A cost-aware strategy for merging differential stores in column-oriented in-memory DBMS

Florian Hübner[1], Joos-Hendrik Böse[2],
Jens Krüger[1], Frank Renkes[2], Cafer Tosun[2], Alexander Zeier[1], Hasso Plattner[1]

[1]Hasso Plattner Institute, [2]SAP AG

Abstract. Fast execution of analytical and transactional queries in column-oriented in-memory DBMS is achieved by combining a read-optimized data store with a write-optimized differential store. To maintain high read performance, both structures must be merged from time to time. In this paper we describe a new merge algorithm that applies full and partial merge operations based on their costs and improvement of read performance. We show by simulation that our algorithm reduces merge costs significantly for workloads found in enterprise applications, while improving read performance at the same time.

1 Introduction

Providing a unified view on real-time data for interactive and ad-hoc data analysis is the goal of column-oriented in-memory DBMS, such as the SanssouciDB outlined in [6]. This DBMS is designed to consolidate the OLTP and OLAP workload of enterprises in one in-memory database, guaranteeing high performance for analytical queries on real-time data as well as for transactional insert operations. While the concept of column-orientation favors mainly analytical query processing [4] and high compression rates [1], changing data in compressed column-structures is expensive. A standard approach to overcome this problem in column-oriented DBMS, e.g. as applied in MonetDB [3] and C-Store [7], is to use an additional write-optimized structure where all changes to the data are accumulated. Since read queries must access both, the read-optimized main store as well as the write-optimized differential store, to derive the consistent state of the DB, overall performance of read queries degrades with increasing size of the latter. To compensate for this performance degradation, changes accumulated in the differential store need to be merged into the read-optimized main store to clear the differential store. Such a merge operation can lead to significant overhead, because it may involve changing the dictionary and re-encoding of all values in a column. In an enterprise environment where columns can easily contain hundreds of millions of records, this may reduce system performance significantly. However, integrating the differential store into the main store from time to time is crucial to maintain viable read performance on frequently updated tables.

In this paper we present a merge strategy that significantly reduces merge overhead for enterprise workloads, while improving read performance at the same time. Specifically, our contributions are the following:

1. We analyse the value distribution of inserts in enterprise systems and present the range of distributions found in typical enterprise applications.
2. We present a merge strategy based on the *partial merge operation*, a merge *trigger strategy* and a *cost aware decision* between full and partial merge operations.
3. We evaluate our approach with a simulation study.

The paper is structured as follows: Section 2 defines our system model and introduces our cost metrics for merge, insert, and select operations. Section 3 presents an analysis of value distributions in columns and their influence on the dictionary encoding in column-oriented data structures. Section 4 gives a detailed description of our algorithms, which is evaluated with a simulation study in Section 5. In Section 6 we conclude and summarize our findings.

2 Architecture, System and Cost Model

The abstract design of a table in SanssouciDB is depicted by Figure 1. Both structures, the read-optimized main store and the write-optimized differential store rely on dictionary encoding [1] for compression. Each column in the main store consists of a column vector C_M and a dictionary D_M. D_M defines a bijective mapping $valueID \leftrightarrow value$, with $valueID$ simply being the position of $value$ in the dictionary vector. Values in D_M are ordered. The number of bits required to encode an entry in C_M is given by $\log_2(|D_M|)$, with $|D_M|$ being the number of distinct values in C_M. A column of the differential store also consist of a column vector C_D and a dictionary D_D. In contrast to the main store, the differential store's dictionary D_D is unsorted but uses a CSB+ tree based index structure to translate value ids to values with logarithmic complexity as shown in Figure 1.

2.1 Merge Cost Model

In order to maintain acceptable read performance over time, the main and differential data structures have to be merged. As described in [6], the *merge algorithm* combines the two dictionaries D_M and D_D to a new dictionary D'_M and then combines C_M and C_D with respect to it. Merging the two dictionaries can be done in $O(\mid D_M \mid + \mid D_D \mid)$ by performing a merge sort while traversing both in order. However, this process can trigger two events influencing the combination of the two columns C_M and C_D. First, the number of bits needed to represent all distinct values in D'_M may be increased by the additional values from D_D. Then every value in C_M has to be updated in order to provide additional memory per entry. We call this event *dictionary overflow* Ξ. It happens whenever the number of distinct values is doubled and becomes more and more infrequent with a growing D_M due to the larger steps and also the mature dictionary with

