Journal of Zhejiang University SCIENCE
ISSN 1009-3095
http://www.zju.edu.cn/jzus
E-mail: jzus@zju.edu.cn

Construction and compression of Dwarf*

XIANG Long-gang (向隆刚), FENG Yu-cai (冯玉才), GUI Hao (桂 浩)

(*School of Computer Science, Huazhong University of Science and Technology, Wuhan 430074, China*)

E-mail: lg_xiang@hotmail.com; fyc@dm2.com.cn; tigerguihao@sina.com

Received Apr. 20, 2004; revision accepted Aug. 1, 2004

Abstract: There exists an inherent difficulty in the original algorithm for the construction of Dwarf, which prevents it from constructing true Dwarfs. We explained when and why it introduces suffix redundancies into the Dwarf structure. To solve this problem, we proposed a completely new algorithm called PID. It bottom-up computes partitions of a fact table, and inserts them into the Dwarf structure. If a partition is an MSV partition, coalesce its sub-Dwarf; otherwise create necessary nodes and cells. Our performance study showed that PID is efficient. For further condensing of Dwarf, we proposed Condensed Dwarf, a more compressed structure, combining the strength of Dwarf and Condensed Cube. By eliminating unnecessary stores of "ALL" cells from the Dwarf structure, Condensed Dwarf could effectively reduce the size of Dwarf, especially for Dwarfs of the real world, which was illustrated by our experiments. Its query processing is still simple and, only two minor modifications to PID are required for the construction of Condensed Dwarf.

Key words: Data cube, Dwarf, Suffix coalescing, Prefix path, MSV partition, Condensed Dwarf
doi:10.1631/jzus.2005.A0519 **Document code:** A **CLC number:** TP311.13

INTRODUCTION

The CUBE BY operator (Gray *et al*., 1996) is an essential facility for data warehousing and OLAP. It is a multidimensional extension of the standard GROUP BY operator, computing all possible combinations of the grouping attributes in the CUBE BY clause. A CUBE BY with N grouping attributes will compute 2^N group-bys. In the real world, a fact table is often very large and sparse. In such cases, the size of a group-by is possibly close to the size of the fact table. So the size of a data cube increases exponentially after computation, and so the inherent difficulty with the CUBE BY operator is its size, both for computing and storing it.

Dwarf (Sismanis *et al*., 2002) is a highly compressed tree-like structure for computing, storing and querying data cubes. Dwarf solves the storage space problem by identifying prefix and suffix redundancies among cube tuples and factoring them out of the store. Any prefix of a cube tuple will appear in $2^{(n-d)}$ group-bys (n is the number of cube dimensions and d is the prefix length), and possibly many times in each group-by. Suffix redundancy occurs whenever two or more partitions share common participating dimensions and cover the same subset of fact table tuples. For example, for the fact table shown in Table 1, prefix (1, 0) appears in cube tuples: (1, 0, *), (1, 0, 0,) and (1, 0, 1), partitions (1, 0) and (*, 0) always have the same measure for any value of dimension C. Every unique prefix is stored exactly once in the Dwarf structure and all cube tuples with the prefix share its storage. Suffix redundancies will construct identical sub-Dwarfs. They are recognized and only one copy is stored in the Dwarf structure. For the fact table shown in Table 1, the corresponding Dwarf is presented in Fig.1. The number illustrates the order in which nodes close, and the dashed line indicates where the suffix coalescing occurs.

The Dwarf construction algorithm proposed in (Sismanis *et al*., 2002) consists of two interleaved processes: the prefix expansion and the suffix coalescing. The prefix expansion would create a tree if it

*Project (No. 20030487032) supported by the Specialized Research Fund for the Doctoral Program of Higher Education, China

were not for suffix coalescing. The suffix coalescing tries to identify identical sub-Dwarfs and coalesce their store. Though simple it is, a serious problem is hidden deeply in the algorithm, which prevents it from constructing true Dwarfs (i.e., without prefix and suffix redundancies). As an example, for the fact table shown in Table 2, the "Dwarf" constructed according to the algorithm is presented in Fig.2, one can see it clearly that there exists one suffix redundancy in the "Dwarf". Anther problem of the algorithm is that when a node is being closed, it persists to insert the "ALL" cell and create the sub-Dwarf for it, even if the node contains exactly two cells: one is the "ALL" cell and the other is a normal cell. Actually, we can safely remove the "ALL" cell from such Dwarf nodes, while any cube tuple can still be directly answered from this condensed version of Dwarf.

