ATTRIBUTE VALUE REORDERING FOR EFFICIENT HYBRID OLAP

OWEN KASER AND DANIEL LEMIRE

ABSTRACT. The normalization of a data cube is the process of choosing an ordering for the attribute values, and the chosen ordering will affect the physical storage of the cube's data. For large multidimensional arrays, proper normalization can lead to more efficient storage in hybrid OLAP contexts that store dense and sparse chunks differently. We show that it is NP-hard to compute an optimal normalization even for 1×3 chunks, although we find an exact algorithm for 1×2 chunks. When attributes are nearly statistically independent, we show that an optimal normalization is given by dimension-wise attribute frequency sorting, which can be done in time $O(dn\log(n))$ for data cubes of size n^d. When attributes are not independent, we propose and evaluate a number of heuristics.

Our optimized hybrid OLAP storage mechanism was observed to be 44% more storage efficient than ROLAP and the gains due to normalization alone accounted for 45% of this increase in efficiency.

Data Cubes, Normalization, Chunking, Multidimensional Binary Arrays, OLAP, MOLAP, HOLAP

1. INTRODUCTION

On-line Analytical Processing (OLAP) is a database acceleration technique used for deductive analysis [Goi99]. The main objective of OLAP is to have constant-time or near constant-time answers for many typical queries. For example, in a database containing salesmen's performance data, one may want to compute on-line the amount of sales done in Ontario for the last 10 days, including only salesmen who have 2 or more years of experience. Using a relational database containing sales information, such a computation may be expensive. Using OLAP, however, the computation can typically be done on-line. To achieve such acceleration one can create a *cube* of data, a map from all attribute values to a given measure. In the example above, one could map tuples containing days, experience of the salesmen, and locations to the corresponding amount of sales.

We distinguish between two types of OLAP engines: Relational OLAP (ROLAP) and Multidimensional OLAP (MOLAP). In ROLAP, data is itself stored in a relational database whereas with MOLAP, a large multidimensional array is built with the data. In MOLAP, an important step in building a data cube is choosing a *normalization*, which is a mapping from attribute values to the integers used to index the array. One difficulty with MOLAP is that the array is often sparse. For example, not all tuples (day, experience, location) would match sales. Because of this sparseness, ROLAP can be far more efficient in terms of storage. Additionally, there are compression algorithms to further decrease ROLAP storage requirements [DER01, NR97, SDRK02]. On the other hand, MOLAP can be much faster, especially if subsets of the data cube are dense [ZDN97]. Many vendors such as Speedware and Microsoft are thus using Hybrid OLAP (HOLAP), storing dense regions of the cube using MOLAP and storing the rest using a ROLAP approach. In other words,

Date: October 23, 2003.

In DOLAP'03, New Orleans, Louisiana, November 7, 2003. NRC 46510.

	<1 yrs	1-2 yrs	>2 yrs
Ottawa			$732
Toronto			$643
Montreal			$450
Halifax	$43	$54	
Vancouver	$76	$12	

	<1 yrs	1-2 yrs	>2 yrs
Halifax	$43	$54	
Montreal			$450
Ottawa			$732
Vancouver	$76	$12	
Toronto			$643

TABLE 1. Two tables representing the volume of sales for a given day by the experience level of the salesmen. Given that three cities only have experienced salesmen, some orderings (top) will lend themselves better to efficient storage (HOLAP) than others (bottom).

a convenient and efficient representation of a sparse data cube is achieved when the dense regions are stored as multidimensional arrays and the sparse remainder is represented as a list of values as in a relational database.

While various efficient heuristics exist to find dense sub-cubes in data cubes [CZK+99, CZK+01, Kas02b], one problem we still face is that the dense sub-cubes are normalization-dependent, so that the same data with attribute values ordered differently may have completely different dense sub-cubes and may be stored significantly more efficiently[DRSN98]. A related problem with MOLAP or HOLAP is that the attribute values may not have a canonical ordering, so that the exact representation chosen for the cube is arbitrary. In the salesmen example, imagine that "location" can have the values "Ottawa", "Toronto", "Montreal", "Halifax", and "Vancouver". How do we order these cities: in terms of population, latitude, longitude, or alphabetically? Consider the example given in Table 1: it is obvious that HOLAP performance will depend on the normalization of the data cube. We also believe that a storage-efficient normalization may often lead to better query performance.

One may object that normalization only applies when attribute values are not regularly sampled numbers. One argument against normalization of numerical attribute values is that storing an index map from these values to the actual index in the cube amounts to extra storage. Consider a data cube with n attribute values per dimension: we say such a cube is *regular* or *n-regular*. The most naive way to store such a map is for each possible attribute value to store a new index as an integer from 1 to n. Assuming that indices are stored using $\log n$ bits, this means that $n \log n$ bits are required. However, array-based storage of a regular data cube uses $\Theta(n^d)$ bits. In other words, unless $d = 1$, normalization is not a noticeable burden and all dimensions can be normalized. The case $d = 2$ is similar to image compression, where reordering pixels is a widely used technique [NM80]. For high dimensional data cubes, the possible gains are substantial.

1.1. **Contributions and Organization.** The contributions of this paper include a detailed look at the mathematical foundations of normalization, including notation for the remainder of the paper and future work on normalization of block-coded data cubes (Sections 2 and 3).

Section 4 then considers the computational complexity of normalization. If data cubes are stored in tiny (size-2) blocks, an exact algorithm can compute the best normalization, whereas for larger blocks, it is conjectured that the problem is NP-hard. As evidence, the case of size-3 blocks is shown NP-hard. An important class of "slice-sorting" normalizations is investigated in Section 5. These normalizations can be efficiently calculated, but the quality of their solutions is sometimes poor. Using a notion of statistical independence, a major contribution (Theorem 5.1) is an easily computed approximation bound for a heuristic called "Frequency Sort", which we show works well when the cube possesses enough independence. Section 6 discusses additional heuristics that could be used when the cube is not sufficiently independent. In Section 7, experimental results compare the performance of heuristics on a variety of synthetic and "real-world" data sets. The paper concludes with Section 8.

Due to space constraints, many proofs and details have been deferred to the full paper [KL03].

