Datacube Computation in the Presence of Attribute Hierarchies

M. Madhava Krishna and P. Sreenivasa Kumar
Department of Computer Science and Engineering
Indian Institute of Technology Madras
Chennai - 600 036
India

{madhav, psk}@aries.iitm.ernet.in

ABSTRACT

Datacube queries have become an important class of aggregate queries in decision support applications. These queries compute aggregates over all possible combinations of a specified list of group-by attributes. Substantial amount of research has been carried out for computing the datacube efficiently over large volumes of data. Previous works focussed on computing a data cube over a single relation, in which all CUBE-BY attributes are present. In OLAP applications it is common for attributes to have a hierarchy associated with them, which is captured in dimension table corresponding to the attribute. Computing the datacube over a set of attributes, of which few are hierarchical attributes, requires a join between fact table and dimension tables. Joins are typically expensive operations, particularly when the relations involved are substantially larger than main memory. Our algorithm extends the BUC algorithm [K. Beyer, R. Ramakrishnan. Bottom-Up Computation of Sparse and Iceberg CUBEs. In *proc. of the ACM SIGMOD Conf .*, pages 359-370, 1999], to compute the datacube in the presence of attribute hierarchies. By using Encoded Bitmap Index, we avoid the actual join of fact and dimension tables and also substantially improve the counting sort phase of BUC algorithm.

1. INTRODUCTION

Decision Support systems enable the analysis of enterprise data for making better decisions in a reasonable amount of time. OLAP (On Line Analytical Processing) is an interactive decision support tool that assists the analyst in extracting business information from the data. OLAP provides consolidated and aggregated data at various levels of detail and across many dimensions. Recently J. Gray et.al. [6] proposed *datacube* operator that has turned out to be one of the core operators of OLAP. It generalizes the standard SQL *group-by* operator to compute multiple group-bys for every combination of grouping attributes. Given d number of CUBE-BY attributes, datacube operator computes 2^d group-bys. For instance, consider a simple multi-dimensional data model with a single relation *Transactions* (ProductID, StoreID, Date, Sales). Datacube query over this relation with ProductID, StroeID, Date as CUBE-BY attributes results in aggregating *Sales* by ProductID, StoreID, Date; *Sales* by ProductID and StoreID; *Sales* by ProductID, Date; *Sales* by StoreID, Date; *Sales* by StoreID; *Sales* by ProductID; *Sales* by Date; and finally overall *Sales*.

OLAP tools are supported by databases that adopt Multi Dimensional Data (MDD) model. *Measure* and *Dimension* attributes are basic components of MDD model. Measures are the objects of interest, each of which are mapped from a set of dimensions. In the above example, *Sales* is the measure value which is analyzed over the dimensions ProductID, StoreID, Date. Dimensions usually have hierarchies associated with them, which can be used in viewing the data at different granularities. For instance, in the above example StoreID can have the hierarchy $StoreID \rightarrow City \rightarrow Region \rightarrow Country$.

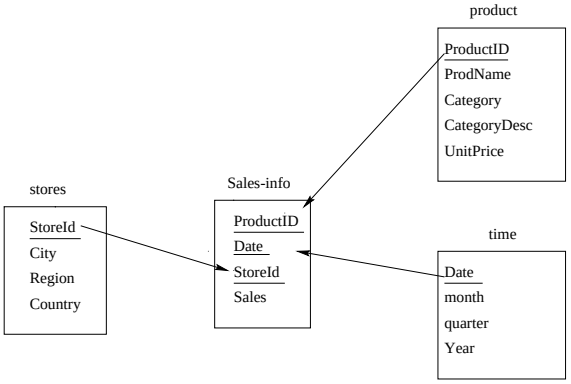

Figure 1: Star schema

Star Schema [8, 4] is a popular approach used for implementing OLAP in relational model. In star schema, data is stored in *fact* tables and *dimension* tables. Fact table is characterized by a set of dimension attributes and contains

ADVANCES IN DATA MANAGEMENT 2000
Krithi Ramamritham, T.M. Vijayaraman (Editors)
Tata McGraw-Hill Publishing Company Ltd.

the corresponding measure values. Each dimension attribute is described in a separate relation dimension table, if it has any related information including hierarchy. Referential integrity holds between fact table and dimension tables. For example, above mentioned multi-dimensional model can be implemented in star schema as shown in Figure 1.

Existing cube computational algorithms do not discuss the computation of cube in the presence of attribute hierarchies. For computing the datacube in the presence of attribute hierarchies, the corresponding dimension tables have to be explicitly joined with fact table. In this paper, we address the issue of efficiently computing datacube in the presence of attribute hierarchies. The following contributions are made in this paper.

1. Introduction of a new approach for computing star joins without using any additional access structure, apart from the index structure on the attribute.

2. BUC algorithm has been extended to handle attribute hierarchies and compute the datacube queries that may have some of the attributes from the hierarchies.

3. An improvement in the performance of counting sort used for partitioning the input over a specified attribute.

Paper outline Section 2 describes the previous work on computing the CUBE and their limitations. In Section 3, we discuss the draw backs of join, join indexes and the new type of join technique. Section 4 introduces Encoded Bitmap Index, which we have used to avoid explicit joins. In Section 5, we discuss about how we have extended BUC [3] to handle attribute hierarchies in computing the CUBE. Section 6 presents the performance results of the proposed algorithm. We conclude this paper in Section 7.