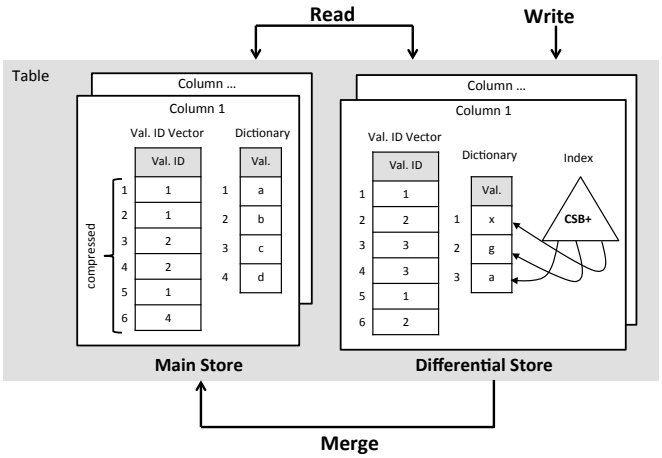


Fig. 1. Data structures of main and differential store.

regard to inserted values. Second, in order to preserve the ordering within the dictionary, any value $v \in D_D$ has to be inserted into D'_M before every $u \in D_M$ with $v < u$, hence the value id for each of those u has to be updated — not only in the dictionary but for every entry representing u within the main store column C_M, resulting in a complexity of $O(\mid C_M \mid + \mid C_D \mid)$ (note that the cost for lookups in the dictionaries can be avoided by creating a mapping between the old and the new dictionary while merging D_M and D_D). We call this event *dictionary re-ordering* Γ.

In practice, whenever C_M and C_D are being combined, a copy of the complete main column is held in memory to enable concurrent read operations while updating the encoding of C_M. This implies that the amount of memory needed for the database may be doubled when merging and also that every merge operation leads to $\mid C_M \mid$ write operations, resulting in the invalidation of all caches holding the column's data. Therefore, we count the number of write operations needed to perform a merge as a proxy for its overall cost. It depends on the current status of the main and differential structure and is defined as follows.

$$mergeCost = \mid D_M \cup D_D \mid + \mid C_M \mid + \mid C_D \mid \tag{1}$$

The sizes of C_D and D_D have to be optimized in order to be small enough to achieve acceptable response times for queries despite the unordered dictionary, and as large as possible in order to have as few merge operations as possible. The latter objective is underlined by the observation that the number of merge

operations performed before any given point in time has no effect on the cost of future merge operations as all summands are invariant.

2.2 Select and Insert Cost

In order to objectively evaluate the performance of our in-memory database system, we define measures that separately look at insert and select operations and discuss their effect on typical workloads. Insert operations are meant to insert single values into a column. Select operations can be divided into point selects and range selects. As different approaches to organizing columns and dictionaries within this paper do not affect the DB's operators after the selected values have been found in a column, we do not further investigate their behavior.

Insert Cost Looking at the complexity of insert operations and taking into consideration what we already know about effects on a dictionary encoded column, it becomes clear that the cost for inserting a single value is strongly dependent on all previous insert operations. When inserting a single value, it is first added to the differential structures. Looking up the value in D_D is of logarithmic complexity $O(log_2(\mid D_D \mid))$ due to the CSB+ index, and adding the found value id to C_D requires a single write operation. If a merge event is triggered, it results in another $\mid D_M \cup D_D \mid + \mid C_M \mid + \mid C_D \mid$ write operations with regard to the cost defined in (1). To evaluate the overall insert performance, we count the number of write operations needed to perform n single insert operations of the values $\{v_1, \ldots, v_n\}$ and evaluate the amortized figures over time. With $writes(v_i)$ being the number of write operations needed to insert v_i including the cost for any merge event triggered before the next insert, we define the amortized insert cost per value $insertCost(n)$ after n inserts as follows.

$$insertCost(n) = \frac{\sum_{i=0}^{n} writes(v_i)}{n} \tag{2}$$

Select Cost A point select operation is a query that returns any entry of a column (or its position) whose attribute value x equals a given value v. It has to be performed on the main and the differential data structures. Looking up the value id of v in D_M and D_D requires $\log_2(\mid D_M \mid)$ and $\log_2(\mid D_D \mid)$ read operations each. If the value id is found in either dictionary, the respective columns have to be scanned requiring $\mid C_M \mid$ and $\mid C_D \mid$ compare instructions (we assume, that no inverted index exists, as we opperate on any given column of a transactional DBMS). It can be seen that the number of operations needed to perform a point select operation is the same within the main and the differential data structures and therefore will be neglected for the purpose of this paper.