2. Block-Coded Data Cubes

In the rest of this paper, d is the number of dimensions (or attributes) of the data cube C and n_i, for $1 \leq i \leq d$, is the number of attribute values for dimension i. Thus, C has size $n_1 \times \ldots \times n_d$. To be precise, we distinguish between the *cells* and the *indices* of a data cube. "Cell" is a logical concept and each cell corresponds uniquely to a combination of values $(v_1, v_2, \ldots, v_d)$, with one value v_i for each attribute i. In Table 1, one of the 15 cells corresponds to (Montreal, 1-2 yrs). *Allocated* cells, such as (Vancouver, 1-2 yrs), store measure values, in contrast to unallocated cells such as (Montreal, 1-2 yrs). From now on, we shall assume that some initial normalization has been applied to the cube and that attribute i's values are $\{1, 2, \ldots n_i\}$. "Index" is a physical concept and each d-tuple of indices specifies a storage location within a cube. At this location there is a cell, allocated or otherwise. *(Re-) normalization changes neither the cells nor the indices of the cube; (Re-)normalization changes the assignment of cells to indices.*

We use $\#C$ to denote the number of allocated cells in cube C. Furthermore, we say that C has *density* $\rho = \frac{\#C}{n_1 \times \ldots \times n_d}$. To support exact answers for queries, we seek an efficient storage mechanism to store all $\#C$ allocated cells.

There are many ways to store data cubes using different coding for dense regions than for sparse ones. For example, in [Kas02b] a single dense sub-cube (chunk) with d dimensions is found and the remainder is considered sparse. It is also possible to use a broader class of regions.

To determine the best storage strategy for data cubes, we have considered image compression, where simple blocks are used in many image compression formats including JPEG [DVDD98]. While some attempt to improve the current formats by dividing images into arbitrarily shaped regions through adaptive algorithms [PM00], others provide evidence that non-adaptive algorithms suffice [CD99]. It is not clear *a priori* that more complex shapes lead to more efficient storage. Another argument for block-coded data cubes is that many efficient buffering schemes for OLAP range queries rely themselves on block coding [GAAS99, Lem02].

We follow [Goi99, SS94] and store the data cube in *blocks*[1], which are disjoint d-dimensional sub-cubes covering the entire data cube. We consider blocks of constant size $m_1 \times \ldots \times m_d$; thus, there are $\lceil \frac{n_1}{m_1} \rceil \times \ldots \times \lceil \frac{n_d}{m_d} \rceil$ blocks. For simplicity, we usually assume

[1] Many authors use the term "chunks", but it seems that term does not mean exactly the same thing to each author.

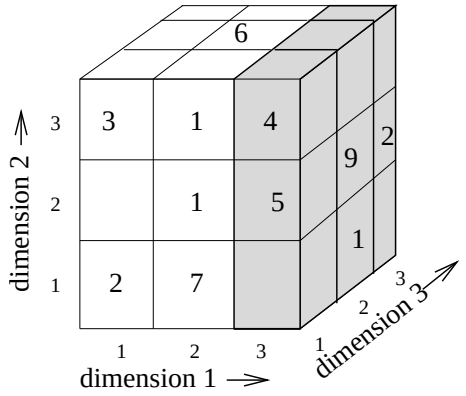

FIGURE 1. A $3 \times 3 \times 3$ cube C with the slice C_3^1 shaded.

that m_k divides n_k for all $k \in \{1, \ldots, d\}$. Each block can then be stored in an optimized way depending, for example, on its density. Rather than considering various classes of compression schemes [LS02], we will consider only the two most widely used coding schemes for data cubes, corresponding respectively to simple ROLAP and simple MOLAP. That is, either we represent the block as a list of tuples, one for each allocated cell in the block, or else we code the block as an array. For both extreme cases, a very dense or a very sparse block, MOLAP and ROLAP are respectively *efficient*.

Assuming that a data cube is stored using block encoding, we need to estimate the storage cost. A simplistic model is given as follows. The cost of storing a single cell sparsely, as a tuple containing the position of the value in the block as d attribute values and the measure value itself, is assumed to be $d/2+1$. For example, while we might store 32-bit measure values, the number of values per attribute will likely be less than 2^{16}. Even if there are more, we need only distinguish between the (fewer than 2^{16}) values used in a given block. Thus, densely storing a block with D allocated cells costs $M = m_1 \times \ldots \times m_d$, but storing it sparsely costs $(d/2+1)D$.

It is more economical to store a block densely if $(d/2+1)D > M$, that is, if $\frac{D}{m_1 \times \ldots \times m_d} > \frac{1}{d/2+1}$. We could show that this block coding is least efficient when a data cube has uniform density ρ over all blocks. In such cases, it has a sparse storage cost of $d/2+1$ per allocated cell if $\rho \leq \frac{1}{d/2+1}$ or a dense storage cost of ρ per allocated cell if $\rho > \frac{1}{d/2+1}$. Given a data cube C, $H(C)$ denotes its storage cost. We have $\#C \leq H(C) \leq n_1 \times \ldots \times n_d$. Thus, we measure the cost per allocated cell $E(C)$ as $\frac{H(C)}{\#C}$ with the convention that if $\#C = 0$, then $E(C) = 1$. Notice that $1 \leq E(C) \leq d/2+1$. A weakness of the model is that it ignores obvious storage overheads proportional to the number of blocks, $\frac{n_1}{m_1} \times \ldots \times \frac{n_d}{m_d}$. However, as long as the number of blocks remains constant, it is reasonable to assume that the overhead is constant. Such is the case when we consider the same data cube under different normalizations using fixed block dimensions.

3. MATHEMATICAL PRELIMINARIES

Now that we have defined a simple HOLAP model, we review two of the most important concepts in this paper: slices and normalizations. Whereas a slice amounts to fixing one of the attributes, a normalization can be viewed as a tuple of permutations.

