Sparse Dictionaries for In-Memory Column Stores

Johannes Wust, Jens Krueger, Martin Grund, Uwe Hartmann, Hasso Plattner
Hasso Plattner Institute for Software Engineering
University of Potsdam
Potsdam, Germany
Email: johannes.wust@hpi.uni-potsdam.de, jens.krueger@hpi.uni-potsdam.de
martin.grund@hpi.uni-potsdam.de, uwe.hartmann@hpi.uni-potsdam.de
hasso.plattner@hpi.uni-potsdam.de

***Abstract*—A common approach to achieve high read and write performance for column-oriented in-memory databases is to separate the data store into a read-optimized main partition and a write-optimized differential buffer. The differential buffer has to be merged into the main partition periodically to preserve read performance and increase memory efficiency. If data is dictionary-compressed, this merge process becomes time-consuming since it involves rewriting the whole main tables if dictionary mappings are changed. To reduce the merge time, we introduce the concept of sparse dictionaries, which can avoid rewriting the whole table in many cases. The basic idea is to place gaps in the dictionaries of the main partition which allows us to merge the differential buffer into the main partition without expensive recompression of the main tables. We leverage known data characteristics to optimize our algorithms for enterprise applications.**

***Keywords*-Column-store; In-memory databases; Dictionary compression; Write-optimized Store; Read-optimized Store**

I. Introduction

An In-Memory Database (IMDB) is a database system where the primary persistence resides entirely in the main memory [1]. In recent years, the introduction of a 64 bit address space in commodity operating systems and the constant drop in hardware prices make large capacities of main memory in the order of terabytes technically feasible and economically viable. Together with ever increasing computing power due to multicore CPUs, this change enables the storage and processing of large sets of data in memory and opens the way for general-purpose in-memory data management.

Recently, column-oriented in-memory DBMS were proposed to consolidate transactional and analytical workloads in a single database system, which provides the potential for new enterprise applications and a reduction of the total costs of operating enterprise IT landscapes [2]. Following the data characteristics found in enterprise systems, the proposed architecture relies on dictionary compression per column to utilize memory efficiently. While a column-oriented storage model favors read-mostly analytical workloads, fast write operations on dictionary-compressed column structures are challenging. A common concept in columnar databases is to split the storage in two parts [3], [4]: a read optimized main partition and a write optimized differential buffer or delta store. For dictionary-compressed data, the read optimized store operates on a sorted dictionary, whereas the write-optimized store appends new values to its dictionary.

We call the process of unifying the two parts a merge. The main performance bottleneck of the naive algorithm as described in [5] is that the whole main partition needs to be copied, leading to a time- and memory-consuming operation – experiments with an implementation of this algorithm in our storage engine HYRISE [6] revealed that copying the structures consumes up to 50% of the total execution time of a merge.

Copying the columns to be merged is required as the dictionary mappings change throughout the merge and the value IDs of each record need to be changed. As the dictionary of the main partition is sorted, this happens each time a value is inserted. Our basic idea is to avoid this by placing gaps in the dictionary, leading to so-called sparse dictionaries. That way, we can add new values to the dictionaries of the main partition without having to change all following value IDs and merge differential buffer into the main partition *in-place*. We can perform this intermediate merge several times until the gaps have been filled, before we need to execute the full merge that involves copying the columns.

A. Contributions

Specifically, our contributions presented in this paper are the following:

1) A new strategy to merge dictionary-compressed read-optimized and write-optimized stores based on a novel data structure called sparse dictionary
2) A cost model for estimating the cost of inserting a value into a sparse dictionary depending on the underlying data characteristics
3) A performance evaluation that compares the runtime of the regular merge process [5] with our optimized sparse merge

Copyright (c) IARIA, 2012. ISBN: 978-1-61208-185-4

B. Related work

Our work is based on the system model of an in-memory database as described in [7]. Besides this architecture specifically targeted for enterprise applications, other in-memory database have been recently developed. From a research point of view, MonetDB [8] and H-Store [9] have been the most influential systems; from a commercial perspective, SAP's In-memory computing engine, IBM's SolidDB and Oracle's Times Ten are best known.

Targeting the challenge of order-preserving dictionaries if the domain size is not known a priori, Binnig et al. [10] describe a data structure specific to string compression.

Concerning the merge algorithm as described in [5], another improvement has been proposed in [11]. This algorithm reduces memory consumption by merging single columns. Our object is to improve run-time and we consider the contribution in this paper as complimentary to this work.

C. Structure of this Paper

This remainder of this paper is structured as follows: Section II gives an overview of the two proposed merge processes, the full sparse merge and the intermediate sparse merge. In Section III, we introduce the sparse dictionary and define the underlying data structure. Section IV describes a cost model for inserting values into a sparse dictionary, based on which we define operations on the sparse dictionary in V. Section VI compares the performance of our proposed sparse merge to the regular merge and we close this paper in Section VII with some concluding remarks.