2. RELATED WORK

Datacube operator was introduced in [6]. Aggregate functions used by CUBE operator are categorized into *distributive, algebraic* and *holistic* functions. Distributive functions allow data to be partitioned into disjoint sets that can be aggregated independently and later combined to get the final output. For instance COUNT, MIN, MAX, SUM. Algebraic functions can be expressed in terms of distributive functions e.g., AVG, Standard deviation, MaxN, MinN. Holistic functions can not be computed in parts and later combined to get the final result e.g., Median and Rank.

Consider a datacube query with four CUBE-BY attributes *A, B, C, D* from a relation *R* with aggregate function SUM on measure attribute *M*.

```
SELECT    A, B, C, D, Sum (M)
FROM      R
CUBE-BY   A, B, C, D
```

This query results in computing 16 (2^4) group-bys. Among them $ABCD$ is the most detailed group-by and ALL (group-by NONE) is the most aggregated group-by.

The property of distributive functions allows to compute less detailed aggregates from more detailed aggregates. Same property holds good with algebraic functions, since they can be expressed in terms of distributive functions. This property induces a partial ordering on all the group-bys of the cube that can be captured in terms of a lattice [7]. Each node in the lattice is a group-by. Let $\{Q_1, Q_2, \ldots, Q_n\}$ be the set of nodes in a lattice. A node Q_i is said to be a child of parent node Q_j, if Q_i can be computed from Q_j. Figure 2 depicts lattice of the datacube considered above. Following subsections discuss the existing techniques for computing the datacube efficiently.

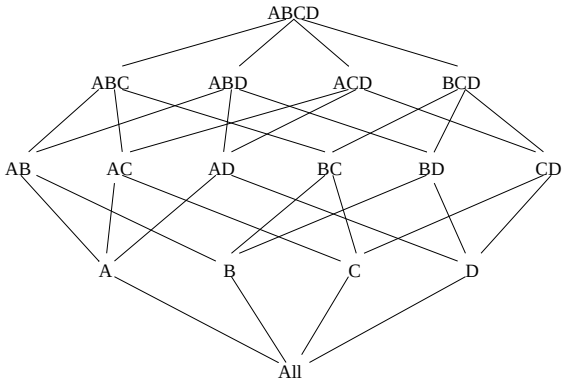

Figure 2: Cube view lattice

2.1 Pipe Sort

Sharing the cost of sorting across the computation of multiple group-bys is the advantageous property of the pipesort method [13, 1]. Considering the above mentioned example if the input is sorted in $ABCD$ order, then all the group-bys with sort order prefix to $ABCD$, such as ABC, AB, A and ALL can be computed simultaneously without requiring any additional sorts. Pipeline technique is used to achieve simultaneous computation of selected group-bys e.g., $ABCD \rightarrow ABC \rightarrow AB \rightarrow A \rightarrow ALL$. Pipesort first converts the given lattice into a minimum spanning tree such that each group-by is connected to its smallest parent and the number of sorts required is also minimum. Then pipeline paths are extracted from the tree and are executed one-by-one. Each pipeline path requires a sort on input in the order same as that of the root of the path. Remaining group-bys that are prefixes to that of the root node can be computed without any additional sorts in a pipeline fashion. Since all the group-bys in the pipeline path are computed in a single scan of the input, disk scan cost is reduced. At any time of execution, pipesort requires only one tuple to be memory resident.

Observations: The lower bound of sorts required in pipesort is given by $k\mathrm{C}_{\lceil k/2 \rceil}$, where k is the number of CUBE-By attributes.

Limitation: When the underlying relation is sparse and larger than main memory, external sorting is required which is a time consuming process.

2.2 Pipe Hash

Pipehash method [13, 1] also computes group-by from its smallest parent. It converts the given lattice into a minimum spanning tree (MST) such that each group-by is connected to a single group-by that is its smallest parent. If

the available memory is sufficient to fit hash tables of all the group-bys, whole cube can be computed in a single scan of input. Otherwise, it splits MST into smaller sub-trees that can fit in memory and process each one independently within memory constraints.

Observations: For each group-by, all its children are computed in a single scan of the group-by.

Limitations: It does not overlap much computation cost as pipesort, since it computes multiple group-bys from a parent group-by sorted in the required order. In contrast, pipehash builds different hash tables for each group-by. Thus it must rehash the data for each group-by. It needs significant amount of memory to store the hash tables for the group-bys.

2.3 Overlap Method

The basic idea of this method [5, 1] is to utilize the existing sort order of a parent group-by to compute all possible group-bys within the memory constraints. Thus existing sort order is not only used to compute group-bys that are prefixes to it but also to compute the partially matched group-bys. Overlap chooses a sort order for the root of the lattice, depending on which sort orders of remaining group-bys are fixed. The heuristic used in selecting the parent for a group-by is to choose the group-by with longest prefix matching. The resulting tree derived from cube-view lattice by applying the above said heuristic is called processing tree, after which partition size of each group-by is evaluated. The partition size of a group-by depends on the cardinality of common prefix. For example, while computing ABC and ABD from $ABCD$ the partition size for ABC is one tuple since it is proper prefix of $ABCD$. Where as, for ABD partition size depends on $|D|$. Group-bys can be either in *partition* state or *sort-run* state. If entire required memory is allocated then it is said to be in partition state, otherwise in sort-run state for which only one page of memory is allocated. The allocation of memory to group-bys is done, by traversing the MST in a breadth-first order, giving priority to group-bys with smaller partition sizes and with larger attribute list. If enough memory is available, overlap computes the entire datacube in a single pass. Otherwise it splits the processing tree into subtrees in such a way that each subtree can be computed within the memory constraints. Thus the number of passes required to compute the datacube is proportional to the number subtrees.

Limitations: I/O cost of Overlap is at least quadratic in k which is significantly large, for sparse data sets where k is the number of attributes.

