Condensed Cube: An Effective Approach to Reducing Data Cube Size

Wei Wang
Dept. of Computer Science
Hong Kong University of Sci. & Tech.
Hong Kong, China
fervvac@cs.ust.hk

Jianlin Feng*
School of Computer Science
Huazhong University of Sci. & Tech.
Wuhan, China
jlfeng@public.wuhan.cngb.com

Hongjun Lu†
Dept. of Computer Science
Hong Kong University of Sci. & Tech.
Hong Kong, China
luhj@cs.ust.hk

Jeffrey Xu Yu
Dept. of Sys. Eng. & Eng. Management
The Chinese University of Hong Kong
Hong Kong, China
yu@se.cuhk.edu.hk

Abstract

Pre-computed data cube facilitates OLAP (On-Line Analytical Processing). It is a well-known fact that data cube computation is an expensive operation, which attracts a lot of attention. While most proposed algorithms devoted themselves to optimizing memory management and reducing computation costs, less work addresses one of the fundamental issues: the size of a data cube is huge when a large base relation with a large number of attributes is involved. In this paper, we propose a new concept, called a condensed data cube. The condensed cube is of much smaller size of a complete non-condensed cube. More importantly, it is a fully pre-computed cube without compression, and, hence, it requires neither decompression nor further aggregation when answering queries. Several algorithms for computing condensed cube are proposed. Results of experiments on the effectiveness of condensed data cube are presented, using both synthetic and real-world data . The results indicate that the proposed condensed cube can reduce both the cube size and therefore its computation time.

1. Introduction

In order to effectively support decision support queries, a new operator, `CUBE BY`, was proposed [7]. It is a multi-dimensional extension of the relational operator `GROUP BY`.[1] The `CUBE BY` operator computes `GROUP BY` corresponding to all possible combinations of attributes in the `CUBE BY` clause.

It is obvious that `CUBE BY` is an expensive operator and its result is extremely large, especially when the number of `CUBE BY` attributes and the number of tuples in the base relation are large. Given a base relation R with n attributes, the number of tuples in a k-attribute cuboid (`GROUP BY`), $0 \leq k \leq n$, is the number of tuples in R that have distinct attribute values on the k-attributes. The size of a cuboid is possibly close to the size of R. Since the complete cube of R consists of 2^n cuboids, the size of the union of 2^n cuboids is much larger than the size of R. Consequently, the I/O cost even for storing the cube result tuples becomes dominative as indicated in [13, 4]. The huge size of a data cube makes data cube computation time-consuming. Although the cheap and high volume memory chips are available, it is difficult to hold the whole data cube of a large relation in the main memory.

Although issues related to the size of data cubes have attracted the attention of researchers, and various algorithms have been developed aiming at fast computation of large sparse data cubes [2, 18, 13, 4], relatively fewer papers concentrated on solving the complexity problem of data cube computation from its root: *reducing the size of a data cube.*

In this paper, we introduce a novel concept, *condensed cube*, for reducing the size of a data cube and hence its computation time and storage overhead. A condensed cube is a fully computed cube that condenses those tuples, aggregated from the same set of base relation tu-

*The work was done when the author was at Hong Kong University of Science and Technology.

†This author's work is partially supported by a grant from the National 973 project of China (No. G1998030414).

[1] The `GROUP BY` operator, first, partitions a relation into groups based on the values of the attributes specified in the `GROUP BY` clause, and then applies aggregation functions to each of such groups.

Proceedings of the 18th International Conference on Data Engineering (ICDE'02)
1063-6382/02 $17.00 © 2002 IEEE

ples, into one physical tuple. Taking an extreme case as an example. Let relation R have only one single tuple $(a_1, a_2, \ldots, a\ , m)$. Then, the data cube for R will have 2 tuples, $(a_1, a_2, \ldots, a\ , V_1)$, $(a_1, *, \ldots, *, V_2)$, $(*, a_2, *, \ldots, *, V\ )$, $(*, *, \ldots, *, V\ )$, where $m = 2$.[2] Since there is only one tuple in relation R, we have $V_1 = V_2 = \ = V\ = a\ (\ )$. Therefore, we only need to physically store one tuple $(a_1, a_2, ..., a\ , V)$, where $V = a\ (\ )$, in the cube together with some information indicating that it is a representative of a set of tuples. For queries on any cuboid, we can return the value V directly without the need of further aggregation. That is, for this example, the cube for R with 2 tuples can be condensed into one tuple. In general, tuples from different cuboids in a cube, that are known being aggregated from the same set of tuples, can be condensed to one tuple.

Compared to the related approaches proposed in the literature, a condensed cube has the following unique features.

- *A condensed cube is not compressed.* Although the size of a condensed cube is smaller than the complete cube, it is not compressed. It does not require decompression when the condensed cube is used to answer queries. Hence, no extra overhead will be introduced.
- *A condensed cube is a fully computed cube.* It is different from those approaches that reduce the size of a data cube by selectively computing some, but not all, cuboids in the cube. Therefore, no further application of aggregation function is required when a condensed cube is used to answer queries.
- *A condensed cube provides accurate aggregate values.* It is different from those approaches that reduce the cube size through approximation with various forms, such as wavelet [16], multivariate polynomials [3], mixed model by multivariate Gaussians [15], histogram [12], sampling [1] and others [8].
- *A condensed cube supports general OLAP applications.* It is different from those proposals to reduce the size of a cube by tailoring it to only answering certain types of queries [10].

Contributions of our work can be summarized as follows: We introduced the concept of condensed cube which can greatly reduce the size of a data cube without additional query overhead. A condensing scheme, namely, basic single tuple (BST) condensing is proposed. Several algorithms have been proposed to efficiently compute the condensed cube. We show the effectiveness of the proposed algorithms with experiment results using both synthetic and real-world data.

The remainder of the paper is organized as follows. Section 2 gives the motivations and definitions of condensed cube. Section 3 presents our algorithm to compute the condensed cube. Section 4 presents the two heuristic algorithms for fast condensed cube computing. The effectiveness of our algorithms, based on the experiments from both synthetic data and real data set, is presented in Section 5. We discuss the related work in Section 6. Finally, Section 7 concludes the paper and discusses future work.

[2] $*$ represents the special value ALL.

2. Condensed Cube: Preliminaries and definitions

In this section, we introduce the concept of condensed cube, and a condensing scheme, base single tuple (BST) condensing, is introduced as well.

2.1. Relation and data cube

The CUBE BY operator [7] is an n-dimensional generalization of the relational GROUP BY operator. For a base relation R with n dimension attributes $(\ _1, \ldots, \ )$ and one measure attribute M, the complete data cube of R (denoted as (R)) on all dimensions is generated by CUBE BY on $_1, \ldots, \ $. This is equivalent to the 2 GROUP BYs on each different subset of the n dimension set. We refer to each GROUP BY as a *cub* i [13]. The cuboid on all n attributes is called *base cuboid*.

