A Clustered Dwarf Structure to Speed up Queries on Data Cubes

Yubin Bao, Fangling Leng, Daling Wang and Ge Yu
School of ISE, Northeastern University, P.R.China
{baoyb, lengfangling, dlwang, yuge}@mail.neu.edu.cn

Dwarf is a highly compressed structure, which compresses the cube by eliminating the semantic redundancies while computing a data cube. Although it has high compression ratio, Dwarf is slower in querying and more difficult in updating due to its structure characteristics. We all know that the original intention of data cube is to speed up the query performance, so we propose two novel clustering methods for query optimization: the recursion clustering method which clusters the nodes in a recursive manner to speed up point queries and the hierarchical clustering method which clusters the nodes of the same dimension to speed up range queries. To facilitate the implementation, we design a partition strategy and a logical clustering mechanism. Experimental results show our methods can effectively improve the query performance on data cubes, and the recursion clustering method is suitable for both point queries and range queries.

Categories and Subject Descriptors: E.1 [**DATA STRUCTURES**]: ; E.2 [**DATA STORAGE REPRESENTATIONS**]:

General Terms: Data Cube, Clustering

Additional Key Words and Phrases: clustered Dwarf, hierarchical cluster, logical clustering, query optimization, recursion cluster

1. INTRODUCTION

Dwarf [Y. Sismanis and Kotidis. 2002] is a highly compressed structure, which compresses the cube by eliminating the prefix and suffix redundancies while computing a data cube. What makes Dwarf practical is the automatic discovery of the prefix and suffix redundancies without requiring knowledge of the value distributions and without having to use sophisticated sampling techniques to figure them out. But the structure of Dwarf (see Fig.1) has the following characteristics. (1) It is a tree structure if ignoring the suffix coalition, which has a common root and some nodes derived from the root. (2) The node cells store the node location information of the next dimension, which is similar to an index. (3) The nodes have different size. These characteristics make it inefficiently to store a Dwarf into a relational database. Of course, we can model the node structure by ER model and disassemble the nodes and store them into the database. (The relationship between Dwarf and its nodes, between the node and its cells are both one-to-many.) But it causes the performance to be reduced, because reading each node will cause the join operations. So Dwarf is usually stored in a flat file, which has a good performance of the construction [Y. Sismanis and Kotidis. 2002]. However, due to not considering the characteristics of queries and the high cost of maintaining files, it brings a low query performance and is hard to be updated. In order to solve the query performance problem, [Y. Sismanis and Kotidis. 2002] gave a clustering algorithm, which clusters nodes according to the computational dependencies among the group-by relationships of the cube. This method can improve the query performance in a certain degree. For a cube, point queries and range queries are two important types of queries on it. We note that for point queries and range queries, the access order to nodes is according to the relationship between parents and children of the nodes from root to leaves instead of the group-by relationships. Therefore, we propose two novel clustering algorithms to improve the query performance on Dwarf. As the data increase in complexity, the ability to refresh data in a data warehouse environment is currently more important than ever. The incremental update procedure in [Y. Sismanis and Kotidis. 2002] expands nodes to accommodate new cells for new attribute values (by using overflow pointers), and recursively updates those sub-dwarfs which might be affected by one or more of the delta tuples. The frequent incremental update operations slowly deteriorate the original clustering feature of the Dwarf structure, mainly because of creating the overflow nodes. Since Dwarf typically performs updates in periodic intervals, a process in the background periodically runs for reorganizing the Dwarf and transferring it into a new file with its clustering restored [Y. Sismanis and Kotidis. 2002]. To avoid reorganizing the Dwarf and maintaining the clusters, we design a partition strategy and a logical cluster mechanism.

In [W. Wang and Yu. 2002], three algorithms are described for discovering tuples whose storage can be coalesced: MinCube guarantees to find all such tuples, but the computation consumption is very high, while BU-BST and RBU-BST are faster, but only discover fewer coalesced tuples. Much research work has been done on approximating data cubes through various forms of compression such as wavelets [J. S. Vitter and Iyer. 1998] or by sampling [Gibbons and Matias. 1998] [S. Acharya and Poosala. 2000] or data probability density distributions [J. Shanmugasundaram and Bradley. 1999]. While these methods can substantially reduce the size of the cube, they do not actually store the values of the group-bys, but rather approximate them, thus not always providing accurate results. Relatively Dwarf is a promising structure to be discussed deeply.

The rest of this paper is organized as follows: Section 2 presents two clustering methods for Dwarf, the recursion clustering method and the hierarchical clustering method. Section 3 describes the physical structure of the clustered Dwarf, the paging partition strategy, and the logical clustering mechanism. Section 4 presents the experiments and the result analysis. The conclusion is given in Section 5.

2. CLUSTERING DWARF

Currently the disk management system of computer architecture is organized by disk-cylinder-sector. The basic storage unit is the sector, and the size of each sector is 512 bytes. Because of the limited capability of sectors, I/O will consume a great deal of time when operating system reads and writes the disk using the sector as the unit. To solve the problem the unit of I/O in current system is set to a cluster, which is constituted with several amounts fixed, sequential sectors. And the amount of the sector is decided when formatting the disk. The default cluster size of Windows NT system is 4 KB, which consists of 8 sectors.