II. Full Sparse Merge and Intermediate Sparse Merge

We replace the regular merge process as described in [5] with two merge processes: a full sparse merge and an intermediate sparse merge.

Intermediate merges are faster than full merges, as they perform an *in-place merge* leveraging well-placed gaps of the dictionaries of the main partition in order to insert new values. As no, or only a small number of value IDs change, we do not need to copy the tables of the main partition, but change the values in the columns of the main partition in place using an index. Intermediate merges can still merge the whole differential buffer into the main partition, but the size of the new values in the dictionary of the buffer is limited by the number of the remaining gaps in the dictionary and the value domain (bit width) of the value IDs.

However, we cannot only rely on intermediated merges: at some point in time the number of gaps will be depleted and/or the width of the value IDs has to be increased. In the first case new gaps have to be added and they should be distributed in a way that supports the intermediate merge, once again requiring a rewrite of a large part of the table. In the latter case all value IDs will have to be rewritten. For these cases we propose the full sparse merge. In contrast to the intermediate sparse merge, it does not operate in-place on the main store and always has to rewrite the whole store. While merging the differential and main dictionary, and potentially increasing the resulting new main dictionary, it redistributes the gaps as efficiently as possible to speed up succeeding intermediate sparse merges.

In order to decrease the number of full merges required, the increasing width of the value IDs and the addition and redistribution of the gaps should be synchronized. It turns out that at a high enough filling level of the dictionary, it becomes more expensive to shift the values in a way that uses the last gaps than to do a full merge. Moreover subsequent intermediate merges benefit largely from the full merge.

In the following, we introduce sparse dictionaries, the data structure our proposed merge processes operate on, and describe the operations on this dictionary required for the full and intermediate sparse merge.

III. Sparse Dictionary

The basic idea of a sparse dictionary is to leave gaps in regular intervals within the dictionary. As a consequence new value IDs can be inserted in the dictionary without moving other values around. In practice, the concept of sparse dictionaries entails a number of implementation specific questions, such as:

- How can gaps be identified?
- Where should the gaps be placed?
- How many gaps should the dictionary have?

In this section, we introduce some definitions to describe our data structures and algorithms throughout the remainder of this paper, and we evaluate data structures used to mark gaps in the dictionary vector.

When it comes to the other two questions, the naive approach would be adding a fixed number of gaps during each full merge and uniformly distribute the gaps over the dictionary vector. However, as some value IDs are used more often than others, moving them around in case no gap is located close to them is more expensive then others. Therefore, we introduce a cost model for changing values in the sparse dictionary in Section IV and discuss optimized operations in Section V.

A. Definitions

To formally describe our data structures and algorithms throughout this paper we use the following notation:

$\mathcal{M}$: Attribute vector of the main partition of the considered column

$\mathcal{U}_{\mathcal{M}}$: Dictionary of the Main partition of the considered column prior to a merge process

$\mathcal{S}$: (new) Sparse Dictionary of the Main partition of the considered column after a merge

$\mathcal{G}_{\mathcal{S}}$: set of value IDs representing sparse elements / gaps within the sparse dictionary

Copyright (c) IARIA, 2012. ISBN: 978-1-61208-185-4

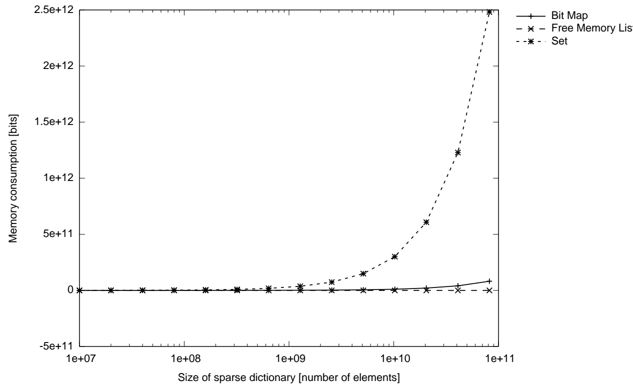

Figure 1. Additional Memory Consumption caused by gap identification with *30%* sparse values running on a 64 bit machine.

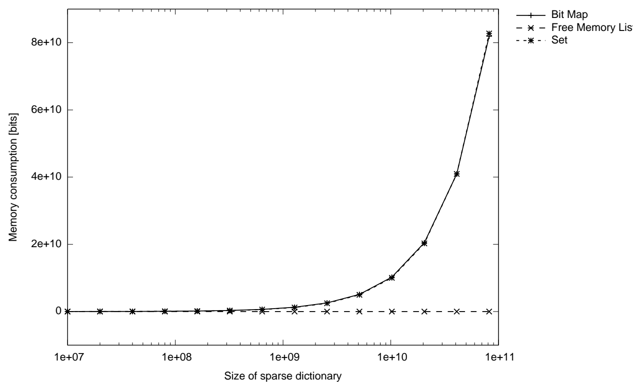

Figure 2. Additional Memory Consumption caused by gap identification with *1%* sparse values running on a 64 bit machine. Memory Consumption of bit map implementation and set are nearly equal.