3.1. **Slices.** Consider an n-regular d-dimensional cube C and let $C_{i_1,\ldots,i_d}$ denote the cell stored at indices $(i_1,\ldots,i_d) \in \{1,\ldots,n\}^d$. Thus, C has size n^d. The *slice* C_v^j of C, for index v of dimension j ($1 \le j \le d$ and $1 \le v \le n$) is a $d-1$ - dimensional cube formed as $C^j_{v\,i_1,\ldots,i_{j-1},i_{j+1},\ldots,i_d} = C_{i_1,\ldots,i_{j-1},v,i_{j+1},\ldots,i_d}$. See Figure 1 for an example, and observe that normalization does not affect the collection of cells in a slice— only the name of the slice and the cells' indices within the slice.

For the normalization task, the actual data values are unimportant and we simply need know which indices contain allocated cells. Hence we often view a slice as a $d-1$ - dimensional Boolean array and we denote the corresponding slice by $\widehat{C}_v^j$. We can also view a slice as a vector of length n^{d-1}, containing either measure values or Booleans, depending on our requirements. For instance, the slice C_3^1 identified in Figure 1 might be viewed as either $[4,0,0,5,9,2,0,1,0]$ or $[1,0,0,1,1,1,0,1,0]$, if we represent non-allocated cells by zeros. Let $\#\widehat{C}_v^j$ denote the number of allocated cells in slice C_v^j.

3.2. **Normalizations and Permutations.** Given a list of n items, there are $n!$ distinct possible permutations and we denote the set of all such permutations as Γ_n. The identity permutation is denoted ι. In contrast to previous work on database compression (e.g., [NR97]), with our HOLAP model there is no performance advantage from permuting the order of the attributes themselves. (Blocking treats all dimensions symmetrically.) Instead, we focus on normalizations, which affect the order of each attribute's values. A normalization π of a data cube C is a $d-$tuple $(\gamma_1,\ldots,\gamma_d)$ of permutations where $\gamma_i \in \Gamma_n$ for $i = 1,\ldots,d$, and the normalized data cube is given by $\pi(C)_{i_1,\ldots,i_d} = C_{\gamma_1(i_1),\ldots,\gamma_d(i_d)}$ for all $(i_1,\ldots,i_d) \in \{1,\ldots,n\}^d$. Recall that permutations, and thus normalizations, are not commutative. However, normalizations are always invertible, and there are $(n!)^d$ normalizations for an n-regular data cube. Let Ξ_n denote the set of all normalizations, and we simply write Ξ when the cube dimensions are known. The identity normalization is denoted $I = (\iota,\ldots,\iota)$. We say that two data cubes C and C' are equivalent ($C \sim C'$) if there exists a permutation π such that $\pi(C) = C'$. The cardinality of an equivalence class is the number of distinct data cubes C in this class. The maximum cardinality is given by $(n!)^d$ and there are such equivalence classes: consider the equivalence class generated by a "diagonal" data cube $C_{i_1,\ldots,i_d} = 1$ if $i_1 = i_2 = \ldots = i_d$ and 0 otherwise. However, there are also singleton equivalence classes since some cubes are invariant under normalization: consider a null data cube given by $C_{i_1,\ldots,i_d} = 0$ for all $(i_1,\ldots,i_d) \in \{1,\ldots,n\}^d$.

To count the cardinality of a class of data cubes, it suffices to know how many slices C_v^j of data cube C are identical, so that we can take into account the invariance under permutations. Considering all n slices in dimension r, we can count the number of distinct slices d_r and number of copies $n_{r,1},\ldots,n_{r,d_r}$ of each. Then, the number of distinct permutations in dimension r is $\frac{n!}{n_{r,1}!\times\ldots,\times n_{r,d_r}!}$ and the given equivalence class's cardinality is $\prod_{r=1}^{d}\left(\frac{n!}{n_{r,1}!\times\ldots,\times n_{r,d_r}!}\right)$. For example, the equivalence class generated by $C = \begin{bmatrix} 0 & 1 \\ 0 & 1 \end{bmatrix}$ has a cardinality of 2, despite having 4 possible normalizations.

Recalling that $E(C)$ measures the cost per allocated cell, we define the *kernel* $\kappa_{m_1,\ldots,m_d}$ as the set of all data cubes C of given dimensions such that $E(C)$ is minimal ($E(C) = 1$) for some fixed block dimensions $m_1,\ldots,m_d$. In other words, it is the set of all data cubes C where all blocks have density 1 or 0. We define the *kernel radius*

$$\overline{\rho} = \max_{\pi\in\Xi, C\in\kappa_{m_1,\ldots,m_d}} E(\pi(C)).$$

Because permutations are invertible, we have that there exists a data cube C such that $E(C) = \overline{\rho}$ and such that it can be renormalized using some π such that $E(\pi(C)) = 1$. In other words, the kernel radius gives us a measure of how much impact normalization can have. Because $1 \leq E(C) \leq d/2+1$, we say that the kernel radius is maximal if $\overline{\rho} = d/2+1$. Having the kernel radius maximal means that the storage requirement can be reduced up to a factor of $d/2+1$.

We can construct an example to show the following result:

Lemma 1. *For data cubes of dimensions $n_1, \ldots, n_d$ with blocks of dimensions $m_1, \ldots, m_d$, if $(d/2+1)$ divides n_k and $\frac{n_k}{d}$ divides n_k and m_k for some $k \in \{1, \ldots, d\}$, then the kernel radius is maximal.*

4. Complexity of Optimal Normalization

It appears that it is computationally intractable to find a "best" normalization $\pi \in \Xi$ (i.e., π minimizes $E(\pi(C))$) given a cube C and given the blocks' dimensions. Yet, when suitable restrictions are imposed, a best normalization can be computed (or approximated) in polynomial time. This section focuses on the effect of block size on intractability.

4.1. **Tractable Special Cases.**

Our problem can be solved in polynomial time, if severe restrictions are placed on the number of dimensions or on block size. For instance, it is trivial to find a best normalization in 1-d. Another trivial case arises when blocks are of size 1, since then normalization does not affect storage cost. Thus, any normalization is a "best normalization." The situation is more interesting for blocks of size 2; i.e., which have $m_i = 2$ for some $1 \leq i \leq d$ and $m_j = 1$ for $1 \leq j \leq d$ with $i \neq j$. A best normalization can be found in polynomial time, based on weighted-matching [Gab76] techniques described next.