Table 1 Example table 1

A	*B*	*C*	*M*
0	1	1	7
0	2	0	4
1	0	0	9
1	0	1	5

Table 2 Example table 2

A	*B*	*C*	*D*	*M*
0	0	0	0	8
0	1	0	1	5
1	0	1	1	10

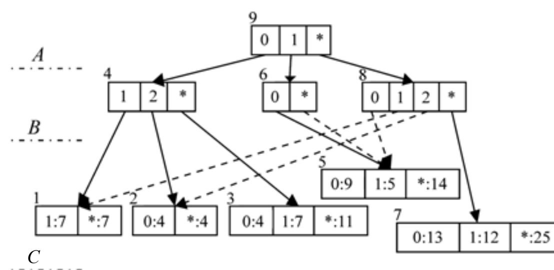

Fig.1 The true Dwarf of Table 1

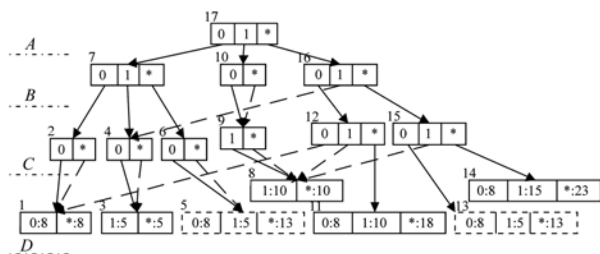

Fig.2 The false Dwarf of Table 2

ORIGINAL ALGORITHM

Two processes: the prefix expansion and the suffix coalescing, govern the Dwarf construction, according to the original algorithm. Prior to the construction, the fact table is sorted using a fixed dimension order. The prefix expansion sequentially scans over the sorted fact table. Every time a tuple is read, necessary nodes and cells are created, by comparing the current tuple with the previous one. When a leaf node is closed, the "ALL" cell is introduced by aggregating the contents of the other cells in the node. When an inner node is closed, the "ALL" cell is created and call the suffix coalescing to create the sub-Dwarf dominated by this cell. The suffix coalescing algorithm is presented in Fig.3. It requires as input a set of Dwarfs and merges them to construct the resulting Dwarf. For the root node of the resulting Dwarf, the sub-Dwarf of the cell with value *k* is constructed by merging those sub-Dwarfs of the cells in the top nodes of the input Dwarfs with value *k*, and the sub-Dwarf of the "ALL" cell is constructed by merging its brother Dwarfs (i.e., the sub-Dwarfs of other cells within the same node). If there is just one Dwarf to be merged, then coalescing happens immediately, since the result of merging one Dwarf will obviously be the Dwarf itself.

Algorithm: SuffixCoalesce

Input: inputDwarfs = set of Dwarfs
1: if only one Dwarf in inputDwarfs then
2: return Dwarf in inputDwarfs;
3: end if
4: while unprocessed cells exist in the top nodes of inputDwarfs do
5: find unprocessed key $Key_{\min}$ with minimum value in the top nodes of inputDwarfs;
6: toMerge ← set of Cells of top nodes of inputDwarfs having keys with values equal to $Key_{\min}$;
7: if already in the last level of structure then
8: curAggr ← calculateAggrgate(toMerge.aggregateValues);
9: write cell [$Key_{\min}$, curAggr];
10: else
11: write cell [$Key_{\min}$, SuffixCoalesce(ToMerge.SubDwarfs)];
12: end if
13: end while
14: create the ALL cell for this node either by aggregation or by calling SuffixCoalesce, with the sub-Dwarfs of the node's normal cells as input;
15: return position in disk where resulting Dwarf starts;

Fig.3 Algorithm SuffixCoalescing

When constructing the sub-Dwarf *SD* of the "ALL" cell in node *N*, the suffix coalescing does not take into account the contents of the sub-Dwarfs *ASD*s of the "ALL" cells in the brother nodes of *N*.