A range select operation is a query that returns any entry of a column (or its position) whose attribute value x lies within a given interval defined by its minimum value v_{min} and its maximum value v_{max}. To perform the query on the main data structures, first the interval $[v_{min}, v_{max}]$ has to be looked up

in D_M. The smallest value $v_i > v_{min}$ and the largest value $v_j < v_{max}$ and their value ids id_{min} and id_{max} can be found with $2 \cdot \log_2(\mid D_M \mid)$ compare operations. After that, the column C_M is scanned and for each value id i it has to be checked, whether $id_{min} \leq i \leq id_{max}$ holds, resulting in $2 \cdot \mid C_M \mid$ compare instructions. Performing the same operation on the differential data structure is more complex. As the dictionary D_D is unsorted, the interval $[v_{min}, v_{max}]$ can no longer be represented by two values, but all values $u \in U = [v_{min}, v_{max}] \cap D_D$ and their value ids have to be calculated in $O(\mid D_D \mid)$. Then C_D has to be scanned, but instead of 2 compare instructions per value v, $min(\mid U \mid, 0.5 \cdot \mid D_D \mid)$ compare instructions are needed to evaluate $v \in U$. While the complexity for a range select in the main data structures is linear in $O(\mid C_M \mid)$, performing the same operation on the differential data structures is in $O(\mid C_D \mid \cdot \mid D_D \mid)$ and therefore has quadratic computational complexity. The overall performance of a range select query is defined by the number of compare operations needed and depends on the sizes of the main and differential data structures. For the purpose of this paper, the cost of a range select query is calculated as follows.

$$selectCost = 2 \cdot \mid C_M \mid + 0.5 \cdot \mid C_D \mid \cdot \mid D_D \mid \tag{3}$$

Previous work shows that OLTP as well as OLAP enterprise workloads comprise of mainly read operations. E.g. [5] mentions that even in OLTP workloads logged on enterprise systems about 84% of all operations are read queries, in OLAP workloads even 93%. In the context of this article we are especially interested in the fraction of range selects among read queries, since the performance of range queries degrades with a growing $|C_D|$. From [5] we obtained that in a typical enterprise OLTP workload 30% of all queries select a range of values, while in a OLAP environment 44% of all queries are range selects. Hence, it can be stated that the performance of range selects is crucial to the overall database performance in enterprise workloads.

3 Data Characteristics of Enterprise Systems

The complexity of merge operations depends on the distribution of values represented by D_M and D_D at merge time. Our algorithm improves merge complexity by leveraging characteristics of these distributions. To evaluate the benefit of our algorithm in a typical ERP environment, we derived the characteristic value distributions of columns in enterprise applications found in SAP's Business Suite [2].

We analyzed the distributions of values within columns in SAP's financial (FI) and in the sales and distributions (SD) module of the Business Suite. We examined tables of both applications in two customer installations, while only the master and line items tables were considered, since these are the most frequently accessed tables. In total we analyzed 1864 columns. For every column we computed the frequency of each value and ranked them accordingly in the resulting histogram. In a first step we excluded all histograms, which had more than one value and every value occurs exactly once. For these key or id columns

dictionary encoding would not be applied. Also, columns with only one distinct value were not considered any further, as they allow for avoiding most of the costs for inserts and queries by engineering.

The remaining histograms (in total 770) have more complex value distributions and thus induce more complex computations for calculating query results. In order to describe their characteristics more generally, they were partitioned into two sets: (i) columns with a distribution best approximated by a zipf pdf, and (ii) columns where a uniform distribution provides a smaller rooted means square error (RMSE) than a zipf pdf. Both pdf functions rely on a given number of distinct values as a parameter. The zipf distribution was chosen because it describes frequency distributions we recognized in a lot of columns of FI and SD, which follow a power-law distribution, i.e., a small set of values occurs frequently, while the majority of values (long tail) are rare.

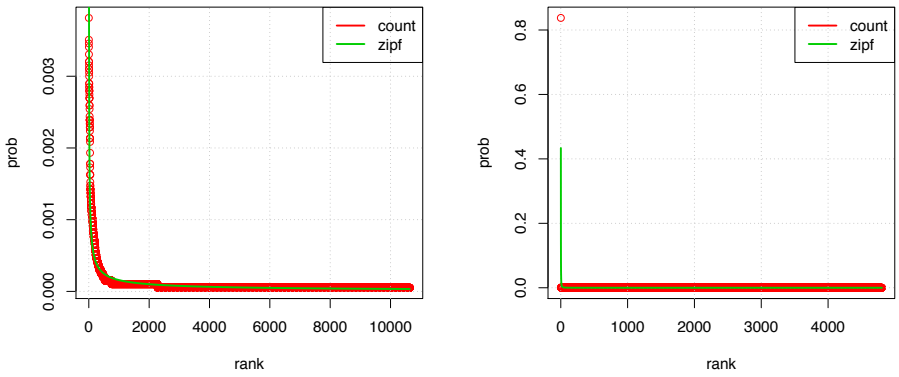
pdf	***fraction***	***parameter range***	***RMSE***
zipf	21.03%	$\alpha_{min} = 0.001$ $\alpha_{Q1} = 1.030$ $\alpha_{Q2} = 1.581$ $\alpha_{Q3} = 2.586$ $\alpha_{max} = 4.884$	$e_{min} = 0.000$ $e_{Q1} = 2.599e^{-16}$ $e_{Q2} = 3.234e^{-15}$ $e_{Q3} = 3.095e^{-13}$ $e_{max} = 7.630e^{-13}$
uniform	20.02%	$k_{min} = 2$ $k_{Q1} = 2$ $k_{Q2} = 5$ $k_{Q3} = 86$ $k_{max} = 216$	$e_{min} = 5.749e^{-17}$ $e_{Q1} = 3.116e^{-16}$ $e_{Q2} = 2.572e^{2}$ $e_{Q3} = 1.465e^{5}$ $e_{max} = 2.398e^{5}$

Table 1. Value distributions and their parameter five-number summaries.