Operating system usually reads the disk using a cluster as a unit. When a certain disk is read, the whole cluster will be read to the buffer. For example, when a byte with the offset 5000 bytes from the start position in the data area is read, the whole second cluster (physical address is between 4097 and 8192 bytes) will be read to the buffer. This will bring the pre-fetch function, and it is an application of locality principle.

With this function, if we can write the possible sequential access nodes to the neighbor location (to the same cluster if allowable) when the Dwarf is constructed. Then the I/O cost will be reduced by the cluster-read characteristic of operating system, and the query performance will be improved obviously. And the above-mentioned process is clustering.

In [Y. Sismanis and Kotidis. 2002] a clustering algorithm was proposed, which clustered the nodes according to the computational dependencies between the group-bys of the cube. But this method may not improve the performance, because it ignores the characteristics of queries on Dwarf. When querying on Dwarf, the access orders are according to the relationships between predecessor and successor of the nodes from root to leaf whether in point queries or in range queries. The method breaks the relationship.

For the following explanation, we give a sample data set in Table I. Fig.1 shows the Dwarf structure of the sample data in Table I according to the method in [Y. Sismanis and Kotidis. 2002], where node (5) is stored behind node (7) according to the computational dependencies, and the parent of node (5) is node (2).

Table I. A sample data set.

Store	Customer	Product	Price
S_1	C_2	P_2	70
S_1	C_3	P_1	40
S_2	C_1	P_1	90
S_2	C_1	P_2	50

When querying from node (2) to node (5), one more time I/O operation (node (7)) will be executed in order to read node (5). Therefore, we propose to cluster the Dwarf according to the relationship between parents and children of the nodes. Two clustering methods are designed to optimize Dwarf for improving the performance of point queries and range queries.

Suppose that the example query is :

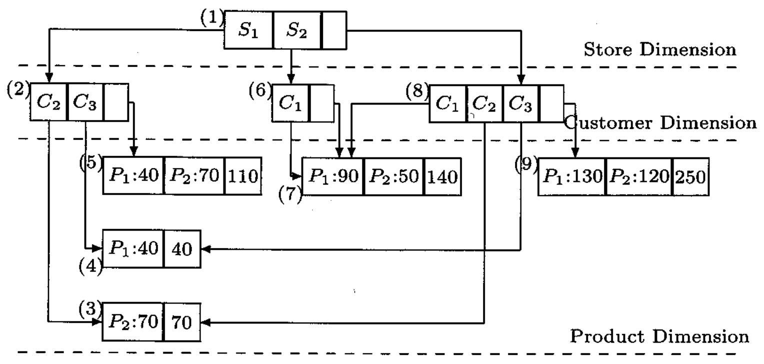


Figure 1. The Dwarf structure of the sample data set.

```
SELECT price
FROM    example_cube
WHERE (store=S1) AND (customer=ALL) AND (product=P1).
```

2.1 Recursion Clustering

Recursion clustering is an optimization of the point query on Dwarf. Often we can see that when a point query is performed, the results must be in the sub-Dwarf pointed by the corresponding cell. That is a lengthways access manner. For the example query, when the condition *store* $= S_1$ is executed, the results must be in the sub-Dwarf pointed by cell S_1, i.e. node (2)(3)(4)(5). So if we can store the four nodes on the same cluster, the query performance will be improved.

In Fig.1, the root node has three sub-trees, rooted by node (2), node (6), and node (8). So they should be stored together. Furthermore, node (2) also has three sub-trees, rooted by (3), (4), (5), and they should be stored together too. When answering the point queries with this method, the distance between the related two nodes in the path will become smaller and smaller with the query going ahead to the leaves. So it is an incremental shrinking method. These clustering steps are in the recursive manner. So we call our method as the recursion clustering method (see Algorithm 1).

For the Hierarchical Dwarf [Y. Sismanis 2003], we treat the roll-up relation (node (2) and node (5) in Fig.2) as a relationship between predecessor and successor too (the sub-Dwarf with the root node (5) is a sub-Dwarf of node (2)). So we can extend the above-mentioned cluster method to the hierarchical Dwarf. The node tags in Fig.2 show the storage order of the node on the disk under this clustering method.

Apart from the leaf nodes, all internal nodes of a Dwarf contain *ALL* cells to indicate the summary information on the corresponding dimensions. [Y. Sismanis and Kotidis. 2002] designed the recursion suffix coalition algorithm to compute the *ALL* cells. But it makes the clustering feature lost. We improve it by eliminating the recursive operations. Fig.3 gives an example of suffix coalition. Here the suffix coalition occurs at the *ALL* cell of node (1), and the following nodes are constructed with it. The node sequence writing to the disk in Fig.3 should be (5)(3)(6)(4)(2)(1).