The introduction of a special value ALL makes the data cube result compatible with the schema of the base relation R by extending the domain of each dimension attribute with ALL. Such tuples in the complete cube are referred to as the *cube tuples*.

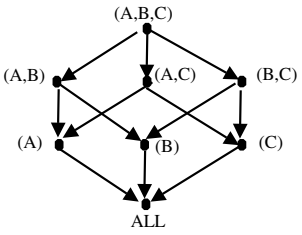
In this paper, we will use relation $R(A, B, \ , M)$ with five tuples in Figure 1(a) as a running example. R has three dimension attributes: A, B and , and one measure attribute M. The aggregate function used is SUM. The complete cube is shown in Figure 1(b), where we use * to denote the special value ALL. There are 30 tuples in the complete cube, each of which belongs to one of the eight cuboids, *ABC, AB, AC, BC, A, B, C,* and *ALL*, as denoted in the Cuboid column.

Cuboids in a cube are related to each other to form a lattice. The cuboid lattice for this 3-dimensional attribute relation is shown in Figure 1(c). The nodes in the lattice are cuboids. An arrow between two cuboids means that one cuboid can be computed from the other. For example, cuboids on AB, A , and B can be computed from cuboid AB .

Without losing generality, we assume, in this paper, that $_1 \ldots$ functionally determines M. In other words, we assume that the base relation is in fact the same as the base cuboid, the cuboid on n attributes. We can always compute the base cuboid first if the original relation does not satisfy the condition.

TID	A	B	C	M
1	0	1	1	50
2	1	1	1	100
3	2	3	1	60
4	4	5	1	70
5	6	5	2	80

(a)



(c)

TID	Cuboid	A	B	C	M
1	ALL	*	*	*	360
2	ABC	0	1	1	50
3	A	0	*	*	50
4	AB	0	1	*	50
5	AC	0	*	1	50
6	ABC	1	1	1	100
7	A	1	*	*	100
8	AB	1	1	*	100
9	AC	1	*	1	100
10	ABC	2	3	1	60
11	A	2	*	*	60
12	AB	2	3	*	60
13	AC	2	*	1	60
14	B	*	3	*	60
15	BC	*	3	1	60
16	ABC	4	5	1	70
17	A	4	*	*	70
18	AB	4	5	*	70
19	AC	4	*	1	70
20	ABC	6	5	2	80
21	A	6	*	*	80
22	AB	6	5	*	80
23	AC	6	*	2	80
24	C	*	*	2	80
25	BC	*	5	2	80
26	B	*	1	*	150
27	B	*	5	*	150
28	C	*	*	1	280
29	BC	*	1	1	150
30	BC	*	5	1	70

(b)

Figure 1. (a) An example relation R, (b) Its complete cube (R), and (c) Cuboid lattice

2.2. BST: Base single tuple

Tuples in a cube are grouped (partitioned) and aggregated from tuples in the base relation. For the cube (Figure 1(b)) and the base relation R (Figure 1(a)), the cube tuples, , $_2$ and $_2$ are aggregated from the partitions, $\{R_1\}$, $\{R_1, R_2\}$, and $\{R_1, R_2, R\ ,R\ \}$, where $_i$ and R_i are tuples in and R with $\texttt{TID} = i$, respectively. In general, there are a large number of base relation tuples in each of such partitions, especially in cuboids with a small number of attributes. However, there exist such partitions that contain only one tuple. We name such special base relation tuple as a *base single tuple*.

Definition 2.1 (Base Single Tuple) *Given a set of dimension attributes,* SD $\{\ _1,\ _2, \ldots,\ \}$*, if is the only tuple in a partition when the base relation is partitioned on* SD*, we say tuple is a base single tuple on* SD*, and* SD *is called the single dimensions of .*

Based on the definition, tuple R_1(0, 1, 1, 50) in our example in Figure 1 is a base single tuple on $\{A\}$, since it is the only tuple in the partition of A-value 0, when R is partitioned on A. On the other hand, R_1 is not a base single tuple on $\{B\}$, because both R_1 and R_2 will be in the same partition of B-value 1, when R is partitioned on B. A tuple could be a single tuple on more than one single dimension set. For example, R is a BST on $\{A\}$ and is also a BST on $\{B\}$. R_1 and R_2 are BST on A but are not on B since two tuples will be in the same partition when R is partitioned on B.

Lemma 2.1 *If a tuple is a base single tuple on* SD*, then is also a single tuple on any superset of* SD*.*

In our example, since tuple R_1(0, 1, 1, 50) is a base single tuple on A, it is also a base single tuple on $\{A\}$, $\{AB\}$, $\{A\ \}$ and $\{AB\ \}$. To represent the fact that a tuple can be a base single tuple on different dimensions, we associate each base single tuple a set of SDs, called SDSET, rather than a single SD. One optimization based on Lemma 2.1 is that we can use only the minimum set of SDs but not their supersets.

Property 2.1 *If is a base single tuple on attribute sets* SDSET*, then cube tuples generated from in all cuboids on* $\texttt{SD}_\texttt{i} \in$ SDSET *have the same aggregation value* $a = a\ (\ )$.

Property 2.1 can be derived directly from the definition of the base single tuple. Cube tuples of the cuboid on SD are obtained by partitioning R on SD. If is a base single tuple on SDSET, it is the only tuple in its partition when relation R is partitioned on any set of attributes, $\texttt{SD}_\texttt{i} \in$ SDSET. Therefore, the aggregation function is applied on tuple only, hence all cube tuples will have the same aggregation value.

Property 2.1 forms the base of the concept of condensed cube. For base single tuple , we only need to store one base cuboid tuple, $_b = (a_1, a_2, ..., a\ , M_b\ )$, where $a_i = \texttt{ALL}$ for $1 \le i \le n$ and $M_b = a\ (\ )$, in the condensed cube physically. Cube tuples corresponding to in a non-base cuboid on $\texttt{SD}_\texttt{i} \in$ SDSET can be omitted from the condensed cube. When needed, they can be expanded from the base cuboid tuple defined below.

