Marcus Paradies, Christian Lemke, Hasso Plattner, Wolfgang Lehner, Kai-Uwe Sattler, Alexander Zeier, Jens Krueger

**How to juggle columns: an entropy-based approach for table compression**

**Erstveröffentlichung in / First published in:**

*IDEAS '10: Fourteenth International Database Engineering & Applications*, Montreal 16.-18.08.2010. ACM Digital Library, S. 205-215. ISBN 978-1-60558-900-8.

DOI: https://doi.org/10.1145/1866480.1866510

# How to Juggle Columns: An Entropy-Based Approach for Table Compression

Marcus Paradies
SAP AG
Walldorf, Germany
marcus.paradies@sap.com

Christian Lemke
SAP AG
Walldorf, Germany
c.lemke@sap.com

Hasso Plattner
Hasso-Plattner-Institute
Potsdam, Germany
hasso.plattner@sap.com

Wolfgang Lehner
SAP AG
Walldorf, Germany
wolfgang.lehner@sap.com

Kai-Uwe Sattler
Ilmenau University of Technology
Ilmenau, Germany
kus@tu-ilmenau.de

Alexander Zeier
Hasso-Plattner-Institute
Potsdam, Germany
alexander.zeier@hpi.uni-potsdam.de

Jens Krueger
Hasso-Plattner-Institute
Potsdam, Germany
jens.krueger@hpi.uni-potsdam.de

## ABSTRACT

Many relational databases exhibit complex dependencies between data attributes, caused either by the nature of the underlying data or by explicitly denormalized schemas. In data warehouse scenarios, calculated key figures may be materialized or hierarchy levels may be held within a single dimension table. Such column correlations and the resulting data redundancy may result in additional storage requirements. They may also result in bad query performance if inappropriate independence assumptions are made during query compilation. In this paper, we tackle the specific problem of detecting functional dependencies between columns to improve the compression rate for column-based database systems, which both reduces main memory consumption and improves query performance. Although a huge variety of algorithms have been proposed for detecting column dependencies in databases, we maintain that increased data volumes and recent developments in hardware architectures demand novel algorithms with much lower runtime overhead and smaller memory footprint. Our novel approach is based on entropy estimations and exploits a combination of sampling and multiple heuristics to render it applicable for a wide range of use cases. We demonstrate the quality of our approach by means of an implementation within the SAP NetWeaver Business Warehouse Accelerator. Our experiments indicate that our approach scales well with the number of columns and produces reliable dependence structure information. This both reduces memory consumption and improves performance for nontrivial queries.

## Categories and Subject Descriptors

H.2.8 [**Database Applications**]: Data mining

## 1. INTRODUCTION

Databases, especially those created for analytic purposes, often show an extremely large degree of correlation within the set of columns. Correlation between columns appears in a huge variety of ways and has been studied in a huge number of research projects. In this paper, we take a slightly different approach to exploiting the appearance of correlations in databases. First, we are only interested in (soft) functional dependencies between columns. Further, we are not concerned with tuning a relational query optimizer by providing accurate selectivities or a good guess of the number of distinct values over an expression with correlated columns. We worked in the context of the SAP NetWeaver Business Warehouse Accelerator (BW accelerator) engine, where our focus was on improving the efficiency of memory utilization by detecting functionally dependent columns. Before we dive into detail, we first take a more comprehensive look at the specific application problem and then show the potential of appropriately considering correlated columns within the BW accelerator.

### Where Is Our Algorithm Useful?

The BW accelerator is designed to answer analytic queries efficiently in large enterprise-sized data warehouse scenarios. In such a setup, we are faced with correlated columns (in terms of "soft" functional dependencies) in the fact tables and within the dimension tables of a data cube. By a soft functional dependency between two columns we mean a functional dependency, which holds not for all values but for a large part of values. This is a generalization of the classical hard functional dependency, where the values of one column completely determine the values in the other column. Fact tables, especially from specifically generated data marts, often contain generated key figures in a materialized form. For example, turnover figures are generated during a data mart load in order to take the computational logic out of the final OLAP system but keep it within the central loading infrastructure. Also, the

Final edited form was published in "IDEAS '10: Fourteenth International Database Engineering & Applications. Montreal 2010", S. 205-215, ISBN 978-1-60558-900-8.
https://doi.org/10.1145/1866480.1866510

same numeric values are often materialized multiple times in different currencies. Within dimension tables, we find a relatively high level of dependencies between columns, not only between classical OLAP hierarchies but especially between (logically) independent columns, such as properties of articles within a product dimension. For example, consider LCD TV models described by a large set of different features (contrast, size, resolution, reaction time, HDMI input, Scart input, PC input, and so on). In real customer data sets, we found only a very small number of different combinations of the possible feature space. For example, a size of 32 inch implies $1366 \times 768$ resolution in 86% of all 32-inch cases; dependencies also exist in the number and type of input ports, size, and resolution. Surprisingly, we also discovered dependencies in almost all other dimensions, such as shop, region, and customer. Leveraging our knowledge of these dependencies in analytic databases is of the utmost importance for a scalable and memory-efficient system.

## Why Is Our Algorithm Useful?

The BW accelerator is designed as a column store using a large variety of different compression techniques to keep all data in memory. A typical customer setup may run on 10 to 16 blades, each with two Intel Xeon quad-core processors and 16 gigabytes (GB) of main memory. The accelerator has been run on installations with up to 140 nodes. Data for the accelerator is compressed using integer coding and dictionary lookup. Integers represent the text or other values in table cells; the dictionaries are used to replace integers by their values during post-processing. In particular, each record in a table has a RowID and each value of a characteristic or an attribute in a record has a ValueID (Figure 1). Dictionary-based compression schemes greatly reduce the average volumes of processed data. Altogether, data volumes and flows are reduced in typical customer scenarios by an average factor of ten. The overall goal of this reduction is to improve the utilization of memory space and to reduce I/O within the accelerator. Finding such correlations enables us to "merge" multiple columns by storing multiple logical columns as a single physical column with only one dictionary

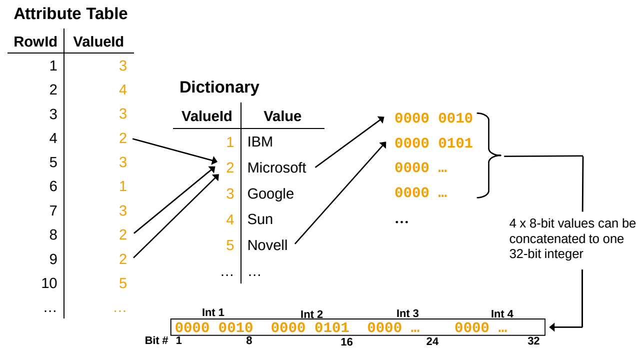

**Figure 1: Data compression using integers**

To demonstrate the effect of using column correlation to increase the compression ratio, we consider a table with 10 million rows (e.g., a fact table in an OLAP context) and look in detail at column $c_1$ with 1000 distinct string values and column $c_2$ with 16 000 distinct string values. With standard dictionary compression, the memory consumption (neglecting the dictionary, which takes 2.4 million bits in the case of an average string length of 20 bytes) computes to 10 million $*$ 10 bits (for $c_1$) $+$ 10 million $*$ 14 bits (for $c_2$) $=$ 240 million bits. Under the assumption that the columns are strongly correlated, such that we can generate a combined dictionary of 20 000 distinct values (by means of string concatenation of the combined occurrences), only one reference column has to be stored. The reference column needs 10 million $*$ 15 bits $=$ 150 million bits, so just 62.5% of the original size. The additional dictionary mapping here amounts to about 7 million bits, which is a relatively negligible increment to memory consumption.