2.4 Fast Cube

Fastcube method [12] is designed for computing datacube over sparse data relation. This method adopts divide and conquer strategy for computing a datacube over a relation that do not fit in memory. If the available memory is sufficient to accommodate the input relation, then the entire datacube can be computed in a single pass. Otherwise it recursively partitions the base relation until each resulting partition fits in memory. For each partition that fits in memory, group-bys are computed which contain the partitioning attributes. A datacube is split into sub-datacubes that can be computed within the memory constraints. Remaining sub-datacubes are computed in the subsequent passes. Instead of using the input relation everytime to compute a sub-datacube, most detailed group-by that is already computed in the previous pass is used. This algorithm uses the same pipeline technique of *pipesort* in computing the datacube for an input that fits in memory. Algorithm uses optimal number of pipeline paths in computing the datacube over the given input.

Observations: It was shown that optimal number is equal to the minimum number of paths required in computing a datacube. It is given by $k\mathrm{C}_{\lceil k/2 \rceil}$, where k is the number of CUBE-BY attributes.

2.5 BUC

BUC [3] looks at the problem of cube computation in a different way. In addition to the computation of entire cube, it also considers the computation of *Iceberg-cube*. The Iceberg-cube computation (IBCC) problem is to compute aggregates for every combination of cube-by attributes over only those partitions that satisfy the user specified condition, in contrast to the basic cube computation problem that computes aggregates for every combination of cube-by attributes. Cube computation problem is a special case of Iceberg-cube computation. Iceberg-cube problem is different from Partial Cube Computation (PCC) [7]. In PCC, a subset of group-bys is selected apriori before the actual cube computation starts. The selection of these group-bys is based on the disk space availability, cost of computing. IBCC considers the partitions in deciding whether to perform aggregation for that particular group-by. Here selection is done dynamically with the execution of cube computation.

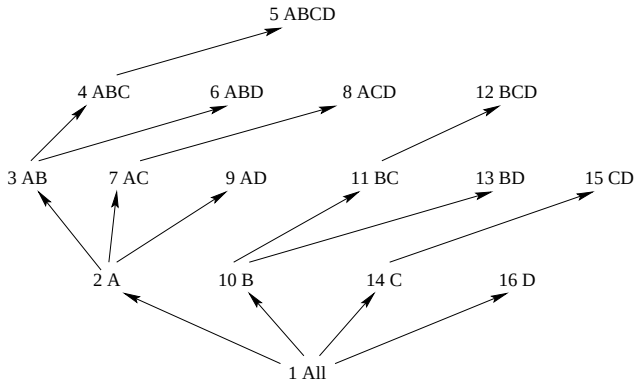

Figure 3: Processing tree of BUC

BUC adopts diffrent approach from previous algorithms to compute the data cube. Previous algorithms compute the cube starting from the most detailed group-by to the most aggregated group-by. Thus they traverse the cube lattice in top-down fashion while computing the cube. In contrast, BUC traverses the cube lattice in bottom-up fashion, which is just opposite to that of the previous algorithms. That is, it starts computing cube from the most aggregated group-by to the most detailed group-by. For instance, the processing tree of BUC while computing the cube lattice given in Figure 2 is as shown in Figure 3.

The idea behind this approach is pruning the computa-

tion of group-bys that do not meet the minimum support specified. Condition can be specified using the HAVING clause in SQL. For example, user wants to consider those partitions that contain more than N tuples for aggregation. This Iceberg query can be expressed in SQL with CUBE-BY clause as follows:

SELECT	A, B, C, D, COUNT (*), SUM (X)
FROM	R
CUBE-BY	A, B, C, D
HAVING	COUNT (*) $\geq$N

Where N is called *minimum support* or *MinSup*. Iceberg cube query with Minsup equal to 1 is identical to full cube computation problem. To achieve pruning, BUC proceeds from the bottom of the lattice, and works its way up, towards the larger and less detailed group-bys. While computing the cube if a partition does not meet the MinSup, all it's ancestors in the lattice also do not meet the MinSup and hence there is no need to perform further aggregation and partition.

BUC does not share any aggregation cost between parent and child group-bys in the lattice, as it argues that there is no much reduction in computation cost by sharing the aggregation cost. It has shown that overall cost can be reduced, by sharing the partition costs.

One more important work on hierarchies was done by Baralis et al [2]. They proposed a formal frame work for the definition of hierarchies. They focussed on selection of views from cube lattice for materialization in the presence of attibute hierarchies. They tried to eleminate the redundancy in views that arise due to the functional dependancy. For example, consider a query grouping on a dimension key d_i and also on an attribute a_j of the same dimension D_i. Since there exists a functional dependacy from d_i to a_j, a query grouping on $\{d_i, a_j\}$, must produce the same result of the query grouping on $\{d_i\}$. Due to this elemination process, number of views to be computed in the lattice are reduced resulting in less time for the computation of cube. But they do not propose a method to compute the datacube in the presence of hierarchies.

3. MOTIVATION

Consider a multi dimensional database for a grocery store that owns member stores located in different places. The schema of the database is as shown in Figure 1. It consists of a fact table *Sales-info* that provides the sales information on which the actual analysis is performed and three dimension tables *stores, time, product*. The hierarchies associated with ProductID, StoreID and Date are as follows.

$$ProductID \rightarrow ProdName \rightarrow UnitPrice$$
$$StoreID \rightarrow city \rightarrow region \rightarrow country$$
$$Date \rightarrow month \rightarrow quarter \rightarrow year$$

Consider the computation of datacube query over the attributes ProductID, city, month with SUM as the aggregate function over sales.