Table 1 summarizes some results of our analysis. It shows that 51% of the relevant columns (i.e., 21.03% of all columns analyzed) are best approximated by a zipf distribution, while 49% are best approximated by a uniform distribution. For those zipf and uniform distributions Table 1 gives the five-number summary of the parameter range. The five-number summary for the zipf parameter α shows a range [$\alpha_{min} = 0.001$, $\alpha_{max} = 4.884$], with a median at $\alpha_{Q2} = 1.581$. We also show the rooted mean square error (RMSE) of the fitted distributions.

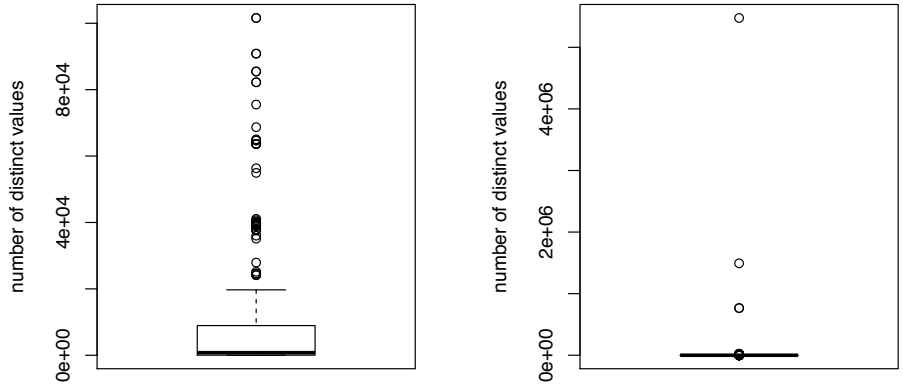
Figure 2 b) shows the ranked histogram for one analyzed column with a zipf coefficient close to α_{Q2}. Figure 2 a) gives another example with zipf parameter α=0.651. Both examples show the typical power-law distribution of values: there are some values that occur frequently, while the majority is rare. Our algorithm will especially leverage the power-law characteristics of such distributions to reduce merge costs.

Besides the zipf parameter α, the total number of distinct values in a column influences the distribution and is the single parameter of uniform distributions. Figure 3 a) depicts the total number of distinct values for all columns that follow a zipf distribution, while Figure 3 b), depicts the same for columns with uniform distributed values using a boxplots.



(a) $\alpha = 0.651$ and $n = 10642$ (b) $\alpha = 1.588$ and $n = 4806$

Fig. 2. Histograms and zipf fits.



(a) zipf (b) uniform

Fig. 3. Number of distinct values.

4 Cost-Aware Merge Strategy

In order to reduce the costs for merging values from the differential data structures described in Section 2 into the main store data structures, we propose an algorithm that is able to merge only a subset of D_D into D_M so that the gain in performance is high but the cost for merging is significantly lower in total. The main idea is to decide the largest subset of values in D_D and C_D that does not trigger expensive re-ordering events Γ when being inserted into D_M (see Section 2.1). As a consequence, it becomes necessary to decide between performing a *partial merge* and a *full merge* whenever a merge event is triggered. In the following, we describe the partial merge algorithm in detail as well as strategies when to trigger merge events and to decide between partial and full merge.

4.1 Partial Merge Algorithm

The partial merge algorithm as described in Algorithm 1 decides the largest subset of values in D_D and C_D that can be merged into the main store data structures D_M and C_M without making the expensive re-ordering of the dictionary and the implied re-encoding of the main column C_M necessary (see Section 2.1). This is achieved by only merging all values v that either are already contained in D_M or whose rank in the ordering of D_M is higher than the ranks of all other values in the dictionary. If v is already contained in D_M, the corresponding value id $valueID(v)$ will be used to encode v in C_M. If the value v is not contained in D_M but of greater rank than all values in D_M, then $valueID(v)$ can be added at the end of D_M without having to update any information in D_M. In both cases, the encoded value can be added at the end of the main store column C_M without affecting all previously inserted values, i.e., no re-encoding is necessary. Finding the set of all value ids *mergeVals* that can be merged without triggering a re-ordering event can be done in $O(\mid D_M \mid + \mid D_D \mid)$ by traversing both dictionaries' values in order. At the same time, we calculate a mapping from D_D to D_M and mark all values contained in *mergeVals*. For each value in C_D we then check, whether it is contained in *mergeVals* and either add it to C_M or to the updated C_D, all of which can be done in $O(\mid C_D \mid)$. The overall complexity of the partial merge algorithm with respect to the columns and dictionaries used for the main and differential store results to $O(\mid D_M \mid + \mid D_D \mid + \mid C_D \mid)$. This is significantly faster compared to the complexity of the full merge in $O(\mid D_M \mid + \mid D_D \mid + \mid C_M \mid + \mid C_D \mid)$ as described in Section 2.1. It can be seen that the complexity of the partial merge is no longer related to the size of the main column. Therefore, the frequency for triggering a partial merge in order to optimize read performance can be significantly higher without compromising on the overall performance.