## Contribution and Outline of the Paper

The main challenge in exploiting correlations for table compression is to identify the pairs of columns which are correlated at all. First of all, in a typical customer scenario the number of columns a data warehouse schema may easily reach 200 columns or more, resulting in $\binom{k}{2}$ possible pairs for $k$ columns. Second, because we do not restrict ourself to explicitly defined constraints or hard functional dependencies, each candidate pair has to be checked for correlations based on the actual data meaning to analyze millions of rows.

Thus, we propose a new algorithm based on sampling, entropy estimation, and pruning, which outperforms the existing algorithms by factors and scales to very large databases. For each column pair and every value pair, it collects frequencies and estimates single and combined entropy based on this frequency information. If the entropy correlation coefficient does not exceed a given limit, pruning takes place and the given column pair is ignored. The output of the algorithm is a dependency graph, which is a model of the dependency structure of the underlying database, also known as a Bayesian network graph [6].

We present a deep integration of our algorithm into a main memory column store und show that this integration results in a significant performance improvement over standard algorithms. By utilizing the different compression schemes, such as standard dictionary coding, block coding, or run-length coding, the algorithm shows better performance on average.

We propose a new compression schema for leveraging the calculated dependency graph. We demonstrate that usage of the dependencies results in a better compression ratio, up to factor 2, with performance improvements on nontrivial queries.

Based on this novel core idea, the paper illustrates aspects of implementing this algorithm efficiently within a column store database engine. We first highlight some measures, e.g., entropy and coentropy which we use in our approach. Then we present our core algorithm with additional optimizations, e.g., pruning heuristics and optimized entropy calculation. Finally, we show the results of experiments using real customer data as well as a synthetic data set. The paper concludes with indications of further work towards an improved physical database design for column stores.

## 2. MATHEMATICAL BACKGROUND

Before describing in detail the core of the algorithm and optimization techniques, we review some fundamental prerequisites for our column pairing approach. In particular, we review the basic characteristics of the entropy measure and introduce the notions of coentropy and entropy estimation.

### Entropy

The use of entropy in statistics has its origin in information theory as introduced by Shannon [16]; for details of the relationship between information theory and statistics, see [11]. In general, entropy is used in statistics as a quantitative measure of the distribution of a variable. For a random variable $X$, the information about the distribution of $X$ is determined by the frequency with which each value occurs. The entropy of a variable $X$ reflects the degree of dispersion in its distribution. For a given variable $X$ with distribution $(p_1, p_2, \dots, p_n)$ the entropy $H$ is defined by the expression:

$$H_X = -\sum_{i=1}^{n} p_i \cdot \log p_i \qquad (1)$$

In equation (1), $p_i$ stands for the probability of a given value to occur in the data set. These probabilities can be calculated by dividing the frequencies from the contingency table by the population size. Further, $H$ is always nonnegative and has its maximum value for a uniform distribution ($p_1 = p_2 = \dots = p_n = 1/n$) and its minimal value if and only if some $p_i = 1$. Figure 2 clarifies this for the case of two given probabilities. As we can see, the entropy gets its maximal value if both probabilities are 0.5 and reaches its minimal value if and only if the probabilities are 0 and 1, respectively.

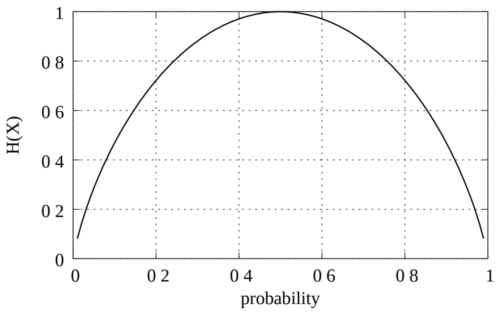

**Figure 2: Entropy curve for probabilities $p$ and $1-p$**

## Coentropy

The entropy measure is not restricted to single variable distributions but can be generalized to yield information about bivariate distributions. The combined entropy, coentropy [1], is not only a measure of the dispersion but also provides information about the mutual dependence of two variables [17]. The coentropy of a two-dimensional joint distribution $(p_{11}, \dots, p_{rc})$ where $r$ denotes the number of rows in a given contingency table and $c$ refers to the number of columns, is defined as:

$$H_{XY} = -\sum_{i=1}^{r}\sum_{j=1}^{c} p_{ij} \cdot \log p_{ij} \qquad (2)$$

In our discussion, we use the following relationships between the entropy and coentropy measure.

**Additivity:** If two variables $X$ and $Y$ are independent, then the coentropy of these variables is less than or equal to the sum of the local entropy measures:

$$H_{XY} \leq H_X + H_Y \qquad (3)$$

**Lower Bound:** The co-entropy value of the variables $X$ and $Y$ is always greater than or equal to the larger local entropy measure:

$$H_{XY} \geq \max\{H_X, H_Y\} \qquad (4)$$

## Mutual Dependence Information

We exploit the dependence information $I_{XY}$ of two variables $X$ and $Y$, which was introduced by [17]. The dependence information is based on the entropy measures and defined as follows::

$$I_{XY} = H_X + H_Y - H_{XY} \qquad (5)$$

$I_{XY}$ is nonnegative, and is zero if and only if $X$ and $Y$ are independent. Another interpretation for the mututal dependence information is highlighted in Figure 3. As we can see, $I_{XY}$ is the intersection of the single entropies $H_X$ and $H_Y$ and $H_{X,Y}$ is the complement of $I_{XY}$. If the entropies $H_X$ and $H_Y$ do not intersect, then $X$ and $Y$ are independent. On the other hand, consider the case that $H_X$ is a complete subset of $H_Y$. Then the set $H_{XY} = H_Y$ and $I_{XY}$ is maximal. For this case, we have a complete dependency, where each value in $Y$ uniquely determines a value in $X$. Based on the notion of dependency information, Theil introduces a correlation coefficient called the uncertainty coefficient $U$ [17]. The main benefit of introducing the concept of information dependency between two statistical variables is that Theil's $U$ expresses a directed dependency between two variables.
The uncertainty coefficient, also called the entropy coefficient, has an information-theoretic interpretation as proportionate reduction in error. It is a measure of the percentage reduction in uncertainty in predicting the dependent variable based on knowledge of the independent variable. When $U_C = 0$, the independent variable is of no help in predicting the dependent variable. The uncertainty coefficient $U_C(X|Y)$ for predicting the variable $X$ with respect to variable $Y$ is defined as the ratio of the value of the mean dependence information for both columns and the local entropy of $Y$:

$$U_C(X|Y) = \frac{I_{XY}}{H_Y} \qquad (6)$$

The uncertainty coefficient $U_C(X|Y)$ is an asymmetric measure. Its value differs depending on which is the independent and which the dependent variable. The uncertainty coefficient $U_C(Y|X)$ with regard to the independent variable $X$ is then defined as:

$$U_C(Y|X) = \frac{I_{XY}}{H_X} \qquad (7)$$

In general, the value of the uncertainty coefficient varies from 0 to 1. The value indicates complete dependence if $U_C(X|Y) = 1$, and statistical independence if $U_C(X|Y) = 0$. We use this measure to indicate a (soft) functional dependence between two columns when we have at least one entropy coeffecent with a value near to 1. The choice of the lower bound for the uncertainty coefficient to detect a functional dependence depends on the use case. For example, in our compression test cases we set a lower bound of 0.9.

