content type is as old as the idea of compression itself. In the context of row-store databases, Cormack [13] proposes a compression scheme that switches between a set of Huffman trees, each optimized for a different implicit field type, thus adapting compression to different classes of content. Domain encoding itself is also a general way to particularly compress a column with a certain domain. By selecting the dictionary format with the best compression rate for each column, we do something similar on another level.

There is a large body of work about automatic physical design for disk based database systems. All major traditional database vendors offer tools for assisting or replacing the database administrator (DBA) in many physical design decisions, including Microsoft [12, 11, 3, 8], IBM [5], and Oracle [17]. There are also publications from research [2]. Most of the above works concerns indexes and use variants and optimizations of an off-line *what-if* analysis but there exist also proposals for on-line solutions [37, 36, 8]. However, all of the above concentrate on secondary data structures such as indexes, while the dictionary as storage of the column content is a primary data structure. This slightly changes the problem: while the question about index selection is *whether or not* to create them, the question about dictionaries is *which format* to choose.

None of the above include compression as physical design decision. The first work on this topic was done by Idreos et al. [24], which was later improved by Kimura et al. [25]. For the first time, they take size and performance of several index compression schemes into account when selecting a set of indexes for a system. Like we do with our dictionaries, they describe strategies to estimate the size of their compressed indexes. Since these estimation methods are very specific to their index format, they cannot directly be applied to dictionaries.

In column-stores, there has been some work tackling the problem to select the right column format. Abadi et al. [1] give a decision tree for DBAs and Paradies et al. [32] propose how to automatically select the format of the column depending on compression properties. While the above work concentrates on the vector of references into the dictionary, we propose a similar approach for the dictionary itself.

3. SURVEY OF DICTIONARY FORMATS

In this section, we survey a selection of string dictionary formats from literature in the context of domain coding in column-stores. We implemented a large number of variants of them, which we evaluate and compare on a variety of data sets. This survey is the base of the subsequent section, where we show how to estimate the size of our dictionary variants using sampling strategies.

3.1 Basics

We start by defining the requirements for string dictionaries in the context of an in-memory column-store.

DEFINITION 1 (STRING DICTIONARY). *A* string dictionary *is a read-only data structure that implements at least the following two functions:*

- *Given a value ID id,* extract(id) *returns the corresponding string in the dictionary.*
- *Given a string str,* locate(str) *returns the unique value ID of str if str is in the dictionary or the value ID of the first string greater than str otherwise.*

In the context of an in-memory column-store, the following properties are desirable:

- An access to a string attribute value in a column-store database often corresponds to an *extract*-operation in a string dictionary. Thus, it is important that *extract* operations can be performed very fast.
- A typical use case for the *locate* operation is a **WHERE**-clause in an SQL statement that compares a string attribute against a string value. Here, only one *locate* operation is needed to execute the statement. Hence, the performance of the *locate* operation is not as critical as the *extract* performance.
- We concentrate on the static dictionaries of the read-optimized store, so no updates are needed. However, when the write-optimized store is merged into the read-optimized store, their dictionaries also have to be merged. Hence, the construction time should be minimized, too.

DEFINITION 2 (DICTIONARY COMPRESSION RATE). *Let str_1, $str_2, \dots, str_n$ be a set of strings stored in a string dictionary d and $|d|$ the memory size of the compressed dictionary. The* compression rate *of d, $comp(d)$, is defined as*

$$comp(d) = \frac{\sum_{i=1}^{n} |str_i|}{|d|}$$

3.2 Compression Schemes and Data Structures

There is a large corpus of text compression schemes in the literature that can be used to compress the strings in the dictionary. A very popular statistical technique is *Huffman encoding* [23]. It creates minimum redundancy codes based on the occurrence frequency of characters. Huffman codes are prefix codes, i. e., given the starting point of a sequence of Huffman codes, the beginning and the end of each code can be determined without additional information.

Hu-Tucker codes [26] are similar to Huffman codes, except for an additional restriction. Given an ordered sequence of characters $c_1 < c_2 < \cdots < c_n$, the corresponding Hu-Tucker codes $h(c_i)$ have the same binary order $h(c_1) < h(c_2) < \cdots < h(c_n)$. This leads to a slightly worse compression ratio but the binary order of two text strings compressed by Hu-Tucker encoding is the same as the initial order of the uncompressed strings. Since the elements in a dictionary usually are stored in ascending order, the order preserving property can be used to improve the search for a string and thus, the performance of the *locate* operation.

If the strings consist of only a small set of characters, a folklore compression technique is *Bit Compression*. Here, each character occurring in the string dictionary is represented by a constant number of bits $b \leq 8$, b being the smallest number of bits sufficient to represent every occurring character. If the codes representing the characters are ordered according to the order of the characters and the uncompressed strings are binary sorted, the initial sort order is preserved. Due to the fixed code length, Bit Compression can be implemented more efficiently than Huffman encoding, which especially increases the performance of the *extract* operation.

A different approach to compress text is to create a minimal grammar from which the text can be constructed [10]. The *Re-Pair* algorithm [27] is an approximate solution to this approach. It replaces frequent pairs of symbols by a new symbol. Pairs of symbols are replaced until there is no more pair occurring at least two times. The compression algorithm can be implemented to run in $\mathcal{O}(n)$ for an input text of length n. Despite the linear complexity, this compression algorithm is quite complex and hence, increases the construction time of a string dictionary.

Another folklore approach that operates on sequences of characters is the *N-Gram* compression technique. It collects frequent character sequences of fixed length n (n-grams) and replaces them by 12 bit codes. Since 12 bits are usually not enough to encode all sequences, only the $2^{12} - 256 = 3840$ most frequent n-grams are mapped to 12 bit codes. The remaining 256 codes are used to encode single characters. Due to the fixed code length, the *extract* operation can be implemented very efficiently. On the other hand, this algorithm does not preserve the sort order of the strings and hence, is not optimal for *locate* intensive dictionaries.

Literature also provides us with a variety of dictionary data structures. Front Coding [43] is a common technique to store sorted string dictionaries in a compressed way. The method makes use of the fact that consecutive text strings in a sorted string dictionary tend to share a common prefix. If two strings share a common prefix, it has to be be stored only once. The remaining suffixes can be further compressed by a statistical or dictionary-based compression algorithm.

In [7], Brisaboa et al. give an overview of string dictionaries. They mention *Hashing* [14] as a popular method to realize a basic dictionary. The strings are mapped to an index by a hash function. The evaluations show that the *locate* performance of this approach is quite good, yet both *extract* performance and compression rate are dominated by other approaches. Hence, hashing is not considered in this work.

There are other classes of pointer-based structures that could implement string dictionary functionality, like compressed text self-indexes [31, 20, 21], prefix trees [26, 22], suffix trees [39, 30], compressed suffix trees [35, 9], or directed acyclic word graphs [6, 16]. We do not consider these data structures since practical implementations of the many pointers are highly non-trivial, although there are interesting advances in recent literature (for example [4]).