4.2 Merge Trigger Strategies

The responsibility of a trigger strategy is to monitor the state of the differential store and to initiate a merge to maintain viable *selectCost*. In general, the cost

overhead for merging has to be compensated by the cost saved on select queries, therefore the distribution of queries in the workload influences the choice of trigger parameters. The strategy proposed here aims at keeping read-costs at a low level and is called *fraction of read optimum (fro)*. In Section 5 we will evaluate this strategy against the naive approach, which is called *fraction of main* (*frm*).

```
   Data: C_M, D_M, C_D, D_D
   Result: Updated columns and dictionaries
 1 // find values to merge by comparing dictionaries D_M and D_D
 2 // and calculate mapping between the dictionaries
 3 mergeVals = []
 4 mapping = []
 5 i_m = 0
 6 i_d = 0
 7 while i_m < size(D_M) and i_d < size(D_D) do
 8 |   if D_M[i_m] == D_D[i_d] then
 9 |   |   mergeVals.add(D_D[i_d])
10 |   |   mapping[i_d] = i_m
11 |   |   i_m++
12 |   |   i_d++
13 |   else if D_M[i_m] > D_D[i_d] then
14 |   |   i_d++
15 |   else if D_M[i_m] < D_D[i_d] then
16 |   |   i_m++
17 |   end
18 end
19 if i_m = size(D_M) and i_d < size(D_D) then
20 |   for i_d = i_d to size(D_D) - 1 do
21 |   |   mergeVals.add(D_D[i_d])
22 |   |   mapping[i_d] = i_m++
23 |   |   D_M.add(D_D[i_d])
24 |   |   D_D.remove(v)
25 |   end
26 end
27 // find values in differential column and attach to main column
28 for each v in C_D do
29 |   if v in mergeVals then
30 |   |   C_M.add(v)
31 |   |   C_D.remove(v)
32 |   end
33 end
```

Algorithm 1: Partial Merge Algorithm

Fraction of read optimum (fro) The basic idea of this strategy is to maintain a read-cost level that has a defined distance to the theoretical optimal read-costs. We define the optimal read-costs after n inserts, denoted by $selectCost^{opt}(n)$, as the worst case costs of a range query with $|C_D| = 0$, i.e., $selectCost^{opt}(n)$ describes the maximum costs of a range query when the differential store is fully merged into the main store. The fro strategy allows $selectCost(n)$ to grow until it exceeds a defined fraction f_{opt} of $\text{selectCost}^{opt}(n)$. Thus, a merge is triggered every time (4) is true. As $selectCost(n)$ and $selectCost^{opt}(n)$ are defined by the sizes of the data structures, both values can easily be calculated at runtime.

$$selectCost(n) > f_{opt} \cdot selectCost^{opt}(n) \tag{4}$$

Fraction of main (frm) When applying the *fraction of main (frm)* strategy, a merge is triggered, whenever $|C_D|$ reaches a threshold size s_t relative to $|C_M|$.

$$s_t \cdot \mid C_M \mid < \mid C_D \mid \tag{5}$$

4.3 Full merge vs. Partial Merge

We leverage the performance functions defined in Section 2.2 to choose between a full and a partial merge whenever a merge event is triggered. If the expected performance after a partial merge $selectCost_P$ is comparable to that after a full merge $selectCost_F$, a partial merge will be executed. We define the factor f_{to} to represent the tolerated overhead with regard to the merge costs saved, thus a partial merge is performed whenever (6) evaluates true, a full merge otherwise. If a merge causes a dictionary overflow Ξ, we always choose a full merge operation.

$$selectCost_P \leq f_{to} \cdot selectCost_F \tag{6}$$

$$2 \cdot \mid C'_M \mid + 0.5 \cdot \mid C'_D \mid \cdot \mid D'_D \mid \leq f_{to} \cdot 2 \cdot (\mid C_M \mid + \mid C_D \mid) \tag{7}$$

The optimal value of f_{to} is dependent on the workload. With increasing ratio between range queries and insert operations, the overhead induced by sub-optimal select performance gains influence on the overall database performance, thus the frequency of full merges should be increased by decreasing f_{to}.