Definition 2.2 (Expand) $Expan$ *takes a base cuboid tuple $_b$, and its single dimensions* SD *or single dimension sets* SDSET *as input, and produces one or a set of cube tuples, respectively.*

- $Expan\ (\ _b\ ,\texttt{SD}) =$ *such that* $M(\ ) = M(\ _b\ )$ *and* $_i(\ ) = \ _i(\ _b\ )$*, for* $_i \in$ SD *and* $_j(\ ) =$ ALL*, for* $_j \notin$ SD*.*
- $Expan\ (\ _b\ ,\texttt{SDSET})$
  $= \{\ |\ = Expan\ (\ _b\ ,\texttt{SD}_\texttt{i})\ \ \texttt{SD}_\texttt{i} \in \texttt{SDSET}\}$
  $= \bigcup_{\texttt{SD}_\texttt{i}\ \texttt{SDSET}} Expan\ (\ _b\ ,\texttt{SD}_\texttt{i})$.

Example 2.1 *Given a base cuboid tuple $_b$ (0, 1, 1, 50) and* $\texttt{SDSET} = \{\{A\}, \{AB\}, \{A\ \}\}$*, we can generate cube tuples (0, *, *, 50), (0, 1, *, 50), and (0, *, 1, 50), where * represents the special value* ALL*, by applying the operator* $Expan$.

It is important to note that the operation $Expan$ is a simple operation that requires copying the original tuple and replacing certain attributes value with a special value. No aggregation or other computation is required.

2.3. BST-condensed data cube

With the discussion so far, we show that there exist some special tuples in a relation, the base single tuples. For those tuples, only one tuple in the base cuboid needs to be stored physically, and tuples in other cuboids can be generated without much computation. This leads to our new approach to reducing the size of a data cube: condensing those tuples that can be generated from base cuboid tuples into their corresponding base cuboid tuples. Since the process of condensing is based on BST (base single tuples), we name such size-reduced cubes as BST-condensed cubes, as defined below.

Definition 2.3 (BST-Condensed Cube) *A BST-condensed cube has an augmented schema* ($_1, \ldots,$ $, M, S$ SE). *Its tuple* ($_1, \ldots,$ $, M($ $), S$ SE) *is denoted by* (. , $.S$ SE) *for brevity, where* . $=$ ($_1, \ldots,$ $,$ $M($ $)$). *For tuples in non-base cuboids,* S SE $= \{\}$. *Tuples in a BST-condensed cube* (R) *of a base relation* R *must satisfy the following conditions.*

1. *For all tuples* in non-base cuboids, there does not exist a tuple $_b$ *in the BST-condensed cube such that* . $\in Expan$ ($_b$. , $_b.S$ SE)
2. $\prod_{\{\bigcup^{1}}$ (R) $\bigcup$ ^{n} $Expan$ (. , $.S$ SE)$\}$ $=$ (R), *where* (R) *is the data cube for* R.

The first condition in the above definition guarantees that the cube is condensed based on BST-principle. That is, all tuples that can be generated from the base single tuples are not included in the condensed cube. The second condition guarantees the correctness. That is, after expansion, the complete cube for R can be obtained.

Figure 2(a) and 2(b) depict the complete cube for relation R and a BST-condensed cube, respectively. We use the curly brackets and arrows to denote the set of cube tuples in Figure 2(a) that are condensed into one tuple in the BST-condensed cube Figure 2(b). For example, tuple (0, *, *), (0, 1, *) and (0, *, 1) are condensed into $_1$(0, 1, 1).

The above definition only requires that the complete cube can be reconstructed from the base cuboid tuples and their stored SDSET. It does not require to make use of *all* possible base single tuples on *all* their single dimensions to condense a cube. As an extreme case, we can view the complete cube as a special case of a BST-condensed cube, with the SDSET fields of all tuples in the base cuboid set to $\{\}$. We attempt to reduce the size of a data cube as much as possible, which leads to the following definition of minimal BST-condensed cube.

Definition 2.4 (Minimal BST-Condensed Cube) *A minimal BST-condensed cube* $_i$ (R) *of base relation* R *is a BST-condensed cube; and there does not exist any tuple in non-base cuboids that can be generated from based single tuples.*

TID	Cuboid	A	B	C	M
1	ALL	*	*	*	360
2	ABC	0	1	1	50
3	A	0	*	*	50
4	AB	0	1	*	50
5	AC	0	*	1	50
6	ABC	1	1	1	100
7	A	1	*	*	100
8	AB	1	1	*	100
9	AC	1	*	1	100
10	ABC	2	3	1	60
11	A	2	*	*	60
12	AB	2	3	*	60
13	AC	2	*	1	60
14	B	*	3	*	60
15	BC	*	3	1	60
16	ABC	4	5	1	70
17	A	4	*	*	70
18	AB	4	5	*	70
19	AC	4	*	1	70
20	ABC	6	5	2	80
21	A	6	*	*	80
22	AB	6	5	*	80
23	AC	6	*	2	80
24	C	*	*	2	80
25	BC	*	5	2	80
26	B	*	1	*	150
27	B	*	5	*	150
28	C	*	*	1	280
29	BC	*	1	1	150
30	BC	*	5	1	70

(a)

TID	A	B	C	M	SDSET
1	0	1	1	50	{{A},{AB},{AC},{ABC}}
2	1	1	1	100	{{A},{AB},{AC},{ABC}}
3	2	3	1	60	{{A},{AB},{AC},{ABC}}
4	4	5	1	70	{{A},{AB},{AC},{ABC}}
5	6	5	2	80	{{A},{AB},{AC},{ABC}}
6	*	3	*	60	{}
7	*	3	1	60	{}
8	*	*	2	80	{}
9	*	5	2	80	{}
10	*	1	*	150	{}
11	*	5	*	150	{}
12	*	*	1	280	{}
13	*	1	1	150	{}
14	*	5	1	70	{}
15	*	*	*	360	{}

(b)

TID	A	B	C	M	SDSET
1	0	1	1	50	{{A},{AB},{AC},{ABC}}
2	1	1	1	100	{{A},{AB},{AC},{ABC}}
3	2	3	1	60	{{A},{B},{AB},{AC},{BC},{ABC}}
4	4	5	1	70	{{A},{AB},{AC},{BC},{ABC}}
5	6	5	2	80	{{A},{C},{AB},{AC},{BC},{ABC}}
6	*	1	*	150	{}
7	*	5	*	150	{}
8	*	*	1	280	{}
9	*	1	1	150	{}
10	*	*	*	360	{}

(c)

Figure 2. (a) The complete cube , (b) A BST-Condensed cube of R and (c) A minimal BST-Condensed cube of R

The intuitive interpretation of the definition is that, a minimal BST-condensed cube must make use of all base single tuples and their single dimensions to condense the cube till no further condensing is possible. The BST-condensed cube shown in Figure 2(b) is not a minimal one since some of single tuple properties are not explored. For example, base relation tuple R and R are base single tuples not only on $\{A\}$, but also on $\{B\}$ and $\{$ $\}$ respectively. Applying this information, we obtain a BST-condensed cube as shown in Figure 2(c). It can be seen that the cube size is further reduced, from 15 to 10.