3.3 Dictionary Implementations

After reviewing the data structures and compression schemes in the literature, we now show how to combine them to implement compressed string dictionaries. In particular we implemented the following string compression schemes:

- Huffman / Hu-Tucker Compression (*hu*): Hu-Tucker compression is used only if the order preserving property is needed.
- Bit Compression (*bc*)
- N-Gram Compression (*ng2* / *ng3*): Frequent 2-grams (*ng2*) or 3-grams (*ng3*) are replaced by 12 bit codes.
- Re-Pair Compression (*rp 12* / *rp 16*): Re-Pair Compression using either 12 bits (*rp 12*) or 16 bits (*rp 16*) to store a rule.

We apply these compression schemes to two main dictionary data structures:

- Array (*array*): One class of dictionary implementations is based on a simple consecutive array containing the string data. Pointers to each string in this array are maintained in a separate array.
- Front Coding (*fc block*): The strings of a dictionary are divided into blocks, which are encoded using Front Coding as explained above. The resulting blocks are then stored in a consecutive array. Pointers to each block are maintained in a separate array. The prefix length values of one block are stored in a header at the beginning of the block.

All of the string compression schemes (plus using uncompressed strings) can be applied to the strings of both data structures yielding a total of 14 variants. We denote them by concatenating the names of the data structure and the compression scheme, e. g., *array* for an array with uncompressed strings or *fc block hu* for a front coded dictionary with Huffman-encoded prefixes and suffixes.

Additionally, we implemented four special-purpose variants:

- Inline Front Coding (*fc inline*): In order to improve sequential access, we implement a Front Coding variant that stores the prefix lengths interleaved with the string suffixes.
- Front Coding with Difference to First (*fc block df*): In order to trade some space for speed, we implement another Front Coding variant that stores the suffixes differing from the first string of a block instead of the difference to the previous string. Hence decompression of a string essentially consists of two `memcpy`s.
- Fixed Length Array (*array fixed*): For very fast access to small dictionaries, we realized an *array* implementation that does not need pointers to the string data. For each string, the same amount of space is allocated in a consecutive array.
- Column-Wise Bit Compression (*column bc*): For columns with strings that all have the same length and a similar structure, we devised a special compression scheme. First we divide the dictionary into blocks. Then we vertically partition each block into character columns, which are then bit compressed.

Note that all dictionary variants presented above are order-preserving, i. e., they do not change the mapping between values and IDs. Futhermore they all allow access to a single tuple without decompressing other tuples or even the entire dictionary.

3.4 Evaluation of Dictionary Implementations

In the previous section, several dictionary implementations were introduced. To evaluate their performance, we test them on several data sets. We selected the following data sets to cover the most common cases. More details about them can be found in [33].

- Ascending decimal numbers of length 18, padded with zeros (*asc*),
- A list of English words[2] (*engl*),
- Tokens from Google Books[3], based on the 1-gram dataset version 20120701, consisting of all words occurring 3 or more times, with special characters removed (*1gram*),
- Salted SHA hashes of passwords, all starting with the same prefix describing the hash algorithm (*hash*),
- Material numbers extracted from a customer system (*mat*),
- Strings of length 10, containing random characters (*rand1*),

[2]http://code.google.com/p/shooting-stars/source/browse/trunk/Collaborative+Text+Editor/dictionary/fulldictionary00.txt (state: 2013/03/27)
[3]http://storage.googleapis.com/books/ngrams/books/datasetsv2.html (state: 2013/04/13)

- Strings of variable length, containing random characters (*rand2*),
- Source code lines, contained in a column of a customer system (*src*), and
- URL templates, extracted from a test system (*url*).

The implementation is realized in C++. All dictionary variants are integrated in a unified test framework. The tests are compiled with GCC 4.3.4 and run on a machine with 24GB of main memory and two Intel Xeon X5550, each with 4 cores running at 2.67GHz. The operating system is Ubuntu 10.04 x86_64.

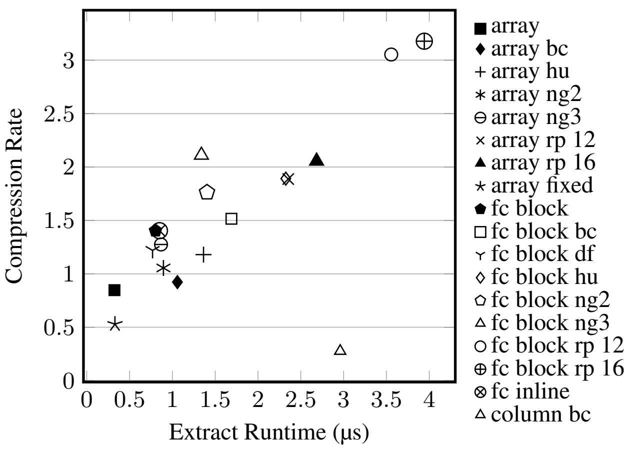

Figure 3: Trade-off between compression rate and *extract* runtimes for all dictionary variants on the *src* data set.

Figure 3 shows the trade-offs between compression rate and *extract* runtime of our 18 dictionary implementations on the *src* data set. We will show later how these results differ for the other data sets. The *src* set contains a lot of redundancy, so almost all compression schemes work as intended. Consequently most of the implementations are close to a pareto optimal curve, yielding the expected trade-off between "fast but big" and "small but slow". Generally speaking the Front-Coding variants are smaller and considerably slower than their array equivalents with the same string compression scheme. Similarly the compression schemes range from very fast and big (uncompressed), over slightly slower but considerably smaller (*ng2*, *ng3*, *bc*, *hu*) to maximal compression with considerably worse performance (*rp 12*, *rp 16*). Variants with *ng2*, *ng3*, and *bc* obviously incur computing overhead compared to uncompressed schemes, but since their fixed size code words can be extracted with more CPU friendly code, they are faster than *hu* with its variable size codes and much faster than *rp 12*'s and *rp 16*'s grammar evaluation. As expected *fc block df* is just a bit faster but larger than *fc block*, and *fc inline* is just a bit slower on the random extracts of this test. Still all three variants of Front Coding have very similar performance and provide a very interesting trade-off between space and speed. *array fixed* and, to quite an extreme extent, *column bc* have very unattractive properties on this data set, since they are factors larger than the data itself (about 2 and 3.5 times respectively) without improving extract performance. However, their large size is not surprising, since both variants are optimized for fixed length columns, which is not at all given in the *src* data set.