Calculating $selectCost_P$ at runtime requires maintaining a list of all values that are merged during a partial merge. Whenever a value $v < max(D_M)$ is added to D_D it has to be looked up in D_M. Additionally, the number of values for each value id in D_D has to be remembered. The total cost for this is in $O(log_2(\mid D_M \mid))$ amortized over the number of inserts between two merge events.

5 Evaluation

We simulate the behavior of the database for inserts sampled from the distributions described in Section 3. To evaluate the performance of our cost-aware merge

strategy introduced in Section 4, we compare it to the behavior of the standard algorithm described in Section 2.1 and analyze the insert cost over time as well as the worst case costs for range select queries after each insert (see Section 2.2).

5.1 Simulation Environment and Parameters

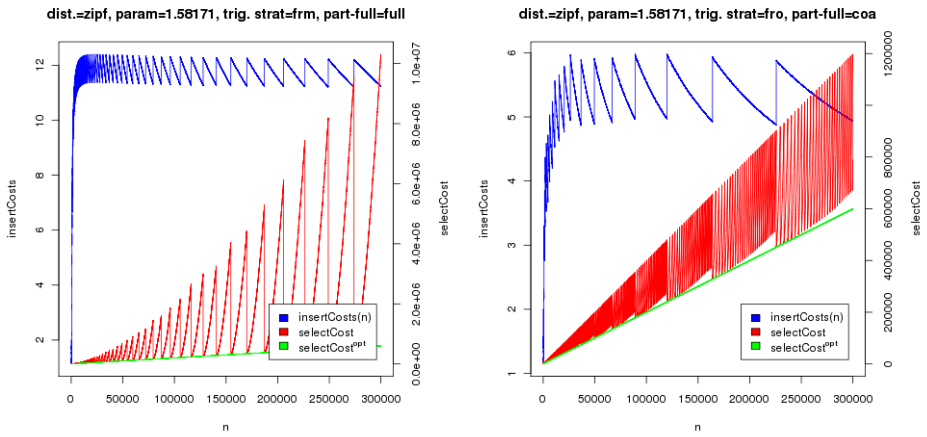
The simulation has been implemented in R. We leverage the median parameters from the value distributions shown in Section 3 in order to generate a set of $n = 300k$ values. 6403 distinct values are generated for the zipf distribution as well as 216 distinct values for the uniform distribution. We then simulate the insertion of the values v_1 to v_n and keep track of lists representing the status of the columns C_M and C_D as well as the dictionaries D_M and D_D over time. For the fraction of optimum (fro) trigger strategy we set the threshold factor $f_{opt} = 2.0$, guaranteeing that the select cost will never be more than two times as high as in a fully read optimized architecture without differential store. The fraction of main (frm) trigger strategy applies a threshold size of 10%, i.e., a merge is triggered whenever the differential column's size $\mid C_D \mid$ is 10% of the main column's size $\mid C_M \mid$. In order to decide between a full and a partial merge in case of our cost-aware algorithm, we set $f_{to} = 1.2$, i.e., a partial merge will be performed as long as the read costs after the merge are not more than 20% higher than they would be after a full merge.

5.2 Analysis of Simulation Results

The simulation study shows that for common distributions observed in enterprise environments, write and range select costs are reduced significantly by our cost-aware merging strategy. To show the potential reduction in costs for zipf distributed columns, we plot the costs for two representative zipf distributions in Figures 4 and 5.

Figure 4 shows the improvement of our algorithm for a workload sampled from a zipf distribution parameterized with the median $\alpha_{Q2} = 1.58171$ of zipf parameters found in our customer data analysis in Section 3. The standard approach with frm trigger and exclusive full merges leads to write costs shown by the blue curve in Figure 4 a) and worst case costs of range selects as depicted by the red curve. The green line marks the optimal read costs as computed by $selectCost^{opt}(n)$. *insertCosts* increase fast and stabilize around 12 write operations per insert, with a slightly decreasing trend. The 12 write operations per value inserted are mainly defined by $s_t = 10\%$, which leads to a mathematical sequence inducing the frequency of merges as well as its change over time. Increasing s_t (e.g., $s_t = 20\%$) would nearly cut the insert costs in half but also dramatically increase *selectCost*, which depends on the delta column's size $\mid C_D \mid$. The decreasing trend in *insertCost* is due to the effect that for larger merge intervals the chance increases that values inserted are already contained in D_D and can be inserted into C_D with a single write operation instead of two writes. *selectCost* increases with the number of inserts, since more values remain in C_D before a merge is triggered. For 300k inserts *selectCost* reaches

$1.0e^7$ and *insertCost* 12.2 in the default stagey, while our approach causes maximal *insertCost* of 6.0 only and *selectCosts* of $1.2e^6$ as shown in Diagram 4 b). The red line in Figure 4 b) depicts the effectiveness of the partial merge strategy, every partial merge moves the *selectCost* close to the optimum, while causing little extra write costs. It can also be seen where the cost-aware merging strategy decides to perform a full merge instead of a partial merge due to the expected improvement in *selectCost*, as the gap to the optimal *selectCost* is closed and the amortized *insertCosts* are increased by 1.