4.1.1. *Using Weighted Matching.*

Given a weighted undirected graph, the *weighted matching problem* asks for an edge subset of maximum total weight, such that no two edges share an endpoint. If the graph is complete, has an even number of vertices, and has only positive edge weights, then the maximum matching effectively pairs up vertices.

For our problem, observe that normalization's effect on dimension k, for some $1 \leq k \leq d$, corresponds to rearranging the order of the n_k slices C^k_v, where $1 \leq v \leq n_k$. In our case, we are using a block size of 2 for dimension k. Therefore, once we have chosen two slices C^k_v and $C^k_{v'}$ to be the first pair of slices, we will have formed the first layer of blocks and have stored all allocated cells belonging to these two slices. The total storage cost of the cube is thus a sum, over all pairs of slices, of the pairing-cost of the two slices composing the pair. Note that the order in which pairs are chosen is irrelevant: only the actual matching of slices into pairs matters. Consider Boolean vectors $\mathbf{b} = \widehat{C}^k_v$ and $\mathbf{b}' = \widehat{C}^k_{v'}$. If both $\mathbf{b}_i$ and $\mathbf{b}'_i$ are true, then the i^{th} block in the pair is completely full and costs 2 to store. Similarly, if exactly one of $\mathbf{b}_i$ and $\mathbf{b}'_i$ is true, then the block is half-full. Under our model, a half-full block also costs 2, but an empty block costs 0. Thus, given any two slices, we can compute the cost of pairing them by summing the storage costs of all these blocks. If we identify each slice with a vertex of a complete weighted graph, it is easy to form an instance of weighted matching. (See Figure 2 for an example.) Fortunately, cubic-time algorithms exist for weighted matching, and n_k is often small enough that cubic running time is not excessive. Unfortunately, calculating the $\Theta(n_k^2)$ edge weights is expensive; each involves two large Boolean vectors with $\frac{1}{n_k}\prod_{i=1}^{d} n_i$ elements, for total edge-calculation time

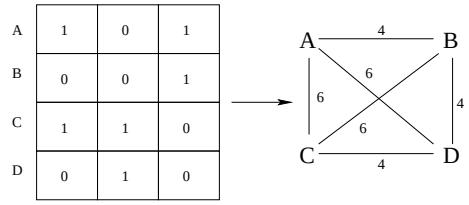

FIGURE 2. Mapping a normalization problem to a weighted matching problem on graphs. Rows are labeled and we try to reorder them, given block dimensions 2×1 (where 2 is the vertical dimension). In this example, optimal solutions include A, B, C, D and C, D, B, A.

of $\Theta\left(n_k \prod_{i=1}^{d} n_i\right)$. For more general block shapes, this algorithm is no longer optimal but nevertheless provides a basis for sensible heuristics.

Theorem 4.1. *For blocks of size* $\overbrace{1 \times \ldots \times 1}^{k-1} \times 2 \times 1 \ldots \times 1$, *the best normalization can be computed in* $O(n_k \times (n_1 \times n_2 \times \ldots \times n_d) + n_k^3)$ *time.*

4.2. **An NP-hard Case.** In contrast with 1×2-block situation, we next show that it is NP-hard to find the best normalization for 1×3 blocks. To simplify the proof, we assume a slightly different cost model: the cost of a block coded densely is twice the size of the block ($2M = 6$), and the cost of a sparsely coded block is triple its number of allocated cells ($3D = 0, 3, 6, 9$). Under this new cost model, a block with three allocated cells ($D = 3$) stores each of them at a cost of 2, whereas each block with fewer than three allocated cells stores each allocated cell at a cost of 3.

The proof involves a reduction from the NP-complete problem Exact 3-Cover (X3C) [GJ79], a problem which gives a set S and a set $\mathcal{T}$ of three-element subsets of S. The question, for X3C, is whether there is a $\mathcal{T}' \subseteq \mathcal{T}$ such that each $s \in S$ occurs in exactly one member of $\mathcal{T}'$.

We sketch the reduction next. Given an instance of X3C, form an instance of our problem by making a $|\mathcal{T}| \times |S|$ cube. For $s \in S$ and $T \in \mathcal{T}$, the cube has an allocated cell corresponding to $(T, s) \Leftrightarrow s \in T$. Thus, the cube has $3|\mathcal{T}|$ cells that need to be stored. The storage cost cannot be lower than $9|\mathcal{T}| - |S|$ and this bound can be met if and only if the answer to the instance of X3C is "yes".

Theorem 4.2. *It is NP-hard to find the best normalization, if* 1×3 *blocks are used.*

We conjecture that it is NP-hard to find the best normalization whenever the block size is fixed at any size larger than 2.

5. SLICE-SORTING NORMALIZATION FOR QUASI-INDEPENDENT CUBES

In practice, whether or not a given cell is allocated may depend on the corresponding attribute values independently of each other. For example, if a store is closed on Saturdays almost all year, a slice corresponding to "weekday=Saturday" will be sparse irrespective of the other attributes. In such cases, it suffices to normalize the data cube using only an attribute-wise approach. Moreover, as we shall see, one can easily compute the degree of

independence of the attributes and thus decide whether or not potentially more expensive algorithms need to be used.

We begin with one of the simplest classes of normalization algorithms, and we will assume n-regular data cubes for $n \geq 3$. We say that a sequence of values $x_1, \ldots, x_n$ is sorted in increasing (respectively, decreasing) order if $x_i \leq x_{i+1}$ (respectively, $x_i \geq x_{i+1}$) for $i \in \{1, \ldots, n-1\}$.

Algorithm 1. *(Slice-Sorting Normalization) Given an n- regular cube C, then slices have $S = n^{d-1}$ cells. Given a fixed function $g : \{true, false\}^S \to \mathbb{R}$, then for each attribute j, we compute the sequence $f_v^j = g(\widehat{C}_v^j)$ for all attribute values $v = 1, \ldots, n$. Let γ^j be a permutation such that $\gamma^j(f^j)$ is sorted either in increasing or decreasing order, then a Slice-Sorting normalization is given by $(\gamma^1, \ldots, \gamma^d)$.*

Algorithm 1 has time complexity $O(n^d)$. Notice that it is possible to compute the number of allocated cells per slice $\#\widehat{C}_v^j$ as the data cube is constructed, thus speeding up the normalization phase and making it $O(dn\log(n))$.