$l(\mathcal{U})$: denotes the length in bits of a single value in the given dictionary

$\mathcal{D}$: Attribute vector of the differential buffer of the considered column

$\mathcal{U}_\mathcal{D}$: Dictionary of the Differential buffer of the considered column prior to a merge

$\mathcal{I}_\mathcal{S}$: index to identify which values in a dictionary are sparse

$d(vid) : [0, |\mathcal{S}|[\mapsto W$: dictionary mapping for a column with domain W

Note that we store two vectors for each column: A dictionary vector that holds the mapping from value IDs to values for all distinct values of the considered column, and an attribute vector, that holds the value IDs.

Furthermore we define a metric called *Sparsity* Ξ that represents the relative number of gaps in a sparse dictionary $\mathcal{S}$. It is defined as follows:

$$\Xi(\mathcal{S}) = \frac{|\mathcal{G}_\mathcal{S}|}{|\mathcal{S}|} \quad (1)$$

B. Evaluation of Data Structures for the Sparse Dictionary

Our sparse dictionary $\mathcal{S}$ is implemented as a vector holding the values. To quickly identify gaps in the vector, we need an index $\mathcal{I}_\mathcal{S}$. The value of a gap can remain undefined – however, we can set it to the value of a preceding entry that is not sparse to assist standard binary search algorithms on the vector. As index structure, we have evaluated three options based on memory consumption and access time: a bit map, free memory lists [12] and a set storing the indexes of gaps.

The bit map stores one bit per entry of the dictionary and therefore leads to a memory consumption of $|Dictionary|$ bits and an access time into the vector of $O(1)$.

The free memory list stores in each gap the index of the next gap. As all values in the dictionary are distinct, the number of bits needed to encode the values is always bigger or equal to the width of the index. So the index always fits into the dictionary slot. In this way the memory overhead of the free memory list is zero (Note that we cannot store the values of preceding entries in the gaps, as mentioned above). The downside of the free memory list is that each check whether a certain value ID is free requires iterating over the whole free memory list. Furthermore each iteration generates a cache miss as the gaps are by purpose distributed over the dictionary. Hence the access time is $O(n)$, with n being the number of gaps.

When using a *set* for storing the indexes of the gaps, memory consumption is heavily implementation dependent, but would scale up with number of gaps and only logarithmically with the whole number of elements in the dictionary as it is the case with the bit map. For our calculation we assume a self-balancing binary search tree, as it is used in the standard c++ library. This leads to a memory overhead δm in bits of

$$\begin{aligned} \delta m = (log2(|Dictionary|)) * n \\ +size(Pointer) * (n-1) \end{aligned} \quad (2)$$

and an access time of $O(log(n))$, n being the number of gaps.

Figures 1, 2, and 3 compare the memory consumption of each of the three data structures. Comparing figure 1 and 3 shows that the bit map has a far smaller memory footprint with a larger fraction of gaps than the set. Yet, when using smaller fractions of free spaces in the dictionary, the set outperforms the bitmap. As one can see in Figure 2 the break even point is about 1%. The free memory list does not need any additional memory but has a linear runtime for containment lookups. In conclusion it depends on the scenario to choose the right data structure: If memory consumption is a concern, the free space list is the right choice. If performance is the most important priority, the bit map offers the best characteristics, but the space characteristics of the set are better if only very little gaps (below 1%) are used.

Copyright (c) IARIA, 2012.

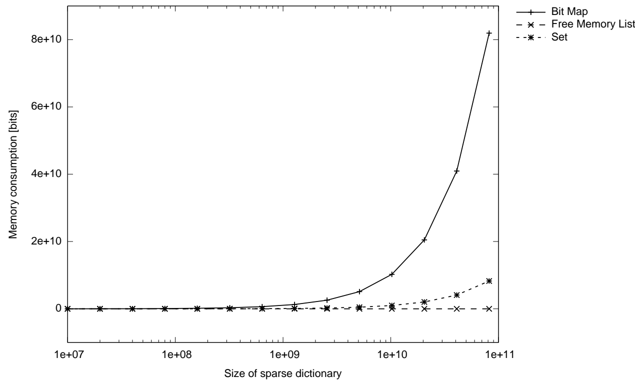

Figure 3. Additional Memory Consumption caused by gap identification with *0.1%* sparse values running on a 64 bit machine.