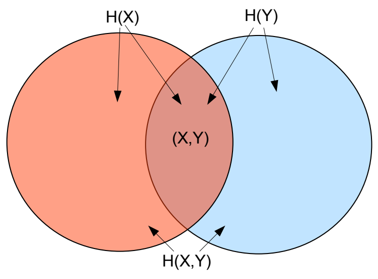

**Figure 3: Mutual dependence information**

## Entropy Estimation

To compute entropy-based measures efficiently, we rely on sampled information. In particular, we have to specify the basic sample blocks of our column signatures. Entropy estimation is a well studied area and a vast set of different estimators for different situations is available. In our setup, we relied on the Chao−Shen estimator [5], which combines the classical Horvitz−Thompson estimator [8] and the concept of sample coverage to adjust for unseen values. Further, the estimator uses Good−Turing [7] frequency estimation, which is a statistical technique for predicting frequencies in a set with unseen values. As evaluated in [18] (and also confirmed in our experiments), the Chao−Shen estimator has remarkably good statistical properties. In our scenarios, there are often many unseen value pairs, which invalidate other approaches. The Chao−Shen estimator performs reasonably well, even when a relatively large fraction of value pairs is missing in the sample. The estimated maximum likelihood (ML) frequency is defined as

$$\theta_k^{ML} = \frac{y_k}{n}$$

where $y_k$ denotes the occurrency of value $k$ and $n$ the size of the sample. On this basis, Good defined a corrected frequency measure $\hat{\theta}_k^{GT}$ (equation 8) for frequency estimation where $m_1$ denotes the number of singletons (values with occurrency 1) [7]:

$$\theta_k^{GT} = (1 - \frac{m_1}{n})\theta_k^{ML} \tag{8}$$

The Chao−Shen estimator $\hat{H}^{CS}$ for $n$ unique values calculates entropy using the Good−Turing corrected frequencies $\hat{\theta}_k^{GT}$ weighted by the probabilities for their incidence:

$$\hat{H}^{CS} = -\sum_{k=1}^{p} \frac{\hat{\theta}_k^{GT} \log \hat{\theta}_k^{GT}}{(1-(1-\hat{\theta}_k^{GT})^n)} \tag{9}$$

## Summary

We have outlined the mathematical foundations and highlighted the uncertainty coefficient, which yields a unidirectional measure of dependence between two statistical variables. Mapping statistical variables to single columns leads us to the column pairing algorithm.

## 3. COLUMN PAIRING

In this section, we outline the general column pairing (CP) algorithm that we use in our scenario to detect (soft) functional dependencies between columns. The basic idea of our solution is to rely on the correlation coefficient introduced in the previous section. For each pair of columns, we detect any dependencies and generate a dependency graph, which is the basis for the pairing step. We do not assume any specific order for the underlying data set. Moreover, we compute all quantitative measures in a single scan over (a subset) of the database. To simplify the treatment, we make the standard assumption that we know the number of distinct values for each column. Optionally, we may store local entropy values for the columns in a metadata catalog or compute them on the fly.

### Preliminaries

We define a column pair $P_k$ as a tuple $(C_i, C_j)$ with $i \neq j$ where $C_i$ denotes the $i$-th column in a given table $R$ and $C_j$ denotes the $j$-th column in $R$. The column pairs define a set $P = \{P_i \mid i = 1, \dots, m\}$ of pairs $P_i$, where $m$ is the number of possible column pairs. Also, we define $|C_i|$ as the number of distinct values in column $C_i$ and $|C_i, C_j|$ as the number of distinct value pairs in columns $C_i$ and $C_j$. The calculated set of dependence candidates $D \subseteq P$ includes all column pairs $P_i \in D$ for which a dependence has been detected.

### Entropy Computation

To compute the correlation coefficient, we scan the database and consider single columns $C_i$ and $C_j$ and pairs of columns $(C_i, C_j)$. To facilitate estimation of the entropy values, we maintain two additional data structures. As Figure 5 shows, the $dict$ structure records the frequency of each distinct value in a column $C_i$. When a new column value is scanned, the $dict$ structure is consulted and either a corresponding entry is found or a new entry is created. Also, the $freq$ structure that records the cumulative frequencies of the distinct values is updated. Since the procedure maintains a $freq$ structure for each candidate column, $C_i$ or $C_j$, and also each candidate column pair $(C_i, C_j)$, the algorithm estimates the entropy (function $ENTROPY(frequencies_{C_i})$ in algorithm 1) and coentropy $COENTROPY(frequencies_{C_i,C_j})$ using the Chao−Shen estimator.

### Creating a Dependence Graph

To construct a directed dependence graph $G$, we need to consider a potential correlation in two directions: dependence of $C_i$ on $C_j$ and/or of $C_j$ on $C_i$. We identify dependent column pairs by calculating the entropy correlation coefficient in both directions and testing whether it satisfies the condition for dependence. To test whether a detected dependence is significant (and therefore recorded as a candidate pair), we define a measure $\epsilon_2 \in [0, 1]$. As the value of $\epsilon_2$ is decreased, increasingly soft functional dependencies (which give increasingly uncertain predicted values in the second column) can be detected. For our use case of improving compression ratios, an $\epsilon_2$ near the upper bound gives the best results (in our scenarios we used $\epsilon_2 = 0.975$). If columns $C_i$ and $C_j$ are dependent, we append $P_k = (C_i, C_j)$ to our set $D$ of candidates. Algorithm 1 illustrates the basic steps to compute the correlation coefficient and check for significant dependencies.

To draw graph $G$, we now define as a node each column $C_i$ that occurs in the result set $D$ in any column pair $P_k = (C_i, C_j)$. Then we represent a given functional dependence $C_i \to C_j$ as a directed edge from node $C_i$ to node $C_j$. We can determine whether $C_i \to C_j$ or $C_j \to C_i$ holds by comparing the calculated entropy coefficients.

**Algorithm 1** Naive column pairing algorithm

```
 1: $D = \emptyset$
 2: for all $P_c \in P$ do
 3:     for $i = 0$ to $n$ do
 4:         get $val_1$ for $C_i$
 5:         get $val_2$ for $C_j$
 6:         update $freq_{C_i}$ with $val_1$
 7:         update $freq_{C_j}$ with $val_2$
 8:         update $freq_{C_i,C_j}$ with $val_1$ and $val_2$
 9:         if $|freq_{C_i,C_j}| \geq (1+\epsilon_1) \cdot \max\{|C_i|, |C_j|\}$ then
10:             break
11:         end if
12:     end for
13:     $h_{C_i}$ = ENTROPY($freq_{C_i}$)
14:     $h_{C_j}$ = ENTROPY($freq_{C_j}$)
15:     $h_{P_c}$ = COENTROPY($freq_{C_i,C_j}$)
16:     $u_{C_j,C_i} = (h_{C_i} + h_{C_j} - h_{P_c})/h_{C_j}$
17:     $u_{C_i,C_j} = (h_{C_i} + h_{C_j} - h_{P_c})/h_{C_i}$
18:     if $(u_{C_j,C_i} \geq \epsilon_2) \vee (u_{C_i,C_j} \geq \epsilon_2)$ then
19:         $D = D \cup \{P_c\}$
20:     end if
21: end for
```

Final edited form was published in "IDEAS '10: Fourteenth International Database Engineering & Applications. Montreal 2010", S. 205-215, ISBN 978-1-60558-900-8.
https://doi.org/10.1145/1866480.1866510

## Exploiting a Dependence Graph

The result dependence graph $G$ may be divided into several subgraphs (Figure 4) that are not connected to each other. Our aim is to define structures that can optimize the resulting compression ratios. Such optimal structures are paths between two or more nodes between which the dependencies are as strong as possible. To avoid storing a single column more than once, and in different ways, we choose to use each node at most once when defining a path.

As a measure for finding optimal paths, we currently use the entropy coefficients calculated by the algorithm. But this measure for assessing edges has to be adapted to several further facts, such as the number of distinct values in the columns. In some cases, to reach optimal compression rates in combination with existing smart compression methods, it may be appropriate to choose a different path for compression. To cover such cases, we need to find a more suitable measure to maximize the benefit, which is planned for future work.