An important property of the minimal BST condensed cube is that they are all equivalent, i.e., their projection on $_1, \ldots,$ $, M$ are the same.

Theorem 2.1 *All the minimal BST-condensed cubes of a relation are equivalent.*

The implication of Theorem 2.1 is as follows. Although there might exist multiple different minimal BST-condensed cubes for a base relation, they should only differ on the values on the SDSET field. Both the number of tuples in the cube and the values of each tuple in the dimension attributes and measure attribute should be the same.

2.4. Effectiveness of BST-condensing

In this subsection, we briefly discuss the effectiveness of BST-condensing. From our example, it is clearly shown that BST-condensing can effectively reduce the number of tuples in a data cube. There are 30 tuples in the complete cube in Figure 2(a), and only 10 tuples in the BST-condensed cube in Figure 2(c). The size reduction ratio with respect to the number of tuples is as high as 66.67%.

A few factors contribute to the effectiveness of BST-condensing. First, since a base single tuple on SD is also a BST on any superset of SD, it is likely that more than one tuple will be condensed into the base single tuple. In our example, there are 5 base single tuples. Two of them reduce 3 tuples each and one 4 tuples and the other 5 tuples respectively. Second, the single tuple phenomenon is quite common. If a cuboid is sparse, i.e., the product of the cardinality of k dimensions is much larger than the number of tuples in the base relation, it is highly likely that most base relation tuples will take up a different cell in the k-dimensional space, resulting in a large number of base single tuples. For example, consider a base relation R' with 1M tuples (following the uniform distribution), which has 10 dimensions and the cardinality of each dimension is 1000. We record the number of single tuples on k-attribute cuboids ($1 \leq k \leq$) in Table 1. We can see from the table that for most cuboids (specifically all k-attribute cuboids with k 3, which amounts to 94.53% of the 2^{10} cuboids), on average more than 99.90% tuples are single tuples. This indicates the huge potential benefit of applying our single tuple condensing technique.

Table 1. Single tuple percentage

k-attribute cuboid	Percentage of single tuples
1	0.00%
2	36.79%
3	99.90%
4	100.00%

3. MinCube: Computing a minimal BST-condensed cube

We have seen that single tuple condensing can effectively reduce the stored cube size. In this section, we present an algorithm, $Min\ \ ube$, that computes a minimal BST-condensed cube for a given relation.

3.1. The algorithm

$Min\ \ ube$ is a constructive algorithm that computes the minimal BST-condensed cube from a base relation. The main strategy is to process each cuboid in the cuboid lattice[3] in a breadth first and bottom-up fashion, and compute all the BSTs identified on each cuboid. It is also optimized to utilize the pruning effectiveness of BSTs to speed up the computation.

Algorithm 1 MinCube(R)

```
1:    t uct   tt ce( )
2:  Reset the BST bitmap index for .
3:  for all cuboid  in the lattice in a bottom-up fashion do
4:     e e  te        te u  eI  ex( );
5:    Sort the candidate tuples according to GroupByDim( ).
6:    Initialize the cuboid BST bitmap index to the candidate tuple index.
7:    Initialize the accumulator for .
8:    for all subsequent candidate tuple  do
9:      if  is the last tuple in the current group then
10:       if  is not the only tuple in the current group then
11:           ut ut          u  e( ccu  u  t  );
12:       else  is a BST on GroupByDim( )
13:         Set the corresponding entry for  in the cuboid BST bitmap Index.
14:         if the corresponding entry for  in the BST bitmap index is set then
15:             S      S  (      u          ( ));
16:         else If  is a newly-identified BST
17:           Set the entry for  in the cuboid BST bitmap index.
18:             ut ut  S  (      u          ( ));
19:         end if
20:       end if
21:     else
22:       Combine  with the accumulator for the aggregate function.
23:     end if
24:   end for
25:   Save the cuboid BST bitmap index.
26: end for
```

The algorithm uses three classes of bitmap indexes and each tuple in the base relation corresponds to 1 bit in the index respectively. The three indexes are: BST bitmap index, candidate tuple index and cuboid BST bitmap index. The first one records whether a base relation tuple has been identified as a BST so far. The second one is built for each cuboid and used by the pruning optimization. Only base relation tuples selected from this index are fed into the processing of each cuboid. The last index is generated after processing of each cuboid and keeps track of the BSTs identified in that process.

The initialization step of $Min\ \ ube$ algorithm includes building the lattice for n dimensions and initializing the BST bitmap index for R to 0 for all entries (line 1–2).

The algorithm then processes each cuboid in the lat-

[3] An example lattice is shown in Figure 1(c).

tice in a bottom-up and breadth first manner, i.e. all the cuboids of k GROUP BYs are processed before any of the cuboids of $k+1$ GROUP BYs can be processed. For each cuboid, a candidate tuple index is created by subroutine $Gene\ a\ e\ \ an\ i\ a\ e\ \ upleIn\ ex(\ )$ (Algorithm 2), and the candidate tuples (with 0 in the corresponding entry) form a subset of R. The candidate tuples are then sorted on the GROUP BY-attributes of cuboid $(G\ \ upBy\ \ im(\ ))$ (line 5). Line 6–7 initialize two data structures, a $cub\ i\ BS\ \ bi\ mapin\ ex$ and an $accumula$ for . The cuboid BST bitmap index is also used for the pruning optimization, and the accumulator is used to store the aggregate result of accumulated cube tuples. The sorted candidate tuples are sequentially processed in groups (line 8–24). A group of tuples is a set of tuples that have the same dimension values on all dimensions in $G\ \ upBy\ \ im(\ )$. No output is made until the last tuple in the current group is hit. If a tuple is not the last tuple in the current group, it is simply combined into the accumulator (line 22). Otherwise, we judge if it is the only tuple in the group. If so, it is identified as a BST on $G\ \ upBy\ \ im$ (line 12–20). We further look up the BST bitmap index for R to determine if is a new BST, i.e., the corresponding bit in the bitmap is not set. A tuple is written to the output BST-condensed cube by calling $Ou\ pu\ BS$, and the BST bitmap index is updated if is a new BST. Otherwise we simply locate and update the SDSET part of the existing BST-condensed cube tuple corresponding to . If is not a BST, the content of the accumulator is written to the output and the SDSET field is set to {} (line 11). At last, the cuboid BST bitmap index is saved (line 25). The correctness of the algorithm comes from the following lemma.