Algorithm 1: Recursion Clustering Algorithm.

```
1:  initialize the stack $S$
2:  push the begin node and its sub-Dwarf into $S$
3:  while $S$ is not empty do
4:    pop the top element $n$ in $S$
5:    if $n$ has non-computed units then
6:      go to the first non-computed unit $c$ of $n$
7:      compute the sub-Dwarf of $c$ using the sub-Dwarf of $n$
8:      push $n$ into $S$
9:      if $c$ can be suffix coalition//not require creating new nodes
10:       write $c$'s suffix coalition position into top element unit
11:     else //require creating new nodes
12:       create the new node $n_1$ using the sub-Dwarf of $c$
13:       push the $n_1$ and its sub-Dwarf into $S$
14:     end if
15:   end if
16: end while
```

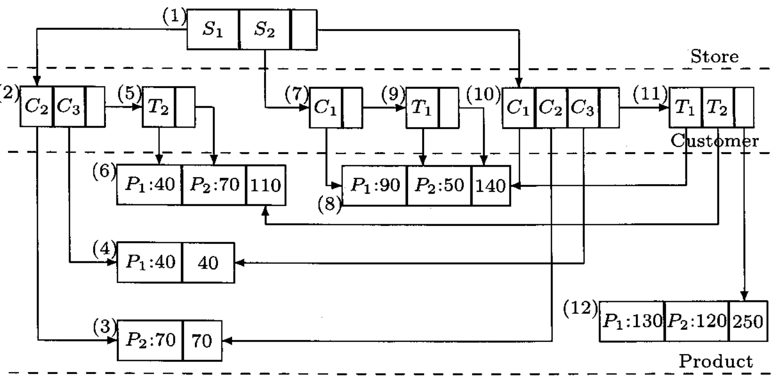


Figure 2. The Dwarf structure with concept hierarchies.

We use a stack to implement the recursion clustering algorithm. The elements in the stack are the nodes to be closed and their sub-Dwarfs. According to Algorithm 1, the suffix coalition of the Dwarf is in Fig.2 and the stack changing are shown in Table II.

2.2 Hierarchical Clustering

A range query often accesses several child nodes of a node, and the sibling nodes of the node will be accessed together. Because the sibling nodes are of the same dimension, we suggest the nodes of the same dimension should be clustered together.

Algorithm 2 shows the hierarchical clustering algorithm, which also supports the suffix coalition. Different from the recursion clustering, a queue is used in it. The elements of the queue are the nodes to be closed and the sub-Dwarfs rooted by these nodes. The process is similar to the breadth-first searching of graph. Table III shows the node sequence and the changing of the assistant queue of Fig.3 according to the Algorithm 2. The storage sequence of the nodes in Fig.3 should be (1)(2)(3)(4)(5)(6) according to the hierarchical clustering. Contrast with the

Table II. The stack changing in recursion clustering.

action	elements	action	computing
1	(1)	PUSH(1)	initialize the stack
2	(1)(2)	PUSH(2)	top element (1) is unfinished, create (2)
3	(1)(2)(3)	PUSH(3)	top element (2) is unfinished, create (3)
4	(1)(2)(3)(5)	PUSH(5)	top element (3) is unfinished, create (5)
5	(1)(2)(3)	POP(5)	compute and write (5) back
6	(1)(2)	POP(3)	compute and write (3) back
7	(1)(2)		compute the cell B_2 in top element (2)
8	(1)(2)(4)	PUSH(4)	top element (2) is unfinished, create (4)
9	(1)(2)(4)(6)	PUSH(6)	top element (4) is unfinished, create (6)
10	(1)(2)(4)	POP(6)	compute and write (6) back
11	(1)(2)	POP(4)	compute and write (4) back
12	(1)	POP(2)	compute and write (2) back
13	NULL	POP(1)	compute and write (1) back

Algorithm 2: Hierarchical Clustering Algorithm.

```
1:  initialize the queue $Q$, and put the begin node into $Q$
2:  **while** $Q$ is not empty do
3:      get a node $n$ from $Q$
4:      **if** $n$ is the leaf node then
5:          compute all the units of $n$
6:          write $n$ to the disk
7:          continue
8:      **end if**
9:      create the child nodes $n_1$, $n_2$, $\cdots$, $n_k$ of $n$ and their sub-Dwarfs, and write
        nodes numbers into the corresponding units of $n$
10:     put the child nodes into $Q$ by the creation time
11:     write $n$ to the disk, and add $n$ to the node index table
12: **end while**
```

recursion clustering, it has the following problems. Before the next layer nodes being written to the disk, i.e. the location of the next layer nodes is unknown, the node should finish writing to the disk. For example, if node (3) and node (4) have not been written to the disk, i.e. the values of cell B_1 and B_2 of node (2) are unknown, node (2) has to be written to the disk first to satisfy the cluster characteristics.

To solve this problem, we propose a node index approach. It assigns a distinct index number to each node, and the cell in a node stores the index number but not the location on the disk of the next layer node. At the same time a node index table is used to maintain the mapping between the index number and the physical location of the node. For example, when writing node (2) to the disk, we only need to construct node (3) and node (4), and fill the index number of them into the cells of node (2). When the child nodes are written to the disk, their addresses will be filled into the node index table. Such index enables to write the parent node before its child nodes.

Restricting the impact of the update of a Dwarf is another advantage of the node index approach. The update is very difficult because of the nodes with unequal size. When updating a Dwarf, some cells or nodes will be created or deleted. The

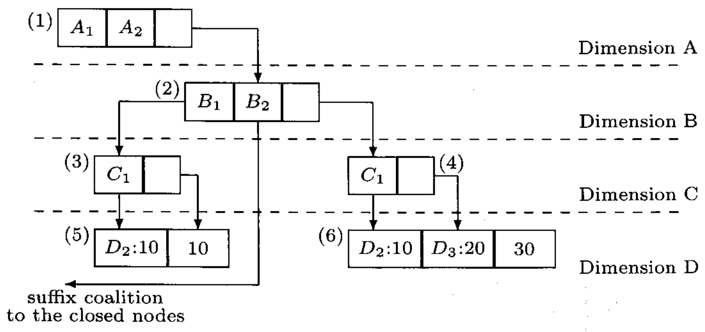


Figure 3. An example of suffix coalition.

