Maik Thiele, Wolfgang Lehner

Shrinked Data Marts Enabled for Negative Caching

Erstveröffentlichung in / First published in:

10th International Database Engineering and Applications Symposium (IDEAS'06). Delhi, 11.-14.12.2006. IEEE, S. 148-158. ISBN 0-7695-2577-6

DOI: https://doi.org/10.1109/IDEAS.2006.41

Final edited form was published in "10th International Database Engineering and Applications Symposium (IDEAS'06)", Delhi, 2006. ISBN 0-7695-2577-6.
https://doi.org/10.1109/IDEAS.2006.41

Shrinked Data Marts Enabled for Negative Caching

Maik Thiele, Wolfgang Lehner
Dresden University of Technology
01069, Dresden
{maik.thiele,lehner}@tu-dresden.de

Abstract

Data marts storing pre-aggregated data, prepared for further roll-ups, play an essential role in data warehouse environments and lead to significant performance gains in the query evaluation. However, in order to ensure the completeness of query results on the data mart without to access the underlying data warehouse, null values need to be stored explicitly; this process is denoted as negative caching. Such null values typically occur in multi-dimensional data sets, which are naturally very sparse. To our knowledge, there is no work on shrinking the null tuples in a multi-dimensional data set within ROLAP. For these tuples, we propose a lossless compression technique, leading to a dramatic reduction in size of the data mart. Queries depending on null value information can be answered with 100% precision by partially inflating the shrunken data mart. We complement our analytical approach with an experimental evaluation using real and synthetic data sets, and demonstrate our results.

1 Introduction

Efficient processing of aggregation operations is one of the most crucial requirements within OLAP (OnLine Analytical Processing). The notion of online processing emphasizes the need of short response times for a highly interactive data exploration. Common techniques to speed up OLAP queries include the precomputation of aggregates and caching. Data marts storing pre-aggregated data, prepared for further roll-ups, play an essential role in OLAP and lead to significant performance gains in the query evaluation. However, resulting null values must be stored explicitly, since the non-existence of a tuple is a necessary, valuable piece of information [13]. It denotes the difference between a computed aggregate value which was null, i.e. no data was available and an aggregate which is not existent within the data mart, whose value is unknown. Hence, negative caching, i.e. storing null values, enables us to revise for every query whether it could be answered completely or not and avoids the round-trip to the underlying data warehouse.
Thus, we do not focus on the computation of summarized cubes; instead, we provide a summarization scheme for null values to enhance cache enabled data marts storing user-defined aggregates.

Example

We will sketch our main idea for a typical example of an OLAP query, asking for international car sales, as illustrated in Figure 1. The example table contains *sales* and *turnover* data for a particular *country*, by *month* and by *car family*, further divided by a set of features, e.g. *brand* and *color* (the numbers are fictitious). Here, symbol "*" in a dimension means that the dimension is generalized such that it matches any value in its domain. Consider the *car family* "Pickup," which is quite popular in Canada but very rarely seen in Europe. As it points out, there are a lot of sales in Canada, very few in Germany, and no sales at all in Italy for that car type. Furthermore, we see that the *brand* "Dodge" is not well established in Germany since it has no sales in "May 2006" (see tuple 7). From this fact, it can be inferred that there cannot be any sales for Dodge pickups at all in that period in Germany, independent of additional feature combinations, e.g. colors. So, by preserving the information that tuple 8 ("Dodge," "silver") and tuple 9 ("Dodge," "white") are inferable from tuple 7 ("Dodge," null), both tuples can be deleted. In the case of Italy all null tuples can be deleted which are inferable from tuples 12 and 14. In reality very few countries, if any, sell all the car brands monitored by economists. Thus, it is very likely that the number of null tuples exceeds the number of tuples which have a value for the measure of interest.

Contribution

The example from the previous section should show from a conceptual point of view that null values appear frequently

Final edited form was published in "10th International Database Engineering and Applications Symposium (IDEAS'06)", Delhi, 2006. ISBN 0-7695-2577-6.
https://doi.org/10.1109/IDEAS.2006.41

in multi-dimensional data sets and that there exist structural dependencies between them. To generalize our idea, we want to extract a set of dependency rules from a multi-dimensional data set in order to achieve high compression ratios by applying these rules. So, our notion of compression is to shrink the data mart by deleting null tuples and maintaining full ability to answer any query. In detail we make following contributions:

- We propose an algorithm which extracts a set of dependency rules from a multi-dimensional data set R. By applying these dependency rules null tuples in R can be deleted (deflation step).
- In order to guarantee that all queries relying on the null value information can be completely answered we propose a partial inflation mechanism, which uses the shrinked relation and the dependency rules to reconstruct the original null values.
- An important aspect of each compression scheme is the handling of updates. We propose some techniques to handle updates, insertions and deletions with as little work as possible, avoiding the complete de/recompression of the data.
- We conduct a comprehensive set of experiments on both real and synthetic datasets. Our experiments show very high tuple reduction ratios for uniformly distributed data as well as for highly skewed data.

Organization of This Paper

In section 2 we discuss related work. Section 3 presents our algorithm to shrink the null tuples. Section 4 discusses methods to handle updates on the shrunken data mart. The effectiveness of our algorithms, based on the experiments from both synthetic data and real data sets is analyzed in Section 5. Finally, Section 6 concludes the paper and discusses future work.

	classification attributes			feature attributes			measures	
	country	month	family	brand	color		sales	turnover
1:	Canada	May-06	Pickup	*	*		14.300	150.2 Mio
2:	Canada	May-06	Pickup	Dodge	*		3.800	43.4 Mio
3:	Canada	May-06	Pickup	Dodge	silver		1.900	21.4 Mio
4:	Canada	May-06	Pickup	Dodge	white		1.150	13.8 Mio
5:	...	...	...	...	...		...	...
6:	Germany	May-06	Pickup	*	*		255	4.8 Mio
7:	Germany	May-06	Pickup	Dodge	*		null	null
8:	Germany	May-06	Pickup	Dodge	silver		null	null
9:	Germany	May-06	Pickup	Dodge	white		null	null
10:	Germany	May-06	Pickup	VW	silver		230	4.3 Mio
11:	...	...	...	...	...		...	...
12:	Italy	May-06	Pickup	*	*		null	null
13:	...	...	...	...	...		...	...
14:	Italy	June-06	Pickup	*	*		null	null
15:	...	...	...	...	...		...	...