SELECT	ProductID, city, month, SUM (sales)
FROM	Sales-info, stores, time
CUBE-BY	ProductID, city, month

It is observed in the above section that all the discussed algorithms compute the cube over a single relation. To execute the above query, join of the dimension tables *stores* and *time* with *sales-info* is performed and the resultant table is given as input to the algorithms. Drawbacks of join are as follows:

- Join is typically a time consuming and resource consuming process, especially when the participating relations require memory exceeding the available main memory.
- The tuple size of the resulting table after the join becomes large. This needs more memory at the time of computation, in turn reducing the efficiency of algorithm.

Computing join operation efficiently is a critical part of the process. A number of techniques have been evolved to avoid explicit join between the relations. These techniques use a pre-computed access structure, namely *join index*. A join index between two relations maintains a pair of identifiers of tuples that would match in case of a join. The join index is to be maintained by the database system, and updated when tuples are inserted and deleted in the underlying relation. *Star joins* have been used to denote the join of a fact table with multiple dimension tables. The star join condition typically includes a selection condition on dimension tables as well.

Various types of *bitmap join indices* have been proposed to efficiently deal with star joins [9, 10]. Before discussing the bitmap join index we shall introduce the *Simple Bitmap Index* (SBI), the first member of the bitmap indexing family introduced in [9, 11]. The basic idea is to use a string of bits (0 or 1) to indicate whether an attribute in a table is equal to a specific value or not. For example, consider a simple bitmap index on an attribute *GENDER*. Let the domain of *GENDER* be Male, Female. Then the bitmap index on it consists of two bit vectors, say B_{Male} and B_{Female}. For B_{Male}, the bit is set to 1, if the associated tuple has the attribute ($GENDER = Male$), otherwise the bit is set to 0. The sample bitmap index on attribute *GENDER* is shown in Figure 4.

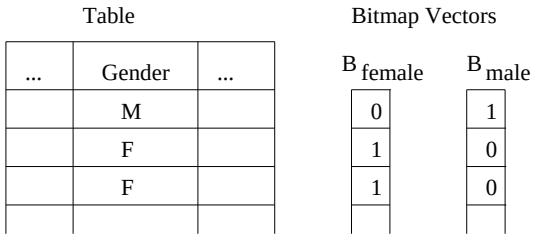

Figure 4: Simple bitmap indexing

A bitmap join index is a bitmap index on the fact table based on a single attribute of a dimension table. The bitmap join index is useful precisely when the OLAP queries consist of a selection condition on attribute A of dimension table T and a join condition between the fact table F and the dimension table T. Join indices can only provide the matching tuple IDs of the dimension table for the tuples of the

fact table. The exact value can be obtained either by reading the corresponding dimension table or the value index of that particular hierarchical attribute if it exists. Drawbacks of join indices are as follows:

- Join indices have to be computed apriori to the actual computation of cube.
- If materialized, a need to update the join indices in addition to the original bitmap indices.
- Space requirement increases, as the table size becomes large.

We propose a new approach of joining relations for which referential integrity holds. The proposed method doesn't need any pre-computed access structure to perform the join and also no need to go for dimension table. We will show in the following section, how we can make use of Encoded Bitmap Index to avoid explicit join between relations.

4. OVERVIEW OF ENCODED BITMAP INDEX

Many variations of bitmap indexing have been proposed to solve the problems arising due to sparsity in data. Each provide significant performance advantages for certain class of queries. It may be desirable to have more than one type of index available on an attribute, so that the best index can be chosen for the query at hand.

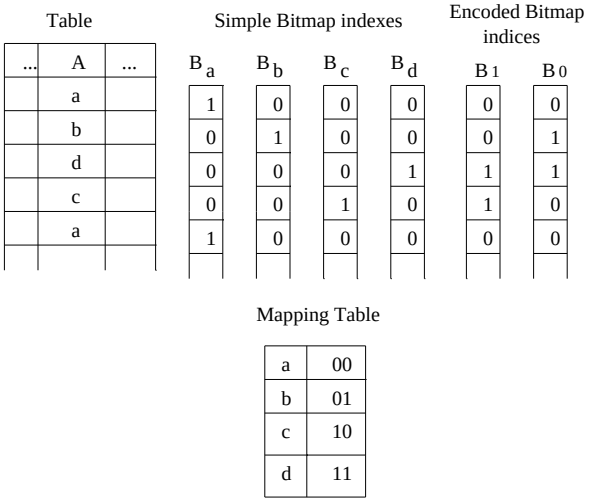

Figure 5: Encoded bitmap index

Recently [15] introduced a new type of bitmap index called *Encoded Bitmap Index* (EBI), which is basically an extension to simple bitmap indexing. This indexing mechanism outperforms all other bitmap index mechanisms either in the aspect of space requirement or response time for queries. EBI applies an encoding function or mapping on the attribute domain and builds a binary-based bit-sliced index on the encoded domain. For example, consider an attribute A with domain *a, b, c, d.* EBI for this attribute consists of $\lceil \log_2 4 \rceil = 2$ bitmap vectors B_0, B_1. In general, the bitmap vector B_i stores the i^{th} bit of the encoded value of attribute A. Figure 5 shows the bitmap vectors B_0, B_1 and the mapping table used to encode the values of attribute A.