While the *qualitative* message of Figure 3 is representative, the *quantitative* picture is quite different on other data sets: For each

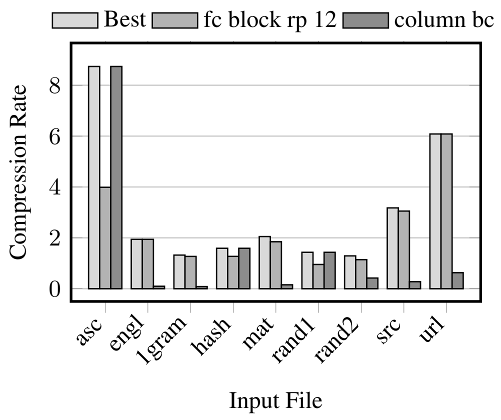

Figure 4: Compression rate of smallest dictionary implementations on different data sets

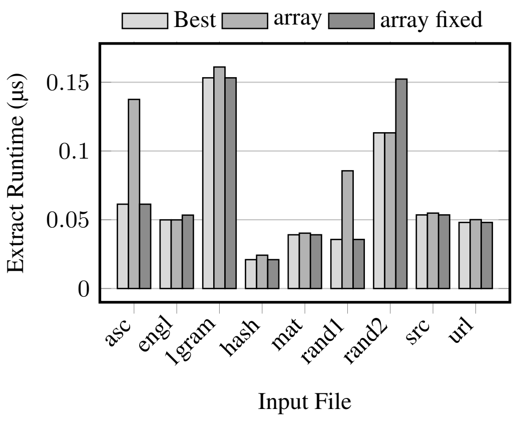

Figure 5: Extract runtime of the fastest dictionary implementations on different data sets

data set, there are "fast but big" and "small but slow" variants and everything in between. However the variants actually lying on the pareto optimal curve are considerably different. For example *ng2* and *ng3* have an interesting trade-off between compression rate and speed when the entire text only consists of the about $2^{12} - 256$ n-grams that have proper 12-bit codes. However if the text contains many more n-grams, only backup codes will be used, resulting in bad, possibly negative compression rates and slower extract runtimes. The bad properties of *array fixed* and *column bc* in Figure 3 are other examples.

Figure 4 and Figure 5 show how the best variants for the two extremes of the trade-off, highest possible compression rate and fastest possible extract time, vary depending on the data set. The plots show two generally attractive variants and, for each data set, the best value achieved by any variant (*"Best"*). Figure 4 shows that the best compression rate is often achieved by *fc block rp 12*, but in three cases *column bc* is considerably better — three data sets with constant string lengths. For all other data sets, *column bc* performs very badly: the compressed dictionary is larger than uncompressed data. In the case of the completely random data of the *rand1* data set, *fc block rp 12* also has a compression rate below 1. Figure 5 shows a similar picture for the fastest variants: on most data sets, the uncompressed variants *array* and *array fixed* achieve the same overall fastest performance, but in some cases *array fixed* is considerably better (again data sets with constant string lengths), whereas the picture is inverted in other cases.

So far we have only seen the trade-off between size and *extract* times. Matters are further complicated if we also take *locate* and construction times into account. Due to space constraints, we cannot present our according findings in this paper, but a more extensive evaluation of the dictionary variants can be found in [33].

3.5 Summary

As our evaluation has shown, there are two challenges in picking the “right” dictionary variant: First, their characteristics concerning size and access performance heavily depend on the data they contain. To solve this challenge, we present advanced sampling techniques in the next section. Second, there are various trade-offs to pick from and the “best” trade-off depends on the usage pattern of each particular dictionary instance. In the subsequent Section 5, we sketch how our compression manager selects a trade-off taking the state of the database system into account.

4. PREDICTING RUNTIME AND COMPRESSION RATE

In this section, we describe a prediction framework that models the properties of the different dictionary implementations for a given string column. The predictions of the framework will serve as the basis for an automatic dictionary selection in the subsequent section.

4.1 Runtime

We model the runtimes of each of the three methods of a dictionary as a constant *time per access* (for *extract* and *locate*) or *time per tuple* (for *construction*). We determine these constants with microbenchmarks as the average of the respective runtimes of the methods on the datasets of Section 3.4. New dictionary variants can be added simply by determining their constants with the same benchmarks.

Note that this is a rather simplistic model. However in experiments not presented here, we investigated the accuracy of more sophisticated models for the runtimes and did not find more robust runtime predictions. For details we refer to [33]. We conclude that constant runtimes are a good approximation and leave more precise modelling as an open question for further research.

4.2 Compression Rate

For estimating the dictionary sizes, we propose more sophisticated models than for the runtime. In particular they preserve the properties of the compression better than naively compressing a sample of the strings and extrapolating the resulting size to the entire data set, while being cheaper to calculate.

4.2.1 Compression Models

In the following we give formulas for all variants presented in Section 3.3, breaking down the size of a dictionary to properties of the data set that are either known beforehand or can be sampled. Table 1 describes the properties used by the models. The properties that are later sampled are written in italic. As input we assume a (sorted) dictionary, which in our case is the output of the domain encoding of the corresponding string column. If not otherwise mentioned, all sizes are given in bytes.

First we model the effect of the dictionary class on its size. The ***array*** **class** dictionaries consist of the (possibly compressed) data of the entire strings and of pointers to the beginning of the dictionary

Property	Description / Sampling Method
# strings	Number of strings in the dictionary
|pointer|	Platform dependent, usually 4 or 8 byte
# blocks	Number of blocks (calculated from *# strings*)
|block header|	Block header size (implementation dependent)
# raw chars	Sum of all string lengths
|data|	Size of the compressed strings of a dictionary format
|*raw data*|	Size of the uncompressed strings of a dictionary format
# chars	Number of distinct characters in a sample of strings / suffixes
entropy₀	0-order entropy of the characters on a sample of strings / suffixes
coverage	Calculated with # covered n-grams and |*raw data*| on sample of strings / suffixes
compr rate	Compression rate of *Re-Pair* on a sample of strings / suffixes
|*max string*|	Maximum of string lengths of sample of strings / suffixes
avg block size	Average block size of sample of blocks

Table 1: Properties of dictionaries used for the compression models.

entries. Their size can be calculated as

$$\text{size} = |\text{data}| + \#\,\text{strings} \cdot |\text{pointer}|$$

The ***Front Coding*** **class** dictionaries (including *fc inline* and *fc block df*) only need a pointer per block, but also a block header. Furthermore, the front coding reduces the number of characters per block. We calculate their size as

$$\text{size} = |\text{data}| + \#\,\text{blocks} \cdot (|\text{pointer}| + |\text{block header}|)$$

Now we model the effect of the string compression schemes applied to the two dictionary classes. The intent is to first use the formulas given above to calculate the overhead needed by the class itself as well as which parts of the strings are stored and then to calculate independently the space needed by these strings after string compression. We denote the size of the original string parts *raw data* and the size of their compressed form |data|.

For **uncompressed** strings the size of the data is simply

$$\text{data} = |\textit{raw data}|$$

For ***Bit Compression***, every character is replaced by a new code of size $\lceil \log_2 \#\, chars \rceil$ bits if there are $\#\, chars$ distinct characters in the dictionary. The data size can thus be modelled as

$$\text{data} = |\textit{raw data}| \cdot {}^1\!/\!_8 \cdot \lceil \log_2 \cdot \#\textit{chars} \rceil$$

Huffman / Hu-Tucker **compression** approximates a coding where each character is assigned a code with a number of bits equal to the characters order-0 entropy. We assume that the difference due to rounding is not too big in practice and model the size of Huffman encoded text as

$$\text{data} = |\textit{raw data}| \cdot \textit{entropy}_0$$

We model ***n-Gram*** **compression** with the ratio of n-grams in the text covered by the $2^{12} - 256$ proper (i. e., non-backup) codes. We call this ratio *coverage* and calculate it as follows: First we count the occurrences of all n-grams in the dictionary, select the 2^{12} −