Figure 1. Multi-dimensional Data Set

2 Related Work

In order to reduce the size of database tables, an obvious solution is the use of traditional data compression methods which are based on statistics or dictionaries [16] [6]. Such methods are syntactic in nature, since they view a table as a large byte string. To avoid the de/recompression of the whole database during a table access, separate compression of individual tuples and individual attributes is possible. However, syntactic compression is usually not effective on small byte sets. Other compression methods which eliminate duplicate values in a database block perform better but do not consider the specific characteristics of multi-dimensional data [10].

In the domain of MOLAP, mapping-complete compression methods, such as header compression [4] and chunk-offset compression [15], which require adjacency preservation of the multi-dimensional data, are widely used.

More recently, compression techniques which take semantics of the table into consideration during compression are presented in [1, 8, 7, 9, 14]. Recent work on *Condensed Cubes* [14] and *Quotient Cubes* [8, 9] is perhaps the closest in spirit to our work. Both of this fields try to group tuples of the cube with similar aggregation values into partitions. Since we focus on aggregates with null values, this can be seen as one large partition consisting of tuples that all have the same constant value. Condensed cubes explore the properties of single tuples so that a number of tuples in a cube can be condensed into one tuple and an additional single dimension set. A simple operation can expand a condensed tuple to all the "hidden" cube tuples it represents. This is similar to our idea of seed tuples from which covered tuples can be inflated; however, we use a top-down approach to find the most general tuples and to keep the rules set as small as possible. Similar to condensed cubes, a quotient cube is a representation of a cube in terms of classes of cells. Each class contains cells that have the same aggregate value. Compression is achieved because only the lower and upper bound cells for each class have to be stored.

The notion of iceberg cubes is to compute only the group-by partitions with an aggregate value above some minimum support threshold and was first introduced by [2]. In practice, it is often difficult to decide whether an aggregate value is interesting or not, especially in our approach, where we even store aggregates with null values. The Minimum Description Length principle (MDL) has been applied in various studies to provide summaries. In the domain of OLAP applications, [3] studied the problem of finding summaries for k-dimensional cubes with tree hierarchies, where S generalizes the query results and H describes all the exceptions to the generalization. *Dwarf Cubes* are compressed data structures that can reduce the size of the data cubes by removing prefix and suffix redundancy. The degree of com-

Final edited form was published in "10th International Database Engineering and Applications Symposium (IDEAS'06)", Delhi, 2006. ISBN 0-7695-2577-6.
https://doi.org/10.1109/IDEAS.2006.41

pression is based on the density of multi-dimensional raw data [11, 12].
Database compression with data mining methods is proposed in [5]. Their contribution is to find rules in a relational database using the *Apriori Algorithm* and storing the data using these rules to achieve high compression ratios. The separation into a shrunken data set and a collection of rules from which the original state can be inferred is similar to our approach.
In spite of relationships to the work above, we do not focus on cube computation from the scratch. Instead, we are interested in shrinking user-defined multi-dimensional query results which are rich of null values in a caching scenario. Because of that background, we also focus on the handling of updates, which is one of the most crucial aspects in the domain of caching.

3 Shrinking the Data Mart

In this section, we develop algorithms to deflate the data mart as well as a compact data structure to store and search the shrunken data mart efficiently.

Preliminaries

Multi-dimensional data sets are stretched by orthogonal dimensions, which can be further divided into classification and feature attributes. Classification attributes CA_i ($i = 0, ..., n$) define a hierarchy of dimensional elements and are ordered according to their functional dependencies. In contrast, feature attributes FA_i ($i = 0, ..., n$) represent descriptive properties of the dimensional elements and solely belong to a classification hierarchy with top element "ALL."
In this paper, we will use relation $R(A, B, C, D, E, F, M)$ as a running example. R consists of six feature attributes, A to F, and one measure attribute M. The relation is filled with a number of user-defined aggregates as it is shown in Figure 2(a), where we use "*" to denote the special value "ALL." A combination of feature attributes which occurs in R is denoted as feature set or node F, i.e. $F = \{FA_1, ..., FA_n\}$. An instance of a feature set F is a tuple which solely has values for the appropriate feature attributes $FA_i \in F$. The feature sets form a lattice $(\Gamma, \prec)$, illustrated in Figure 2(b). Γ represents all occurring feature sets in a relation R whereas $\prec$ specifies the depencies between the single feature sets. $(\Gamma, \prec)$ is not a representation of the full cube, since not every feature set occurs in R, e.g. R does not contain instances of feature set $(*, B, *, *, *, *)$ abbreviated as B; hence, it is not part of the lattice. The reason is that not every feature set represents an interesting fact and therefore not every feature combination appears in the relation. The edges are defined as $F(FA_1, ..., FA_n) \prec F'(FA'_1, ..., FA'_n)$; whenever $FA'_i \neq *$, we have $FA_i = FA'_i$, for all i.

Null Ratio and Feature Dependency

In addition to the lattice structure above, we are further interested in the *null ratio* of each node.

Definition 3.1 (Null Ratio) *Given a feature set F in a relation R, $C_v(F)$ denotes the cardinality of tuples which are instances from F and having a measure value $M \neq null$ in R. Similarly, $C_n(F)$ is the cardinality of tuples which have a measure value $M = null$. Given both values, we define the null ratio ρ of a feature set F as*

$$\rho(F) = \frac{C_n(F)}{C_v(F) + C_n(F)}$$

The intuition behind this definition is the following: Consider feature set $F = (A)$ in Figure 2(b). It has two instances in relation R, one having a null value for measure M, and one having a value of 600 (tuple 1 and 2). Therefore, the null ratio of A is $\rho_A = 0.5$. Under the assumption of a uniform distribution it can be inferred that every descendant feature set of A must have a null ratio larger or equal then $\rho(A)$, e.g. $\rho(AB) = 0.83 > \rho_A = 0.5$.