In general, Algorithm 1 does not produce a unique solution given a function g because there could be many different valid ways to sort. We say that a normalization $\varpi = \{\gamma^1, \ldots, \gamma^d\}$ is a *solution of the slice-sorting problem* if it provides a valid sort for the slice-sorting problem stated by Algorithm 1 . Given a data cube C, denote the set of all slice-sorting normalizations as $\mathcal{S}_{C,g}$. We say that two functions g_1 and g_2 are *equivalent* with respect to the slice-sorting problem if $\mathcal{S}_{C,g_1} = \mathcal{S}_{C,g_2}$. It turns out that we can characterize such equivalence classes using monotone functions. Recall that a function $h : \mathbb{R} \to \mathbb{R}$ is strictly monotone nondecreasing (respectively, nonincreasing) if $x < y$ implies $h(x) < h(y)$ (respectively, $h(x) > h(y)$).

We write $g_1 \sim g_2$ if there exists a strictly monotone function h such that $g_1 = h \circ g_2$. We further say that a Slice-Sorting algorithm is *stable* if the normalization of a normalized cube can be chosen to be the identity, that is if $\varpi \in \mathcal{S}_{C,g}$ then $I \in \mathcal{S}_{\varpi(C),g}$. We say that the algorithm is *strongly stable* if for any normalization $\varpi \in \Xi$, $\mathcal{S}_{\varpi(C),g} \circ \varpi = \mathcal{S}_{C,g}$. Strong stability implies stability and there exist stable schemes that are not strongly stable. Strong stability means that the resulting normalization does not depend on the initial normalization. This is a desirable property because data cubes are often normalized arbitrarily at construction time.

Define $\tau : \{true, false\}^S \to \mathbb{R}$ as the number of *true* values in the argument. In effect, τ counts the number of allocated cells: $\tau(\widehat{C}_v^j) = \#\widehat{C}_v^j$ for any slice $\widehat{C}_v^j$. If the slice $\widehat{C}_v^j$ is normalized, τ remains constant: $\tau(\widehat{C}_v^j) = \tau\left(\varpi\left(\widehat{C}_v^j\right)\right)$ for all normalizations $\varpi \in \Xi$. Therefore τ is strongly stable. The converse is also true if $d = 2$, but not if $d > 2$.

Lemma 2. *A Slice-Sorting algorithm based on a function g is strongly stable if $g = h \circ \tau$ for some function h. For 2-d cubes, the condition is necessary.*

In the above lemma, whenever h is strictly monotone, then $g \sim \tau$ and we call this class of Slice-Sorting algorithms *Frequency Sort* [Kas02b]. We will show that we can estimate *a priori* the efficiency of this class (see Theorem 5.1).

It is useful to consider a data cube as a probability distribution in the following sense: given a data cube C, let the *joint probability distribution* Ψ over the same n^d set of indices be

$$\Psi_{i_1,\ldots,i_n} = \begin{cases} 1/\#C & \text{if } C_{i_1,\ldots,i_n} \neq 0 \\ 0 & \text{otherwise} \end{cases} .$$

Heuristic	Synthetic Kernel-Based Data Sets				"Real-World" Data Sets		
	$\kappa^{base}_{2,2,2,2}$	$\kappa^{sp}_{2,2,2,2}$	$\kappa^{sp}_{2,2,2,2}$+N	$\kappa^{sp}_{4,4,4,4}$+N	CENSUS	FOREST	WEATHER
FS	61.2	56.1	85.9	70.2	**78.8**	94.5	88.6
GS	61.2	87.4	86.8	72.1	79.3	94.2	89.5
ISC ($2 \times \ldots \times 2$)	51.5	33.7	**49.4**	98.8	86.3	—	—
ISC ($4 \times \ldots \times 4$)	63.6	48.7	74.6	**49.6**	96.2	—	—
IM	**51.5**	**33.7**	**49.4**	97.8	79.0	**86.5**	**86.7**

TABLE 2. Performance of heuristics. Compression ratios are in percent and are averages. Each number represents 100 test runs for the synthetic data sets and 50 test runs for the others. Each experiment's outcome was the ratio of the heuristic storage cost to the default normalization's storage cost. Smaller is better.

The underlying probabilistic model is that allocated cells are uniformly likely to be picked, whereas the unallocated cells are never picked. Given an attribute $j \in \{1,\ldots,d\}$, consider $\#\widehat{C}^j_v$ for $v \in \{1,\ldots,n\}$: we can define a *probability distribution* φ^j along attribute j as $\varphi^j_v = \frac{\#\widehat{C}^j_v}{\#C}$. From these φ^j for all $j \in \{1,\ldots,d\}$, we can define the *joint independent probability distribution* Φ as $\Phi_{i_1,\ldots,i_n} = \prod_{j=1}^{d} \varphi^j_{i_j}$.

Given a joint probability distribution Υ and the number of allocated cells $\#C$, we can build an *allocation cube* A by computing $\Upsilon \times \#C$. Unlike a data cube, an allocation cube stores values between 0 and 1 indicating how likely it is that the cell be allocated. Using allocation cubes, there is an efficient way to determine whether Frequency Sorting is sufficient as Theorem 5.1 shows. It should be noted that we give an estimate valid independently of the dimensions of the blocks; thus, it is necessarily suboptimal.

Theorem 5.1. *Given a data cube C, let ϖ be an optimal normalization and fs be a Frequency Sort normalization, then*

$$H(fs(C)) - H(\varpi(C)) \leq \left(\frac{d}{2}+1\right)(1-\Phi\cdot B)\#C$$

where B is the allocation cube of C and Φ is the joint independent probability distribution. The symbol $\cdot$ denotes the scalar product defined in the usual way.

This theorem says that $\Phi \cdot B$ gives a rough measure of how well we can expect Frequency Sort to perform over all block dimensions: when $\Phi \cdot B$ is very close to 1, we need not use anything but Frequency Sort whereas when it gets close to 0, we can expect Frequency Sort to be less efficient. We call this coefficient the *Independence Sum*.