256 most frequent ones, and calculate the sum of their occurrences denoted *# covered n-grams*. *coverage* can then be calculated as

$$\text{coverage} = \#\,\text{covered n-grams}/(|\textit{raw data}| - n + 1)$$

As the compression replaces either n characters (for a covered n-gram) or a single character (as backup) by a single 12 bit code, we calculate the size of n-gram compressed data as

$$\text{data} = \frac{12}{8} \cdot (\frac{1}{n} \cdot \textit{coverage} + (1 - \textit{coverage})) \cdot |\textit{raw data}|$$

The complex, grammar-based approach of ***Re-Pair*** **compression** makes it difficult to be modelled precisely. We fall back to assuming a uniform compression rate for the entire dictionary and calculate the size of the compressed data as

$$\text{data} = \frac{|\textit{raw data}|}{\textit{compr rate}}$$

Finally, we model the special purpose variants in isolation. In the ***array fixed*** variant, all strings take the same space, i. e., the space of the longest string. We calculate the size of a dictionary as

$$\text{size} = \#\,\text{strings} \cdot |\textit{longest string}|$$

For ***column bc***, we reduce the size of a dictionary to the average size of its blocks:

$$\text{size} = \#\,\text{blocks} \cdot \textit{average block size}$$

The above formulas reduce the size of each dictionary variant to properties that are either known a priori or can be sampled (see Table 1). For sampling the properties, we draw samples uniformly at random with a granularity of dictionary entries or blocks as described in the tables. The formulas above are then instantiated with the sampled properties in order to produce an estimation of the size of each dictionary variant for a given data set. A new dictionary variant can be added to the framework by defining an according model and possibly adding new properties that need to be sampled.

Note that for ease of presentation, some of these formulas are slightly simplified. In order to make the predictions more precise, they can be extended for example by corrections for cut-off due to half-used machine words. For details we refer again to [33].

4.2.2 Evaluation

We now evaluate the accuracy of the predictions of the dictionary sizes by our framework. In particular we empirically answer the question of how much sampling is needed to get reasonably good estimations. To that aim we compare the estimated sizes with the actual memory consumption and calculate the relative error of the predictions as

$$\textit{err} = \left| \frac{\text{Real Size} - \text{Predicted Size}}{\text{Real Size}} \right|$$

We determine this prediction error for all our dictionary variants and for all data sets of Section 3.4 for a variety of sample sizes. The result is shown in Figure 6. For a particular sample size, the distribution of errors for all combinations of dictionary variants and data sets is summarized as a box plot. In this plot, the line inside of each box indicates the median, the box itself indicates the first and third quartiles, and the whiskers indicate the closest datum closer than 1.5 times the inter quartile range away from the quartiles, while the small crosses indicate outliers[4].

The first observation concerns the sample size 100%, where the properties from the previous section are determined precisely and

[4]This is the default configuration of box plots in R and PGFPlots.

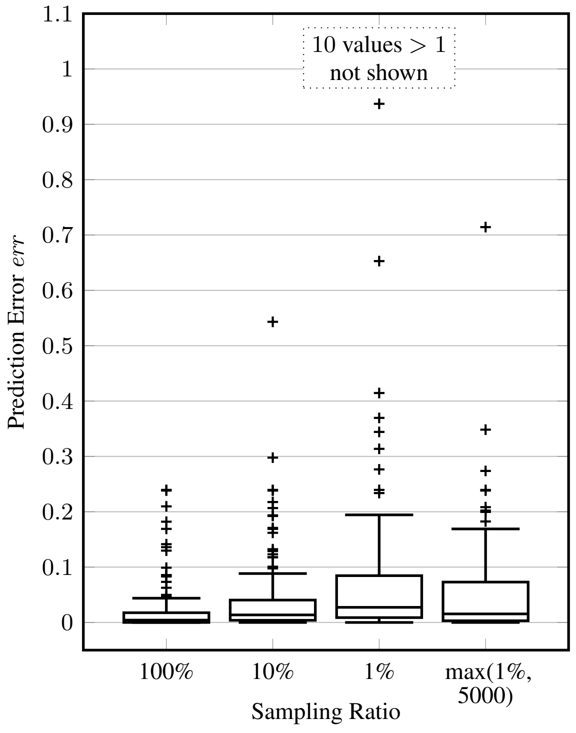

Figure 6: Prediction error of the compression models

the deviance of the prediction from the actual size is due to a simplification by the model and not due to sampling. The plot shows that more than 75% of all the predictions are less than 2% off the correct value and all values except some outliers are off by less than 5%.

We now discuss how sampling affects the prediction error. For a sample size of 10%, the prediction errors increase with 75% of all estimations still having an error below 4%. For a sample size of 1%, the estimations become significantly worse. 25% of all estimations now deviate more than 10% from the correct value. While this may still be acceptable, there is now a high number of outliers and some of them have extreme errors of $100 - 500\%$ (outside the plotted range). However, these extreme mispredictions stem mostly from very small dictionaries, where 1% sample size represents too few entries. We fix this corner case by taking at least 5000 string into the sample. The last column of the plot shows that we now get an error of less than 8% in more than 75% of the predictions and virtually all predictions are less than 20% off. This seems like a good trade-off between sampling costs and accuracy and is therefore used throughout the rest of the paper.

4.3 Summary

The prediction framework presented in this section enables us to estimate the *extract*, *locate*, and *construct* runtimes as well as the compression rate of the different dictionary implementations. For a given string column, this gives us the possibility to choose the trade-off between speed and space of its dictionary similar to what Figure 3 shows in Section 3.4. Using sampling, we are able to greatly reduce the estimation overhead while the precision remains sufficiently good. Until here our work can be used in a tuning advisor to assist the database administrator in taking the decision of the format of the most important dictionaries manually. In the following section, we will go a step further and draft a compression manager that selects an appropriate dictionary variant completely automatically.

5. AUTOMATIC SELECTION

In this section, we present the different components of our compression manager, which decides for every string dictionary what implementation should be used. First we motivate on an intuitive level how the compression manager should make its decisions. We then show how we automate these decisions, before giving details about the different steps involved in the process.

5.1 Problem Statement and Solution Overview

There is a large number of factors that may influence the fact which dictionary variant is the optimal one for a given string column. Our aim is to monitor or collect these factors and translate them into an automatic decision. Intuitively, the following factors should be taken into account:

- The access pattern of a column is an important factor. Columns that are accessed very frequently should use a fast dictionary implementation, while mostly passive data should be compressed more heavily and intermediate cases should have something in-between. If either *locate* or *extract* dominates, we may want to choose an implementation that optimizes the relevant method over the other.
- Similarly, update-intensive columns need a string dictionary supporting fast construction times.
- As discussed in detail in Section 3.4, the properties of the implementations, in particular their compression rate, depend on the content of the column. Furthermore these properties may change depending on the hardware.
- The size of the dictionary should also be taken into account. As we have seen in Figure 2, a very small fraction of the dictionaries dominates their overall memory consumption. Compressing these huge dictionaries more heavily is therefore beneficial for the overall system. However, the memory consumption of these large dictionaries should be put into relation with the memory consumption of the rest of the column. A dictionary roughly as large as its column vector should be compressed more than a dictionary whose size is dominated by its column vector.