Definition 3.2 (Uniform Distribution) *Uniform distribution in our context means that every instance of a feature attribute in F_i also occurs in every descendant feature set $F_j \prec F_i$.*

So, the two instances of feature set A, a_1 and a_2, should also occur in each feature set AB, AC, AD, ABC, and so on. Due to the way how multi-dimensional data is explored, this assumption is held more or less by many real data sets. However, we are not restricted to a uniform distribution; it simply represents just the best case. We will also evaluate the efficiency of our algorithms for skewed data in Section 5.
Finally, we need to define the *width* of a feature set:

Definition 3.3 (Width) *The width of a feature set F_i is defined as the number of feature attributes which occur in F_i.*

Seed Tuple Characteristics

During the previous discussion, we have shown that for a multi-dimensional relation R, a lattice structure exists whose nodes consist of the occurring feature sets in R and the appropriate null ratio ρ. This leads to our new approach, which is to reduce the size of that partition of R which consists of null values only, denoted as $R_{null} \subseteq R$. The process of shrinking R_{null} is based on so called seed tuples, which we characterize below.
Consider feature set A, which has two instances a_1 and a_2 in relation R. Their values are $null$ and 600 respectively,

Final edited form was published in "10th International Database Engineering and Applications Symposium (IDEAS'06)", Delhi, 2006. ISBN 0-7695-2577-6.
https://doi.org/10.1109/IDEAS.2006.41

	A	B	C	D	E	F	M
1	a_1	·	·	·	·	·	null
2	a_2	·	·	·	·	·	600
3	·	·	c_1	·	·	·	null
4	·	·	c_2	·	·	·	40
5	·	·	...	·	·	·	...
6	·	·	c_{10}	·	·	·	305
7	·	·	·	d_1	·	·	90
8	·	·	·	d_2	·	·	null
9	·	·	·	d_3	·	·	150
10	·	·	·	d_4	·	·	200
11	·	·	·	d_5	·	·	110
12	a_1	b_1	·	·	·	·	null
13	a_1	b_2	·	·	·	·	null
14	a_1	b_3	·	·	·	·	null
15	a_2	b_1	·	·	·	·	400
16	a_2	b_2	·	·	·	·	null
17	a_2	b_3	·	·	·	·	null
18	...	...	...	...	...	...	null
19	·	·	·	·	e_1	·	null
20	·	·	·	·	e_2	·	null

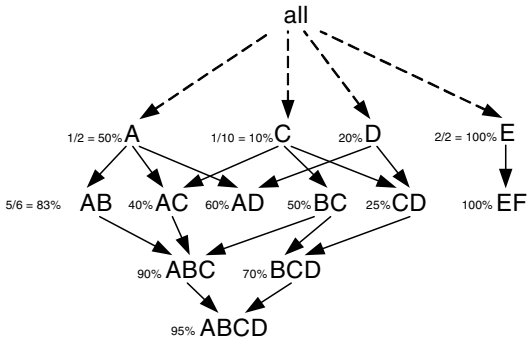


Figure 2. a) Example Relation R and b) the appropriate Feature Set Lattice

which leads to a null ratio of $\rho_A = 0.5$. Under the assumption that *Definition 3.2* holds, it can be concluded that 50 percent of all null tuples $t \in F_i \prec F_A$ must be null, too. For example, tuples $12 - 14$ in Figure 2(a) are null, since they contain a_1 as instance of feature attribute A; we denote A as seed feature set and say $(a_1, *, *, *, *, *)$ is a seed tuple because it covers $(a_1, b_1, *, *, *, *)$. Seeds are those null tuples which need to be stored physically and from which a set of other null tuples can be generated. In detail, we distinguish three groups of null tuples:

- Redundant tuples, $R_{redundant} \subseteq R_{null}$, are those tuples whose existence can be inferred from the set of seed tuples R_{seed}, and which can be deleted from R_{null}, e.g. $(a_1, b_1, *, *, *, *)$. When deleting the redundant tuples, we say R_{null} deflates to R'_{null}.

- The set of seeds $R_{seed} \subset R_{null}$, consists of those tuples from which the existence of other null tuples can be inferred and which must be kept explicitly, $R_{seed} \subset R'_{null}$.

- The outliers, $R_{outlier} \subseteq R_{null}$, are those tuples which are not covered by a seed tuple and therefore cannot be deleted; $R_{outlier} \subset R'_{null}$.

Before we discuss the required qualifications of a tuple to become a seed tuple, we need to define the *cover ratio* of a feature set. Consider the feature sets A and D in Figure 2(b). From these sets, it can be derived that 50% of all null tuples in feature set AD are covered by A and 20% are covered by D. In general, the cover ratio of a feature set F_n, covered by a set of feature sets C, is defined as follows:

Definition 3.4 (Cover Ratio) *The cover ratio $\omega(F_n, C)$ of a feature set F_n covered by a set of feature sets C, where $F_n \prec F_i$ for each $F_i \in C$ is defined to be*

$$\omega(F_n, C) = 1 - \prod_{F_i \in C} (1 - \rho(F_i))$$

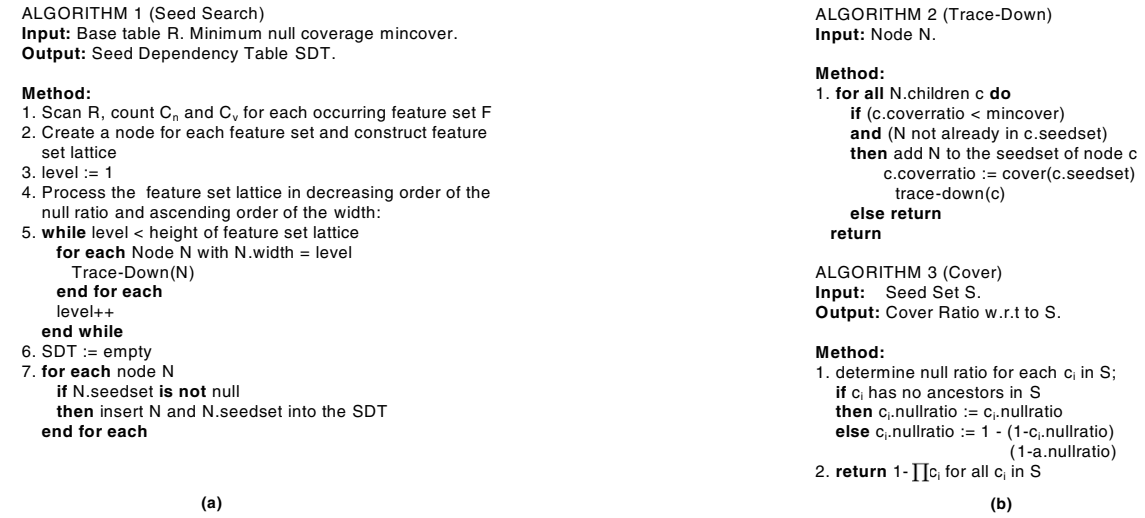
under the assumption that Definition 3.2 holds.