Once these optimal structures have been built up, the paths derived from them can be used as a basis for further analysis (especially in case of soft functional dependencies). In the simplest case of hard functional dependencies, the algorithms gives the columns that can be compressed immediately.

## 4. MAKING THE ALGORITHM WORK

In the previous section, we have introduced an algorithm that scans over the set of column pairs and enumerates column pairs that show soft functional dependencies. This naive approach to compute the dependency degree of two columns shows an exponential runtime behavior with regard to the number of columns. To ensure scalability of the algorithm for large databases with a relatively high number of columns and make it feasible for real-world scenarios, the naive CP algorithm needs to be improved by optimization techniques and pruning heuristics. This can be achieved on two ways:

- reducing the effort for frequency computation by considering only subset of the rows,
- reducing the number of pairs which have to be taken into account during processing by applying heuristics for pruning the search space.

In the following, we discuss appropriate techniques for both approaches and study their impact in an experimental evaluation.

### Blockwise Processing

A first step to reduce the overhead of computing the entropy measures is to maintain the statistics for computing the entropy measures not after every row but after a block of 1024 entries. Batching the computation reduces the processing time significantly but does not change the number of operations and rows considered for entropy measures.

### Sampling

A second step toward a more reliable and efficient procedure is to rely on a subset of the overall database. Thus we pick only an arbitrary number of blocks and base the estimation of the entropy measures on all rows within these blocks. The sampling factor, i.e., the number of blocks to be considered for the entropy estimation, may either chosen in advance or tied to the error bound of the entropy estimator. We study both influencing factors in the evaluation section.

### Static Pruning

The prohibitive factor for the naive algorithm is the sheer number of pairing combinations to be potentially considered as dependent. Pruning the set of candidate columns is therefore of utmost importance to make the algorithms applicable for large scenarios and reduce runtime sufficiently. Our CP algorithm provides two ways to prune columns: static and online.

Static pruning relies in the given cardinality information and distinguishes two different situations where a column is no longer considered a good "driver" for a combined dictionary.

- **Case 1:** The number of distinct values of a candidate column is 1 or 2. In this case, the column storage engine provides a special compression scheme that is more efficient than a combined dictionary. Especially in analytic scenarios, these types of columns are extremely popular (e.g., to store yes/no answers) and the pruning potential is significant.
- **Case 2:** The number of distinct values of a candidate column is equal or almost equal to the number of rows in the table. Such "almost key" columns are also dismissed, because they obviously functionally determine (almost all) other columns. We defined a constant $\epsilon_3 \in [0, 1]$ in Algorithm 2 to be able to exclude this kind of "almost key" columns. Again, in the context of analytic databases, product numbers or additional identifiers are very frequent.

Algorithm 2 summarizes the static pruning strategies applied to the set of candidate columns before starting to scan a sample of the underlying database.

**Algorithm 2** Pruning column candidates

```
 1: for all $C_i \in R$ do
 2:     get distinct values for $C_i$ from dictionary
 3:     if ($|C_i|_R < 3$) then
 4:         prune $C_i$
 5:         continue
 6:     end if
 7:     if ($|C_i|_R \geq (1 - \epsilon_3) \cdot |R|$) then
 8:         prune $C_i$
 9:         continue
10:     end if
11: end for
```

### Online Pruning

In addition to the static pruning step before starting to analyze the columns, we also apply an online pruning technique to cut off the analysis of a column pair as soon as it proves not to exhibit significant dependence. The overall goal is to reduce the number of candidate pairs as quickly and as much as possible.

To illustrate the value of online pruning, we consider a small example. Assume two columns $C_1$ and $C_2$ with cardinalities $|C_1| = 10$ and $|C_2| = 13$ and table size $|R| = 10\ 000\ 000$. Given this information, we would not prune the column pair by applying the static pruning heuristics. However, we can prune the column pair as soon as the number of detected distinct value pairs exceeds $\max(|C_1|, |C_2|) \cdot \epsilon_1$, where $\epsilon_1$ is a dependence threshold to cut off analysis of soft dependencies. The maximum possible number of distinct value pairs for our example is $|C_1| \cdot |C_2| = 130$. In this case, $C_1$ and $C_2$ are completely independent. The lower bound of distinct value pairs is $\max(|C_1|, |C_2|)$. In this case, the column with $\max(|C_1|, |C_2|)$ distinct values completely determines each

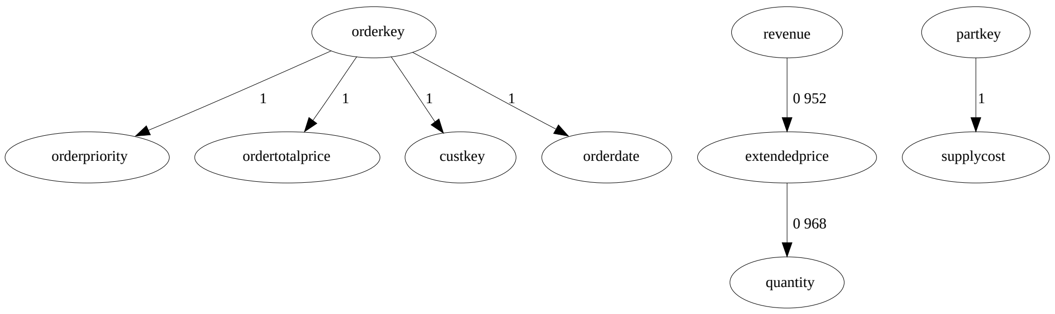

**Figure 4: Dependency graph for *lineorder***

value in the other column (which indicates a hard functional dependence). We are very careful in exploiting transitivity to more aggressively prune column pair candidates. For example, if the two correlated column-pairs $(C_1, C_2)$ and $(C_2, C_3)$ are known, we cannot defer that $(C_1, C_3)$ is also correlated. It may even be the case that $(C_1, C_3)$ shows an even larger benefit than exploiting the correlated column-pairs $(C_1, C_2)$ and $(C_2, C_3)$. For the sake of illustration, consider the following example: we have $revenue \rightarrow extendedprice$ with coefficient $0.9518$; $extendedprice \rightarrow quantity$ with coefficient $0.968$ and $revenue \rightarrow quantity$ with coefficient $0.84$. There is a near transitivity, which unfortunately cannot be exploited by the pruning strategy. Additionally with $C1 \rightarrow C2$ and $C2 \rightarrow C3$, we cannot assume that $C1 \rightarrow C3$ is also a candidate for further compression. In the above example with a threshold of $0.95$, the first $2$ column pairs are candidates, the third pair does not qualify.

## Optimized Entropy Calculation

The time required to calculate entropy measures is primarily determined by expensive (in relation to native operations like shifting or addition) logarithm calculations.

In a naive approach, we would calculate entropy terms (by term we mean a summand $p_i * \log p_i$ in the entropy equation) for each value and for each value pair, respectively. However, in our experiments we discovered that many frequencies occur several times. With the simple mapping shown in Figure 5, we avoid calculating equal terms (corresponding to the same frequencies) again and again. In the figure, the leftmost part displays $9$ rows of a column in our database. The middle part of the figure maps the values to their occurrences in the column and the rightmost part holds information about the frequency for each occurrence. For example the values $a$ and $c$ have both a frequency $2$ and in a naive approach we would calculate the same entropy value two times. The dotted lines in the figure illustrate the transition between two calculation steps. In our example the last scanned value was $b$. The frequency of $b$ is $1$ and the number of occurences with values with frequency $1$ is $2$ ($e$ and $b$). In the next step another $b$ is scanned and the frequency of $b$ increases by $1$. However, we have to decrease the number of values with occurency $1$ by $1$ and increase the number of values with occurency $2$ by $1$. The general idea behind the mapping is not to collect frequencies for each value or value pair but to count how often a frequency occurs. This reduces the number of term calculations by at least an order of magnitude.