Table III. The queue changing in hierarchical clustering.

action number	elements in queue	action of queue	action of queue
1	(1)	IN(1)	initialize the queue
2	NULL	OUT(1)	establish (2), and write the ALL cell of (1)
3	(2)	IN(2)	write (1) to the disk
4	NULL	OUT(2)	establish (3) and (4), and write all the cell of (2)
5	(3)(4)	IN(3)(4)	write (2) to the disk
6	(4)	OUT(3)	establish (5), and write all the cell of (3)
7	(4)(5)	IN(5)	write (3) to the disk
8	(5)	OUT(4)	establish (6), and write all the cell of (4)
9	(5)(6)	IN(6)	write (4) to the disk
10	(6)	OUT(5)	compute (5), and write (5) to the disk
11	NULL	OUT(6)	compute (6), and write (6) to the disk

Dwarf is usually stored in a flat file, and the frequent node insertion or deletion operations will produce many file fragments. Compared with deletion, inserting a new node or a new cell is very hard to deal with. To the adjoining nodes on the disk, the increase of a node may not be simple as the increase of an array in memory. Since the long time of disk operation, we have to abandon the original node space and add a new updated node at the end of the file. If there is no node index, the operations will lead the parent node of the updated node to be updated passively because the corresponding cell of the parent node stores the offset of the updated child node in the file. (In Fig.3, the update of node (3) will cause the cell B_1 of node (2) updated.) Since Dwarf only hold the mapping from the father node to its child nodes, the update of the father node becomes more trouble. After using the node index, the problem of updating scope is restricted within the child node. If the location of the child node is changed, we only need to update the node index of the child node, and other nodes will not be affected. But the changing of the node location should be avoided in order to maintain the cluster feature.

According to the hierarchical clustering algorithm, the storage sequence of the nodes in Fig.1 should be (1) (2) (6) (8) (3) (4) (5) (7) (9). To the hierarchical Dwarf, the extended hierarchy should be regarded as the same hierarchy, and should be stored in the same cluster. To the hierarchical Dwarf in Fig.2, the storage sequence should be (1) (2) (5) (7) (9) (10) (11) (3) (4) (6) (8) (12).

In hierarchical clustering, the suffix coalition of node (2) in Fig.1 is executed before node (1). If the nodes are always written to the end of the file, node (2) and node (8), which should be put together, will be separated by node (3)(4)(5). Hence, the physical organization structure should be considered further.

3. PHYSICAL STRUCTURE OF DWARF

3.1 Paging Partition Strategy

To solve the above clustering problem, we propose a partition organization strategy for Dwarf files. Firstly, allocate a certain space (called a dimension *partition*) on the disk for the node cluster of each dimension, which uses the idea of chunk in [Y. Zhao 1997]. Secondly, a dimension partition only stores the corresponding dimension nodes. When the remaining space cannot contain a new node, a new partition for that dimension will be appended to the end of the file. In this way, when the node is written to the disk, it may not be appended to the end of the file, but be written to its dimension partition. In this case, the hierarchical clustering feature is guaranteed.

Generally, we can extend the concept to the recursion clustering. The unit of the disk I/O of current operating system is a disk cluster. So the amount of the cluster produced by different clustering methods should be an integer, at the same time a node if and only if belongs to a cluster, i.e. the node spanning two clusters are not allowed, the intension of which is reducing the additive I/O operations.

Generally speaking, the size of the memory page equals to the size of the disk cluster (the two values are all 4 KB in Windows NT), which is convenient for reducing the cost of memory management because a disk cluster can just fill a memory page. So our partition management of Dwarf is designed as the paging storage management. Several pages consist of a partition, and a node is not allowed to span the pages. When clustering, a certain amount of pages is pre-allocated for each cluster, and a new partition is added at the end of the file only if a partition cannot contain the new nodes. Since a node is not allowed to span the pages, the fragments occur. But they can be used to save the broken cluster by updating operations from another point of view. This will be addressed in the next section.

How many pages should we pre-allocate for a cluster? The amount should be neither small nor big. Too few pages will cause the frequently adding the new partitions, which will break the clusters to a certain extent. And too many pages will cause the waste of the space because some pages may not be used forever. So the pre-allocation of the recursion clustering and the hierarchical clustering should be taken with different strategies. But the basic idea is the same that pre-allocating many more pages for the bigger cluster.

(1) For the recursion clustering, a cluster is corresponding to a sub-Dwarf tree actually. In Fig.3, the cluster of nodes (2)(3)(4)(5)(6) is related with the sub-tree of *ALL* cell of node (1), and the cluster of node (3)(5) is related with the sub-tree of cell B_1 of node (2). Obviously, the cluster size is related with the height of the sub-tree. So when we pre-allocate the pages, we only need pre-allocate enough pages for each sub-tree of the root, and the low layer sub-trees recursively get the space through partitioning the pages. Suppose that there is a D-dimensional cube,

we should pre-allocate $N = 2^D$ pages for each sub-tree of the root. In the above expression, the number 2 indicates the qualitative analysis of the space. Its actual size should be studied further according to the data statistic features.