For the example above, this gives $\omega(F_{AD}, \{F_A, F_D\}) = 1 - (1 - 0.5)(1 - 0.2) = 0.6$, i.e. 60% of a null tuples in AD are covered by A and D. In order to maximize the compression ratio, we will use the *cover ratio* to find adequate seed feature sets. Therefore, we propose a new algorithm called *Seed Search*, which is described in detail in the following section.

Identifying Seeds

The details of *Seed Search* are shown in Figure 3(a) and (b). The first step is to determine the ratio of null tuples for each feature set. The ratios can be determined in a single scan over the base table R by counting the occurences of null C_n and non-null instances C_v of each feature set. In the second step, the feature set lattice is constructed, as described in the previous section. For each node, three pieces of information are maintained: C_n and C_v, from which the null ratio can be computed and a seed set, which is initialized as empty. The node data is stored in a relation *FeatureSetNode(featureset, C_v, C_n, seeds)* and the feature set relations in *FeatureSetEdge(featureset,parent)*.
In a next step, the lattice structure is traversed in ascending order of the width and descending order of the null ratios, i.e. nodes with a low number of features are preferred since they cover lots of other nodes due to the lattice structure. The details of the traversing can be seen in Figure 3(b). For each descendant node of N, it is checked whether its *cover ratio* already exceeds a defined limit *mincover*. If the *cover ratio* is lower than the *mincover* parameter N is added to the seed set of the actual node and the coverage of that

Final edited form was published in "10th International Database Engineering and Applications Symposium (IDEAS'06)", Delhi, 2006. ISBN 0-7695-2577-6.
https://doi.org/10.1109/IDEAS.2006.41



ALGORITHM 1 (Seed Search)
Input: Base table R. Minimum null coverage mincover.
Output: Seed Dependency Table SDT.

Method:
1. Scan R, count C_n and C_v for each occurring feature set F
2. Create a node for each feature set and construct feature set lattice
3. level := 1
4. Process the feature set lattice in decreasing order of the null ratio and ascending order of the width:
5. **while** level < height of feature set lattice
   for each Node N with N.width = level
   Trace-Down(N)
   end for each
   level++
   end while
6. SDT := empty
7. **for each** node N
   if N.seedset **is not** null
   then insert N and N.seedset into the SDT
   end for each

(a)

ALGORITHM 2 (Trace-Down)
Input: Node N.

Method:
1. **for all** N.children c **do**
   if (c.coverratio < mincover)
   and (N not already in c.seedset)
   then add N to the seedset of node c
   c.coverratio := cover(c.seedset)
   trace-down(c)
   else return
   return

ALGORITHM 3 (Cover)
Input: Seed Set S.
Output: Cover Ratio w.r.t to S.

Method:
1. determine null ratio for each c_i in S;
   if c_i has no ancestors in S
   then c_i.nullratio := c_i.nullratio
   else c_i.nullratio := 1 - (1-c_i.nullratio)
   (1-a.nullratio)
2. **return** 1-$\prod c_i$ for all c_i in S

(b)

Figure 3. (a) Algorithm Seed Search and (b) Trace-Down and Coverage

node is recomputed (Algorithm 3 in 3(b)). Otherwise, the processing of the descendant node is skipped. The traversal stops when each node exceeds the minimum coverage or all nodes in $Gamma$ have been processed. Next, the lattice is traversed a second time to extract the dependency rules between a feature set and its seeds. Each node whose seed set is not empty is inserted together with the appropriate seed set into the seed dependency table SDT.

Example Figure 4(a) illustrates *Seed Search*. The parameter minimum coverage is set to 90%, i.e. 90% of the null tuples at each node should be covered by seed tuples. Through the processing in descending order of the null ratio, the first node which qualifies for the traversing is E. The only child of node E is EF, for which E is added to its seed set. The next node is A, which covers AB, AC, AD, ABC and $ABCD$. The sixth node to be processed is AB. This node covers ABC, which is already covered by A and C with a cover ratio of $1-(1-0.5)(1-0.1) = 0.55$, which is less then minimum coverage of 0.9. The new cover ratio of ABC can not be computed by adding $(1-0.83)$ to equation above, since the 83 % of AB are partly covered by A which is already an element of the seed set of AB. Therefore, the null ratio of AB, with respect to the existing seed set S, needs to be computed:

$$\rho(AB, \{A, C\}) = 1 - \frac{(1-0.83)}{(1-0.5)} = 0.66$$

This null ratio is now used to compute the new cover ratio for ABC which is $1-(1-0.5)(1-0.1)(1-0.66) = 0.85$. The resulting seed dependencies for that example are illustrated in Figure 4(b). It should be mentioned that a seed feature set can also be covered by other seeds, e.g. AB. Furthermore, under the assumption that Definition 3.2 holds, seed sets with overlapping feature attributes, as $\{CD, C, D\}$ for BCD, are redundant, since C and D cover the same as CD. However, since real data does not always meet this requirement exactly this additional rule can enhance the overall compression ratio.

Deflate the Data Mart

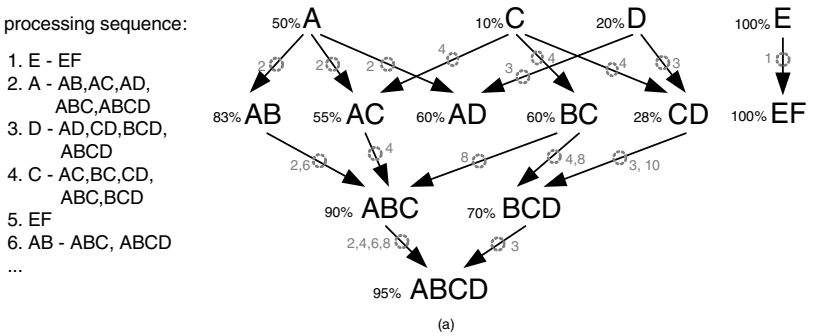
The *seed dependency table* heavily restricts the search-space of potential seed candidates, since it only stores feature sets with a high cover ratio. It allows the very fast determination of all seeds for a given tuple. The function which computes the seed tuples for a given tuple t_i is denoted as *grow*.