Therefore the probability exists that the same merging work that has been done in the construction of *ASD*s will be executed once again during constructing *SD*. As an example, consider again the false Dwarf shown in Fig.2. According to the original algorithm, Dwarf 15 (i.e., node 15 and all the nodes that can be visited via it. Such kind of reference is used throughout this paper) is constructed based on Dwarfs 4 and 12. Then, Dwarf 13 is constructed by merging Dwarfs 1 and 3. But this merging work has already been done in the construction of Dwarf 5. The suffix coalescing fails to be aware of the existence of Dwarf 5, just because it does not take Dwarf 6 as input when constructing Dwarf 15.

Let us first define some terms to help in the understanding of the problem hidden in the suffix coalescing. A prefix path covers a unique subset of fact table tuples, which take the same values with the prefix path for those dimensions of the prefix path with a normal value. Assuming *PP* is a prefix path in a Dwarf, and dimension k of *PP* takes the value of "ALL". If a single value v_k of dimension k appears in all the fact table tuples covered by *PP*, then dimension k is called a missing single value (MSV) dimension; otherwise it is called a normal missing (NM) dimension. As an example, consider again the Dwarf of Fig. 1. The prefix path of (*, 1) covers one fact table tuples: (0, 1, 1: 7), the first dimension of this prefix path is an MSV dimension. While for the prefix path of (*, *), the first dimension is an NM dimension.

In general, for a prefix path *PP* with the form of $\alpha.*_i.\beta.v_j$, where α and β are two sequences of dimension values, $*_i$ means that the value of "ALL" appears on dimension i, and v_j means that dimension j (j>i) takes the value of v_j (including "ALL"). If dimension i is an MSV dimension (assuming the single value for dimension i is v_i), and if there exists no MSV dimension and at least one NM dimension in β, then the suffix coalescing will lose its magic to coalesce the sub-Dwarf of *PP* with the sub-Dwarf of the prefix path of $\alpha.v_i.\beta.v_j$, though they are identical (because they share common participating dimensions and cover the same subset of fact table tuples). Assuming that dimension k (j>k>i) is an NM dimension of β (i.e., at least two different values of dimension k appears in the cube tuples covered by *PP*), and no MSV dimension exists in β. Therefore the sub-Dwarf of $\alpha.v_i.\beta.v_j$ must be constructed by merging other sub-Dwarfs. Since the suffix coalescing does not take as input the sub-Dwarf of $\alpha.v_i.\beta.v_j$, the merging work constructing the sub-Dwarf of $\alpha.v_i.\beta.v_j$ is done once again to construct the sub-Dwarf of *PP*. Consequently, the sub-Dwarf of *PP* fails to be coalesced with the sub-Dwarf of $\alpha.v_i.\beta.v_j$, resulting in a redundant sub-Dwarf in the Dwarf structure. As an example, the prefix path of (*, *, 0) in the false Dwarf of Fig.2 introduces the redundant sub-Dwarf, because dimension A is an MSV dimension and dimension B is an NM dimension.

Then, can we construct the sub-Dwarf of an "ALL" cell as normal cells? (Remove line 14 from the suffix coalescing shown in Fig.3, take into account the value of "ALL" at line 4, and let it to be the biggest value in the domain of any dimension.) Such modification will certainly avoid the problem stated above. However, it will introduce another type of suffix redundancy, which occurs more commonly! Since when constructing the sub-Dwarf of an "ALL" cell, it does not take as input its brother sub-Dwarfs, it loses the chance to share the merging works in the brother sub-Dwarfs. As an example, consider the fact table shown in Table 3. With the modified version of the suffix coalescing, the resulting "Dwarf" is presented in Fig.4. Dwarf 12 fails to be coalesced with Dwarf 11 because when constructing Dwarf 12, its input Dwarfs are Dwarfs 1 and 4, so the merging work, though already being done in the construction of

Table 3 Example table 3

A	*B*	*C*	*D*	*M*
0	0	0	0	5
1	0	0	1	3
1	1	1	1	4

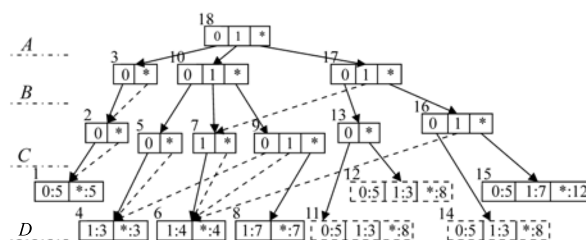

Fig.4 The false Dwarf of Table 3

Dwarf 11, is executed once again. The redundancy of Dwarf 12 may be removed due to the fact that there exists only one normal cell in node 13; however the redundancy of Dwarf 14 cannot be easily eliminated.