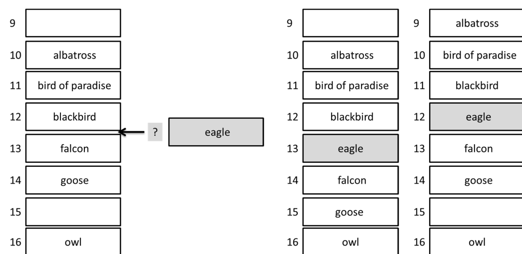

Figure 4. Insertion of a new element into a sparse dictionary and two possible outcomes

IV. Cost Model for Changes in the Sparse Dictionary

In this section, we address the question of how the right places for the gaps can be found and based on this question, where to insert a new value in the dictionary. For inserting new values, the problem is trivial if we hit a gap at the position we want to insert. But a common case is that we must create an empty space at the desired position by shifting values in an existing gap. Figure 4 illustrates this problem. The two cases have different costs depending on the number of occurrences of each changed value ID. Consequently we define a cost function to approximate the resulting cost for changing a specific value ID in the dictionary. We also apply this for cost function later for placing the gaps in case we create a new sparse dictionary during a full sparse merge (Section V-A). The idea is to leave more gaps around values that implicate high costs when shifted.

Change Cost Functions: We define a change cost function π that assigns the cost of changing the associated value to each value ID:

$$\pi(vid) : [0, |\mathcal{S}|[\mapsto \mathbb{N} \tag{3}$$

In the following we discuss potential cost functions.

1) Simple Cost Function: The simple cost function returns 1 for every value ID.

$$\pi_{simple}(vid) = 1 \tag{4}$$

Although this function causes no computation overhead, it is only useful, if the occurrence of distinct values in the main store follows a uniform distribution. However, an analysis of the data characteristics of enterprise systems in [13] showed that the value distribution of the columns with more than one distinct values is in more than 50% of the cases better approximated by a zipf distribution.

2) Quantitative Cost Function: A more advanced change cost function is the so-called quantitative cost function. It calculates the number of elements in the main storage that have to be rewritten as a result of changing the given value ID, which becomes computationally expensive in case of large tables. In case of a small number of distinct values, a potential solution would be to maintain a list of occurrences of each value; however, the additional memory consumption increases in case of many distinct values.

$$\pi_{quantitative}(vid) = |x : x \in \mathcal{M} \wedge x = vid| \tag{5}$$

3) Pareto Cost Function: The problem with the simple cost function introduced in IV-1 is that it assumes an even distribution, whereas the quantitative cost function suffers from a long computation time or high space consumption to save pre-calculated values. As our cost model should support the decision for the placement of gaps, we do not require an optimal result, but rather a heuristic that is fast to calculate. We therefore have defined the pareto cost function, which is based on the observed data characteristics in the enterprise application domain discussed in [13].

Analyzing the characteristics of the data sets discussed in [13] we found out that in most columns at most 20% of the distinct values are sufficient to cover at least 80% of all rows. We use this observation to categorize the values in the dictionary in two categories: The first category, called FC, consists of the 20% of the distinct values that cover the most rows in a given column. The second category, called SC, contains all remaining values. The pareto cost function is basically the quantitative cost function with a reduced co-domain. Based on the observed value distribution, we give those two categories a relative value. In our analyzed data set we have seen that 20% of values covered between 73.53% up to 94.02% of the of elements in the main store, on average 86.91%. Assuming an even distribution within the categories this means that each 1% of the first class distinct values of the first category covers on average $\frac{86.91\%}{20\%} = 4.34\%$ of the rows in a column. In contrast to that, 1% of the second

Copyright (c) IARIA, 2012. ISBN: 978-1-61208-185-4

class values cover only $\frac{100\%-86.91\%}{80\%} = 0.16\%$ of the values. Accordingly one distinct value of the first category covers on average *27 times* more values than a value of the second category.

$$\pi_{pareto}(vid) = \begin{cases} 27, & d(vid) \in FC \\ 1, & d(vid) \in SC \end{cases} \tag{6}$$

Note that the assigned values for the pareto cost function depend on the expected value distribution of the data set at hand. The assigned values to each class can be adjusted for different value distributions.

V. Operations on the sparse dictionary

In this section, we describe operations on the sparse dictionary that are required when performing the proposed full and intermediate sparse merge. These operations are the creation of a new sparse dictionary in a full merge and insertion of values into the dictionary in an intermediate merge.

A. Creating the Dictionary

Creating a new sparse dictionary during the full sparse merge is considered as a merge of two existing dictionaries. At startup, the initial main dictionary is empty. Before we describe the proposed algorithm in detail, we discuss the rationale for choosing the number of gaps.

The Size of the Dictionary and the Number of Gaps: As discussed in Section III, we use a vector as the underlying data structure for a dictionary. Having the sizes of the two input dictionaries as input, we need to determine how much size we want to reserve for our new dictionary. There are several points to consider:

1) The more gaps there are in the dictionary, the faster the intermediate merges are as it reduces the likelihood of shifting values.
2) The more gaps there are in the dictionary, the more intermediate merges can be done before another expensive full merge is required.
3) The more gaps there are in the dictionary, the more space is used by the dictionary.
4) The more gaps there are in the dictionary, the less values fit into a cache line when iterating over the dictionary values, slowing down the iteration.
5) If the total number of elements in the dictionary excesses the size of the next power of 2, a full merge is required to increase the bitmask for a value ID and the marginal cost of additional gaps is increased.
6) The minimum required size of our new dictionary might not be $|\mathcal{U}_\mathcal{M}| + |\mathcal{U}_\mathcal{D}|$ since values might be contained in both dictionaries. This is actually the upper bound of the required minimum size.