Definition 3.4 (Grow) *$Grow$ takes a null tuple t_i and the seed dependency table SDT as input, and produces a set of tuples coarser than the input tuple t_i such that $Grow(t_i, SDT) = FA_{out}$ and $t_i \prec t_j$ for each $t_j \in FA_{out}$.*

For example, the tuple $\{A = a_2, B = b_1, C = c_2\}$ would match the feature set ABC in the *seed dependency table* in Figure 4(b). The application of *grow* to that tuple would lead to four potential seeds $\{A = a_2, B = b_1\}$, $\{B = b_1, C = c_2\}$, $\{A = a_2\}$ and $\{C = c_2\}$. Without having the *seed dependency table*, each subset of a given feature set would need to be checked for its seed properties. For the feature set $ABCD$, that would mean that instead of the five feature sets derived from the SDT, all 14 subsets must be considered in the following deflation step.

Relation R is divided into two partitions: R_{value}, containing all tuples with a value unequal to null, and R_{null}, containing all tuples with a null value for measure M. All subsequent operations are restricted to R_{null}. The poten-

Final edited form was published in "10th International Database Engineering and Applications Symposium (IDEAS'06)", Delhi, 2006. ISBN 0-7695-2577-6.
https://doi.org/10.1109/IDEAS.2006.41



feature set	seed sets
AB	A
AC	A, C
AD	A, D
BC	C
CD	C, D
EF	E
ABC	AB, BC, A, C
BCD	BC, CD, C, D
ABCD	AB, BC, A, C, D

(b)

Figure 4. (a) Example for Identifying Seeds and (b) Resulting Seed Dependency Table

tial seeds derived from the SDT are used to deflate the aggregate table by executing the statement illustrated as query graph in Figure 5.
Relation R_{null} is joined with itself to determine whether a certain tuple is covered by a seed tuple ($t2$ and $t3$ in Figure 5). Therefore, the *grow* operation is applied to each feature set in $t3$ to determine all potential seeds for each tuple in R_{null}. Aside from the join predicate over the feature set, it must be guaranteed that only seed tuple pairs within the same classification hierarchy are considered. For example, a tuple $\{color =' white', brand =' VW'\}$ within the car family "Pickup" cannot be covered by a seed tuple $\{color =' white'\}$ at classification node "SUV." For this reason, all classification attributes CA_i of a null tuple and its potential seeds have to be equal.
The resulting tuples of $t4$ are removed from the original relation R_{null} alias $t1$ by a $MINUS$ operation. This also compensates the effect that one tuple can be covered by a set of seed tuples, i.e. that it occurs several times. Finally, the deflated relation solely contains a set of seeds and a set of outlier tuples which were not covered.

4 Deflated Data Mart - Organization and Use

The reverse operation which reconstructs the original state of relation R_{null} must be performed for every incoming query that relies on the existence of null tuples. However, the range of inflation can be restricted to those tuples which are necessary to answer the appropriate query. The corresponding statement, illustrated as query plan in Figure 6, is similar to the deflate statement of the previous section. First of all, the feature sets requested by a query Q must be stored in a temporary table called *Temp_Feature*. This table is joined with that part of the deflated table R'_{null} which fulfils the predicates of the requested query, $CA_i = Pred_i$ and $Pred_i \in Q$. By applying the *grow* operation on each feature set in $t2$, it can be verified whether a requested tuple is covered by tuples in $t1$. If a requested tuple is covered by at least one seed in R'_{null}, it follows that this tuple can be answered from the partially inflated relation R'_{null}. Aside from the covered tuples, each tuple which passes the second join condition, $t1.featureset = t2.featureset$, is part of the inflation result. These tuples are all seed or outlier tuples which could not be deflated. The additional *GROUP BY* guarantees that an inflated null tuple occurs only once, even if it is covered several times by different seed tuples or if it was covered and physically stored due to an update (see the following section). Thus, the inflate statement reconstructs all null aggregates of the original aggregate table.

Incremental Maintenance of the Shrunken Data Mart

Many applications encounter frequent updates; thus, fast maintenance of the shrunken data mart is highly important. Complete de/recompression is usually out of the question, so we propose techniques that can update the system without touching the unaffected data.
Aside updates on the shrunken data mart R', we need to maintain the seed dependency table derived from the lattice structure. That means every time an update affects the null ratio of a node FA, the appropriate counters C_n or C_v in the $FeatureSetNode$ relation need to be updated, too. Based on these values, a process running in the background reorganizes the *seed dependency table* from time to time to optimize the deflation. The reorganization will be described later on in this section.

Insertion

First, we consider the insertion of one tuple into the shrunken dataset R'. Let t_{new} be the new tuple. Two situations may happen:
Case 1: If $M(t_{new}) \neq null$, then insert t_{new} into R'_{value} and update $FeatureSetNode$, i.e. increment C_v by one.
Case 2: If $M(t_{new}) = null$, then determine whether t_{new} is covered by an existing seed:

- If t_{new} is covered, then discard t_{new} and update $FeatureSetNode$, i.e. increment C_n .

Final edited form was published in "10th International Database Engineering and Applications Symposium (IDEAS'06)", Delhi, 2006. ISBN 0-7695-2577-6.
https://doi.org/10.1109/IDEAS.2006.41

- If t_{new} is not covered, then insert t_{new} into R'_{null}, update $FeatureSetNode$, and remove all tuples in R_{null} which may covered by t_{new} as a seed tuple.

From the description above, it appears that no null tuples are inserted into the shrunken data mart R' which are covered by an existing seed tuple, i.e. only seed and outlier tuples are inserted. We denote this mode as *Instant Mode*.
Checking the coverage for a tuple to be inserted can be costly in some cases, since this requires a partial deflation of R' (see Section 5). The idea is to skip the coverage check for t_{new} and simply insert the tuple into R' – independent of its value $M(t_{new})$ – and update the $FeatureSetNode$ relation accordingly. We denote this mode as *Delayed Mode*, which performs faster than the $InstantMode$ but does not maintain the optimal deflation of R'. It is important to note that no double counts can occur due to insertions of covered tuples, since this is compensated by the inflation algorithm from the previous section.