Hence, if the ROLAP storage cost is denoted by *rolap*, the optimally normalized block-coded cost by *optimal*, and the Independence Sum by *IS*, we have the relationship

$$rolap \geq optimal + (1-IS)rolap \geq fs \geq optimal$$

where *fs* is the block-coded cost using Frequency Sort as a normalization algorithm.

6. HEURISTICS

Since many practical cases appear intractable, we must resort to heuristics when the Independence Sum is small. We have experimented with several different heuristics, and

we can categorize possible heuristics as block-oblivious versus block-aware, dimension-at-a-time or holistic, orthogonal or not, or finally by whether they follow an overall strategy such as *slice clustering* or slice sorting.

Block-aware heuristics use information about the shape and positioning of blocks. In contrast, Frequency Sort (FS) is an example of a *block-oblivious* heuristic: it makes no use of block information.

All our heuristics reorder one dimension at a time, as opposed to a "holistic" approach when several dimensions are simultaneously reordered. In some heuristics, the permutation chosen for one dimension does not affect which permutation is chosen for another dimension. Such heuristics are *orthogonal*, and all the strongly stable Slice-Sorting algorithms in Section 5 are examples. Orthogonal heuristics can safely process dimensions one at a time, and in any order. With non-orthogonal heuristics that process one dimension at a time, we typically process all dimensions once, and repeat until some stopping condition is met.

Heuristics can be categorized by the overall approach used. In Slice Sorting, recall that each slice C_v^i is mapped to a real value $g(\widehat{C}_v^i)$, and these $g(\cdot)$ values are used to sort the slices. This abstracts the information of a complicated structure (a $d-1$ - dimensional cube) as a single number, and then groups slices accordingly. *Slice-grouping* heuristics compare entire slices for similarity. For instance, $\widehat{C}_v^i$ can be viewed as a 0-1 vector, and so we can cluster slices using Euclidean distance. The weighted-matching algorithm (from Section 4.1.1) is block-aware and slice-grouping. Another heuristic that is block-aware and slice-grouping is "ISC" (Iterated Slice Clustering). It uses Euclidean distance between "slice-density vectors" that capture information about the block densities of different regions in the chunk. It is best to explain ISC using an example. We start with the following cube

$$\begin{bmatrix} 1 & - & 1 & 1 \\ - & 1 & 1 & - \\ 1 & 1 & - & 1 \\ - & 1 & 1 & - \end{bmatrix}$$

where $-$ means the cell is not allocated. We wish to use a 2×2 block coding, and we begin by rearranging the columns. To do this, we first compute the density of allocated cells in each 2×1 chunk, obtaining $\begin{bmatrix} 0.5 & 0.5 & 1.0 & 0.5 \\ 0.5 & 1.0 & 0.5 & 0.5 \end{bmatrix}$. We then reorder the columns so as to minimize the Euclidean distance between columns in the same block, using a greedy approach starting with the first column. For 2×2 block coding, we are pairing these columns and, in this case, we exchange the first and last columns, getting $\begin{bmatrix} 0.5 & 0.5 & 1.0 & 1.0 \\ 0.5 & 0.5 & 0.5 & 0.5 \end{bmatrix}$ and giving us the new array

$$\begin{bmatrix} 1 & 1 & 1 & - \\ - & - & 1 & 1 \\ 1 & 1 & - & 1 \\ - & - & 1 & 1 \end{bmatrix}.$$

We would then repeat this same process along rows (over all dimensions). Because this algorithm is not orthogonal, we then need to repeat along rows and columns until convergence or, more likely, until we reach a maximum number of iterations (set at 5 in our software). Slice-grouping heuristics are typically slow, since the vectors are large and not quickly compared.

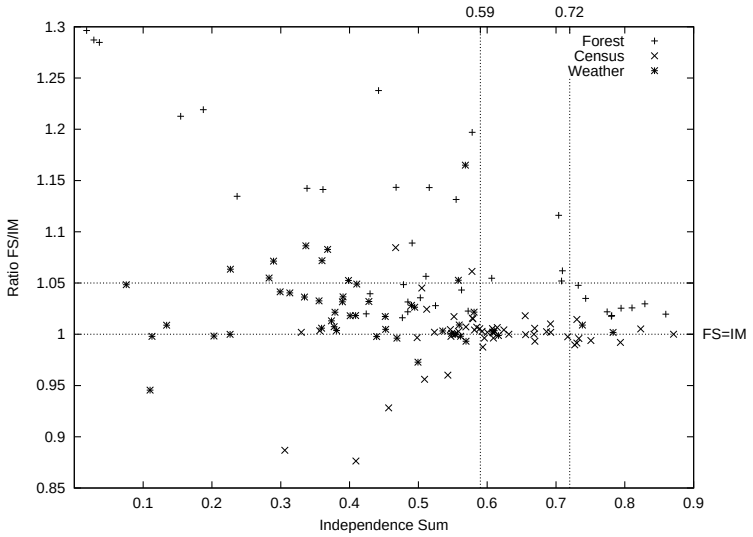

FIGURE 3. Solution-size ratios of FS and IM as a function of Independence Sum. When the ratio is above 1.0, FS is suboptimal; when it is less than 1.0, IM is suboptimal. We see that as the Independence Sum gets closer to 1.0, FS matches IM's performance.

Another heuristic for 2-regular blocks is Iterated Matching (IM), which applies the weighted-matching algorithm to each dimension in turn (see Section 4.1.1 for details). Again, this algorithm is better explained using an example. Applying this algorithm along for the rows of the cube in Fig. 2 amounts to building the graph in the same figure and solving the weighted-matching problem over this graph. The cube would then be normalized to

$$\begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 1 \end{bmatrix}.$$

Like the ISC heuristic, we would then repeat on the columns (over all dimensions). Unlike ISC, however, we would not reapply these steps over rows and columns more than once, because the IM algorithm is orthogonal. A small example, $\begin{bmatrix} 1 & - & 1 & 1 \\ 1 & - & - & - \end{bmatrix}$, demonstrates this approach is suboptimal, since the normalization shown is optimal for 2×1 and 1×2 blocks but not optimal for 2×2 blocks.