Size Depending on Space Efficiency: Points 1 - 4 show us that there is basically a trade-off of size and speed. The size of the gaps is determined by $|\mathcal{G}_\mathcal{S}| * l(\mathcal{S})$. Point 5 shows us that there is a sweet spot for this trade-off: total number of elements including gaps is most efficient just before it excesses any power of 2, requiring additional space per value ID. Thus an easy to make estimate for an efficient element number would be:

$$|\mathcal{S}| = 2^{\lceil \log_2(|\mathcal{U}_\mathcal{M}| + |\mathcal{U}_\mathcal{D}|) \rceil} \tag{7}$$

Distribution of gaps: Having determined the size of the new sparse dictionary, we need to define the distribution of the gaps based on the cost function π. A fairly simple but fast algorithm would be to evenly distribute the gaps – however, that way we would not take our cost function into account. The A-Star algorithm always offers a optimal solution for our problem but its runtime and in particular its memory consumption characteristics make it unsuitable for our case [14]. A more viable alternative is the greedy algorithm [15], although it does not offer an optimal solution.

Algorithm for creating the Dictionary: Algorithm V.1 describes the merge process of two dictionaries and the creation of the new sparse dictionary.

Algorithm V.1: MERGEDICTIONARIES($\mathcal{U}_\mathcal{M}$, $\mathcal{U}_\mathcal{D}$)

- $\mathcal{S} \leftarrow newSparseDictionary(|\mathcal{U}_\mathcal{M}|, |\mathcal{U}_\mathcal{D}|)$
- $\mathcal{I}_\mathcal{S} \leftarrow newSparseDictionaryIndex(|\mathcal{U}_\mathcal{M}|, |\mathcal{U}_\mathcal{D}|)$
- $STemp \leftarrow$ vector of size $|\mathcal{U}_\mathcal{M}| + |\mathcal{U}_\mathcal{D}|$
- $gaps \leftarrow$ vector of size $|\mathcal{U}_\mathcal{M}| + |\mathcal{U}_\mathcal{D}|$
- $mainTemp \leftarrow$ vector of size $|\mathcal{U}_\mathcal{M}|$
- $diffTemp \leftarrow$ vector of size $|\mathcal{U}_\mathcal{D}|$
- $i, j, k \leftarrow 0$
- **while** $i \neq |\mathcal{U}_\mathcal{M}| \vee j \neq |\mathcal{U}_\mathcal{D}|$
  do
  - $value \leftarrow min(\mathcal{U}_\mathcal{M}[i], \mathcal{U}_\mathcal{D}[j])$
  - **if** $value == \mathcal{U}_\mathcal{M}[i]$
    then
    - $mainTemp[i] \leftarrow k$
    - $i \leftarrow i + 1$
    - $value \leftarrow \mathcal{U}_\mathcal{M}[i]$
  - **if** $value == \mathcal{U}_\mathcal{D}[j]$
    then
    - $diffTemp[j] \leftarrow k$
    - $j \leftarrow j + 1$
    - $value \leftarrow \mathcal{U}_\mathcal{D}[j]$
  - $STemp[k] \leftarrow value$
  - $k \leftarrow k + 1$
- $gaps \leftarrow findDistribution(STemp)$
- $i, j, k \leftarrow 0$
- **while** $i < |\mathcal{U}_\mathcal{M}| + |\mathcal{U}_\mathcal{D}|$
  do
  - $j \leftarrow gaps[i]$
  - $value \leftarrow STemp[i]$
  - **while** $j > 0$
    do
    - $\mathcal{S}[k] \leftarrow value$
    - $\mathcal{I}_\mathcal{S}[k] \leftarrow 0$
    - $k \leftarrow k + 1$
    - $j \leftarrow j - 1$
  - $\mathcal{S}[k] \leftarrow value$
  - $\mathcal{I}_\mathcal{S}[k] \leftarrow 1$
  - $k \leftarrow k + 1$
  - $i \leftarrow i + 1$

Copyright (c) IARIA, 2012. ISBN: 978-1-61208-185-4

The merge of the dictionaries is similar to the one described by Krueger et. al. [5] but inserts gaps to the dictionary in a second pass. We require a second pass, as we need to first merge the dictionaries to calculate the distribution of gaps as discussed in the previous subsection.

Having created a temporary dictionary vector with the appropriate size ($STemp$), we iterate over the two ordered input dictionaries from the main partition and the differential buffer in parallel and merge them, while preserving the order of all values. Note that both dictionaries can contain equal values. We enter the old value IDs into the vectors $mainTemp$ and $diffTemp$ to preserve a mapping from old value IDs to new value IDs, required by running transactions. In a next step we calculate the number of gaps that should be inserted into the final dictionary prior to each value ($findDistribution$), applying the cost function π and a greedy algorithm [15], as discussed previously. In the second pass, we iterate over the temporary dictionary, as well as the vector including the gaps and add gaps and values to the new sparse dictionary $\mathcal{S}$. To indicate whether an entry of $\mathcal{S}$ is a gap, we set the according position in the bitmap $\mathcal{I_S}$ to 0.