Deletion

Next, we consider the deletion of a tuple from the shrunken dataset R'. Let t_{del} be the tuple to be deleted. Again, two situations may occur:
Case 1: If $M(t_{del}) <> null$, then delete t_{del} in R'_{value} and update $FeatureSetNode$, i.e. decrement C_v by one.
Case 2: If $M(t_{del}) = null$, then determine whether t_{del} is a seed tuple:

- If t_{del} is a seed tuple, a deletion of t_{del} implicates the deletion of all tuples covered by this seed. The semantic of deleting a tuple t_{del} is that t_{del} is not valid anymore in context of the data mart. Hence, deleting a seed tuple $\{A = a_1\}$ further causes the invalidation of all redundant tuples which it covers, e.g. $\{A = a_1, D = d_7\}$ is implicitly deleted what is correct in terms of our deletion semantic. The $FeatureSetNode$ relation is updated, i.e. the appropriate feature set for t_{del} is decremented by one. Furthermore, the null cardinality values of all feature sets which were covered by t_{del} need to be updated. Since no information about the exact cardinality of covered tuples is maintained we make use of a heuristic approach based on *Definiton 3.2*: The new null cardinality value C'_n of a feature set F_j which was covered by a deleted seed tuple t_{del} with feature set F_i, where $F_j \prec F_i$ is determined as follows:

$$C'_n(F_j) = C_n(F_j) - \frac{C_n(F_j) \cdot \rho(F_i)}{C_n(F_i)}$$

- If t_{del} is not a seed tuple, then delete t_{del} and decrement the appropriate C_n by one.

Updating the Seed Dependency Table

As mentioned in the discussion above, the $FeatureSetNode$ relation is changed during each update of the shrunken data mart, i.e. the values for C_n and C_v are changed. These changes affect the null ratio $\rho(F)$ of the appropriate feature sets, which leads to deviations from the initial null ratio distribution over the time. Since the seed dependency table is inferred from the lattice structure, which is build on $FeatureSetNode$ and $FeatureSetEdge$, these modified null ratios lead to *out-dated* as well as to *new* rules in the SDT. We will consider both cases. To differentiate between the different kinds of rules, we extend the seed dependency table with a new column called "state". Furthermore, the seeds will not be stored set-wise but individually.
We remember the *Seed Search* algorithm from the previous section. By application of this algorithm, a new seed dependency table is created, SDT', which can be different from the original SDT. A dependency rule r, which is part of the new SDT' but does not occur in the previous SDT, $r \in SDT' \wedge r \notin SDT$, is inserted into SDT and tagged as "new". If a dependency rule r is contained in both relations, $r \in SDT' \wedge r \in SDT$, this rule is marked as "current". Rules which are not up-to-date any longer, i.e. where $r \notin SDT' \wedge r \in SDT$, are tagged as "out-dated".

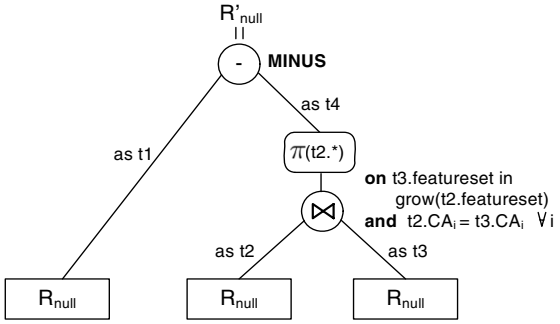


Figure 5. Deflation

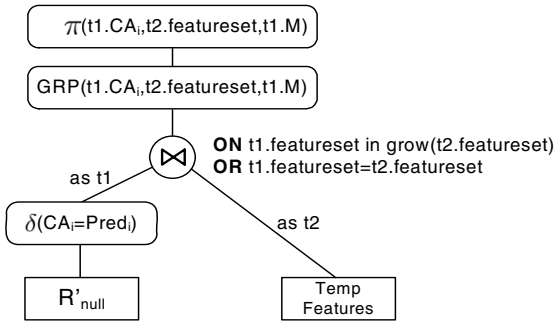


Figure 6. Partial Inflation

Final edited form was published in "10th International Database Engineering and Applications Symposium (IDEAS'06)", Delhi, 2006. ISBN 0-7695-2577-6.
https://doi.org/10.1109/IDEAS.2006.41

Incremental Reshrinking

The following deflation step can make use of the additional state information to enhance the overall efficiency. Under the assumption that updates are accomplished in *instant mode*, only dependency rules tagged as "new" need to be considered by the deflation algorithm. All rules tagged as "current" have already been considered by the incremental deflation during the previous updates. In case that updates were processed in *delayed mode*, all rules tagged as "new" or "current" have to be regarded to assure an optimal deflation of the data mart.
Out-dated dependency rules are not considered by the deflation algorithm, since their application leads to a low tuple reduction or no deflation at all. However, an out-dated rule can not be deleted from the seed dependency table as long as there exist seed tuples in R' which match this rule. The reason is that each seed tuple together with its seed dependency rules covers a set of null tuples. That means deleting a rule r from the SDT implicitly deletes all null tuples which are covered by the seeds in R'_{null} that match this rule. To avoid the overhead of too many rules in the SDT which do not contribute anything to the deflation, a rule can be deleted after it has been checked that no tuple in R' exists which belongs to that rule.

5 Performance Analysis

To assess the effectiveness of the shrinking in terms of tuple reduction as well as its effects on query performance and incremental maintenance, a set of experiments has been conducted. In this section, we present some of the results.
About the datasets In order to examine the effects of various factors on the performance of the algorithms, we generated synthetic datasets with uniform distribution as well as with skewed distribution. Referring to *Property 3.1*, a dataset with skew factor $S = 1.2$ means that 20% of every instance of a feature attribute in F_i do not occur in the child feature set $F_j \prec F_i$. The skew considers the fact that not every aggregate defined by the user was requested at each granularity . In addition, we also used a real market research dataset containing product sales at various shops and a set of product features for each article. It contains 2,874,102 tuples (340.2MB) in 11 dimensions and has no skew. The overall null ratio of the real dataset is about 99%.
All our experiments were run on an Intel Celeron 2,66 GHz machine with a memory of 1GB and running Windows 2003. There are three classes of datasets used in the experiments: The first two are synthetic datasets with uniform and skewed distribution and the third is a real dataset containing market research data. The effectiveness of shrinking can be measured by the ratio between the amount of tuples of the shrunken data mart R' and the amount of tuples of the original data mart R.