Figure 7 summarizes these factors and shows how they are taken into account for the selection of the dictionary implementation: To keep the decision reasonably cheap, we reduce the factors local to the column to the two dimensions space and time. This way we have a variety of space / time trade-offs to choose from for every column provided by the different dictionary variants. The remaining factors are reduced to a single, global trade-off parameter, which is kept up-to-date by the compression manager.

This decouples the local decisions from the global factors: The compression manager monitors the global factors and asynchronously updates the trade-off parameter when necessary. Every time a dictionary is reconstructed, this parameter is taken into account by the selection strategy in order to select an optimal space / time trade-off.

Furthermore the decisions of the dictionary format of different columns are also decoupled. This is important because they are taken at different points in time: Depending on the usage of a table, the write-optimized stored of the table runs full sooner or later and needs to be merged into the read-optimized store. This entails a reconstruction of the dictionaries of the concerned table. The decision of their format is then simply based on the current value of the global trade-off parameter.

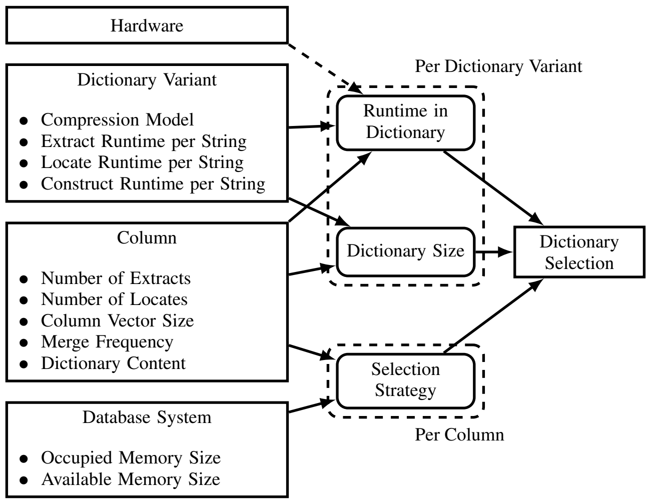

Figure 7: Overview of the information taken into account by the compression manager for selecting the dictionary implementation.

5.2 Reduction of Dimensionality

We now show how we can reduce all of the above factors to either the time dimension or the space dimension. In the space dimension, we have the content of the dictionary and the size of the rest of the column, i. e., the size of the (compressed) column vector. To reduce this to a single value, we simply view the column as a single unit and take the aggregated size of column vector and dictionary.

Formally this can be expressed as follows: Let D be the set of dictionary implementations, c the column in question, and $\texttt{dict\_size}(d, c)$ the dictionary size of c using implementation $d \in D$. Then the size of c using d can be calculated as

$$\texttt{size}(d, c) = \texttt{dict\_size}(d, c) + \texttt{columnvector\_size}(c)$$

We suppose that $\texttt{columnvector\_size}(c)$ is known, since the column vector is a product of domain encoding just like the dictionary. $\texttt{dict\_size}(d, c)$ can be estimated using the prediction models described in Section 4.2.

With this definition, the size of the dictionary is put into relation with the size of the entire column. If the dictionary is small compared to the table, the size will be dominated by the size of the column vector. Consequently differences in the compression rate of different dictionary variants will only have a small influence on the total column size.

In the time dimension, we have the runtime of the three methods *extract*, *locate*, and *construct*. Let us assume that during the lifetime of a single dictionary instance *extract* and *locate* are called $\#\,\texttt{extracts}$ and $\#\,\texttt{locates}$ times respectively and that the dictionary contains $\#\,\texttt{strings}$ entries. Then we can calculate the total runtime $\texttt{time}(d)$ spent in this dictionary instance:

$$\begin{aligned}\texttt{time}(d) = &\#\,\texttt{extracts} \cdot \texttt{time}_e(d) + \\ &\#\,\texttt{locates} \cdot \texttt{time}_l(d) + \\ &\#\,\texttt{strings} \cdot \texttt{time}_c(d)\end{aligned}$$

The runtimes of the methods, $\texttt{time}_*(d)$, are constants determined at installation time by microbenchmarks as described in Section 4.1. We assume that at the point in time when a new dictionary is created, the number of calls to the three methods is known or an approximation can be deduced from the usage statistics of the corresponding column or table.

As the final value for the time dimension, we normalize this runtime over the lifetime of the dictionary, $\texttt{lifetime}(d)$. The lifetime of a dictionary corresponds to the time between two periodic merges of the read-optimized store into the write-optimized store of the column. Formally this translates to

$$\texttt{rel\_time}(d) = \frac{\texttt{time}(d)}{\texttt{lifetime}(d)}$$

With this definition, the construction time of a dictionary is amortized over the lifetime of the object and long living dictionaries can afford more expensive construction time than those that are reconstructed frequently.

5.3 Determining the Global Trade-Off Parameter

We now describe how a global value for Δc is determined, the parameter used to choose a space / time trade-off of the dictionary format. It is periodically updated by the compression manager, which monitors the system state, in particular the memory consumption. If the memory consumption is above a certain threshold, the memory manager decreases the value of Δc. Dictionaries created after this point in time will use implementations favoring small size a bit more over access speed than before. If on the contrary the memory consumption is below a certain threshold, the memory manager increases the value of Δc. New dictionaries will now use faster implementations instead.

One can describe this process as a closed loop feedback control system. The reference input is the desired amount of free memory. The measured output is the currently available free memory. In order to avoid over-shooting, this value is smoothed before feedback. Using the difference between the (smoothed) measured and the desired amount of free memory, the compression manager can then decide to adjust Δc. Figure 8 illustrates the feedback loop.

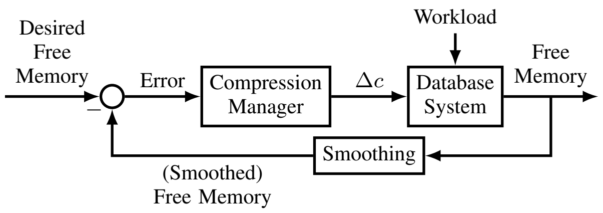

Figure 8: Feedback loop to configure Δc.

Now that we have defined the global trade-off parameter Δc, we can select a dictionary variant. The following section will introduce several possible selection strategies.

5.4 Trade-Off Selection Strategy

In this section we incrementally develop a strategy to use the global trade-off parameter to locally select a space / time trade-off provided by the different dictionary implementations for a given column. The main idea is similar to the approach of Lemke et al. [28]: To select a space / time trade-off for their data structure, they use the fastest variant that is not larger than the smallest variant plus a fraction of Δc. While they have a fixed value for Δc, we let the compression manager control this value. Furthermore, they do not take access frequency into account.