Lemma 3.1 *Algorithm $Min\ \ ube$ computes the minimal BST-condensed cube for the input base relation R.*

3.2. Pruning optimization

We can progressively reduce the number of candidate tuples to be processed at each cuboid by pruning those already identified base single tuples. Lemma 2.1 states that if is a BST on a cuboid , then it must also be a BST on any of 's ancestor cuboids. Furthermore, it is obvious that removal of from the candidate of cuboid $'$ ($'$ is an ancestor cuboid of) would not affect the result of other candidate tuples. Therefore, we can effectively prune the candidate tuple set as follows:

1. We use cuboid BST bitmap indexes associated with each cuboid. These indexes are generated after each corresponding cuboid is processed. An entry is set to 1 if the corresponding tuple is a BST on the the definition of current cuboid.
2. $Gene\ a\ e\ \ an\ i\ a\ e\ \ upleIn\ ex$ (Algorithm 2) constructs the candidate tuple index for each cuboid . It combines all the cuboid BST bitmap indexes of 's child cuboids' using the logical operator OR. The result bitmap index is called a candidate bitmap index, and an entry of value 0 indicates the corresponding base relation tuple is in the candidate tuple set and will be later processed.
3. Line 6, 17 and 25 in Algorithm 1 are used to maintain the cuboid BST bitmap index. Line 6 simply assigns the candidate tuple index to the cuboid BST bitmap index of . Line 17 sets the corresponding bits in the cuboid BST bitmap index for any newly identified BSTs. Finally, line 25 saves the cuboid BST bitmap index of for later use.

Algorithm 2 GenerateCandidateTupleIndex( )

1: Reset the candidate tuple bitmap index.
2: **for all** child cuboid of **do**
3: Combine the cuboid BST bitmap index of into the candidate tuple bitmap index using logical operator **OR**.
4: **end for**

The pruning techniques here progressively reduce the number of tuples in each cuboid, and result in a significant speedup in many datasets, especially when the datasets are not heavily skewed.

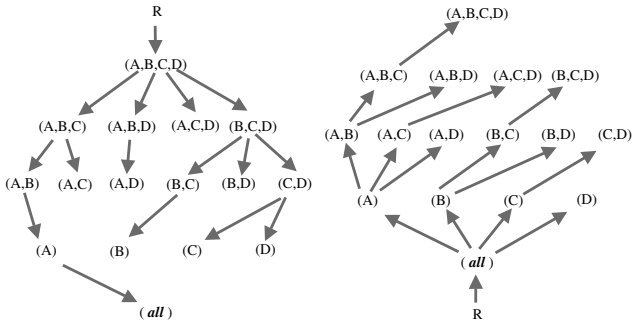
4. Condensed Cube: Heuristic algorithms

Algorithm $Min\ \ ube$ described in the previous section guarantees to find a minimal BST-condensed cube constructively. However, execution time of $Min\ \ ube$ is relatively long. In this section, we propose two heuristic algorithms to compute BST- and PST-condensed cube. They do not guarantee to generate the minimal condensed cube, but require less computation time.

4.1. Bottom-Up BST-condensing

Recall Lemma 2.1 that a base single tuple on SD is also a base single tuple on all supersets of SD. That is, during the computation, if we have identified a base single tuple in a cuboid, there will be no need to consider this tuple in the cuboids whose GROUP BY-attributes are the superset of the current ones. We only need to output the tuple as a base single tuple and record the relative single dimension information.

The cube computation algorithms proposed so far can be categorized into two groups, top down and bottom up. The major difference between these two approaches is the order in which the cuboids are computed. This can be illustrated by the example processing trees shown in Figure 3.



(a) Top-down computation of a cube (b) Bottom-up computation of a cube

Figure 3. Example processing trees for computing a cube

As shown in Figure 3(a), the top-down approach computes the base cuboid first. A cuboid of $n-1$ attributes is computed from an n-attribute cuboid that contains all the $n-1$ attributes. The rationale is to share the cost of partitioning and/or sorting, etc. among computations of related cuboids. Most cube computation algorithms belong to this category [2, 18, 13]. The bottom-up approach shown in Figure 3(b) was originally proposed to process so-called iceberg cube queries [4]. Iceberg cube queries only retrieve those partitions that satisfy user-specified aggregate conditions. Using the bottom-up approach, it is possible to prune off those partitions that do not satisfy the condition as early as possible.

BST condensing requires to identify those partitions with only one tuple, which is essentially similar to the requirement of iceberg queries. The algorithm $B\ \ m\ \ pBS$ is basically a modified version of the original BUC algorithm proposed by Bayer and Ramakrishnan [4].

Algorithm $B\ \ m\ \ pBS$ (BU-BST for short) predefines an order in which the dimensions are processed. Without loss of generality, we can assume the order is $_1,\ _2, \ldots,\ \ _i$, where $num\ \ ims$ is the total number of dimensions. For simplicity, we map this order to integers from 1 to $num\ \ ims$. During the computation, the $u\ pu\ Rec$ contains the current cube tuple, including the dimension attribute values, and the aggregate value and a bitmap. The bitmap is used to encode the single dimension information, `SDSET`, if a tuple is a base single tuple.

The algorithm is rather straightforward. The aggregate function is first applied to the entire input partition *input* (line 1). If the partition contains only one tuple, that is, a base single tuple is found, the bitmap is set (line 3) and the base relation tuple is written out as a single tuple. The partition will not be further processed based on BST principle, as those tuples should be condensed into the current tuple (line 5). Otherwise, the current cube tuple is written out and the computation continues using *d* between *dim* and *numDims* as the partition attribute (line 8–19). Upon return

Algorithm 3 BottomUpBST(input,dim)

```
 1: Aggregate(input, outputRec);   Places result in outputRec
 2: if input.count==1 then
 3:    SetBitmap(bitmap);   Record single dimension info
 4:    Output(input[0], bitmap);   Output the single tuple
 5:    return.   A base single tuple is identified, simply return
 6: end if
 7: Output(outputRec);
 8: for d=dim to numDims do
 9:    C=cardinality[d]
10:    Partition(input,d,C,dataCount[d])
11:    k=0
       For each partition on dimension d
12:    for i=0 to C-1 do
13:       c=dataCount[d][i]
14:       outputRec.dim[d]=input[k].dim[d]
          Recursively compute cuboid starting from next dimension
15:       BottomUpBST(input[k ... k+c],d+1)
16:       k+=c
17:    end for
18:    outputRec.dim[d]=ALL
19: end for
```

from $Pa\ \ i\ i\ n()$ (line 10), *dataCount* contains the number of tuples for each distinct value of the *d* dimension. The inner *FOR* loop (line 12–17) iterates through the partitions (i.e., each distinct value). Each partition becomes the input relation in the next recursive call to $B\ \ m\ \ pBS$ that computes the cuboids for dimensions *d* + 1 to $num\ \ ims$, starting from dimension *d*+1 (line 15). Upon return from the recursive call, we continue with the next partition of dimension *d*. Once all the partitions are processed, we repeat the whole process for the next dimension.