PID ALGORITHM

Based on the above reasoning and examples, we can conclude that the original algorithm has an inherent limitation arising from the sub-Dwarf construction of an "ALL" cell, which prevent it from constructing true Dwarfs. In this section, we propose a completely new algorithm called PID (Partitioning and Inserting Dwarfing) to solve this problem. Before we present the algorithm, we first study some preliminary terms. Assume a fact table has d dimensions, a partition has the form of $(t_1,\ldots,t_i,\ldots,t_k)$ $(k\leq d)$, where value t_i of dimension i can take the special value of "ALL" or any value from its domain. A partition defines a unique subset of fact table tuples, which take the same values with the partition for those dimensions of the partition with a normal value. From this definition, one can conclude that fact table partitions and prefix paths are of one-to-one relationship, i.e., partition $pt=(t_1,\ldots,t_i,\ldots,t_k)$ matches prefix path $pp=(t_1,\ldots,t_i,\ldots,t_k)$. If a partition contains at least one MSV dimension, it is called an MSV partition. Among all partitions that share common participating dimensions and cover the same subset of fact table tuples, the partition with no MSV dimension is called the base partition. As an example, consider again the fact table of Table 1 and the corresponding Dwarf of Fig.1. Partition (1, *) covers the subset of fact table tuples: {(1, 0, 0: 9), (1, 0, 1: 5)}, and it matches the prefix path of (1, *). It is an MSV partition, since dimension B is an MSV dimension. Partition (1, 0) is the base partition for the set of three partitions: {(1, 0), (1, *), (*, 0)}.

The main idea of PID is to bottom-up partition a fact table, i.e., it first partitions the fact table using the first dimension, and then partitions each sub-partitions using the second dimension, and so on. Every time a partition is computed, it is inserted into the Dwarf structure. To achieve coalescing, PID tests each inserted partition. If it is an MSV partition, coalesce its suffix and stop further partitioning. Just as the original algorithm, PID recognizes a suffix redundancy before actually creating it.

The details of PID are presented in Fig.5. First, we test partition *ptn*. If at least one dimension of *ptn* is an MSV dimension, i.e., *ptn* is an MSV partition, first compute partition *ptn'* from partition *ptn*: for each MSV dimension of partition *ptn*, replace the "ALL" value with its single value. Then let the prefix path of *ptn* point to the sub-Dwarf of prefix path *ptn'* (i.e., coalesce the sub-Dwarf of prefix path *ptn*). Lastly exit from this iteration. Next, we test whether partition *ptn* contains a single fact table tuple or not. If it does, directly construct the whole sub-Dwarf of prefix path *ptn* without further partitioning on partition *ptn*: for each dimension in [current dimension, last dimension], create a node with two cells: one is a normal cell and the other is the "ALL" cell, and let the two cells point to the node belonging to the next dimension. Next, we create a new node to hold all distinct values of the current dimension in partition *ptn*. Next, we partition *ptn* on the current dimension. Then, we process sub-partitions one by one: insert into the new node a cell, which take the value of the current dimension in sub-partition *sub-ptn*, and recursively call

```
Algorithm: PID(ptn, dim)
Input:
ptn: the current partition for this iteration;
dim: the current dimension for this iteration.
Method:
1.  if ptn is an MSV partition then
2.    compute its base partition ptn';
3.    let prefix path ptn point to the sub-Dwarf of prefix
      path ptn';
4.    return;
5.  end if
6.  if ptn contains a single fact table tuple then
7.    direct-write-Dwarf(ptn, dim+1);
8.    return;
9.  end if
10. create a new node;
11. Partition(ptn, dim);
12. for each sub-partition sub-ptn
13.   insert a new cell into the new node;
14.   if dim < total-dimensions then
15.     PID(sub-ptn, dim+1);
16.   end if
17. end for
18. insert the "ALL" cell into the new node;
19. if dim < total-dimensions then
20.   PID(ptn, dim+1);
21. end if
22. link the new node with its parent cell;
```

Fig.5 Algorithm PID

PID with sub-partition *sub-ptn* and dimension (*dim*+1) as input. Next, we create the "ALL" cell for the new node and construct the corresponding sub-Dwarf. The last step is to link the newly created node with its parent cell.