B. Inserting into the Sparse Dictionary

The insertion of an new value is trivial if we have to insert it before or after a gap. However, we might also face the issue that we must create an empty space at the desired position by shifting values in an existing gap (see Figure 4). Note that multiple shifts can happen during an intermediate sparse merge. Therefore we note all changes in so-called change map described in the next section and apply all changes to the main partition afterwards.

If we have to create an empty space to insert the new value, we have to choose whether to shift the succeeding or the preceding values to the next gaps. To decide which values to shift, we calculate the estimated resulting cost using π and bit map $\mathcal{I_S}$ we introduced earlier. Afterwards we shift all values between our insert position and the selected gap by one value ID and note this change in a change map. Finally we insert the given value in order.

Creating the Change Map: Having added a new entry to the dictionary and having shifted value IDs in it, we need to reflect these changes in the column's main partition. We need to refresh the value IDs of all values that we shifted in the dictionary. However, it would not be practical to apply those changes to the main data store immediately. There are several reasons for this:

- Double updates: Having changed one single dictionary key requires to change all associated keys in the main partition. If the next insertion into the dictionary causes the first value to change again, all values in the main partition have to be changed once more.
- Unusable without index: The performance penalty will further increase if no index is used. The insertion of each value into the dictionary would require a full table scan in order to update the values.

Consequently, we propose a change map data structure that stores a mapping from the old value ID of a shifted value ID to the corresponding new value ID. We store all changes with the dictionary during the whole merge in this data structure. At the end of the merge, we commit these changes to the main store, avoiding the two problems stated above. To record these changes, we use a CSB+ tree that is optimized for read as well as write operations [16].

We only insert position changes of already existing values, meaning that we have to exclude all new values that were shifted in course of the merge. The reason is the following: Given a new value is inserted at position 12. To do this the old value at 12 needs to be shifted to 13. Thus $12 \mapsto 13$ is added to the change set. Now another insertion causes 12 shift to 13 and 13 shift to 14. In this case there would be two values for 12 in the change map: $12 \mapsto 13$ (the new value) and $12 \mapsto 14$. As soon as we replace value IDs in the main storage we would not know whether we should replace the value ID 12 with 13 or 14. Moreover, there is no point in tracking shifts of new values, since none of them are in the main storage and can be replaced. Thus we create another bitmap similar to the one saving whether a value ID is occupied or not. This bitmap saves whether a value ID points to a new Value or not.

Our goal is a tree structure that maps the value ID for a value before the merge to the corresponding value ID after the merge. If a value ID that has to be refreshed was already mapped, we need to know this and have to find the original ID (pre-merge) that maps to this ID. This operation involves a reverse search on the tree which results in a linear search over all values of the change map. This would have to be done at every insert of a change item and thus potentially multiple times per value insertion. In the following, we propose three methods of reducing the search time.

A Second Mapping: The simplest approach to this problem would be the construction of a second map, or an index structure for the reverse lookup. But this would involve an increase in memory consumption since the keys and values would have to be stored in memory twice. Moreover the write performance, which is important to us because of the number of updates, would suffer as well.

Shared-Leave Structure: Another approach would be the use of a Leaf structure as proposed in [10]. This would avoid storing value IDs twice in memory.

Mapping with Lazy and Fast Search: The two proposals above were general solutions – in the following, we propose a specific solution exploiting the characteristics of our specific case

First, we define the Fast Search: The input of our function is a mapping $x \mapsto y$ of the current state of the dictionary which maps the value ID x via the mapping function $f(K)$ to a new ID y. In the following we call x the key of the

Copyright (c) IARIA, 2012. ISBN: 978-1-61208-185-4

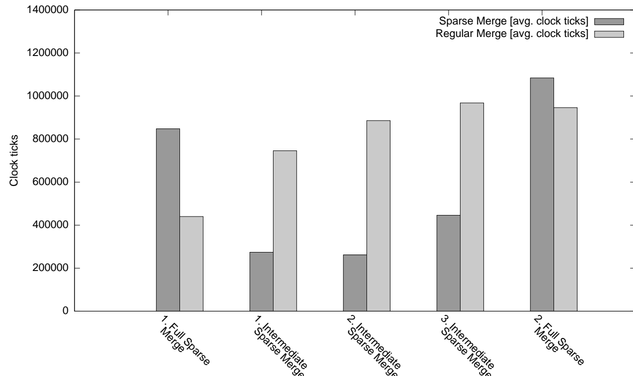

Figure 5. Performance of the Full Sparse Merge and Intermediate Sparse Merge compared to a regular Merge over multiple merge cycles

mapping and y the new value. Hence we need to check the map m, whether it already contains a mapping $z \mapsto x$. In this case we have to replace it by $z \mapsto y$. Searching a mapping with new value x in our map would require $\mathcal{O}(n)$ steps (n being the number of elements in m), as we need to iterate over all values in m in the worst case.