Consider a fact table, *Sales-info*, containing N tuples of data and a dimension table, *Products*, containing information about 12000 different products. Building a simple bitmap index on the *ProductID* dimension of *Sales-info* table, results in 12000 bitmap vectors, each of N bits in length. Space required by this index is $\lceil (12000 * N)/8 \rceil$ bytes. In contrast, creating EBI on *ProductID* requires $\lceil \log_2 12000 \rceil = 14$ bitmap vectors, each of N bits in length. Thus the space requirement of EBI is $\lceil 14 * |N| \rceil$ bits plus a mapping table, which is much lesser compared to simple bitmap indexing. In general, if N is the relation size and M is the cardinality of an attribute A then space requirement of SBI is $\lceil (M * N)/8 \rceil$ bytes. The space required for EBI is $N * \lceil \log_2 M \rceil + (k + \lceil \log_2 M \rceil) * M$ bits. Where k is a constant depending on the size of attribute values. The former term accounts for the storage of bitmap vectors and the later term accounts for the storage of mapping table. For reasonably large N, this cost is much less than that of SBI.

5. EXTENDING BUC TO HANDLE ATTRIBUTE HIERARCHIES

The basic requirement of BUC over the input data is to have all the dimension attribute values mapped to consecutive integers between zero and the attribute's cardinality. This is same as what is being done while constructing an EBI over an attribute. The mapping step of BUC can be eliminated, if all the dimension attributes including hierarchical attributes have EBI. The above assumption is not only helpful in avoiding the mapping of attribute values but also allows us to perform join without any additional access structure. Since the join between fact and dimension tables is an equi-join, each tuple from fact table would match uniquely with a single tuple in dimension table. Typically, datacube queries do not impose any selection predicate on the join attribute. Hence join can be viewed as replacing the values of the dimension attribute with the corresponding hierarchical dimension attribute values.

Returning to our previous example, viewing the sales information by *region* needs to map all the *storeID's* in the fact table with the corresponding regions in which they fall. This mapping information is present in dimension table *stores*. Thus it can be viewed as replacing the attribute *storeID* in the fact table with attribute *region* of dimension table *stores*. This is done using EBI with the following assumptions.

- All the dimension attributes including hierarchical attributes must have EBI.
- All the primary keys of dimension tables must be ordered according to their respective mapped values.
- All the primary keys of dimension tables and the corresponding foreign key attributes in fact tables must use the same mapping functions to map the values to their domain.

5.1 Algorithm

The algorithm ExtendedBUC is as given below. Aggregation on given input is performed at the beginning. Then for each dimension attribute i between d and *dimAttribs* the

loop [8 to 17] is iterated. First, the input is partitioned on attribute i. Then for each partition of attribute i, ExtendedBUC is called recursively if it meets the *Minsup*. BUC is called recursively for each partition that meets the minimum support, to compute the CUBE on that partition for dimensions *i+1* to *dimAttribs*. After the computation of CUBE over all partitions of i^{th} attribute is over, entire process is repeated for the remaining attributes.

Global variables:

dimAttribs:	The total number of CUBE-BY attributes.
hieAttribs :	The number of hierarchical attributes.
Minsup :	The minimum number of tuples a partition must hold, to be considered for cube computation.
cardinality[dimAttribs]:	Stores the cardinality of each CUBE-By attribute.
countArray[dimAttribs][q]:	Stores the size of each partition of a particular CUBE-BY attribute, whose cardinality is q.
bitMap[hieAttribs][ ][ ]:	It is a three dimensional array that stores the encoded bitmap vectors of each hierarchical attribute.

Algorithm: ExtBUC(int *min*, int *max*, int *d*)

Input: A set of tuples from Relation R specified with boundaries *min*, *max* and starting dimension d

```
aggregate(min,max,d);
if((max-min)<2)
{
  \\No need of further aggregation and partition
  return();
}
for(i=d; i<dimAttribs; i++)
{
  partition(min,max,i);
  q = cardinality[i];
  for(j=o; j<q; j++)
  {
    if((countArray[i][j+1]-countArray[i][j])>MinSup)
    {
      ExtBUC(countArray[i][j],countArray[i][j+1],i+1);
    }
  }
}
```

Algorithm: Replace(int *min*, int *max*, int *d*)

Input: A set tuples from relation R specified with boundaries *min*, *max* and the attribute d for which replacement has to be done.

```
Check whether the attribute values in the given
input are replaced or not.
if(not replaced)
{
  q = log2(Cardinality(d));
  for(i=0; i<(max-min); i++)
  {
    temp = 0;
    for(j=0; j<q; j++)
    {
      if(bitMap[d][j][currentRecord[d]] == 1)
      {
        temp += 2pow(j);
      }
    }
    currentRecord[d] = temp;
  }
}
```

The first step of *partition* module is replacement as in algorithm *Replace*. Replacement is done for only those dimension attributes of fact table on which attribute hierarchy is specified in the query. Algorithm *Replace* replaces the current partitioning attribute value of fact table by the specified hierarchical attribute value from the corresponding dimension table. A check is performed to find out whether that attribute values are already replaced or not. If true, it returns to partition module without any replacement. The input for this algorithm are a set of tuples from fact table, attribute to be replaced and the hierarchical attribute along with its index structure. Assuming the EBI of each dimension attribute of the schema to be memory resident during the process of cube computation, disk scan cost can be reduced. Each EBI index file of an attribute contains bitmap vectors stored in transpose fashion. For instance, consider an EBI containing bitmap vectors B_0, B_1 and B_2. The order in which they are stored is B_0, B_1 and B_2.

Consider an instance of fact table *sale-info* and dimension table *stores* as shown in Figure 6 below. The value of *storeID* in fact table directly gives the matching row number of dimension table *stores* from which *region* is to be extracted. This is possible because of our first assumption. Instead of going to dimension table *stores*, index file of *region* is used to get the corresponding mapped value of region. Thus replacement is done in terms of mapped values of the dimension attributes. Due to this, there is no need of mapping the newly replaced values to its domain for counting sort algorithm. Figure 6 shown below illustrates the replace algorithm for an instance of both fact and dimension tables.