Effectiveness of Shrinking

The first set of experiments studies the effectiveness of shrinking. We report two sets of results, for synthetic and real datasets, in Figures 7(a), and 7(b), respectively. The number of tuples for the synthetic dataset was fixed to 2.5M tuples, which were randomly generated based on uniform and skewed distribution with factor 1.25 and 1.5, respectively. The cardinality of each attribute was randomly generated, too. We varied the number of dimensions, i.e. the number of feature attributes, from 1 to 10.
Several observations could be made from the results: The proposed shrinking approach can effectively reduce the amount of null tuples of a data mart. As shown in Figure 7(a), for a 10-dimensional data mart, the shrunken dataset R'_{null} contains only 0.05% of the tuples compared to the original dataset R_{null}. With increasing number of dimensions the tuple ratio becomes better since the feature set lattice gets more complex, which leads to better and more general dependency rules. As expected, the tuple ratio increases with increasing skew S, but still is 4% for $S = 1.5$ with 10 dimensions. For the real dataset the tuple ratio exceeds about 1% for 11 dimensions (Figure 7(b)). Considering the overall shrinking, the number of rules which are generated, i.e. the size of the seed dependency table SDT, can be neglected. For example, for 2.5M tuples only 226 rules need to be stored.
Furthermore, we measured the time for the shrinking against the increasing size of the data mart R and for different null ratios, as illustrated in Figure 7(c). It can be seen that the computation time increases with the cardinality of the data mart and with its increasing null ratio, which seems logically since both increase the size of R_{null}.

Query Answering Performance

In this experiment, we compared the query answering performance using the shrunken data mart R' and the original dataset R. Figures 8(a)-(c) show the result. It becomes clear that for queries which request a low number of tuples the query performance for R' is better than for R, since the cardinality of R' is much lower than the cardinality of R. With an increasing amount of tuples requested by the queries, the performance for R becomes better, since the partial deflation of R' takes increasingly longer.
However, remember our major concern, which is to store null tuples explicitly. This enables us to decide whether a query can be completely answered by a data mart without accessing the underlying data warehouse. Thus, we also compared our approach to the case that no null tuples have to be stored, i.e. a recourse to the data warehouse is necessary to determine the completeness of a query. In this case,

Final edited form was published in "10th International Database Engineering and Applications Symposium (IDEAS'06)", Delhi, 2006. ISBN 0-7695-2577-6.
https://doi.org/10.1109/IDEAS.2006.41

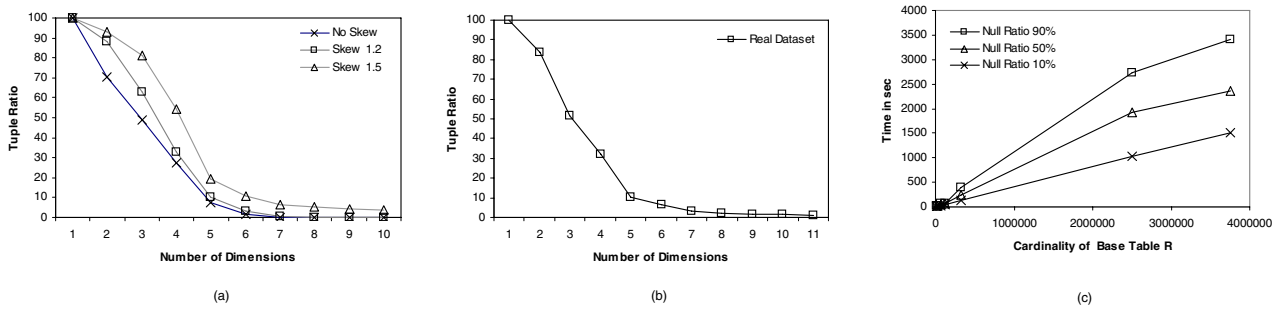


Figure 7. Evaluating the Tuple Ratio on (a) Synthetic Data, (b) Real Data and (c) Runtime

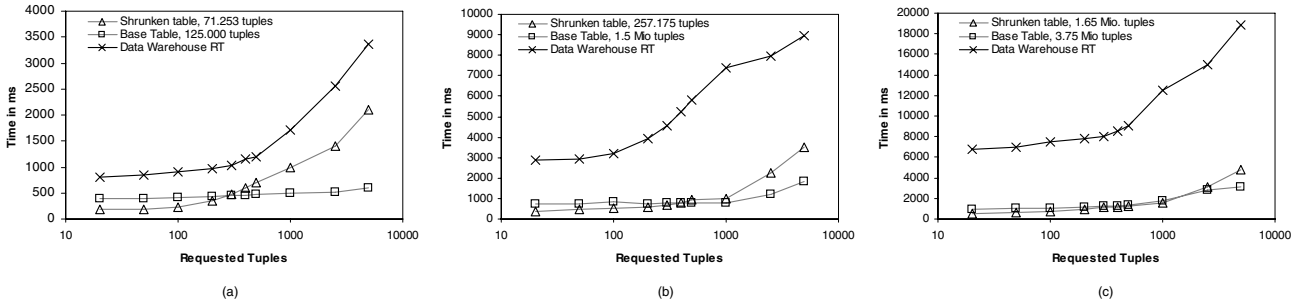


Figure 8. Query Performance for (a) $|R| = 0.125\ Mio$ **(b)** $|R| = 1.5\ Mio$ **(c)** $|R| = 3.75\ Mio$

the query performance on the shrunken data mart is much better, as illustrated in Figures 8(a)-(c).

Maintenance of the Shrunken Data Mart

The last set of experiments studies the effectiveness of maintenance, i.e. insertions, deletions and updates. To test the scalability of the incremental maintenance of the shrunken data mart, we fixed the number of tuples in the base table R to 2.5M with a null ratio of 90%. The tuples to be maintained are randomly generated and separated into three classes of the following proportion: 90% are null tuples, from which 40% are covered and 10% are seed tuples. The remaining tuples either have a value or are null tuples which are not considered by the actual seed dependency table.

In Figure 9(a), we compared the performance of insertions for an increasing number of inserted tuples. It turns out that inserting tuples in the *Delayed Mode* causes the best performance due to the smaller size of R' compared to R and the avoidance of the partial inflation. Insertions into shrunken datasets also depend on the overall null ratio of the base table R. For lower null ratios, e.g. 50%, insertions perform better than on the base table R, whereas for higher null ratios, e.g. 90%, the insertions perform slower. This is due to the inflation step denoted in Section 4, during which all tuples are considered that are covered by a seed. Since a tuple can be covered several times, the generated data space, multiple increases with growing R'_{null}.