BU-BST produces BST-condensed cubes since it stops to process a partition with only one tuple. Let's use an example to illustrate. Assume that we have three tuples: (1, 1, 1, 10), (1, 1, 2, 10) and (2, 1, 1, 10). First, we partition data on A. The first partition has two tuples (1, 1, 1, 10) and (1, 1, 2, 10). We output (1, *, *, 20) as a cube tuple. Next, the partition is partitioned on B, the cuboid tuple output will be (1, 1, *, 20). This recursive process will produce (1, 1, 1, 10), (1, 1, 2, 10), and so on. For second partition with a single tuple (2, 1, 1, 10), the tuple (2, 1, 1, 10) will be written out as a single tuple and the processing for the partition will be stopped. The algorithm will continue to process the set of tuples by partitioning on B. Therefore, the cube tuples (2, *, *, 10), and (2, 1, *, 10) will not be produced.

Note that BU-BST does not generate all base single tuples. The main reason is that it fixes the order of dimensions in which the cube is generated. Therefore, if it is not properly ordered, a base single tuple may be detected late and some redundant cube tuples have already been produced. Let's use an example to illustrate this. Assume the input is

two tuples: (1, 1, 1, 10) and (1, 2, 1, 10). If the processing order is AB , we can only identify they are single tuples on AB, since the only partition of A has two tuples. If the partition order is BA , we will immediately discover that they are single tuples on B. It is obvious that the latter case will produce less tuple than the former one. In fact, the number of cube tuples generated with these two orders is 8 and 6, respectively. Tuples such as (*, 1, *, 10) and (*, 2, *, 10) will not be produced if B is processed first.

The benefit of fixed ordering is twofold. One is that, unlike $Min\ \ ube$, it runs in depth-first fashion and does not try to find all possible base single tuples on any combination of dimensions. Therefore, it is less time consuming. Another benefit is that, we can use a simple method to code the information of single dimensions: we only need to record from which dimensions, the tuple becomes a single tuple.

Two heuristics for ordering the dimensions for bottom-up cube computation are proposed in [4]. The first heuristic is to order the dimensions based on decreasing cardinality. The second heuristic is to order the dimensions based on increasing maximum number of duplicates. When the data is not skewed, the two heuristics are equivalent. For some real dataset, small difference has been found. In our implementation, we order the dimensions based on decreasing cardinality. The rationale is that, with higher cardinality and fixed number of tuples, the probability of a tuple being single tuple because of the attribute is higher. As we mentioned earlier, it had better to find such tuples as early as possible.

4.2. Reordered bottom-up BST-condensing

As illustrated in the discussion about algorithm BU-BST, dimension order in which the computation proceeds affects the effectiveness of BST-condensing. Although the heuristics such as decreasing order of cardinality provide reasonable performance to certain extent, some other orderings might be more effective. However, it is not easy, if not impossible, to find the optimal ordering.

Two observations lead to a heuristic algorithm, reordered bottom-up BST condensing, presented in this subsection. First, a good dimension order can reduce the size of cube dramatically. As experimental results presented in next section indicated, such reduction can be as high as 80–90%. Second, the major cost of cube computation is the I/O cost for writing out the results. As such, we may want to compute the cube to gather some statistics without output. Based on such statistics, we order the dimensions to actually compute the condensed cube. It is hoped that the overhead for the first computation will be smaller than the I/O cost saving. Furthermore, if the characteristics of a dataset do not change, it might be worthwhile to find a better ordering with certain amount of costs.

The algorithm, $Re\ \ e\ e\ B\ \ \ m\ \ pBS$ is a two

Algorithm 4 ReorderedBottomUpBST(input,dim)

1: Order dimensions in their decreasing order of cardinality; Run BottomUpBST to find the involved counts for each dimension
2: involvedCnt = BottomUpBST2(input, 1);
3: Reorder dimensions in their decreasing order of involvedCnt;
4: involvedCnt = BottomUpBST(input, 1);

phase algorithm as shown in Algorithm 4. In the first phase, dimensions are ordered based on their cardinality. A modified version of $B\ \ \ m\ \ pBS$, $B\ \ \ m\ \ pBS\ \ 2$ is executed. It identifies the single tuples, and record the number of occurrence of each dimension in the single dimensions of identified single tuples. We name this number the *involved count*. Then the dimensions are ordered in the decreasing order of their involved count. The actual condensed cube is obtained by executing $B\ \ \ m\ \ pBS$ again. The rationale for using the involved count is that the more time a dimension is involved in the generated single tuples, the higher chance the dimension is among the first few dimensions of the SDs of single tuples.

Experiments conducted indicate that the heuristic is actually quite effective. For certain dataset, it can find the order by which the condensed cube produced is close to the minimal BST-condensed cube.

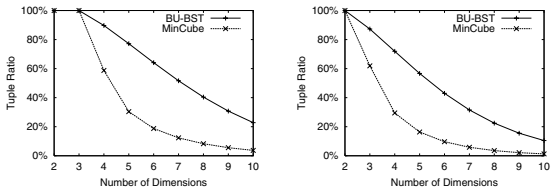
5. An experimental study

To assess the effectiveness of the condensed cube in size reduction as well as its effects on data cube computation, an experimental study has been conducted. In this section, we present some of the results.

All experiments are conducted on an Athlon 900Hz PC with 256M RAM, 20G hard disk. There are three classes of datasets used in the experiments. The first two are synthetic datasets with uniform and Zipf distribution. The other is the real datasets containing weather conditions at various weather stations on land for September 1985 [13]. This weather dataset was frequently used in recent data cube computation experiments [13, 4, 14].

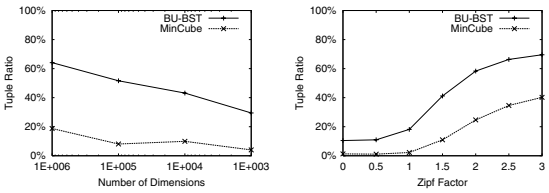
The effectiveness of condensing can be measured by the ratio between the size of a condensed cube and the size of the non-condensed complete cube. The size can be of two meanings, one is the number of tuples and the other is the actual size in bytes. We name them *tuple ratio* and *size ratio*, respectively. Relatively speaking, tuple ratio seems to be a better measure with the following reasons:

- Although a condensed cube introduces some space overhead to store information about single dimensions, the overhead should not be too high. Intuitively, each SD "removes" at least one cube tuple (though some cube tuples may be removed by more than one single



(a) Cardinality=100 (b) Cardinality=1000

Figure 4. Effectiveness of BST-condensing



(a) Tuple Ratio ($D = 6$, $C = 100$) (b) Tuple Ratio

Figure 5. Impact of sparsity and data skew

tuple). The size of one SD should be much smaller than the size of a tuple. Therefore, the tuple ratio should be also a good indication of size ratio.