This replacement can be done either before starting the actual cube computation (*join-apriori*) or dynamically while computing the cube (*join-dynamic*). In BUC computation of group-bys that do not meet the user specified aggregate condition are pruned. This is decided dynamically by checking the condition for each partition. Thus aggregation is performed for those partitions that satisfy the condition. In the same way replacement of the dimension attributes can be pushed into the cube computation. Thus, in join-dynamic partitions that satisfy the user specified condition will only be considered for replacement, due to which computation cost will be reduced. And in addition, we are reducing one scan of entire fact table, unlike join-apriori. As the data becomes sparse, the performance of join-dynamic improves over join-apriori. Section 6 gives the performance results of both join-apriori and join-dynamic.

The majority of the time in BUC is spent in partitioning the data, thus partitioning data is critical with respect to optimizing the BUC. We used counting sort algorithm

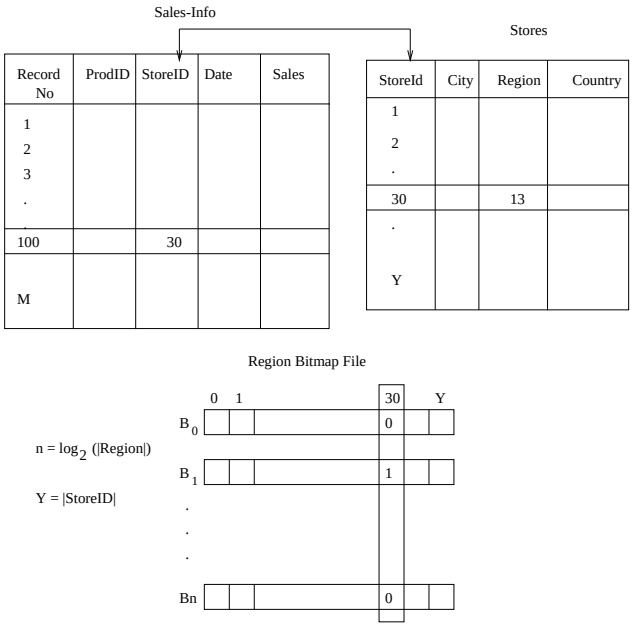

Figure 6: Illustration of replacement

to partition the given set of tuples on a specified attribute. Counting sort scans the input twice to partition it. In the first scan it counts the occurrences of each distinct value and computes the relative place of each one when they are actually sorted. Using this information it sorts the input in the next scan. Thus the time complexity of this algorithm is O(N). The performance of counting sort degrades as the ratio of the input size and the partitioning attribute's cardinality decreases [3]. Thus, quick sort is used when the size of input is less than 1/4 th of cardinality of the partitioning attribute. Insert sort is used when the number of tuples in input is less than 12.

One more important module is *writeAncestors* that is used by *aggregate* module to compute aggregates for a singe tuple. For single tuple partitions, there is no need of further partitioning and aggregating [3]. Thus it computes the aggregates once and output the record for every ancestor of it in the lattice shown in Figure 3. For example, if the first partition of A in the cube example discussed in related work contains only a single tuple, then for all its ancestors $AB, ABC, ABCD, AC, ACD, AD$ no need of performing any partitions or aggregations. The respective output records can be written out by setting the dimension values appropriately. This module also uses *replace* as and when required.

5.2 Memory requirement

BUC is implemented as two modules BUC internal and BUC external and the proper one is selected based on the availability of free memory. If the required memory is available then the cube computation will be performed internal to memory. At any point of time CUBE is computed over a partition. Let N be the number of tuples in a partition, and each tuple requires T bytes. [3] argued that *counting sort* [14] requires a temporary set of pointers to perform the partition on the given input, which is not required in our improved counting sort algorithm. We make use of the existing pointers and one additional pointer to sort the input. By this, we are able to reduce the memory requirement of counting sort algorithm. And another improvement is that we use only one array of counters, which is shared by both counting sort and BUC. Hence there is no need of extra set of counters for counting sort algorithm. Let d be the number of dimension attributes of the table. Let $C_1, \ldots, C_d$ be the cardinality of each dimension, and C_{max} be the maximum cardinality. Counting sort [3] uses C_{max} counters and BUC uses $\sum C_i$ counters. Each tuple requires $T = d*4$ bytes. Assuming counters and pointers are each of four bytes, memory requirement of counting sort algorithm [3] is :

$$N(T+8) + 4C_{max}$$

For the improved version of counting sort algorithm it is given as :

$$N(T+4) + 4$$

Total memory of our algorithm is :

$$\underbrace{N(T+4)+4}_{countingsort} + \underbrace{4\sum_{i=1}^{d} C_i}_{counters} + Sizeof(indexfiles)$$

6. PERFORMANCE OF THE ALGORITHM

We ran our tests on a 450MHz Pentium PC with 64MB of RAM and 4.5 GB hard disk running on linux environment in a single user mode. The implementation uses the file system provided by OS. The algorithm is implemented in C. We considered the total elapsed time for the computation of cube, which includes both I/O time and CPU time. I/O time includes the time taken to output the results and read the input from the disk. To illustrate the gains of our algorithm, we compared the performance of our algorithm with BUC together with star join algorithm. Algorithms are tested against the data sets of sizes 1,00,000 and 5,00,000 tuples by varying the number of dimension attributes from 2 to 8. The cardinality ranges of the attributes are varied from 10 to 1000. Figure 7 shows the graphs obtained for the data set of 1,00,000 tuples and Figure 8 for the data set of 5,00,000 tuples. From graphs it is clear that both join-dynamic and join-apriori achieve a significant improvement over star-join, as the input becomes sparse. The diffrence between join-dynamic and join-apriori is small and negligible. The reason for it is, join-apriori involves only an addtional scan of input but no additional computation cost to perform replacement operation. Since the difference is very small and negligible either of them can be used for datacube computation.