## Exploiting Block Coded Columns

If two columns are compressed using a block coding schema, further speed improvements are possible. If a column is set to be block coded, its structure is divided into equal-sized blocks of values. This is part of the compression scheme provided by the BWA, where equal values are grouped together to compress blocks of values. If a block contains just one distinct value, this block can be marked as compressed and only its value is stored. In combination with our block sampling approach, this can speed up the algorithm in two ways. On the one hand, the costs for I/O operations are reduced and on the other hand the number of insertions for the entropy estimation are minimized to one insertion per block.

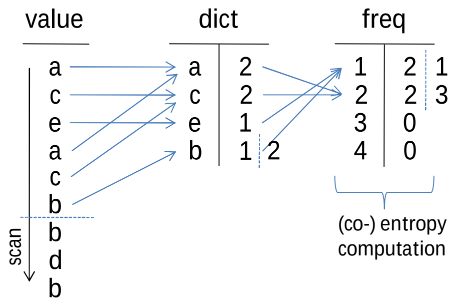

**Figure 5: Incremental computation of entropy values**

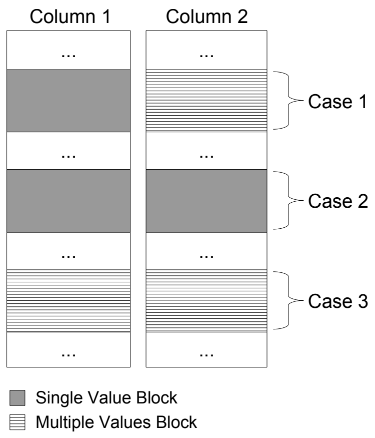

**Figure 6: Possible cases for optimizing block coded columns**

Figure 6 illustrates the cases that are possible during collecting column pair frequencies of two block coded columns. A gray shaded block is a block where only one distinct value occurs and this block will be compressed within the BWA. The striped block illustration indicates, that there are at least two different values within this block and therefore the block is not compressed. In case 1, the number of values to be read from the first column can be reduced because the block only contains one distinct value. Further, we only need to insert as many value pairs into our data structure as distinct values appear in the block of the second column. In case 1, we can gain a little more by optimizing the single entropy calculation for column 1. In case 2, we need to read only one value from each column and insert only one pair, which is a big improvement. If we have a block size of 1024, we can reduce the number of insertions into the data structure to $1/1024$. In case 3, since multiple values appear in both columns, no optimization is possible. Whether a block in a block coded column is compressed can be easily retrieved from the intern data representation of block coded columns in the BWA. There are more compression schemes in the BWA (e.g. run-length coding) which are part of the implementation. These compression schemes can be exploited in a similar way to reduce the execution time of the algorithm.

## Summary

Making the column pairing algorithm a feasible solution requires that we reduce the number of rows and the number of columns to be considered. In this section, we introduced blockwise processing and sampling steps to reduce the number of items. We also introduced two pruning heuristics: static pruning based on the number of distinct values and online pruning based on the cardinality of the column pair values. We evaluate all these factors in the following section.

## 5. EXPERIMENTAL EVALUATION

This section evaluates our algorithm against a wide range of experiments, where we highlight the main characteristics and advantages of our approach. We tested our algorithm against two different data sets: we chose a synthetic data set and a business data set from a BW accelerator customer.

- The synthetic data set (called SSB10 in the figures [14]) is derived from the TPC-H benchmark where we generated a data set based on the fact table *lineorder* with scaling factor 10. The data set consists of 17 columns and about 59 million rows.
- In contrast to the synthetic data set, the business data set (called BIZ in the figures) with 175 columns and about 10 million rows.

Given these data sets, the different optimizations like sampling and pruning heuristics can be described as follows. In our experiments, we focused on the characteristics of our pruning heuristics and compared the resulting errors under different aspects. We also examined the effect of sampling in relation both to our entropy estimations and to the pruning heuristics. In our experiments, where not otherwise noted, we used a sample size of 5% and a block size of 1024.

To illustrate the importance of pruning columns as early as possible, we show in Figure 7 the influence of the number of columns on execution time. As expected, we see an almost exponential increase with the number of columns. The time also shows a linear dependence on the size of the individual columns.

The figure suggests that a naive approach without efficient pruning heuristics will fail on business data sets with many columns. In Figure 8 we tested our business data set against our online pruning approach. We expected that with increasing sample size the algorithm would prune more column pairs. Indeed, with increasing sample size the number of observed value pairs increases and our

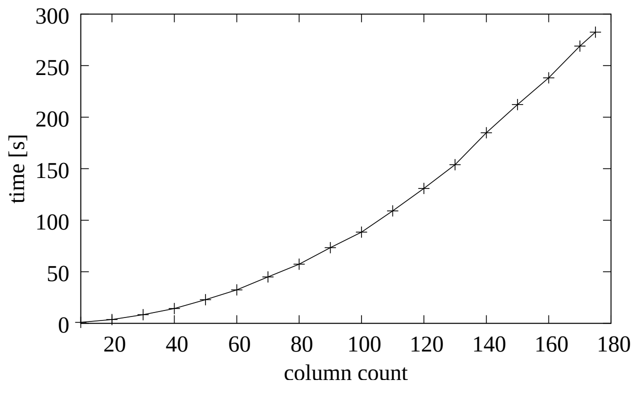

**Figure 7: Effect of the column count on the execution time (BIZ)**

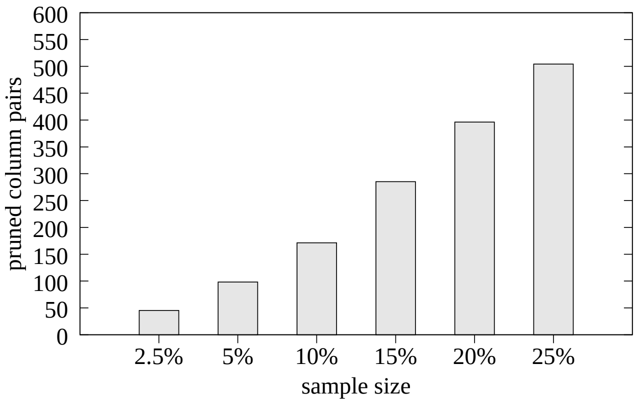

**Figure 8: Effect of the sample size on the number of pruned column pairs (BIZ)**

Final edited form was published in "IDEAS '10: Fourteenth International Database Engineering & Applications. Montreal 2010", S. 205-215, ISBN 978-1-60558-900-8.
https://doi.org/10.1145/1866480.1866510

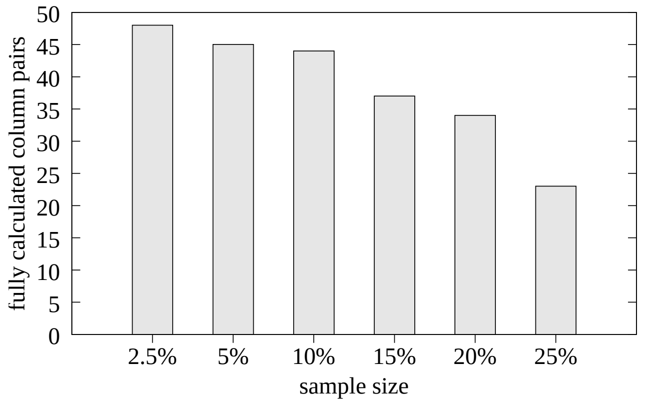

**Figure 9: Effect of the sample size on the number of complete calculated column pairs (SSB10)**

pruning condition becomes true. With the smallest sample size of 2.5%, we achieved a pruning rate of about 0.75%. This increased to 4.75% pruning rate for a 15% sample and up to 8.4% for a reduced dataset with sample size 25%.