As an example, assuming that (a_1, b_1), (a_2, b_2) and (*, b_3) are three partitions of a fact table with four dimensions: *A*, *B*, *C* and *D*. (a_1, b_1) is a normal partition, (a_2, b_2) is a single tuple partition, in which the dimension values for *C* and *D* are c_2 and d_2 respectively, and (*, b_3) is an MSV partition, where dimension *A* takes a single value of a_3. Fig.6 illustrates how to construct the sub-Dwarfs for the three typical partitions.

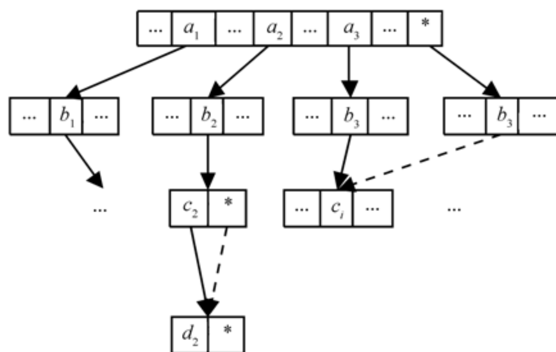

Fig.6 Sub-Dwarfs for three typical partitions

1. Rationale

The correctness of the algorithm is based on the partition inserting and the MSV partition detecting. According to the definition of Dwarf, two or more sub-Dwarfs are identical if their prefix paths cover the same subset of fact table tuples. Consider those partitions that correspond to the prefix paths pointing to an identical sub-Dwarf, they must cover the same subset of fact table tuples, and the participating dimensions of these partitions are identical. The only difference for these partitions is that for any single value dimension of the subset of fact table tuples, some partitions take the single value and the others take the value of "ALL". For each newly computed partition, if it is an MSV partition, PID recognizes its base partition and shares the storage of the sub-Dwarf constructed from the base partition. Otherwise, PID creates a new node to hold the values of the current dimension in the partition. So PID achieves sharing of prefixes and coalescing of suffixes, and so it constructs true Dwarfs.

2. Efficiency

PID eliminates all prefix and suffix redundancies prior to the computation of the redundant values. As a result, not only is the storage space requirement reduced, but its computation is also accelerated. PID builds the Dwarf by computing and inserting partitions, so the sub-Dwarf of an "ALL" cell is constructed just as normal sub-Dwarfs, and the previously constructed sub-Dwarfs are not required to be loaded into the main memory from disk, resulting in a remarkable decrease of I/Os when compared with the original algorithm. A key optimization of PID is the single tuple partition optimization, i.e., for a single tuple partition, the time-consuming partitioning on current dimension is avoided, and if it is not an MSV partition, the whole sub-Dwarf of the partition is directly constructed without further partitioning. Fortunately, data cubes of the real world are often sparse, where large numbers of partitions contain a single fact table tuple. Therefore this optimization can contribute significantly to the overall performance of PID.

CONDENSED DWARF

Note that if a Dwarf node contains exactly two cells: one is the "ALL" cells and the other is a normal cell, it is unnecessary to explicitly store the "ALL" cell, because they point to the same sub-Dwarf. This inspires us to propose a more condensed data structure called Condensed Dwarf. It is a condensed version of Dwarf, which deletes the "ALL" cell from a Dwarf node containing two cells. In fact, only if a partition contains a single fact table tuple, or it takes a single value on the current dimension in the covered fact table tuple, PID will create nodes with two cells. By avoiding unnecessary storing of "ALL" cells, Condensed Dwarf effectively reduces the Dwarf size, especially for Dwarfs of the real world. The reason is as follows: the data is often very sparse and high correlated in such cases. With a high sparseness, many partitions are single tuple partitions. With a high correlation, many partitions take a single value on the current dimension in the covered fact table tuples. Even without the paths following such "ALL" cells, any point query can be directly answered with-

out further aggregation, i.e., Condensed Dwarf is still a complete data cube store. As an example, consider again the fact table of Table 3. The corresponding condensed Dwarf is presented in Fig.7.