(2) For the hierarchical clustering, the nodes of the same dimension constitute a cluster. Obviously, the more close to the leaves the dimension is, the more amounts of the nodes the dimension has. For a D-dimensional cube, the page amount of the cluster of the k-th dimension should be $N_k = 2^k(1 \leq k \leq D)$. The number 2 in the expression also indicates the qualitative analysis of the space.

3.2 Fragments in the Pages

As mentioned above, the fragments couldn't be avoided in the page-style partition clustering structure. The existence of the fragments wastes the storage space, and brings the side effect for compression. We propose a special mechanism, called *idle space manager*, to maintain the fragments space produced in the Dwarf system caused by shrinking, deleting, or enlarging nodes. In the previous section, we mentioned that the fragments contribute the maintenance of the cluster feature of Dwarf during the update. We use a more obvious example to show the effect of the fragments. Suppose the status before updating is as shown in Fig.4.

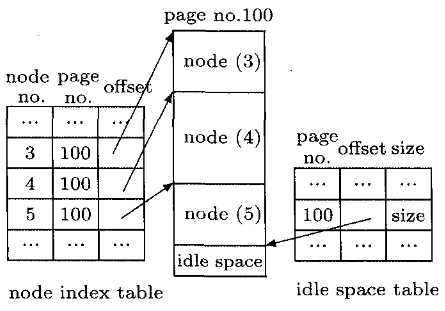


Figure 4. Status before updating.

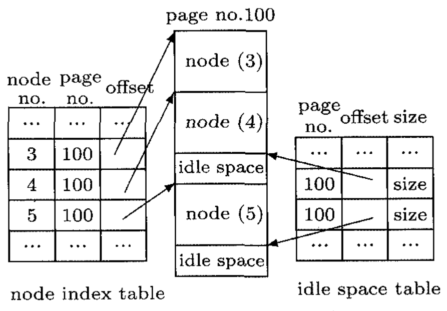


Figure 5. Status after using strategy (a).

The address part of the node index table adopts two-level address mechanism, and one is the page number, the other is the offset within the page. Suppose node (4) is changed.

(1) If the size of node (4) is reduced, there are two strategies to deal with this simple case. (a) Only change node (4) and the node index table, which will produce new fragments, but the maintenance speed is very fast because of not moving other nodes. It only updates the node content, and adds a new record in the idle space table. But more and more such fragments will cause the total size of the idle space to be larger, and it cannot satisfy the need of a small idle space. So in this method operations of constricting the idle space will be performed periodically. After this operation, the Fig.4 will change to the Fig.5. (b) When shrinking the changed node, the fragments in a page is compacted and the following nodes are moved ahead within the page. At the same time, the node index table and the idle space table are updated. This method avoids the fragments, but it will consume more time. Fig.6 shows the status after updating. Our Dwarf system adopts this method.

(2) If the size of node (4) increased, supposing the increment is $\triangle S$, there are three cases according to the size of $\triangle S$. (c) If $\triangle S$ is less than the idle space of the page, the update will be restricted within the current page, and will be implemented by moving the following nodes forward. Fig.7 shows the status after updating. (d) If $\triangle S$ is greater than the idle space of the page, but the size of the node plus the $\triangle S$ is less than a certain idle space or a certain idle page in the current cluster, the original node will be deleted and the current page will be shrunken. At the same time, the updated node is written on the new location of the partition, and the idle space table is updated. This case is similar with the operation for the reducing node. (e) When any idle space in a partition is not satisfied with the increased node, a new partition need to be appended to the end of the file, and the increased node will be written to the new partition. At the same time, the idle space table is updated. In this case, the physical cluster feature is violated. So we propose a logical clustering mechanism.

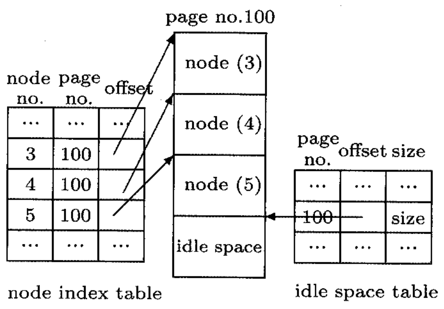


Figure 6. Status after using strategy (b).

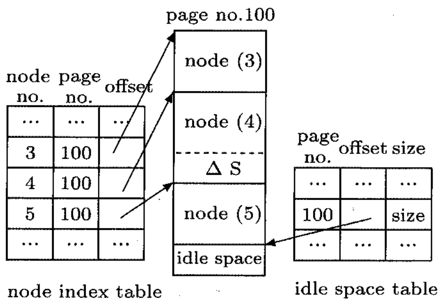


Figure 7. Status after using strategy (c).

3.3 Logical Clustering Mechanism

Besides the case talked in the end of the previous section, the inadequate pre-allocated space will result in the physical cluster feature broken too. To deal with the problem of breaking clusters, we propose a logical clustering mechanism, which is used to keep the physical cluster feature by tracking of the broken clusters. So-called the logical clustering is not a real cluster. In fact, it cannot guarantee the pages in a logical cluster are stored in the sequential pages on the disk. We can get the general idea of updating data cubes and maintaining the clusters from [H. He and Yu. 2005] [Chen and Rundensteiner. 2005] [N. Folkert 2005].