6.1. **One-Dense-Chunk Heuristics.** In [Kas02b], data-cube normalization has previously been examined under a different HOLAP model, where only one block may be stored densely, but the block's size is chosen adaptively. Despite model differences, normalizations that cluster data into a single large chunk intuitively should be useful with our current model. We adapted the most successful heuristic from [Kas02b] and called the result GS (for iterated Greedy Sort). It can be viewed as a block-aware version of Frequency Sort.

7. Experimental Results

In describing the experiments, we discuss the data sets used, the heuristics tested, and the results observed.

7.1. **Data Sets.** Heuristics were tested on a variety of data cubes. Several synthetic $12 \times 12 \times 12 \times 12$ data sets were used, and 100 random data cubes of each variety were taken.

- $\kappa^{base}_{2,2,2,2}$ refers to choosing a cube C uniformly from the set $\kappa_{2,2,2,2}$ and choosing π uniformly from Ξ_{12}. Cube $\pi(C)$ provides the test data; a best-possible normalization will compress $\pi(C)$ by a ratio of $\max(\rho, \frac{1}{3})$, where ρ is the density of $\pi(C)$. (The expected value of ρ is 50%.)
- $\kappa^{sp}_{2,2,2,2}$ is similar, except the random selection from $\kappa_{2,2,2,2}$ is biased towards sparse cubes. The expected density of such cubes is 10%, and thus the entire cube will likely be stored sparsely. The best compression for such a cube is to $\frac{1}{3}$ of its original cost.
- $\kappa^{sp}_{2,2,2,2}+N$ adds noise. For every index, there is a 3% chance that its status (allocated or not) will be inverted. Due to the noise, the cube usually cannot be normalized to a kernel cube, and hence the best possible compression is probably closer to $\frac{1}{3} + 3\%$.
- $\kappa^{sp}_{4,4,4,4}+N$ is similar, except we choose from $\kappa_{4,4,4,4}$, not $\kappa_{2,2,2,2}$.

Besides synthetic data sets, we have experimented with several data sets used in [Kas02a]: CENSUS (50 6-d projections of an 18-d data set) and FOREST (50 3-d projections of an 11-d data set) from the KDD repository [HB00], and WEATHER (50 5-d projections of an 18-d data set) [HWL01][2]. These data sets were obtained in relational form and their initial normalizations can be summarized as "first seen, first when normalized", which is arguably the normalization that minimizes data-cube implementation effort. In [Kas02b] this was called "Order I".

7.2. **Results.** The heuristics chosen for testing were Frequency Sort (FS), Iterated Greedy Sort (GS), Iterated Slice Clustering (ISC), and Iterated Matching (IM). Except for the "$\kappa^{sp}_{4,4,4,4}$+N" data sets, where 4-regular blocks were used, blocks were 2-regular. IM implicitly assumes 2-regular blocks, whereas ISC is used once assuming 2-regular blocks and used a second time assuming 4-regular blocks. Results are shown in Table 2.

Looking at the results in Table 2 for synthetic data sets, we see very similar performance from ISC ($2 \times \ldots \times 2$) and IM. We see that GS was never better than FS; this is perhaps not surprising, because the main difference between FS and GS is that the latter does additional work to ensure allocated cells are within a single hyperrectangle and that cells outside this hyperrectangle are discounted. This was helpful when using the model for which it was devised, but perhaps not so reasonable for our current model. Comparing the $\kappa^{sp}_{2,2,2,2}$ and $\kappa^{sp}_{2,2,2,2}$+N columns, it is apparent that noise hurt all heuristics, particularly the slice-sorting ones (FS and GS). However, FS and GS perform better on $\kappa^{sp}_{4,4,4,4}$+N than $\kappa^{sp}_{2,2,2,2}$+N whereas both ISC ($2 \times \ldots \times 2$) and IM did worse. We explain this improved performance for slice-sorting normalizations as follows: $\#C^i_v$ is a multiple of 4^3 under $\kappa_{4,4,4,4}$ but a multiple of 2^3 under $\kappa_{2,2,2,2}$. Thus, $\kappa_{2,2,2,2}$ is more susceptible to noise than $\kappa_{4,4,4,4}$ under FS because the values $\#C^i_v$ are less separated.

[2] Projections were selected at random but, to keep test runs from taking too long, cubes were required to be smaller than about 100MB.

Of the slice-clustering heuristics, ISC ($4 \times \ldots \times 4$) is also badly affected by noise. IM is less affected, but we note its results are not optimal (estimated as 36%). Thus, while IM (and ISC) are the most effective heuristics for 2-regular blocks, there is room for improvement. We observe the disastrous effects when the actual block size does not match the size assumed by a block-aware heuristic.

Table 2 also contains results for "real-world" data. Unfortunately, our implementation of ISC ran too slowly to use in these tests, except on CENSUS. ISC performs poorly on the CENSUS cubes, in contrast to the good performance obtained when normalizing synthetic cubes. We also measured how many times the block-aware ISC heuristic outperformed the block-oblivious FS on 4-regular blocks: ISC was better only 11% of the time. Hence we concluded that ISC is not promising, at least on the CENSUS cubes. We see that the relative performance of the various heuristics depends heavily on the data set used. Part of this is due to the nature of the data sets: for instance, FOREST contains many measurements of physical characteristics of geographic areas, and there is significant correlation between these characteristics that tends to penalize FS.

7.2.1. *Utility of the Independence Sum.* Despite the differences between data sets, the Independence Sum (from Section 5) seems to be useful. In Figure 3 we plot the ratio $\frac{\text{size using FS}}{\text{size using IM}}$ against the Independence Sum. Note that when the Independence Sum exceeds 0.72, the ratio is always near 1 (within 5%); thus, there is no need to use the more computationally expensive IM heuristic. WEATHER has few cubes with Independence Sum over 0.6, but these have ratios near 1.0. For CENSUS, having an Independence Sum over 0.6 seems to guarantee good relative performance for FS. On FOREST, however, FS shows poorer performance until the Independence Sum is larger ($\simeq 0.72$).