For a first, illustrative approach, we apply this principle in a naive manner to the trade-off introduced above. Let D, c, and $\texttt{size}(d,c)$ be defined as above. Then the size of the smallest dictionary variant $\texttt{size}_{\text{min}}$ can be formalized as

$$\texttt{size}_{\text{min}} = \min_{d \in D}(\texttt{size}(d,c))$$

and the set $\tilde{D}_{\text{const}}$ of variants not larger than $\texttt{size}_{\text{min}}$ plus a fraction of Δc can be formalized as

$$\tilde{D}_{\text{const}} = \{d \in D \mid \texttt{size}(d,c) \leq (1+\Delta c) \cdot \texttt{size}_{\text{min}}\}$$

We can now formally introduce $\texttt{tradeoff\_strategy}_{\text{const}}$, which selects the smallest variant from $\tilde{D}_{\text{const}}$, as

$$\texttt{tradeoff\_strategy}_{\text{const}}(c) = \arg\min_{d \in \tilde{D}_{\text{const}}} (\texttt{rel\_time}(d))$$

To illustrate $\texttt{tradeoff\_strategy}_{\text{const}}$, Figure 9 shows a possible dictionary performance distribution. It was generated using the *src* input file and arbitrarily chosen extract and *locate* frequencies, as well as an arbitrarily chosen merge interval. Absolute size values are not relevant for illustration purposes and are therefore omitted. Each point represents a dictionary variant, the diamond corresponds to the dictionary variant with the smallest size. The parameter Δc can be seen as a dividing line, separating "allowed" dictionary variants (*included*) from "too big" ones (*excluded*). All points below this line correspond to dictionary variants with a compression rate high enough to fulfill the size requirement defined by Δc. From these variants, we choose the one with the lowest runtime, i. e., the leftmost one, plotted as a black dot.

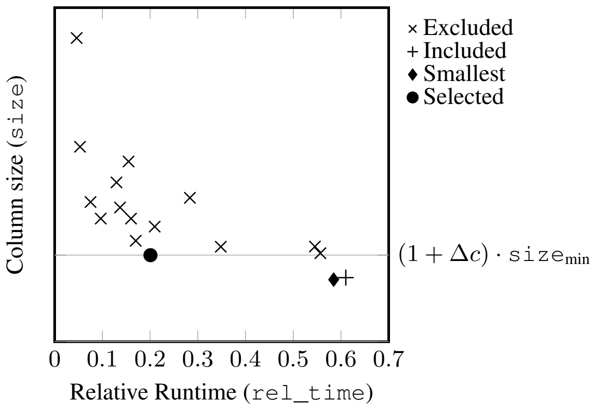

Figure 9: Possible distribution of dictionary performances.

While this naive approach may look convincing on first sight, it does not fulfill important design goals set forth at the beginning of this section. Note that $\tilde{D}_{\text{const}}$ only depends on the sizes of the dictionaries and not on the access frequency of the column. In terms of the plot, changing frequencies only scale the plot on the x-axis and (through a changing mix of the methods *extract*, *locate*, and *construct*) potentially the relative order of the dictionary variants. $\tilde{D}_{\text{const}}$ however is invariant to changes on the x-axis, so the allowed size of variants is not increased by higher access rates.

To address this issue, we extend the principle from Lemke et al. to take access frequencies into account. We keep the general idea to define a subset of the dictionary variants and then to select the fastest one of this subset. But we generalize the approach to use a subset defined by an arbitrary dividing function. For any function f, we define $\tilde{D}_f$ as

$$\tilde{D}_f = \{d \in D \mid \texttt{size}(d, c) \leq f(\texttt{rel\_time}(d))\}$$

and we define $\texttt{tradeoff\_strategy}_f(c)$, the corresponding selection strategy, as

$$\texttt{tradeoff\_strategy}_f(c) = \arg\min_{d \in \tilde{D}_f} (\texttt{rel\_time}(d))$$

The naive approach with a constant offset naturally fits into this definition with $f(t) = f_{\text{const}}(t) = (1+\Delta c)\cdot\texttt{size}_{\text{min}}$. Furthermore we propose two other strategies that take the access frequency into account. Both of them define f in terms of the smallest dictionary variant d_{min}.

- $\texttt{tradeoff\_strategy}_{\text{rel}}$ shifts the dividing line up by a multiple of $\texttt{rel\_time}(d_{\text{min}})$, the runtime of the smallest dictionary variant. It is defined with $f = f_{\text{rel}}$ with

  $$f_{\text{rel}}(t) = (1+\Delta c\cdot(1+\texttt{rel\_time}(d_{\text{min}})\cdot\alpha))\cdot\texttt{size}(d_{\text{min}}, c)$$

  where α is a configuration parameter. Note that $f_{\text{rel}} = f_{\text{const}}$ for $\alpha = 0$. Since a higher access frequency of the dictionary increases $\texttt{size}(d_{\text{min}}, c)$, the size threshold for dictionaries in $\tilde{D}$ is also increased.

- $\texttt{tradeoff\_strategy}_{\text{tilt}}$ tilts the dividing line in favor of faster but bigger variants than d_{min}. In order insure that we include *more* dictionaries than with f_{const}, we define f_{tilt} such that it crosses f_{const} at the x-value of d_{min}, i. e., we define f_{tilt} as

  $$f(t) = -\alpha \cdot \texttt{rel\_time}(d_{\text{min}}) \cdot t + b$$

  such that b is defined by the equation $f(\texttt{rel\_time}(d_{\text{min}})) = (1+\Delta c)\cdot\texttt{size}_{\text{min}}$. Again α is a configuration parameter that specifies the slope of f.

The last open question is how to choose the parameter α used in both above functions. It adjusts how much the new dividing function differs from the dividing line defined by f_{const}. There are at least the following two possibilities:

- Experiment with different values for α and try to find a good trade-off for dictionaries with high access frequencies, or
- Add another constraint to the function $f(t)$ that defines α.

We opt for the latter using the following intuitive constraint: if the runtime of the smallest dictionary variant is greater than or equal to 100% of the available time until the next merge operation, the fastest dictionary variant should be chosen. Formally this translates into setting $\texttt{rel\_time}(d_{\text{min}}) = 1$ and solving the equation $f(\texttt{rel\_time}(d_{\text{speed}})) = \texttt{size}(d_{\text{speed}})$ for α, where d_{speed} is the fastest dictionary variant. Note that for $\texttt{tradeoff\_strategy}_{\text{rel}}$, this constraint cannot be applied for $\Delta c = 0$ since in this case f_{rel} is a constant function (equal to the old dividing line).

5.5 Summary

As presented in this section, the compression manager takes the decision of the dictionary format of a column in two steps: On a global level it maintains a trade-off parameter Δc indicating the need of the overall system to trade space for speed. On a local level upon dictionary reconstruction, it maps all characteristics of a column to the dimensions space and time and uses Δc to select a trade-off between the two dimensions. The next section shows how this works in practice.

6. EVALUATION

In this section, we evaluate an off-line prototype of the compression manager, which is implemented in the following way: The characteristics about the lifetime, the number of calls to *extract* and *locate*, and size estimations of every dictionary instance are determined while running a representative workload on an instrumented version of the SAP HANA database. This information is then be combined to produce a configuration of the system for a given Δc, i. e., a mapping of columns to dictionary formats, using the $\texttt{tradeoff\_strategy}_{\text{tilt}}$ of Section 5. When the system is restarted the next time, the according formats are used upon construction of each dictionary. We believe that the same approach can be used for an online decision.