- Usually, indexes will be built on data cubes to facilitate queries. The size of an index is directly related to number of cube tuples, rather than the actual size in bytes. Low tuple ratio means small number of tuples, hence small indexes.

5.1. Effectiveness of condensing

The first set of experiments studies the effectiveness of condensing. Synthetic data sets were used. The number of tuples in base relation was fixed to 1M tuples, which were randomly generated based on uniform distribution. We varied the number of CUBE BY-attributes from 2 to 10. The cardinality of all attributes were set to the same. We report two sets of results, cardinality equals 100, and 1000 in Figure 4(a), and 4(b), respectively.

Table 2. Tuple ratios ($= 10$)

Cardinality	BU-BST	MinCube
100	22.80%	3.71%
1000	10.48%	1.30%

In Table 2, we show the tuple ratios of condensed cube generated by two different algorithms: $Min\ \ ube$, and BU-BST. Several observations could be made from the results:

- The proposed condensing approach can effectively reduce the size of a data cube. As shown in Table 2, for a 10-dimension data cube, a minimal BST-condensed cube only has less than 2% of the tuples compared to the complete cube. Even using heuristic algorithms, the condensed cube has only about 10% tuples.
- The tuple ratio decreases when the number of CUBE BY-attributes and cardinality of attributes increase. With fixed number of tuples, increases in either CUBE BY-attributes or cardinality of attribute raise the probability of a tuple being a single tuple.
- As expected, $Min\ \ ube$ produces smaller cube than BU-BST. This is because BU-BST does not guarantee minimality, as $Min\ \ ube$ does.

5.2. Sparsity and skewness

The second set of experiments investigates the impacts of sparsity and data skew.

To create different degrees of sparsity, we fixed the number of dimensions ($= 6$) and the cardinality of each dimension ($= 100$ for all the dimensions), and varied the number of tuples from 1,000 to 1,000,000 with uniform distribution. The results, tuple ratio of cubes generated by two algorithms are presented in Figure 5(a). In general, the tuple ratio decreases with the increase of sparsity. This trend is also partially revealed in the previous experiments where data cubes become sparse implicitly.

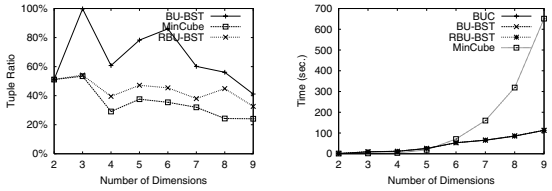
To study the effects of data skew, we generated datasets of 1M tuples following the Zipf distribution with different Zipf factors. A Zipf factor of 0 means that the data is uniformly distributed. By increasing the value of Zipf factor, we increase the skew in the distribution of distinct values in the datasets. The Zipf factor used ranges from 0.0 to 3.0 with 0.5 interval.

Figure 5(b) plots the tuple ratio for two algorithms under varying degree of skew. The tuple ratios of all the algorithms increase with the Zipf factor. It is expected that tuple ratio will increase as skew increases. The reason is that, with high skew, there would be fewer single tuples and the size of the complete data cube also decreases dramatically.

5.3. Experiments with real-world data: The weather data

The third set of experiments were conducted on a real-world dataset, the weather dataset. The data set contains 1,015,367 tuples (about 27.1MB). The attributes are as follows with cardinalities listed in parentheses: *station-id*(7037), *longitude*(352), *solar-altitude*(179), *latitude*(152), *present-weather*(101), *day*(30), *weather-change-code*(10), *hour*(8), and *brightness*(2). We generate 8 datasets with 2 to 9 dimensions by projecting the weather dataset on the first k dimensions ($2 \leq k \leq 9$).

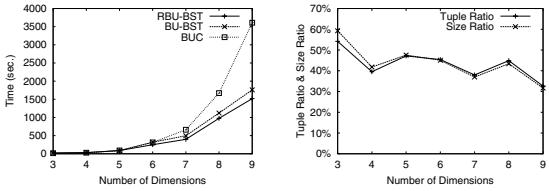
$Min\ \ ube$ algorithm and the two heuristic algorithms were executed during the experiment. Algorithm BU-BST uses the dimension order by the decreasing cardinality



(a) Tuple Ratio

(b) Execution Time (No I/O)

Figure 6. Experiments with weather dataset



(a) Execution Time (with I/O)

(b) Size Ratio vs. Tuple Ratio

Figure 7. RBU-BST vs. complete cube

heuristic. Algorithm RBU-BST first runs BU-BST with the decreasing cardinality dimension order and then runs BU-BST again with the order determined by the involved count heuristic. For example, if we use the cardinality to name the 9 dimensions, the decreasing cardinality heuristic will give the order of (7037, 352, 179, 152, 101, 30, 10, 8, 2) for the 9-dimension weather data set, while our involved count heuristic suggests the order of (30, 7037, 179, 101, 352, 152, 10, 8, 2).

Figure 6(a) shows the tuple ratio for the cubes generated by each of the algorithms. We can see that $Min\ \ ube$ and RBU-BST outperform BU-BST, while $Min\ \ ube$ is still the best in terms of the tuple ratio. The tuple ratios for all the algorithms with the 9 dimension dataset are listed in Table 3.

Table 3. Tuple ratios (weather dataset with $= 9$)

Algorithm	Tuple Ratio
BU-BST	41.13%
RBU-BST	32.62%
MinCube	24.09%

Figure 6(b) shows the running time for each algorithm without writing out the result. RBU-BST requires approximately double time of that of BU-BST since it actually runs BU-BST twice. As will be shown later, the tradeoff of this additional time results in a decrease in the output size, which is beneficial if we really write out the result. $Min\ \ ube$'s running time almost doubles with each increment of the number of dimensions. The running time with writing out the result is discussed in Section 5.4.

5.4. Reordered BU-BST

The last set of experiments investigates the performance of algorithms using single tuple condensing by comparing RBU-BST with a well-known fast algorithm BUC [4]. The experiment was repeated 7 times with the number of dimensions varying from 3 to 9.

Figure 7(a) shows the execution time used for each datasets with different number of dimensions. It can be seen that RBU-BST outperforms BUC algorithm. For example, RBU-BST takes about 42.09% of the running time of BUC. This is mainly because the performance of data cube computation algorithm is largely dominated by the I/O time of writing out the cube. RBU-BST writes out a much smaller condensed cube than the BUC algorithm which writes out a huge complete cube.