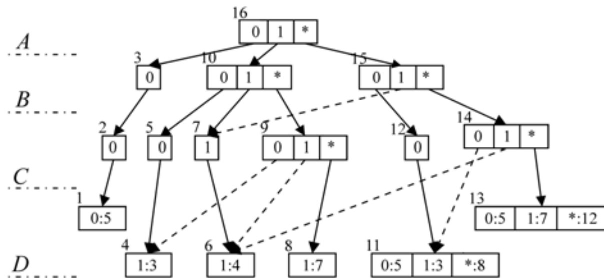

Fig.7 The condensed Dwarf of Table 3

Since the "ALL" cell in any two-cell node is removed in Condensed Dwarf, the query processing over Condensed Dwarf differs from that over Dwarf, but just a minor modification to the Dwarf query processing is required to fit for Condensed Dwarf. Let $c=(t_1,\ldots,t_n)$ be a point query to a condensed Dwarf, where t_i (i=1,…,n) takes any value, including "ALL", and n is the number of dimensions. Answering c requires just a simple traversal on the condensed Dwarf from the root to a leaf. At level i, we search for the cell with value t_i and descend to the next level. If t_i is "ALL" and the "ALL" cell does not appear on the searched node (i.e., it was removed), we follow the pointer of the single normal cell. A range query of Condensed Dwarf is handled similarly with Dwarf, i.e., compile the range query into a set of point queries and answer them one by one. Based on above description, one can see that Condensed Dwarf has equivalent performance as Dwarf in query answering. Regarding query answering, one path of Condensed Dwarf can answer more than one point query, while in Dwarf one path matches a unique cube tuple. Therefore Condensed Dwarf combines the strength of Dwarf and Condensed Cube (Wang *et al.*, 2002), i.e., two or more cube tuples are condensed into one Dwarf path, and such one path can answer more than one point query without computation.

With the PID algorithm presented in Fig.5, we can easily write out the algorithm for the construction of Condensed Dwarf, which differs from PID in two points. One is at line 7, i.e., do not create the "ALL" cell for each newly created node when constructing the sub-Dwarf for a single tuple partition. The other is that the actions of lines 17~20 should be controlled by an "if" statement, i.e., only if more than one distinct value of the current dimension appear in the fact table tuples covered by the current partition, execute lines 17~20; otherwise skip them.

EXPERIMENTAL STUDY

In this section, we conducted comprehensive experiments to validate our performance and storage expectation. All experiments were conducted on an Intel Pentium IV 1.8 GHz PC with 512 MB main memory, running Windows 2000 Advanced Server. All programs were coded in Microsoft Visual C++ 6.0. The times recorded included the initialization time, the computation time and the I/O time.

Both synthetic and real datasets were used in these experiments. Synthetic datasets satisfy *Zipf* distribution. When *Zipf*=0, the data is uniform. As *Zipf* increases, the data is more skewed. The cardinality is the same for each dimension in synthetic datasets. The real dataset records weather conditions at various weather stations on land for September 1985 (Hahn *et al.*, 1994). The dataset contains 1015367 tuples. The dimensions are ordered by cardinality: station-id (7037), longitude (352), solar-altitude (179), latitude (152), present-weather (101), day (30), weather-change-code (10), hour (8) and brightness (2). All datasets contain one measure attribute.

Performance analysis

The first set of experiments reported our performance analysis on constructing Dwarfs. To show the performance of PID, we compared it with the original algorithm (ORGALG for short), though it actually constructs false Dwarfs. Both synthetic and real datasets were evaluated. The experimental results are shown in Fig.8.