The online pruning rate depends strongly on the dependence structure of the underlying database. If the data includes many dependencies, online pruning speeds up the algorithm only marginally. This is because we are likely to discover fewer distinct value pairs than in the case of statistical independence. If the number of distinct values in one column is larger than the sample, then a column pair that includes this column cannot be pruned.

By contrast, Figure 9 shows that online pruning can greatly reduce the search space for the synthetic data set. For this test, we focused on the number of completely calculated column pairs. We see that we can achieve high pruning rates of 60% for a relatively small sample size of 2.5%. This increases to 80% pruning for a sample size of 25%. When we compare these pruning rates with those for the business dataset, we find that the pruning rate varies with the data distribution.

To reduce the amount of data to be processed even further, we used a block sampling approach. Figures 10 and 11 show the results of our evaluation of the impact of different sample sizes for business and synthetic data sets. Independently of the data set used, we observed a linear growth of the execution time with increasing sample size. The figures also show the huge impact of using column and column pair pruning.

Figures 10 and 11 show only the final number of pruned or fully calculated column pairs after the algorithm has run.

In contrast, Figure 12 illustrates the progress of pruning and the steadily decreasing number of potentially dependent column pairs. The pruning occurred in two phases, first static pruning (which took one second in the graph) and then calculation and online pruning. As the graph shows, the number of dependent column pair candidates that remain to be processed decreases rapidly.

Another aspect that we considered in our tests is the effect of the chosen sample size on the relative error for estimating the coentropy for two columns. Figure 13 shows the results. We focused on the coentropy because this is critical in view of the combinatorial explosion in the possible number of distinct value pairs. We compared the calculated relative error on our synthetic data set to the relative error on the business data set. The relative error is the difference between the estimated coentropy and the exact coentropy divided by the exact coentropy.

We expected that the relative error decreases with increasing

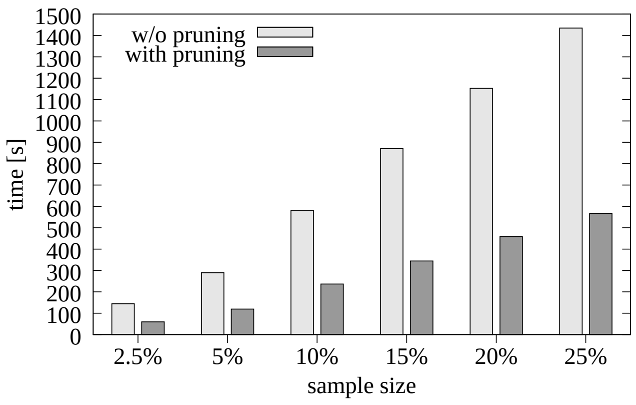

**Figure 10: Effect of the sample size and pruning on the execution time (BIZ)**

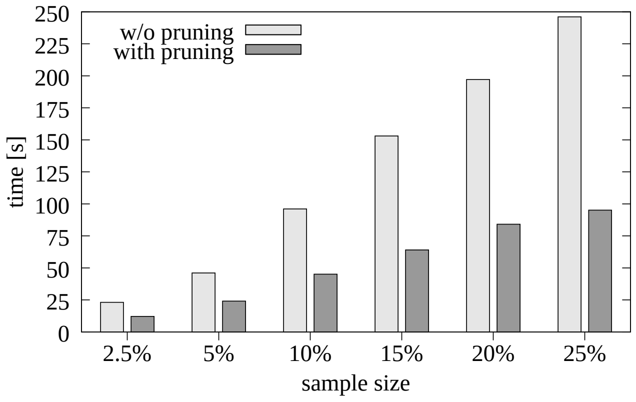

**Figure 11: Effect of the sample size and pruning on the execution time (SSB10)**

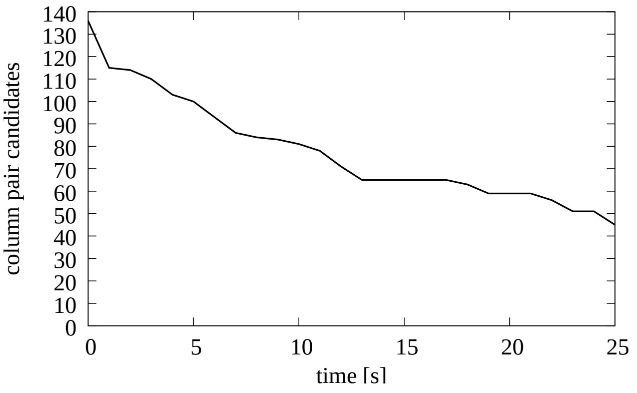

**Figure 12: Number of column pair candidates over time using pruning heuristics (SSB10)**

Final edited form was published in "IDEAS '10: Fourteenth International Database Engineering & Applications. Montreal 2010", S. 205-215, ISBN 978-1-60558-900-8.
https://doi.org/10.1145/1866480.1866510