6.1 Test Setup

We base our experiments on a slightly modified version of the TPC-H benchmark [15]. In particular we modify the schema in the following way: We change the type of all key columns, i. e., all columns whose names end with `KEY` such as `C_CUSTKEY`, from `INT` to `VARCHAR(10)`. This reflects our observations from Section 1, suggesting that real-world business applications use strings for a large fraction of columns including key columns. Since the data of the TPC-H benchmark is synthetic, the achieved compression rates need a careful interpretation, but it allows us to show the most important point of our work, the adaptive selection of the dictionary formats. We use scale factor 1 in the experiments presented here, but punctual comparisons with other scale factors did not reveal significant differences.

For the experiments in this section, we use the same hardware as in Section 3.4.

6.2 Experimental Results

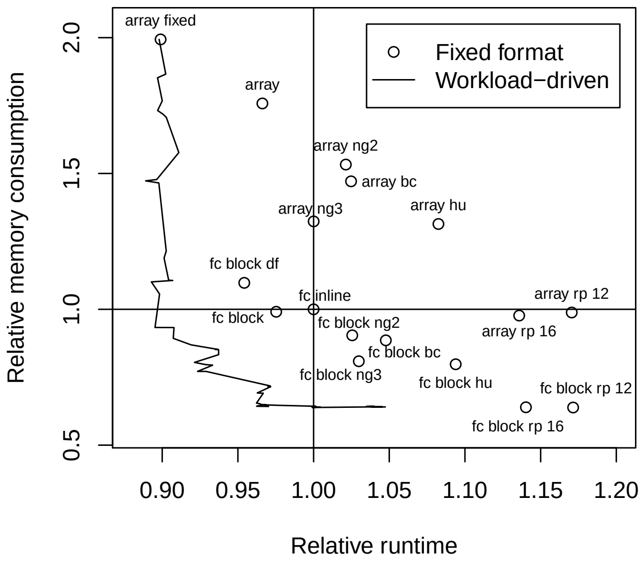

Figure 10: Space / time trade-off of different dictionary format selection strategies on queries of the TPC-H benchmark.

We now study the effect of the dictionary configuration on the runtime of the TPC-H queries and the size of the dictionaries. Figure 10 shows our results. Every point on the plot represents a space / time trade-off of one configuration: The space dimension is the total memory consumption of the TPC-H tables, including column vector and dictionary. The time dimension is the sum of the medians of 100 executions of each of the 22 queries. Both dimensions are normalized against *fc inline*. Each such point is produced by configuring and restarting the system as described above, then running the workload and measuring time and space consumption.

The results of configurations with a fixed format for all dictionaries correspond to the results of the microbenchmarks in Section 3.4: We have "fast but big" formats like *array fixed* and *array*, balanced formats like *fc block* and *fc inline*, and "small but slow" formats like *fc block rp 12/16*. They seem to form a pareto-optimal curve, which dominates some "big and slow" formats but does not reach the "fast and small" region of the plot. *column bc* is outside of the plot. There is a difference in the end-to-end runtime of roughly 25% between the fastest and the slowest format and difference in the total memory consumption of factor 3.5, confirming the importance of the dictionary format.

The same plot also shows workload-driven configurations produced by our compression manager for a logarithmic range between 10^{-3} and 10 as values of Δc. The workload we use to trace the lifetime and the calls to *extract* and *locate* consists of 100 repetitions of all TPC-H queries, which minimizes the influence of the construction time.

The plot shows that all workload-driven configurations are closer to the "fast and small" region than any single format. For every fixed-format configuration, there is a workload-driven configuration that is smaller while maintaining the same speed and another one that is faster while maintaining the same size. For example the most balanced format in this plot, *fc block*, is outperformed by a roughly 10% faster configuration of the same size and its performance can be achieved with a configuration using only two thirds of its space. This shows the benefit of adapting the compression format to the workload. Last but not least, the plot also shows that the space / time trade-off of a configuration produced by the compression manager can be controlled by varying Δc, making it suitable as "trade-off knob".

We now analyze what dictionary formats the compression manager selected depending on Δc in order to understand how the different trade-offs were achieved. Figure 11 shows how Δc affects the distribution of the dictionary formats: Starting from very small values of Δc, i. e., the most heavily compressing configurations, the pointer-free format *array fixed* is used for a large fraction of the dictionaries. The reason is that this is actually the smallest format for the numerous columns with very low cardinalities such as `C_MKTSEGMENT` thanks to its small constant overhead. For small values of Δc, we can also observe the largest diversification of formats. This suggests that the compression manager successfully identified specialized dictionary formats thanks to the compression models. As Δc increases, the usage of heavily compressing formats such as *fc block rp 12/16* and (on specific data) *column bc* declines more and more in favor of more balanced formats such as *fc block df*. Towards the end of the largest values of Δc, even these formats are more and more replaced by the fastest one, *array fixed*, which finally accounts for all columns. All in all, the selections of the compression manager presented in Figure 11 provide an intuitive explanation for the performance presented in Figure 10.

7. SUMMARY

In this paper we studied the question of how to adaptively compress the string dictionaries of in-memory column-store database systems. Our analysis of real-world business applications showed that strings are more commonly used than previously thought. In alignment with the requirement of single tuple access of in-memory column-stores, we studied a broad variety of compressed dictionary formats. We found that for a single dictionary, there is always a trade-off between access time, construction time, and space consumption.

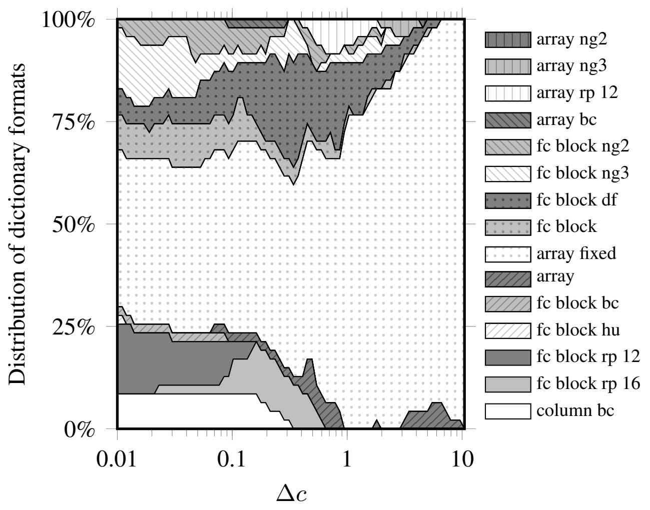

Figure 11: Dictionary formats selected by the compression manager for the TPC-H columns depending on the value of Δc.

In order to improve the space / time trade-off of the overall system, we built a compression manager automatically selecting the most appropriate dictionary format for every column, based on characteristics of the data, usage pattern of the column, and overall system state. The compression manager uses elaborate compression models allowing to predict the size of a dictionary format for a given data set using only a small sample of the data. Furthermore we showed how to decouple local information needed for the format selection from the global information in order to keep the decision cheap.

We confirmed the approach of our compression manager with experiments on a slightly modified TPC-H benchmark. We showed that the adaptive compression can improve overall performance by 10% using the same space than the most balanced single dictionary format or reduce the over space consumption to 60% while maintaining the performance of the single format.