7.2.2. *Comparison with Pure ROLAP Coding.* To place the efficiency gains from normalization into context, we calculated (for each of the 50 CENSUS cubes) c_{default}, the HOLAP storage cost using 2-regular blocks and the default normalization. We also calculated c_{ROLAP}, the ROLAP cost, for each cube. The average of the 50 ratios $\frac{c_{\text{default}}}{c_{\text{ROLAP}}}$ is 0.7 with a standard deviation of 0.14. In other words, block-coding is 30% more efficient than ROLAP. On the other hand, we have shown that normalization brings gains of about 20% over the default normalization and the storage ratio itself is brought from 0.7 to 0.56 in going from simple block coding to block coding together with optimized normalization.

8. CONCLUSION

In this paper, we have given a number of theoretical results relating to cube normalization. Because even simple special cases of the problem are NP-hard, heuristics were needed. However, an optimal normalization can be computed when 1×2 blocks are used, and this forms the basis of the IM heuristic, which seemed most efficient in experiments. Nevertheless, a Frequency Sort algorithm is much faster, and another of the paper's theoretical conclusions was that this algorithm becomes increasingly optimal as the Independence Sum of the cube increases. Unfortunately, our theorem did not provide a very tight bound on suboptimality. Nevertheless, we determined experimentally that an Independence Sum greater than 0.72 always meant that Frequency Sort produced good results.

As future work, we will seek tighter theoretical bounds and more effective heuristics for the cases when the Independence Sum is small. We are also in the process of implementing the proposed architecture for further validation and research by combining an embedded relational database with a C++ layer. An implementation would allow us to quantify our claim that a more efficient normalization leads to faster queries.

REFERENCES

[CD99] Emmanuel J. Candès and David L. Donoho. Curvelets - a surprisingly effective nonadaptive representation for objects with edges. In *Curves and Surfaces*, 1999.

[CZK+99] David Wai-Lok Cheung, Bo Zhou, Ben Kao, Kan Hu, and Sau Dan Lee. DROLAP - a dense-region based approach to on-line analytical processing. In *Database and Expert Systems Applications*, pages 761–770, 1999.

[CZK+01] David Wai-Lok Cheung, Bo Zhou, Ben Kao, Hu Kan, and Sau Dan Lee. Towards the building of a dense-region-based OLAP system. *Data and Knowledge Engineering*, 36(1):1–27, 2001.

[DER01] Frank Dehne, Todd Eavis, and Andrew Rau-Chaplin. Coarse grained parallel on-line analytical processing (OLAP) for data mining. In *Proc, ICCS 2001*, 2001.

[DRSN98] P. Deshpande, K. Ramasamy, A. Shukla, and J. Naughton. Caching multidimensional queries using chunks. In *Proceedings, SIGMOD'98*, pages 259–280, 1998.

[DVDD98] David L. Donoho, Martin Vetterli, Ingrid Daubechies, and Ron A. DeVore. Data compression and harmonic analysis. *IEEETIT: IEEE Transactions on Information Theory*, 44, 1998.

[GAAS99] S. Geffner, D. Agrawal, A. E. Abbadi, and T. R. Smith. Relative prefix sums: An efficient approach for querying dynamic OLAP data cubes. In *Proc, 1999 ICDE*, pages 328–335, March 1999.

[Gab76] H. Gabow. An efficient implementation of Edmond's algorithm for maximum matching on graphs. *Journal of the ACM*, 23:221–234, 1976.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, New York, 1979.

[Goi99] Sanjay Goil. *High Performance On-line Analytical Processing and Data Mining on Parallel Computers*. PhD thesis, Dept. ECE, Northwestern University, 1999.

[HB00] S. Hettich and S. D. Bay. The UCI KDD archive. http://kdd.ics.uci.edu, last checked on 6/7/2003, 2000.

[HWL01] C. Hahn, S. Warren, and J. London. Edited synoptic cloud reports from ships and land stations over the globe (1982-1991). http://cdiac.ornl.gov/epubs/ndp/ndp026b/ndp026b.htm, last checked on 6/7/2003, 2001.

[Kas02a] Owen Kaser. Compressing arrays by ordering attribute values. under review, email author at owen@unbsj.ca, 2002.

[Kas02b] Owen Kaser. Compressing MOLAP arrays by attribute-value reordering: An experimental analysis. Technical Report TR-02-001, Dept. of CS and Appl. Stats, U. of New Brunswick, Saint John, Canada, August 2002.

[KL03] O. Kaser and D. Lemire. Attribute-value reordering for efficient hybrid OLAP. Technical Report NRC 46509, NRC/CNRC, 2003.

[Lem02] Daniel Lemire. Wavelet-based relative prefix sum methods for range sum queries in data cubes. In *Proc, CASCON 2002*, October 2002. also NRC-44967.

[LS02] Jianzhong Li and J. Srivastava. Efficient aggregation algorithms for compressed data warehouses. *IEEE Knowledge and Data Engineering*, 15, May/June 2002.

[NM80] A. N. Netravali and F. W. Mounts. Ordering techniques for facsimile coding: A review. *Proceedings of the IEEE*, 68(7):796–807, 1980.

[NR97] Wee-Keong Ng and Chinya V. Ravishankar. Block-oriented compression techniques for large statistical databases. *IEEE Knowledge and Data Engineering*, 9(2):314–328, 1997.

[PM00] Erwan Le Pennec and Stéphane Mallat. Image representation and compression with bandelets. Technical report, École Polytechnique, 2000.

[SDRK02] Yannis Sismanis, Antonios Deligiannakis, Nick Roussopoulus, and Yannis Kotidis. Dwarf: Shrinking the petacube. In *ACM SIGMOD 2002*, pages 464–475, 2002.

[SS94] Sunita Sarawagi and Michael Stonebraker. Efficient organization of large multidimensional arrays. In *Proc, Eleventh Int. Conf. on Data Engineering*, February 1994.

[ZDN97] Yihong Zhao, Prasad M. Deshpande, and Jeffrey F. Naughton. An array-based algorithm for simultaneous multidimensional aggregates. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, pages 159–170. ACM Press, 1997.