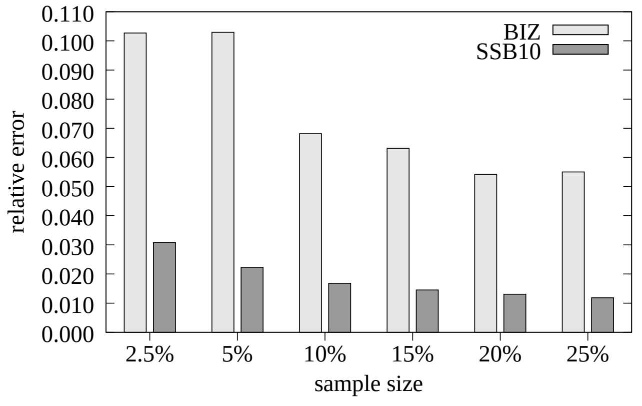

**Figure 13: Effect of the sample size on the relative estimation error**

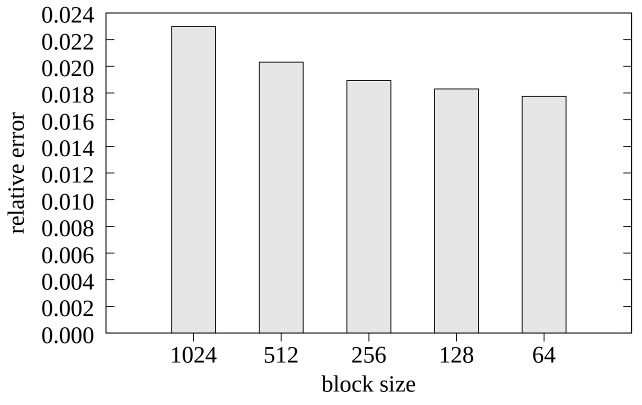

**Figure 14: Effect of the block size on the relative estimation error (SSB10)**

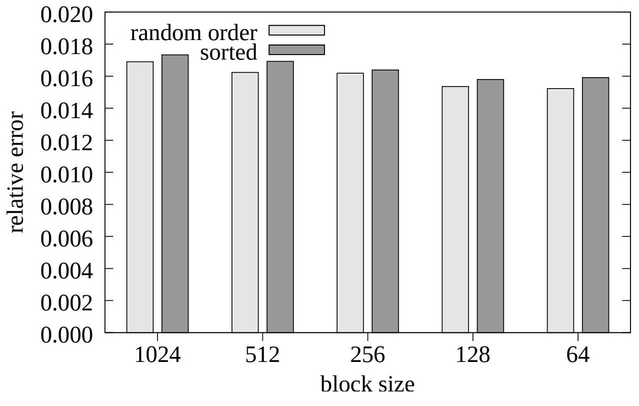

**Figure 15: Effect of the data order on the relative error with different block sizes (SSB10)**

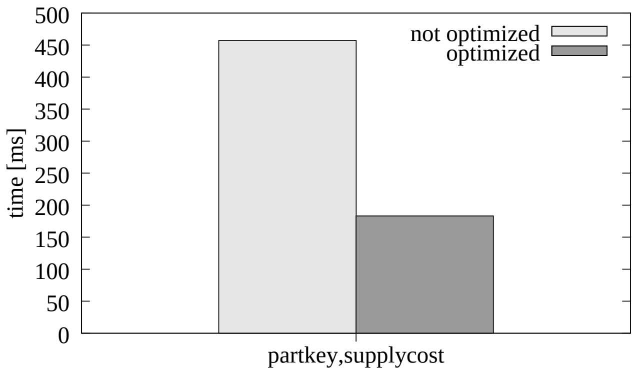

**Figure 16: Effect of block coding on execution time (SSB10)**

sample size. As we can see, the two data sets differ dramatically. Whereas the relative error for the synthetic data set is quite small, at 3% for a sample size of 2.5%, the error in estimating the coentropy for our business data set is well over three times larger. This is caused by an extremely spiky distribution, which is difficult to catch with sampling. In the case of spiky distributions, both the entropies for single columns and the coentropies for column pairs have values near zero, where the calculations are always prone to error. But, as expected, the relative error decreases for both data sets with increasing sample size. The relative error for the business data set was nearly halved for a sample size of 20%. For the synthetic data set we reduced the relative error by a factor three in relation to the error by increasing the sample size by factor ten (from 2.5% to 25%).

Another parameter that may influence the relative error in estimating the coentropy is the block size of our sample data blocks. Figure 14 shows the effect of different block sizes in relation to calculated relative errors. We see that decreasing the block size from 1024 to 64 reduces the relative error by 20%. By using a small block size, say 64, we catch on average more value pairs and can more accurately estimate the coentropy to characterize the dispersion of the joint distribution. If we choose a large block size, such as 1024, we define fewer data blocks and reduce the accuracy of our estimations. On the other hand, choosing a smaller block size results in a runtime overhead, because more data blocks need to be fetched from the database. This may conflict with other optimizations, such as for block-coded columns in the BW accelerator, where a block size of 1024 is predefined. We need to find a compromise between reducing error and reducing algorithm runtime. In the case of the BW accelerator, we chose a block size of 1024 for our sampling approach to improve the runtime at the cost of an acceptable increase in relative error when estimating the coentropy.

Figure 15 shows the results of our evaluation of the effect of the column value sort order in relation to calculated relative errors. We considered two categories: column of values in sort order and column of values in random order. In a column in sort order, the distinct values are grouped, whereas in random order they are all mixed up. We also evaluated the effect of different block sizes. As the figure shows, the relative error is generally larger for sorted data than for random data. Figure 14 also shows that the relative error decreases with decreasing block size. The difference in effect on the error between random order and sorted order is quite small, so we conclude that our algorithm reliably estimates single entropy and coentropy, independently of the structure and order of the underlying data set.

The results of our evaluation of the performance improvement for block-coded columns are shown in Figure 16. We chose a column pair with columns *partkey* and *supplycost* which are both block-coded in our test scenario. The figure highlights the runtime improvement by exploiting block coding in contrast to the case when columns are not block-coded. For the evaluated column pair, we achieved a performance improvement of factor 2.5.

To demonstrate the consequences of combining two columns physically in respect to the query performance, we performed tests

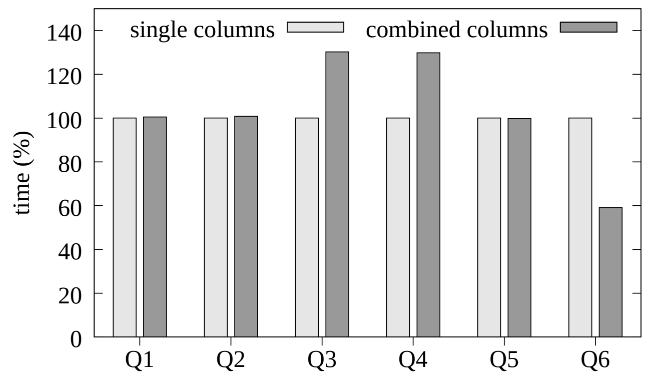

**Figure 17: Possible improvement for combined dictionaries in relation to severel queries (SSB10)**

| Columns | Original size | Reduction | |
|---|---|---|---|
| orderkey, orderpriority | 91 553 kB | 3 662 kB | (96 %) |
| orderkey, ordertotalprice | 156 160 kB | 68 270 kB | (56 %) |
| orderkey, custkey | 110 645 kB | 22 754 kB | (79 %) |
| orderkey, orderdate | 102 548 kB | 14 658 kB | (86 %) |
| revenue, extendedprice | 79 165 kB | 7 935 kB | (90 %) |
| extendedprice, quantity | 35 800 kB | 5 625 kB | (84 %) |
| partkey, supplycost | 43 904 kB | 17 147 kB | (61 %) |

**Figure 18: Reduction in space for combined columns (SSB10)**

with different search predicates using the scenario described in the introduction. We ran two queries Q1 and Q3 evaluating an equality predicate on each column seperately, two queries Q2 and Q4 with a range predicate (20 values), query Q5 concatenating two equality predicates with AND, and query Q6 concatenating two equality predicates with OR.

Because the combined dictionary can only be ordered by the values of one column, values of the other column can appear in random order. As the figure 17 shows, queries on column two (Q3 and Q4) ran slower when the combined dictionary is ordered by column one. The reason is that in this case we have to evaluate a list of values (implemented using a bit vector) which is slower than searching for one value/range. In contrast, query Q6 ran faster on the combined columns because only one scan over the data has to be performed instead of doing two searches and combining the resulting bit vectors. For query Q5 the difference is much smaller because even when using single columns only one full table scan on the more selective column is required. The remaining result rows have to be tested for the requested value of the other column. To achieve the best query performance, previous knowledge and results of query workload analyses are required. In any case, the increased efficiency of memory space utilization is the primary benefit of our approach, independently of runtime considerations.

For this reason figure 18 shows the reduction of memory space for the correlated column pairs of the synthetic data set. Using a greedy strategy for the six chosen columns a total reduction of 93 352 kB (67 percent) was achieved.

## 6. RELATED WORK

Information about data distribution and correlation of table columns has played an important role in the database field for many years. Most work has addressed this issue in the context of collecting and maintaining statistics for query optimization and can be categorized into the questions for which data statistics are needed (e.g. if the uniformity assumption is not valid), questions about how the data characteristics can be represented efficiently (e.g., in the form of histograms), and questions about when and how statistics have to be updated.

One of the main problems in exploiting database statistics is the independence assumption. This is often violated, either because functional dependencies are explicitly modeled using constraints or – more often – exist only implicitly or even not for all tuples. In recent years, this has motivated research on detecting and capturing such dependencies and correlations in databases. Here we give a brief survey of techniques that are related to our approach. We distinguish between two basic classes: approaches where the set of columns is known a priori and approaches aiming at discovering the correlated and dependent columns.

Examples of the first class are approaches based on multidimensional synopses and histograms like STHoles [4] and SIT [3] which both use query feedback for maintaining multicolumn statistics. STHoles histograms are built and refined by analyzing query results. SITs are "statistics on intermediate tables" which are constructed also for intermediate nodes in query execution plans and are general enough to be usable with multidimensional histograms, too.