But we know that we find a mapping with key z in close distance where we would enter $x \mapsto y$ in the map, as each shift increases the distance between x and z by one. Thus we can iterate starting from the closest key to x in both directions, thereby exploiting the cache line by reading chunks of date in each direction. However this assumes that a mapping $z \mapsto x$ is contained in our map which is typically not the case.

In order to deal with this issue we propose the Lazy Search. The idea of Lazy search is to reduce the question: "Which z maps to x: $f^-1(x)$?" to the question "Is there a z that maps to x: $x \in f(K)$?". If we can answer the latter question quickly, we will not run into worst case complexity of the Fast Search, as we abort the search in case $x \notin f(K)$. For the discussion of deciding whether $x \in f(K)$, we define the following:

K	is the set of all value keys of the mappings in the change map
E_{new}	is the set of all value IDs which constitute a gap in the current state of the dictionary during merge
E_{old}	is the set of all value IDs which constituted a gap before the merge started
N	is the set of all new value IDs that belong to a new value from the differential buffer

Furthermore, we know, that $f(x) \neq x$, as we only record changes of a mapping, and that $x \notin E_{new}$, since we only shift values that are not empty. As discussed earlier, we do not store new values in the changed map, thus $x \notin N$. We can exclude these cases.

If the element at x was not shifted since the beginning of the merge ($x \notin K$) and x was not empty at this point in time, meaning $x \notin E_{old}$, we know that the same element is still at this position as only one element can be contained at the same time and because of $x \notin K$, x was never moved. Consequently, no element could have been moved to x and thus $x \notin f(K)$:

$$x \notin K \wedge x \notin E_{old} \Rightarrow x \notin f(K) \tag{8}$$

On the other hand, if x has been shifted since the beginning of the merge and x was not empty at this point of time, we know that another value has been mapped to the value ID x and we find a mapping with new value x in our map. Hence:

$$x \in K \wedge x \notin E_{old} \Rightarrow x \in f(K) \tag{9}$$

If the element at x was empty at the beginning of the merge and we encounter a mapping from value ID x to another ID, we have to find a mapping in m that has assigned a value to x. Hence:

$$x \in E_{old} \Rightarrow x \in f(K) \tag{10}$$

We can summarize these observations in the following truth table:

$x \in K$	$x \in E_{old}$	$x \in f(K)$	Rule
1	0	✓	(9)
0	1	✓	(10)
0	0	×	(8)

As we can see, based on the two conditions $x \in E_{old}$ and $x \in K$ we can quickly decide whether we should perform a fast search. $x \in E_{old}$ can be checked in constant time using the old bitmap and $x \in K$ can be checked in $\mathcal{O}(log(n))$, by searching the CSB+ tree. This way we can efficiently avoid searching the map if the sought-after mapping is not in the map.

Finally, the attribute vector of the main partition must be updated using the change map. This step is quite fast forward. All entries in the main which are equal to a key in the change map must be replaced by the corresponding value. It is best to use an index for this operation because otherwise a full table scan is required. After that, the entries from the attribute vector of the differential buffer must be appended to the main partition using the value IDs of the sparse dictionary of the main storage.

VI. Performance Evaluation

Test Setup: In order to evaluate the performance of the sparse full merge as well as sparse intermediate sparse we set up an experiment that executed five subsequent merges: 2 full merges and 3 intermediate merges and compared the

Copyright (c) IARIA, 2012. ISBN: 978-1-61208-185-4

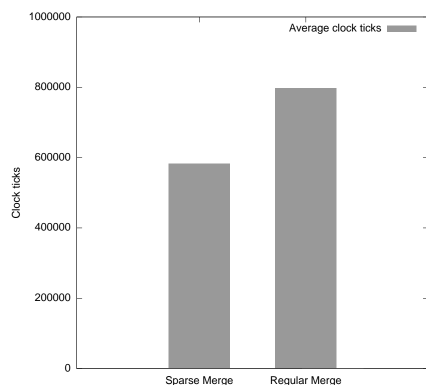

Figure 6. Total time spend on merging in the course of 5 subsequent merges

results against five regular merges. The used table starts with 50000 entries in the main part. At each stage of the experiment, the merge has to add a differential buffer with 20% of the size of the current main partition. Each distinct value is represented on average 10 times in the table. The first full sparse merge creates a sparsity of 50% for the subsequent intermediate merges. The first full merge is required to bring the table and the dictionary in a state that allows the subsequent in-place merges through the gaps in the dictionary that are created in this merge. The last full merge restores the starting state (a dictionary with a lower sparsity) to offer a fair comparison to the regular merge. The performance evaluation was done using the column oriented in-memory database prototype HYRISE [6]. The test machine used a 64 bit Linux (Ubuntu) operating system and was equipped with a 2.4 GHz Core 2 Duo processor with 2 cores and 2 GB of RAM. Figure 5 shows the results of our experiment.