The logical cluster mechanism is similar to the disk chunk chain of FAT file system. We register a record for a page to hold the next page number in logic in a cluster. For example, there are 4 pages satisfied with the cluster characteristic before updating. Pages (0)(1) and pages (2)(3) are separately in the same cluster. Fig.8 shows the array of the logical cluster records.

Here L[i] $=$ k indicates the logic page k is the neighbor of page i, and L[i] $=$ -1 indicates page i is the end of the file in logic, but not physically. To the cluster having not been broken, it is always true that L[i] $= i + 1$, which is the basis of adjusting the broken cluster. Suppose that the node in page (1) is increased, and we have to append a new page after the end of the file to store the updated node. After updating the records of the logical cluster are shown in Fig.9. We can see that L[1] = 4, which tells us page (4) is the neighbor of page (1) in logic but not physically.

1	2	3	-1
L[0]	L[1]	L[2]	L[3]

Figure 8. The records of logical clustering.

1	4	3	-1	2
L[0]	L[1]	L[2]	L[3]	L[4]

Figure 9. The records after updating.

When the current storage layout seriously affects the query performance, the pages in a cluster in logic need to be adjusted together at once. At that time, the logical cluster record will work as a navigator. Since the node index approach is used, the cell values of the nodes need not be changed when moving the pages, and we only need to modify the page numbers of the relative nodes in the node index table, which is another benefit of the node index approach.

4. PERFORMANCE EXPERIMENTS

Compared with the other semantic compression algorithms, Dwarf has the highly compression advantage [Y. Sismanis and Kotidis. 2002]. Dwarf is not sensitive to the data sizes and the dimension numbers. Especially, Dwarf has a considerable expression ratio for the high dimensional, super large-scale data. The experimental route of this paper differs from [Y. Sismanis and Kotidis. 2002]. We focus on the query performance. Because we all know that the original intention of data cube is to speed up query performance. Since Dwarf already has high compression ratio, we consider that it is worthy to get better query performance at the cost of construction time and storage of Dwarf.

In our experiments, the final Dwarf is stored by file, but other inputs and outputs are stored by the relational database system, which facilitates the management but slows the speed down. We will see that in the following experiments.

We generate a data set with 10 dimensions and 4×10^5 tuples. The cardinalities of each dimension are 10, 100, 100, 100, 1000, 1000, 2000, 5000, 5000, and 10000, respectively. The aggregation function is SUM. All the experiments are running on a single Pentium IV with 2.6GHZ processor running Windows XP plus SP2 with 512MB of DDR RAM. The disk is 80GB, and able to read at about 22MB/s and write at about 12MB/s. We use ODBC connection to the Microsoft SQL Server 2000 plus SP4. In the following, non-clustering represents non-clustering Dwarf, Dwarf represents computational dependencies between the group-by relationships, R-Dwarf represents recursion clustering, and H-Dwarf represents hierarchical clustering.

4.1 Construction of Dwarf

Test 1: In this part, we want to test the relationship between the construction time, the storage space and the dimension numbers. We test 7 to 10 dimensions with 4×10^5 tuples data set, respectively. We compare our algorithm with the original Dwarf [Y. Sismanis and Kotidis. 2002] to construct the cube. The construction time costs are shown in Fig.10 and Fig.11.

From the experimental results, we find that the construction time of our algorithm is about one time more than the original one. The storage space of our algorithm is about 0.3 times more than the original one. But our goal is to improve the query performance, and each part of our method is designed toward the goal. The main reasons are as follows: a) Sorting the dimensions by their cardinalities from small to big, which will speed up the query response but weaken the effect of suffix coalition. It is the main reason that the space of our Dwarf becomes bigger. b) A certain internal idle space is reserved by the page-manner partition storage strategy, which will facilitate holding the cluster feature during the update but will increase the space consumption of the final file. c) The node index technique and the logical clustering mechanism speed up the update and facilitate the maintaining of the clusters after updating. But they bring additional time and space cost. From our observation, 40% of the construction time is used to work on the database, which is the main reason that our construction time is much more than the original method. d) It is very interesting that the space utilization is around 95%. In the beginning, we forecast the space utilization decrease with the increasing of the number of the dimensions. The possible reason is the sparseness of the data. e) Our method is sensitive with the number of the dimensions, and the space increases fast with the increasing of the dimensionality. The main reason should be the ordering manner of the dimension. But the original intention is speeding up the query performance.

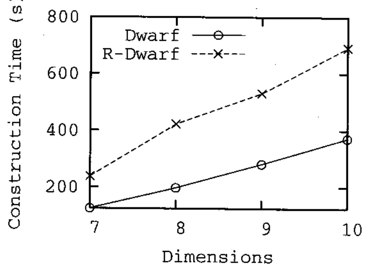


Figure 10. Time vs. Dims (4×10^5 Tuples).

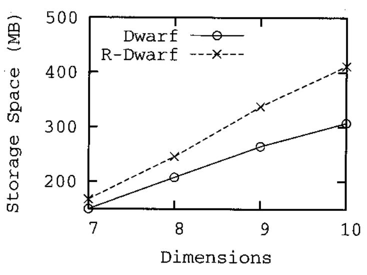


Figure 11. Space vs. Dims (4×10^5 Tuples).