Recall that we mentioned another size ratio at the beginning of this section, the ratio of actual size in bytes. Figure 7(b) shows the tuple ratio and the actual size ratio. It can be seen they are very close to each other, which justified our choice of using tuple ratio as a metric in the experiment.

6. Related work

The work reported in this paper is inspired by data cube computation algorithms proposed in recent years, such as PipeSort, PipeHash, Overlap, ArrayCube, PartitionCube and BottomUpCube [2, 18, 13, 4], in particular, the work by Bayer and Ramakrishinan [4]. Since those algorithms are well noted, and we will not repeat here in detail.

Techniques directly related to size reduction include data compression, a classical technique. However, data compression in relational databases is not that popular. [9] gives a wide discussion on impact of compression technology in database from an architectural view, with main emphasis on the choice of algorithms. In the field of statistical database, transposed table is used and [17] further takes the advantage of encoding attribute values using a small number of bits to reduce the storage space. Run length encoding was also proposed to compress the repeating sub-sequences that are likely to occur in the least rapidly varying columns. Various compression techniques are also adopted in the field of MOLAP, mainly to handle the sparsity issue [18, 11].

Approximate query processing can be viewed as a technique that addresses the size problem from another angle, which is especially popular in the data warehouse environment. [6] argues the importance of the synopsis data structures. Characterized by their substantially small sizes, synopsis data structures can efficiently provide approximate query results in many applications. The statistical methods and models used for the purpose of fast approximate

query answering in the data warehouse environment include wavelet [16], multivariate polynomials [3], mixed model by multivariate Gaussians [15], histogram [12], sampling [1], etc.

Several specialized data structures for fast processing of special types of queries were also proposed. Prefix sum and relative prefix sum methods [5] are proposed to answer range-sum queries using a large pre-computed prefix cube. More recently, the hierarchical compact cube is proposed to support range max queries [10].

7. Conclusions

In this paper, we proposed a new approach to reducing the size of a data cube, and hence its computation time. The approach explores the properties of single tuples so that a number of tuples in a cube can be condensed into one without loss of any information. A simple operation can expand a condensed tuple to all the "hidden" cube tuples it represents. There is no overhead for decompression or further aggregation. A novel condensing scheme, namely the base single tuple (BST) condensing, is introduced. An algorithm that computes a minimal BST condensed cube is proposed. Heuristic algorithms for computing BST condensed cubes are also presented. Experimental results indicate that, the tuple reduction ratio could be very high for uniformly distributed synthetic data. For highly skewed data and a real-world data set, the reduction rate can also reach 70–80%. Since the major cost of cube computation is the I/O cost for writing out the cube tuples, the great reduction of cube size leads to efficient cube computation as well.

This paper, limited by space, concentrates on the basic concepts, definitions, and computation of the proposed condensed cube. We deliberately leave out a number of related issues that require lengthy discussions, which will be included in the full version of this paper.

The concept of condensed cube introduces several interesting issues for the future research. First, in terms of storing the condensed cube, we may also consider to detach the single dimension information from the tuples, and store them at different areas. In this way, the schema of cube does not need to change. The `SDSET` information will be accessed only when it is needed. Second, we aim to develop some query evaluation and optimization techniques so that the size benefit of condensed cubes can be fully explored. Third, we will study reorganizing an existing cube into a condensed one, which might be most beneficial for certain applications where update is infrequent and complete cubes are available.

As a summary, we believe that cube condensing is a promising approach to supporting efficient on-line analytical processing. With the framework specified in this paper, a set of research issues has been identified. We are currently further addressing those issues and aiming at developing an efficient OLAP server based on such a new structure.

References

[1] S. Acharya, P. B. Gibbons, and V. Poosala. Congressional samples for approximate answering of group-by queries. In *SIGMOD 2000*, volume 29, pages 487–498, 2000.

[2] S. Agarwal, R. Agrawal, P. Deshpande, A. Gupta, J. F. Naughton, R. Ramakrishnan, and S. Sarawagi. On the computation of multidimensional aggregates. In *VLDB'96*, pages 506–521, 1996.

[3] D. Barbará and M. Sullivan. Quasi-cubes: Exploiting approximations in multidimensional databases. *SIGMOD Record*, 26(3):12–17, 1997.

[4] K. S. Beyer and R. Ramakrishnan. Bottom-up computation of sparse and iceberg cubes. In *SIGMOD 1999*, pages 359–370, 1999.

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

[6] P. B. Gibbons and Y. Matias. Synopsis data structures for massive data sets. *SODA 1999*, pages 909–910, 1999.

[7] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-total. In *ICDE 1996*, pages 152–159, 1996.

[8] D. Gunopulos, G. Kollios, V. J. Tsotras, and C. Domeniconi. Approximating multi-dimensional aggregate range queries over real attributes. In *SIGMOD 2000*, volume 29, pages 463–474, 2000.

[9] B. R. Iyer and D. Wilhite. Data compression support in databases. In *VLDB'94*, pages 695–704, 1994.

[10] S. Y. Lee, T. W. Ling, and H.-G. Li. Hierarchical compact cube for range-max queries. In *VLDB 2000,*, pages 232–241, 2000.

[11] J. Li, D. Rotem, and J. Srivastava. Aggregation algorithms for very large compressed data warehouses. In *VLDB'99*, pages 651–662, 1999.

[12] V. Poosala and V. Ganti. Fast approximate answers to aggregate queries on a data cube. *SSDBM 1999*, pages 24–33, 1999.

[13] K. A. Ross and D. Srivastava. Fast computation of sparse datacubes. In *VLDB'97*, pages 116–185, 1997.

[14] K. A. Ross and K. A. Zaman. Serving datacube tuples from main memory. *SSDBM 2000*, pages 182–195, 2000.

[15] J. Shanmugasundaram, U. M. Fayyad, and P. S. Bradley. Compressed data cubes for olap aggregate query approximation on continuous dimensions. *SIGKDD 1999*, pages 223–232, 1999.

[16] J. S. Vitter, M. Wang, and B. R. Iyer. Data cube approximation and histograms via wavelets. In *CIKM98*, pages 96–104, 1998.

[17] H. K. T. Wong, H.-F. Liu, F. Olken, D. Rotem, and L. Wong. Bit transposed files. In *VLDB'85*, pages 448–457, 1985.

[18] Y. Zhao, P. Deshpande, and J. F. Naughton. An array-based algorithm for simultaneous multidimensional aggregates. In *SIGMOD 1997*, pages 159–170, 1997.