Results: As shown in Figure 5 the full sparse merge is slower than a regular merge due to the overhead of reorganizing the dictionary. But it enables the faster intermediate merges. However, it is striking that the runtime of the intermediate merges increases faster than of the regular merges. This is due to the dictionary filling more with each additional sparse merge, leading to more shifting of value IDs in the dictionary. The more often we merge, the closer the runtime approaches the regular merge. Figure 6 shows the run times and deviations of all five merges (consequently including the full sparse merges) added together. We can see, that the *total* time the system uses to merge decrease by roughly 25% when using full sparse merges and intermediate sparse merges compared to a regular merge.

VII. Conclusion

In this paper we showed implementation techniques for the concept of an intermediate sparse merge and a full sparse merge both using a sparse dictionary. The full sparse merge reorganizes a dictionary and a table so that several in-place merges using the intermediate sparse merge can be executed. The full sparse merge "charges" the table so that it can use a few very fast intermediate sparse merge that do not require the table to be copied. The combination from full sparse merge and intermediate sparse merge can reduce the total time the machine spends merging by around 25%, thus allowing other operations such as queries use the freed resources. Additionally, the intermediate sparse merge avoids the high base cost of the copy operation. Thus it scales with the size of the differential buffer and the state of the dictionary. Therefore it becomes possible to use the merge more often at a lower cost. This improves the general query performance since the differential buffer is smaller and more values reside in the read optimized main store.

References

[1] H. Garcia Molina and K. Salem, "Main memory database systems: an overview," *IEEE Transactions on Knowledge and Data Engineering*, vol. 4, no. 6, pp. 509–516, 1992.

[2] H. Plattner, "A common database approach for oltp and olap using an in-memory column database," in *Proceedings of the 35th SIGMOD international conference on Management of data*, ser. SIGMOD '09. New York, NY, USA: ACM, 2009, pp. 1–2.

[3] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. R. 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*, Trondheim, Norway, 2005, pp. 553–564.

[4] P. A. Boncz, M. Zukowski, and N. Nes, "Monetdb/x100: Hyper-pipelining query execution," in *CIDR*, 2005, pp. 225–237.

[5] J. Krueger, M. Grund, C. Tinnefeld, H. Plattner, A. Zeier, and F. Faerber, Eds., *Optimizing Write Performance for Read Optimized Databases: Database Systems for Advanced Applications*. Springer, 2010.

[6] M. Grund, J. Krüger, H. Plattner, A. Zeier, P. Cudre-Mauroux, and S. Madden, "Hyrise: a main memory hybrid storage engine," *Proc. VLDB Endow.*, vol. 4, pp. 105–116, November 2010.

[7] H. Plattner and A. Zeier, *In-Memory data management: An inflection Point for Enterprise Applications*, 1st ed. Berlin: Springer, 2011.

[8] S. Manegold, M. L. Kersten, and P. Boncz, "Database architecture evolution: mammals flourished long before dinosaurs became extinct," *Proc. VLDB Endow.*, vol. 2, pp. 1648–1653, August 2009.

[9] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. Jones, S. Madden, M. Stonebraker, and Y. Zhang, "H-store: a high-performance, distributed main memory transaction processing system," *VLDB*, pp. 1496–1499, 2008.

[10] C. Binnig, S. Hildenbrand, and F. Färber, “Dictionary-based order-preserving string compression for main memory column stores,” in *Proceedings of the 35th SIGMOD international conference on Management of data*, ser. SIGMOD ’09. New York, NY, USA: ACM, 2009, pp. 283–296.

[11] Jens Krueger, Martin Grund, Johannes Wust, Alexander Zeier, and Hasso Plattner, “Merging differential updates in in-memory column store,” *DBKDA 2011*, 2011.

[12] R. Fenichel and J. Yochelson, “A lisp garbage-collector for virtual-memory computer systems,” *Communications of the ACM*, vol. 12, no. 11, pp. 611–612, 1969.

[13] F. Huebner, J.-H. Boese, J. Krueger, F. Renkes, C. Tosun, and A. Zeier, “A cost-aware strategy for merging differential stores in column-oriented in-memory dbms,” in *BIRTE workshop in conjunction with VLDB, Seattle*, 2011.

[14] P. Hart, N. Nilsson, and B. Raphael, “A formal basis for the heuristic determination of minimum cost paths,” *Systems Science and Cybernetics, IEEE Transactions on*, vol. 4, no. 2, pp. 100–107, 1968.

[15] V. Chvatal, “A greedy heuristic for the set-covering problem,” *Mathematics of operations research*, pp. 233–235, 1979.

[16] V. Raatikka, “Cache-Conscious Index Structures for Main-Memory Databases,” http://ethesis.helsinki.fi/julkaisut/mat/tieto/pg/raatikka/cachecon.pdf (Accessed February, 2012), Feb. 2004.