Test 2: In this part, we want to test the relationship between the construction time, the space and the size of the data sets. Set the dimension is 10, and we separately test the data sets with 100000, 200000, 300000, and 400000 tuples. The results are shown in Fig.12 and Fig.13.

From the result, we can see that the performance of our method is still behind the original method. The reason is similar with test 1. But it is noticeable that our performance can maintain a linear changing with the increasing of the tuples.

4.2 Query on Dwarf

The focus of our tests is about the query performance since it is our original intention. We make several experiments on different kinds of queries under different strategies.

Test 3: In this part, we test the relationship between the point queries and the clustering styles including the non-clustering, the computational dependencies

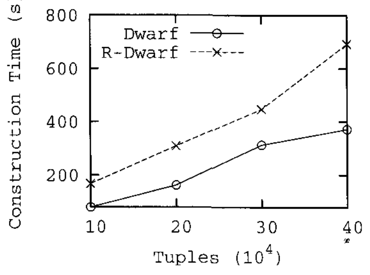


Figure 12. Time vs. Tuples (10 dimensions).

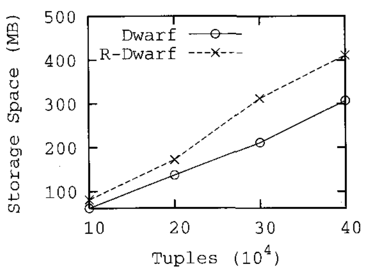


Figure 13. Space vs. Tuples (10 dimensions).

between the group-bys, the recursion clustering, and the hierarchical clustering. The data cubes are the four Dwarfs used in Test 1 with 4×10^5 tuples. We generate 1000 point queries randomly and run them continuously. Fig.14 shows the time cost.

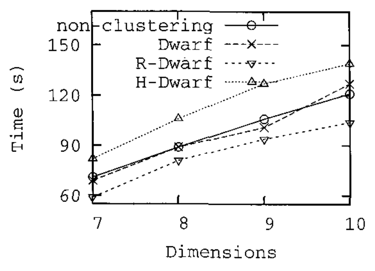


Figure 14. Point query time vs. Dims.

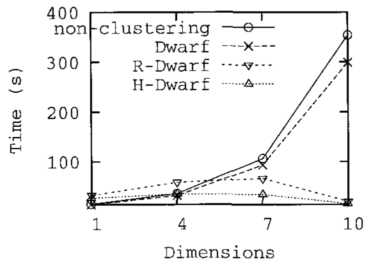


Figure 15. Range query time vs. Dims.

In this test, the recursion clustering aiming at the point query outperforms the computational dependencies method by 10%. Hierarchical clustering behaves too badly, and it drops behind far from others, even the non-clustering method. Because the hierarchical clustering method clusters the nodes of each dimension, and the query on each dimension will produce an I/O cost. To a point query, the average I/O number of hierarchical clustering method should be equal to the dimension number in theory. Here it is less than the dimension number because of the contributions of the high-speed buffer in Windows NT.

Test 4: In this part, we test the relationship between the range queries and the clustering styles mentioned in Test 3. The cube we used is the biggest one of 10 dimensions and 4×10^5 tuples. We want to validate the positive effects of the improved dimension ordering method to the range queries by this test. The sorting order is from small to big. To be fair, the query conditions are on the 1st, 4th, 7th, and 10th dimensions with the cardinalities are 10, 100, 2000, and 10000, respectively. Suppose each range query covers one dimension by the ratio 20%. We create 100 queries on each of the four dimensions, and use the four clustering methods mentioned in Test 2 to answer the queries. Fig.15 shows the response time vs dimension. The I/O times are shown in Fig.16 and Fig.17.

The results are very interesting. The former two methods sort the dimensions with the cardinalities changing from big to small when constructing the Dwarf, and the size of Dwarf is small. But in this test, we can see that with the conditions are pushed near to the root with the maximal cardinality, the time cost is growing fast. On the contrary, the sizes of the latter two methods are large, but the query response times do not fluctuate as much as the former two methods. And it seems that there exists a certain theory similar to negative feedback. We will analyze the reason of it behind.

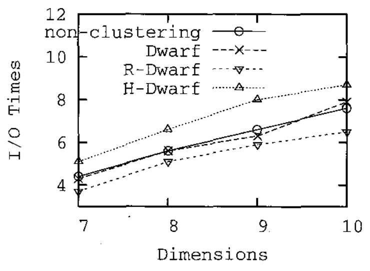


Figure 16. Point query I/O times vs. Dims.

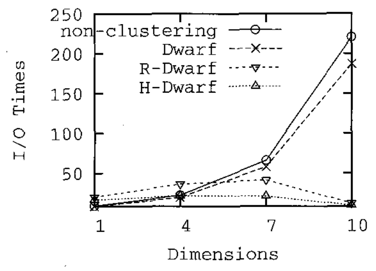


Figure 17. Range query I/O times vs. Dims.