The costs of the insertion steps are compared with each other in Figure 9(b). We can see that with an increase in the number of inserted tuples, the costs of the step to determine whether an inserted tuple is covered, dominates the overall costs with about 85%. This is different to the computation of tuples which are covered by inserted seeds. Since, for our experiments, only 10% of all null tuples are seeds, these costs are comparatively small. The computation of whether or not a tuple is a seed remains constant, since this only requires the seed dependency table, which is very small compared to the size of R'.

Finally, we considered the scalability for deletions as illustrated in Figure 9 (b). In order to delete tuples from the shrunken data mart, it has to be determined whether these tuples are seeds or not. As mentioned in the discussion above, the computation of the seed affiliation is very fast. Hence, deletions from the shrunken data mart perform much better because of the lower cardinality of R' in comparison with R.

6 Summary

The technique of negative caching as a promising approach to support efficient on-line analytical processing has

Final edited form was published in "10th International Database Engineering and Applications Symposium (IDEAS'06)", Delhi, 2006. ISBN 0-7695-2577-6.
https://doi.org/10.1109/IDEAS.2006.41

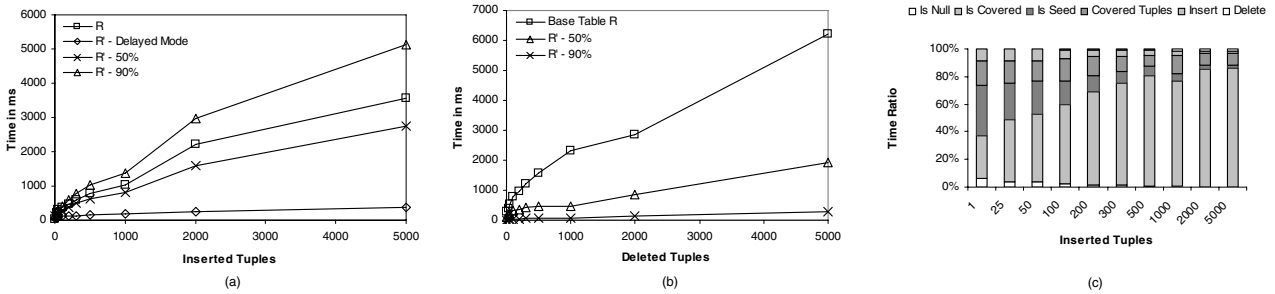


Figure 9. Performance of (a) Insertion (b) Deletion and (c) Distribution of the Multiple Insertion Steps ($|R| = 2.5\ Mio$)

recently been proposed in [13]. It allows us to ensure the completeness of queries to the data mart without accessing the underlying data warehouse. However, the application of this technique leads to a high amount of null tuples, which easily exceed the number of regular tuples in sparse datasets.

In this paper, we proposed a new approach to efficiently shrink the size of such sparse multi-dimensional datasets. The approach explores the properties of so-called seed tuples; these seeds are to be used to deflate a number of tuples without loss of information. A simple grow operation can inflate a seed tuple to all the null tuples it covers. Our approach is compact and elegant in that the only information kept is a small set of rules from which all covered tuples can be inferred. A set of algorithms for query answering and incremental maintenance of the shrunken data mart is also presented.

Experimental results indicate that the tuple reduction ratio is very high for synthetic as well as for real data. Even highly skewed data achieve a reduction rate of 90%. The strong tuple reduction also has positive effects on the update performance, which is noticeably faster in instant mode and significantly faster in delayed mode.

To summarize, we believe that negative caching in conjunction with our shrinking framework specified in this paper is essential to enrich data marts with completeness information, which leads to a much more efficient overall on-line analytical processing.

References

[1] S. Babu, M. Garofalakis, and R. Rastogi. SPARTAN: A Model-based Semantic Compression System for Massive Data Tables. In *SIGMOD 2001*, pages 283–294, New York, NY, USA, 2001. ACM Press.

[2] K. Beyer and R. Ramakrishnan. Bottom-up Computation of Sparse and Iceberg CUBE. In *SIGMOD 1999*, pages 359–370, New York, NY, USA, 1999. ACM Press.

[3] S. Bu, L. V. S. Lakshmanan, and R. T. Ng. MDL Summarization with Holes. In *VLDB 2005*, pages 433–444. VLDB Endowment, 2005.

[4] S. J. Eggers, F. Olken, and A. Shoshani. A Compression Technique for Large Statistical Data-Bases. In *VLDB 1981*, pages 424–434. IEEE Computer Society, 1981.

[5] C.-L. Goh, K. Aisaka, M. Tsukamoto, and S. Nishio. Database Compression with Data Mining Methods. pages 177–190, 2000.

[6] D. A. Huffmann. A Method for the Construction of Minimum-Redundancy Codes. *Proc. Inst. Radio Eng. 40*, pages 1098–1101, 1952.

[7] H. V. Jagadish, R. T. Ng, B. C. Ooi, and A. K. H. Tung. It-Compress: An Iterative Semantic Compression Algorithm. In *ICDE 2004*, page 646, Washington, DC, USA, 2004. IEEE Computer Society.

[8] L. V. S. Lakshmanan, J. Pei, and J. Han. Quotient Cube: How to Summarize the Semantics of a Data Cube. In *VLDB 2002*, pages 778–789, 2002.

[9] L. V. S. Lakshmanan, J. Pei, and Y. Zhao. Qc-Trees: An Efficient Summary Structure for Semantic Olap. In *SIGMOD 2003*, pages 64–75, New York, NY, USA, 2003. ACM Press.

[10] M. Pöss and D. Potapov. Data Compression in Oracle. In *VLDB*, pages 937–947, 2003.

[11] Y. Sismanis, A. Deligiannakis, N. Roussopoulos, and Y. Kotidis. Dwarf: Shrinking the PetaCube. In *SIGMOD 2002*, pages 464–475, 2002.

[12] Y. Sismanis and N. Roussopoulos. The Dwarf Data Cube Eliminates the High Dimensionality Curse, 2003.

[13] M. Thiele, J. Albrecht, and W. Lehner. Optimistic Coarse-Grained Cache Semantics for Data Marts. In *SSDBM 2006*, 2006.

[14] W. Wang, H. Lu, J. Feng, and J. X. Yu. Condensed Cube: An Effective Approach to Reducing Data Cube Size. In *ICDE 2002*, 2002.

[15] Y. Zhao, P. M. Deshpande, and J. F. Naughton. An Array-based Algorithm for Simultaneous Multidimensional Aggregates. pages 159–170, 1997.

[16] J. Ziv and A. Lempel. A Universal Algorithm for Sequential Data Compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.