In both join-apriori and star-join replacement is performed before the actual computation of cube starts. In case of star-join, index file size increases in proportion to the data set size where as in join-apriori, index file size increases logarithmically proportional to the data set size. Thus time taken to perform join operation in case of star-join is high since it involves a large number of comparisions. On the other hand join-apriori does not need any comparisions to perform join operation as it makes use of the replacement technique.

7. CONCLUSIONS

In this paper, we proposed an algorithm that computes datacube efficiently in the presence of attribute hierarchies. It is basically an extension to BUC algorithm, which does not handle cube computation in the presence of attribute hierarchies. The proposed algorithm adopts a new approach, an alternative to join index, to join the relations for which referential integrity holds. This new technique of join does not require any additional pre-computed access structure. It also does not incur any additional overhead of I/O cost, since all the required EBI files can usually be memory resident. The algorithm not only takes the advantage of user specified condition for pruning the computation of group-bys but also for replacing the hierarchial attributes. We performed empirical evaluations and scalability experiments. The results showed that the algorithm scales linearly for large datasets.

8. REFERENCES

[1] S. Agarwal, R. Agrawal, P. M. Deshpande, A. Gupta, J. F. Naughton, R. Ramakrishnan, and S. Sarawagi. On the computation of multidimensional aggregates. In *Proc. of the 22nd VLDB Conference*, pages 506–521, 1996.

[2] E. Baralis, S. Paraboschi, and E. Teniente. Materialized view selection in a multidimensional databases. In *Proc. of the 23rd VLDB Conference*, pages 156–165, 1997.

[3] K. Beyer and R. Ramakrishnan. Bottom-up computation of sparse and iceberg cubes. In *Proc. of the ACM SIGMOD conference*, pages 359–370, 1999.

[4] S. Chaudhuri and U. Dayal. An overview of data warehousing and olap technology. *SIGMOD Record*, 26(1):65–74, March 1997.

[5] P. M. Deshpande, S. Agarwal, J. F. Naughton, and R. Ramakrishnan. Computation of multidimensional aggregates, tr-1314. Technical report, University of Wisconsin, Madison, 1996.

[6] J. Gray, A. Bosworth, A. layman, and H. pirahesh. Datacube: A relational aggregation operator generalization group-by, cross-tab, and sub-totals. In *Proc. of the IEEE ICDE*, pages 152–159, 1996.

[7] V. Harinarayanan, A. Rajaraman, and J. D. Ullman. Implementing data cubes efficiently. In *Proc. of the ACM SIGMOD Conference*, pages 205–216, 1996.

[8] R. Kimball. *The Data Warehouse Toolkit*. John Wiley & Sons, 1996.

[9] P. O'Neil. Model 204 architecture and performance. In *Proc. of the 2nd International Workshop on High Performance Transaction Systems*, pages 40–59, 1987. Lecture Notes in Computer Science, No. 359.

[10] P. O'Neil and G. Graefe. Multi-table joins through bitmapped join indices. *SIGMOD Reocord*, 24(3):8–11, September 1995.

[11] P. O'Neil and D. Quass. Improved query performance with variant indexes. In *Proc. of the ACM SIGMOD Conference*, pages 38–49, 1997.

[12] K. A. Ross and D. Srivastava. Fast computation of sparse datacubes. In *Proc. of the VLDB Conference*, pages 116–125, 1997.

[13] S. Sarawagi, R. Agrawal, and A. Gupta. On computing the data cube, tr-rj10026. Technical report, IBM Almadan Research Center, San Jose, CA, 1996.

[14] R. Sedgewick. *Algorithms in C*, chapter 8, page 112. Addison-wesley Publishing Company, 1990.

[15] M. C. Wu and A. Buchmann. Encoded bitmap indexing for data warehouses. In *Proc. of the 14th ICDE*, 1998.

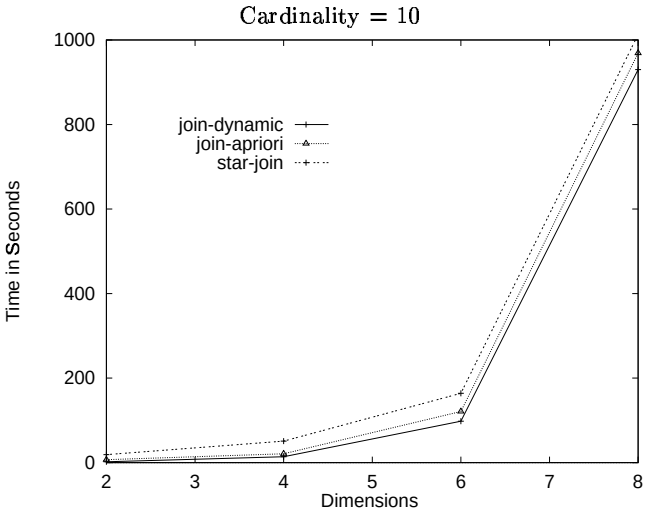

Figure 7(a)