It is obvious that for the query covering a dimension by a fixed proportion, the query time will increase with the increase of the cardinality, which is formed by the special tree structure of Dwarf. When querying at the 10th dimension with 10000 cardinality, the original method has to read about $10000 \times 20\% = 2000$ nodes of the next dimension to decide whether or not to go on, because the 10th dimension is the root. Since the data is sometimes sparse, the tuples matching the conditions are few. So most of the 2000 nodes read will be abandoned, which waste many I/O operations. Here the order is opposite to the original Dwarf, the 10th dimension is set to the leaf nodes. We only need to read a few leaf nodes to answer the queries. That is the reason why the I/O times using hierarchical clustering are close to the dimension number under this condition. In addition, we can see that in our experiments, when query on the root, the performance is still good, because the root has a small fan-out (only 10 cardinalities). To the query covering 20% range, we only need to judge 2 nodes but not the 2000 nodes of the next dimension at most. We call it query negative feedback of reversed dimension ordering. This negative feedback makes the performances of the range query using the recursion clustering and the hierarchical clustering fluctuates not much.

Generally, our clustering methods bring the construction and compression of Dwarf negative effect, but the effect is endurable. And our methods can remarkably improve the query performance on Dwarf. Since we have to maintain some additional structures, the performance of the update operations will be slow down, but it is endurable according to the simplicity of our structures. In the future, we will do some research on updating.

5. CONCLUSION

In this paper we propose and implement two novel clustering algorithms on Dwarf to speed up querying and design a partition strategy and a logical clustering mechanism to facilitate updating and maintain the clusters. Experimental results and theoretic analysis show that the recursion clustering is the best for point queries, and the hierarchical clustering is the best for range queries. In general, the recursion clustering is suitable for both queries.

In the future, we will study the support of the iceberg cube [Beyer and Ramakrishnan. 1999] [D. Xin and Wah. 2003]. Compared with point queries and range queries, the iceberg query is a special type of query. The former searches aggregations from dimension values essentially, but the latter is opposite. Because the aggregations are all stored in the leaf nodes, an iceberg query will access almost all the nodes, which has no significance.

In addition, the high capability and the pre-fetch function of the high-speed memory buffer of Windows NT disk I/O system can reduce the I/O times, which gives us the idea that whether it is effective for the high randomization Dwarf file.

ACKNOWLEDGMENTS

The research reported here is supported by the National Natural Science Foundation of China under Grant No.60773222.

REFERENCES

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

Chen, S. and Rundensteiner., E. A. 2005. Gpivot: Efficient incremental maintenance of complex rolap views. *In ICDE*, 552–563.

D. Xin, J. Han, X. L. and Wah., B. W. 2003. Star-cubing: Computing iceberg cubes by top-down and bottom-up integration. *In VLDB*, 476–487.

Gibbons, P. B. and Matias., Y. 1998. New sampling-based summary statistics for improving approximate query answers. *In SIGMOD*, 331–342.

H. He, J. Xie, J. Y. and Yu., H. 2005. Asymmetric batch incremental view maintenance. *In ICDE*, 106–117.

J. S. Vitter, M. W. and Iyer., B. 1998. Data cube approximation and histograms via wavelets. *In CIKM*, 96–104.

J. Shanmugasundaram, U. F. and Bradley., P. S. 1999. Compressed data cubes for olap aggregate query approximation on continuous dimensions. *In KDD*, 223–232.

N. Folkert, A. Gupta, A. W. e. a. 2005. Optimizing refresh of a set of materialized views. *In VLDB*, 1043–1054.

S. Acharya, P. B. G. and Poosala., V. 2000. Congressional samples for approximate answering of group-by queries. *In SIGMOD*, 487–498.

W. Wang, J. Feng, H. L. and Yu., J. X. 2002. Condensed cube: An effective approach to reducing data cube size. *In ICDE*, 155–165.

Y. Sismanis, N. Roussopoulos, A. D. and Kotidis., Y. 2002. Dwarf: Shrinking the petacube. *In SIGMOD*, 464–475.

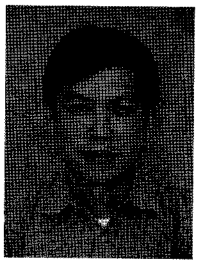
Y. Sismanis, A. Deligiannakis, Y. K. N. R. 2003. Hierarchical dwarfs for the rollup cube. *In DOLAP*, 17–24.

Y. Zhao, P. M. Deshpande, J. F. N. 1997. An array-based algorithm for simultaneous multidimensional aggregates. *In SIGMOD*, 159–170.

Yubin Bao is an associate professor of School of Information Science and Engineering at Northeastern University in China. He received his PhD degree in 2003 from Northeastern University of China. His research interests include data warehouse and OLAP.

Fangling Leng is a doctor candidate of School of Information Science and Engineering at Northeastern University in China. Her research interests include data warehouse and OLAP.

Daling Wang is a professor of School of Information Science and Engineering at Northeastern University in China. She received her PhD degree in 2003 from Northeastern University of China. She is currently a professor and associate PhD advisor in Northeastern University of China. Her research interests include web mining, information retrieval.



Ge Yu is an associate dean of School of Information Science and Engineering, a professor of School of Information Science and Engineering at Northeastern University in China. Ge Yu received his PhD in computer science from Kyushu University in Japan. He is a member of the ACM. His research interests include data warehouse and OLAP, distributed data processing, data mining, data stream, Grid computing.