8. ACKNOWLEDGEMENTS

We would like to thank the SAP HANA database development team for their invaluable support in the design, implementation, evaluation, and write-up of our work, in particular Carsten Thiel and Norman May. We also thank the PhD students in the team as well as Wolfgang Lehner, Peter Sanders, and Michael Gertz for their scientific discussions throughout the project.

9. REFERENCES

[1] D. Abadi, S. Madden, and M. Ferreira. Integrating compression and execution in column-oriented database systems. In *SIGMOD*, page 671. ACM Press, 2006.

[2] M. Abd-El-Malek, W. V. C. II, C. Cranor, G. R. Ganger, J. Hendricks, A. J. Klosterman, M. P. Mesnier, M. Prasad, B. Salmon, R. R. Sambasivan, S. Sinnamohideen, J. D. Strunk, E. Thereska, M. Wachs, and J. J. Wylie. Early experiences on the journey towards self-* storage. *IEEE Data Eng. Bull.*, 29(3):55–62, 2006.

[3] S. Agrawal, S. Chaudhuri, and V. R. Narasayya. Automated Selection of Materialized Views and Indexes in SQL Databases. *PVLDB*, pages 496–505, 2000.

[4] J. Arz and J. Fischer. LZ-Compressed String Dictionaries. *CoRR*, abs/1305.0, 2013.

[5] B. Bhattacharjee, L. Lim, T. Malkemus, G. A. Mihaila, K. A. Ross, S. Lau, C. McCarthur, Z. Toth, and R. Sherkat. Efficient Index Compression in DB2 LUW. *PVLDB*, 2(2):1462–1473, 2009.

[6] A. Blumer, J. Blumer, A. Ehrenfeucht, D. Haussler, and R. M. McConnell. Linear size finite automata for the set of all subwords of a word - an outline of results. *Bulletin of the EATCS*, 21:12–20, 1983.

[7] N. R. Brisaboa, R. Cánovas, F. Claude, M. A. Martínez-Prieto, and G. Navarro. Compressed string dictionaries. *SEA*, 6630:136–147, 2011.

[8] N. Bruno and S. Chaudhuri. An Online Approach to Physical Design Tuning. In *ICDE*, pages 826–835. IEEE, 2007.

[9] R. Cánovas and G. Navarro. Practical Compressed Suffix Trees. *SEA*, 2010.

[10] M. Charikar, E. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Sahai, and A. Shelat. The Smallest Grammar Problem. *IEEE Transactions on Information Theory*, 51(7):2554–2576, 2005.

[11] S. Chaudhuri and V. Narasayya. Self-tuning database systems: a decade of progress. *PVLDB*, pages 3–14, 2007.

[12] S. Chaudhuri and V. R. Narasayya. An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server. *PVLDB*, pages 146–155, 1997.

[13] G. V. Cormack. Data compression on a database system. *Communications of the ACM*, 28(12):1336–1342, 1985.

[14] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 2 edition, 2001.

[15] T. P. P. Council. TPC-H Benchmark. `http://www.tpc.org/tpch/default.asp`, Apr. 2013.

[16] M. Crochemore and R. Vérin. On compact directed acyclic word graphs. In J. Mycielski, G. Rozenberg, and A. Salomaa, editors, *Structures in Logic and Computer Science*, volume 1261 of *LNCS*, pages 192–211. Springer, 1997.

[17] B. Dageville and K. Dias. Oracle's Self-Tuning Architecture and Solutions. *IEEE Data Eng. Bull.*, 29(3):24–31, 2006.

[18] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner. SAP HANA Database - Data Management for Modern Business Applications. *ACM SIGMOD Record*, 40(4):45, 2012.

[19] F. Färber, N. May, W. Lehner, P. Große, I. Müller, H. Rauhe, and J. Dees. The SAP HANA Database – An Architecture Overview. *IEEE Data Eng. Bull.*, 35(1):28–33, 2012.

[20] P. Ferragina, R. González, G. Navarro, and R. Venturini. Compressed text indexes: From theory to practice. *Journal of Experimental Algorithmics*, 13:1–12, 2009.

[21] R. González and G. Navarro. Compressed Text Indexes with Fast Locate. In *CPM*, LNCS 4580, pages 216–227, 2007.

[22] R. Grossi and G. Ottaviano. Fast Compressed Tries through Path Decompositions. *CoRR*, abs/1111.5, 2011.

[23] D. Huffman. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[24] S. Idreos, R. Kaushik, V. R. Narasayya, and R. Ramamurthy. Estimating the compression fraction of an index using sampling. In *ICDE*, pages 441–444, 2010.

[25] H. Kimura, V. Narasayya, and M. Syamala. Compression aware physical database design. *PVLDB*, 4(10):657–668, 2011.

[26] D. E. Knuth. *The Art of Computer Programming, volume 3: Sorting and Searching*. Addison Wesley, 2007.

[27] N. J. Larsson and A. Moffat. Offline Dictionary-Based Compression. *DCC*, page 296, 1999.

[28] C. Lemke. *Physische Datenbankoptimierung in hauptspeicherbasierten Column-Store-Systemen*. PhD thesis, Technische Universität Ilmenau, 2012.

[29] C. Lemke, K.-U. Sattler, F. Färber, and A. Zeier. Speeding Up Queries in Column Stores – A Case for Compression. In *DaWak*, pages 117–129, 2010.

[30] E. M. McCreight. A Space-Economical Suffix Tree Construction Algorithm. *Journal of the ACM*, 23(2):262–272, 1976.

[31] G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1), 2007.

[32] M. Paradies, C. Lemke, H. Plattner, W. Lehner, K.-U. Sattler, A. Zeier, and J. Krueger. How to juggle columns. In *IDEAS*, pages 205–215. ACM Press, 2010.

[33] C. Ratsch. Adaptive String Dictionary Compression in In-Memory Column-Store Database Systems. Master's thesis, Heidelberg University, 2013.

[34] G. Ray, J. R. Haritsa, and S. Seshadri. Database Compression: A Performance Enhancement Tool. In *COMAD*, 1995.

[35] L. M. S. Russo, G. Navarro, and A. L. Oliveira. Fully compressed suffix trees. *ACM Transactions on Algorithms*, 7(4):1–34, 2011.

[36] K.-U. Sattler, I. Geist, and E. Schallehn. QUIET: continuous query-driven index tuning. *PVLDB*, pages 1129–1132, 2003.

[37] K. Schnaitter, S. Abiteboul, T. Milo, and N. Polyzotis. COLT: Continuous On-Line Database Tuning. In *SIGMOD*, page 793. ACM Press, 2006.

[38] 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.

[39] P. Weiner. Linear Pattern Matching Algorithms. In *SWAT (FOCS)*, pages 1–11, 1973.

[40] T. Westmann, D. Kossmann, S. Helmer, and G. Moerkotte. The implementation and performance of compressed databases. *ACM SIGMOD Record*, 29(3):55–67, 2000.

[41] T. Willhalm, I. Oukid, and I. Müller. Vectorizing Database Column Scans with Complex Predicates. In *ADMS*, 2013.

[42] 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.

[43] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann, 2. edition, 1999.

[44] M. Zukowski, S. Heman, N. Nes, and P. Boncz. Super-Scalar RAM-CPU Cache Compression. In *ICDE*, pages 59–59. IEEE, 2006.