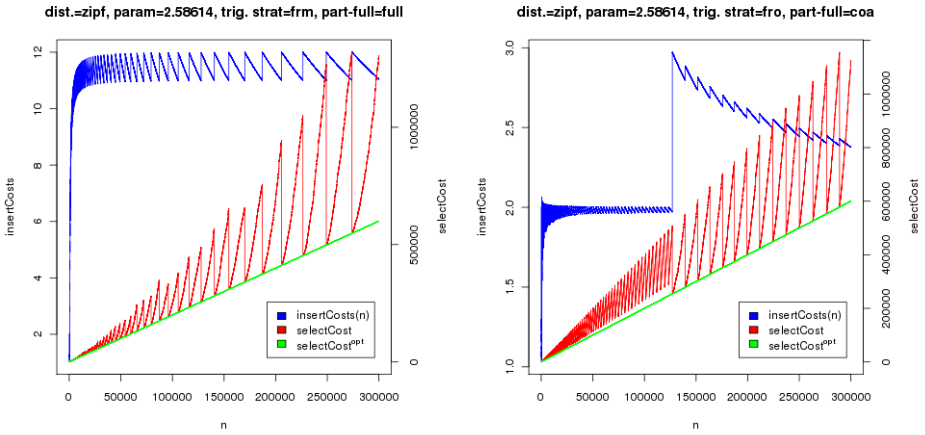
(a) frm trigger and full merge

(b) fro trigger and cost-aware merge

Fig. 4. Development of *insertCost* and *selectCost* for a zipf distributed workload.

Figures 5 a) and b) plot *insertCost* and *selectCost* for a workload sampled from a zipf distribution with a steeper shape parameter $\alpha_{Q3} = 2.58614$ as derived in Section 3, i.e., fewer frequent values and more rare values in the long tail. Since here the probability that values are already contained in D_D and D_M is higher, *insertCost* does not exceed 3 in our approach as shown in Figure 5 b), while in the default approach *insertCost* stays at 12 and is not visibly affected by the distribution as it can be seen in Figure 5 a).

To evaluate the performance of our approach for workloads sampled from uniform distributions, we plot *insertCost* and *selectCost* for a uniform distribution with 216 values, which is the $Q3$ parameter derived in Section 3. Due to the small number of distinct values, D_M is saturated quickly and a partial merge operation can insert all values into C_M, resulting in optimal *selectCost* after every merge operation as shown in Figure 6 b). The drift away from optimal *selectCost* during partial merges, as observed for workloads with lots of distinct values in Figure 4 b) and 5 b) cannot be observed here. Write costs also benefit from the fact that dictionaries are stable, resulting in nearly optimal insert costs



(a) frm trigger and full merge

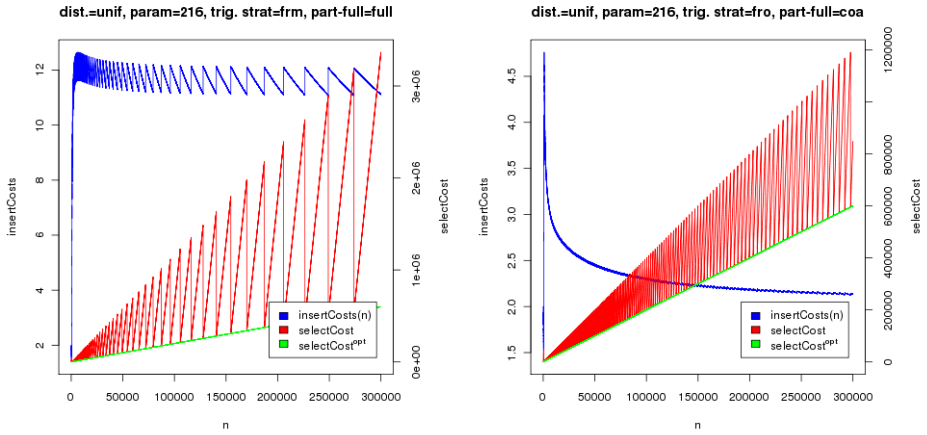
(b) fro trigger and cost-aware merge

Fig. 5. Development of *insertCost* and *selectCost* for a zipf distributed workload.

for large n with a peak of 4.7 for small n as shown in Figure 6 b), while the naive approach causes *insertCost* of maximal 12.2. *selectCost* reduce from at maximum $3e^6$ in the standard approach to $1.2e^6$ as depicted in Figure 6. To demonstrate the effect of increased value domain sizes, we plot *insertCost* and *selectCost* for a uniform distribution with 10000 distinct values in Figure 7. Here, our strategy decides on full merges frequently, resulting in higher *insertCost* for small n than in the default approach, while for large n amortized write costs are reduced to about 10 for our approach vs. 13 for the default strategy, as depicted in Figure 7 a) and b). Read costs are reduced by two orders of magnitude, while the standard approach causes *selectCost* of $1.2e^8$ at $n = 300000$, our approach results in *selectCost* of only $1.2e^6$.