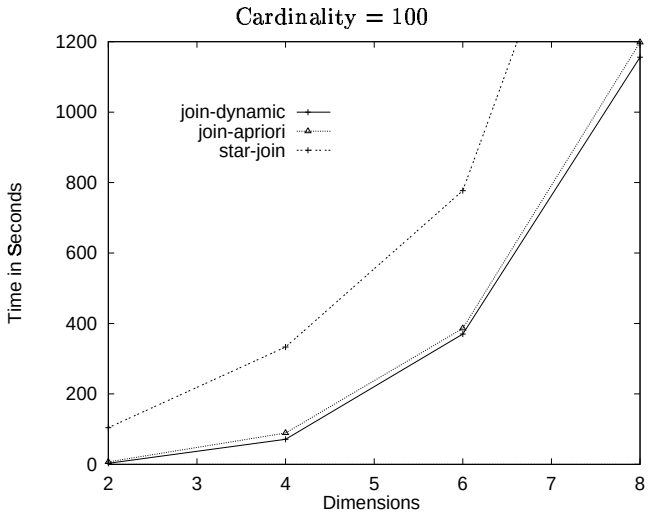

Figure 7(b)

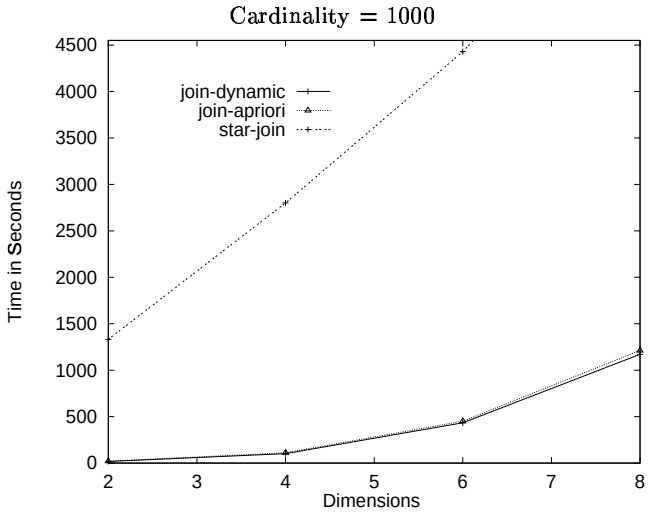

Figure 7(c)

Figure 7: Full Cube Computation over a relation with size 1,00,000 tuples

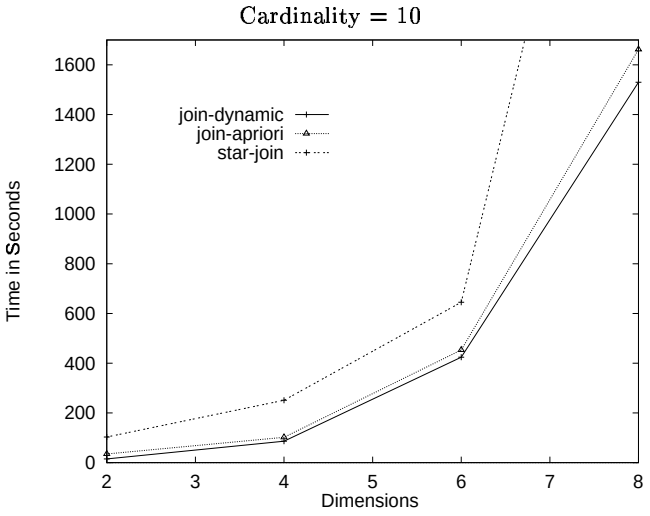

Figure 8(a)

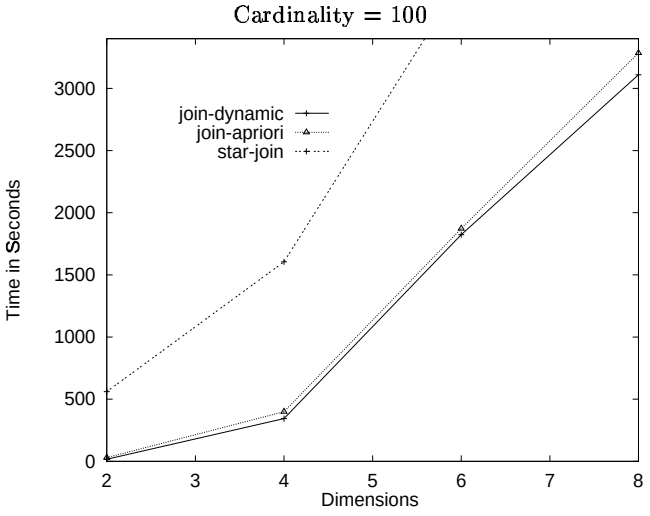

Figure 8(b)

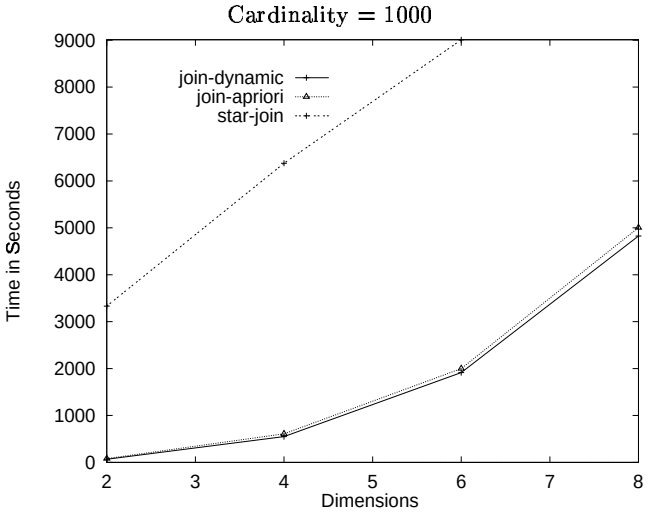

Figure 8(c)

Figure 8: Full Cube Computation over a relation with size 5,00,000 tuples