In contrast, approaches of the second class aim to discover dependencies and/or correlations. Most of the existing solutions are data-centric in the sense that they analyze the base data in order to identify dependency relationships. Apart from early works on detecting strict functional dependencies in support of data modeling and reengineering (e.g. [13, 15]), several approaches have been proposed that also consider fuzzy or soft dependencies.

The goal of SASH [12] is to find the best set of histograms with respect to a given multidimensional query workload. SASH uses junction trees for representing dependency relationships between columns. A junction tree consists of cliques – related columns – whose frequency distribution is approximated by histograms. The junction trees are then used to formulate a search strategy for the histogram set that minimizes a given scoring criterion such as MDL or the error for selectivity estimation. SASH also exploits query feedback to adjust the frequencies and the overall structure incrementally.

CORDS [10] – which extends BHUNT [2] – is a sampling-based approach for correlation detection. It enumerates pairs of columns and applies a chi-squared analysis to a sample of column values in order to identify correlations. The problem of a potentially huge search space is addressed by exploiting several pruning techniques. From the set of candidate pairs, only the top-$k$ pairs with respect to their correlation strength are recommended in the column group statistics, which may then be used by the query optimizer. A crucial point in CORDS is to determine the necessary sample size for the correlation analysis. According to [10] the test for soft functional dependencies requires that the sample $S$ contains "enough information" meaning that $|S| \gg |C_i, C_j|$ which means that the sample size has to be adapted to the number of distinct value pairs. For a large number of distinct value pairs (e.g., combinations with the orderkey column in the SSB data set) this may result in wrongly detected FDs. Fig. 19 illustrates this by comparing the relative estimation error (sample of 5% size vs. exact) from CORDS to our entropy-based approach. Note, that these are only the critical cases – if the sample size can be chosen large enough, CORDS provides results which are as good as ours. On the other hand, in situations with very small entropy values ($< 0.01$) the entropy-based approach is less reliable than CORDS due to rounding errors.

Techniques from data mining have also been applied to the problem of discovering dependencies. An example of such an approach is TANE [9], which is a typical levelwise data mining algorithm

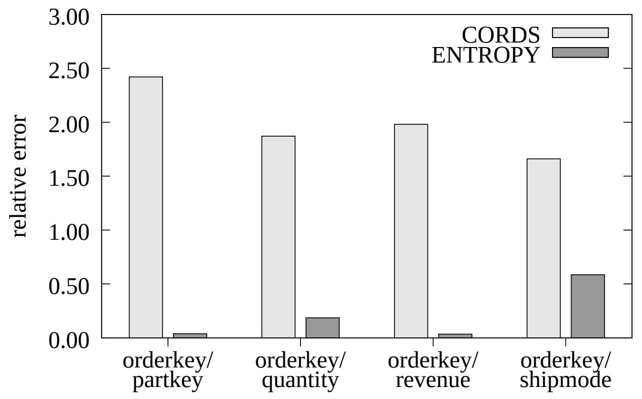

**Figure 19: Comparison of CORDS vs. entropy-based aproach**

based on a lattice representing the containment relationships of all attribute sets. TANE searches levelwise for nontrivial, minimal, fuzzy dependencies starting from the singleton set. In each step, TANE validates the dependencies and eliminates false dependencies as soon as possible. This is performed by partitioning the relation into equivalence classes, i.e., sets of tuples containing the same attribute values. Partitions can be computed from the product of earlier partitions. Like similar methods from data mining, this approach requires both a full scan on the base table and appropriate synopses for maintaining the equivalence classes.

## 7. CONCLUSION

In analytic engines using a column-based approach, such as the BW accelerator, data compression techniques greatly reduce the data volumes and thus enable more data to be held in memory for accelerating queries. Leveraging correlated columns opens the potential to compress multiple logical columns together and to store them as a single physical column with a single dictionary for encoded value pairs.

For this purpose, we have developed a novel algorithm, the column pairing (CP) algorithm, for detecting (soft) functional dependencies based on entropy measures. Our approach exploits sampling and effective pruning heuristics, is easily scalable to large databases, and can detect dependencies between pairs of columns orders of magnitude faster than using other approaches. Our experiments indicate that the CP algorithm performs even well on a database with $n > 100$ columns, where the search space for (soft) functional dependencies scales exponentially with $n$.

We have implemented the CP algorithm in the BW accelerator to enable it to deliver more fully on the advantages of column-oriented main memory databases. In further work, we shall concentrate on analyzing the dependence graph to find an improved structure for compressing detected column pairs and optimizing the mapping scheme, and hence yet further exploit the functional dependencies in the data.

## 8. REFERENCES

[1] J. Astola and I. Virtanen. Entropy correlation coefficient, a measure of statistical dependence for categorized data. Technical Report 4, Lappeenranta University of Technology, 1982.

[2] P. G. Brown and P. J. Haas. Bhunt: Automatic discovery of fuzzy algebraic constraints in relational data. In *Proc. 29th VLDB*, pages 668–679, 2003.

[3] N. Bruno and S. Chaudhuri. Exploiting Statistics on Query Expressions for Optimization. In *Proc. ACM SIGMOD 2002, Madison, WI*, pages 263–274, 2002.

[4] N. Bruno, S. Chaudhuri, and L. Gravano. STHoles: A Multidimensional Workload-aware Histogram. In *Proc. ACM SIGMOD 2001, Santa Barbara, CA*, pages 211–222, 2001.

[5] A. Chao and T.-J. Shen. Nonparametric estimation of shannon's index of diversity when there are unseen species in sample. *Environmental and Ecological Statistics*, 10:429–443, 2003.

[6] J. Cheng, D. A. Bell, and W. Liu. Learning belief networks from data: An information theory based approach. In *Proc. 6th CIKM*, pages 325–331, 1997.

[7] I. J. Good. The population frequencies of species and the estimation of population parameters. *Biometrika 40(3 and 4)*, pages 237–264, 1953.

[8] D. G. Horvitz and D. J. Thompson. A generalization of sampling without replacement from a finite universe. *Journal of the American Statistical Association*, 47(260):663–685, 1952.

[9] Y. Huhtala, J. K. P. Porkka, and H. Toivonen. Efficient Discovery of Functional and Approximate Dependencies Using Partitions. In *Proc. ICDE 1998, Orlando, FL*, pages 392–401, 1998.

[10] I. F. Ilyas, V. Markl, P. J. Haas, P. G. Brown, and A. Aboulnaga. Cords: Automatic discovery of correlations and soft functional dependencies. In *Proc. 30th VLDB*, pages 1341–1344, 2004.

[11] S. Kullback. *Information Theory and Statistics*. Wiley, 1959.

[12] L. Lim, M. Wang, and J. Vitter. SASH: A Self-Adaptive Histogram Set for Dynamically Changing Workloads. In *Proc. VLDB 2003, Berlin, Germany*, pages 369–380, 2003.

[13] H. Mannila and K.-J. Räihä. Dependency Inference. In *Proc. VLDB 1987, Brighton, England*, pages 155–158, 1987.

[14] P. O'Neil, E. O'Neil, and X. Chen. The star schema benchmark. http://www.cs.umb.edu/ poneil/StarSchemaB.PDF.

[15] J.-M. Petit, F. Toumani, J.-F. Boulicaut, and J. Kouloumdjian. Towards the Reverse Engineering of Denormalized Relational Databases. In *Proc. ICDE 1996, New Orleans, LA*, pages 218–227, 1996.

[16] C. E. Shannon. A mathematical theory of communication. *Bell Systems Technical Journal*, 27, 1948.

[17] H. Theil. On the use of information theory concepts in the analysis of financial statements. *Management Science*, 15(9):459–480, 1969.

[18] V.Q.Vu, B.Yu, and R. Kass. Coverage-adjusted entropy estimation. In *Statistics in Medicine*, volume Volume 26 Issue 21, pages 4039–4060, 2007.