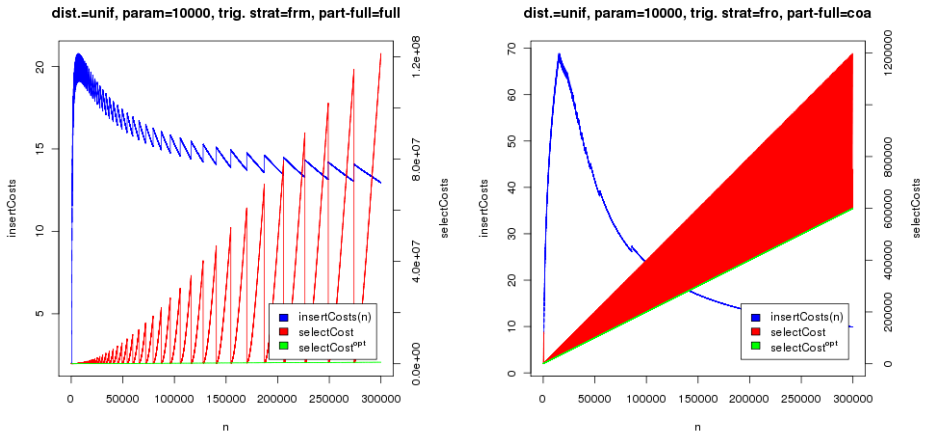
6 Conclusion

In this paper we analyze the behavior of read-optimized column-oriented in-memory databases that leverage a differential store for increasing write performance and therefore face the situation of having to merge the dictionary-compressed main and differential data structures to maintain viable read performance. After examining the complexity of the algorithms used by current databases, we analyze customer data from real-world enterprise systems and present the parameter ranges of zipf and uniform distributions of columns in the relevant tables. We build on the insight how those value distributions influence the dictionaries used for compressing the data structures and introduce the partial merge algorithm, which merges only those values of a column, which do not trigger costly re-ordering events. We integrate the partial merge algorithm into



(a) frm trigger and full merge (b) fro trigger and cost-aware merge

Fig. 6. Development of *insertCost* and *selectCost* for a uniform distributed workload.



(a) frm trigger and full merge (b) fro trigger and cost-aware merge

Fig. 7. Development of *insertCost* and *selectCost* for a uniform distributed workload with 10000 distinct values.

the cost-aware merging strategy, which adaptively triggers merge events and decides between performing a partial merge and the classic full merge algorithm with respect to defined guarantees on amortized insert costs and induced select costs. We show the potential of our cost-aware merging strategy by simulation of database operations using the parameters extracted from the analyzed customer data and comparing it to the classic approach used at present. It can be seen that our approach significantly improves the range select performance while reducing the amortized insert cost at the same time.

As part of our future work, we plan to optimize the cost-aware merging strategy with regard to the workload of the database and more accurate latency of its operators, in order to be able to find a global cost minimum or adapt to service levels for inserts or selects. This will include the analysis of additional customer data in order to add further detail to our current results. Furthermore, we want to investigate the consequences of bringing the concept to tables that may present with different distributions of distinct values for each column.

References

1. D. Abadi, S. Madden, and M. Ferreira. Integrating compression and execution in column-oriented database systems. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 671–682. ACM, 2006.
2. SAP AG. Sap business suite. http://www.sap.com/solutions/business-suite/index.epx.
3. P.A. Boncz, M. Zukowski, and N. Nes. Monetdb/x100: Hyper-pipelining query execution. In *Proc. CIDR*, volume 5. Citeseer, 2005.
4. S. Harizopoulos, V. Liang, D.J. Abadi, and S. Madden. Performance tradeoffs in read-optimized databases. In *Proceedings of the 32nd international conference on Very large data bases*, pages 487–498. VLDB Endowment, 2006.
5. Jens Krueger, Changkyu Kim, Martin Grund, Nadathur Satish, David Schwalb, Jatin Chhugani, Alexander Zeier, Pradeep Dubey, and Hasso Plattner. Fast updates on read-optimized databases using multi-core cpus. *To appear*, 2011.
6. H. Plattner and A. Zeier. *In-Memory Data Management - An inflection point*, volume 1. Springer, 2011.
7. M. Stonebraker, D.J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, et al. C-store: a column-oriented dbms. In *Proceedings of the 31st international conference on Very large data bases*, pages 553–564. VLDB Endowment, 2005.