There are three main points that can be taken from these results. First, PID and ORGALG have similar performance for most data distributions, whereas PID runs much faster than ORGALG for weather data because of the high sparseness of the weather dataset. Second, for large data cubes, the output I/O dominates the cost of construction. Consequently, Condensed Dwarf will not only decrease the I/O requirements, but also improve the constructi-

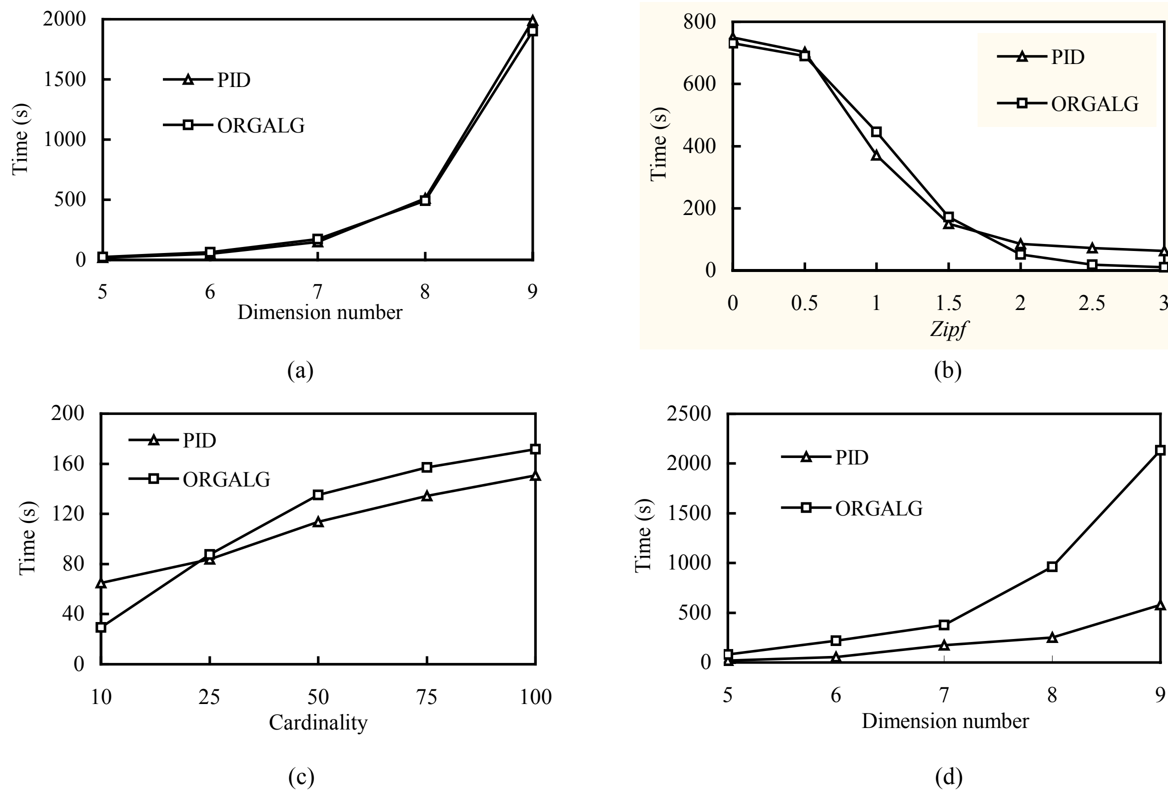

Fig.8 Evaluating the performance of PID

(a) Synthetic data (Cardinality=100, Tuple=1000 thousand, *Zipf*=1.5); (b) Synthetic data (Dimension=7, Cardinality=100, Tuple=1000 thousand); (c) Synthetic data (Dimension=7, Tuple=1000 thousand, *Zipf*=1.5); (d) Weather data

on performance, when compared with Dwarf. Third, as the sparseness increases, the performance improvement gained by the optimization of single tuple partitions increases correspondingly.

Compression analysis

In this subsection, we explored the compression benefits of Condensed Dwarf over Dwarf. The datasets used here are the same as those in the above subsection. To show the compression, we compared the storage size of Condensed Dwarf with that of Dwarf. We also show in the graphs the storage sizes of false Dwarfs constructed by ORGALG. The experimental results are shown in Fig.9.

The results showed that Condensed Dwarf effectively reduces the Dwarf size. One can see that the compression ratio is mainly influenced by three factors, namely sparseness, correlation and cardinality distribution. Given two datasets with similar sparseness, the dimension cardinalities are nearly the same for each dimension in the first dataset, while they are discriminating in the second dataset, just as the case with the weather dataset. The compression ratio for the second dataset is significantly higher than the first. The reason is that large numbers of partitions are single tuple partitions in the second dataset. For example, the condensed Dwarfs in Fig.9 reduce the storage sizes of the Dwarfs: 25% and 11% for the weather data and the synthetic data respectively, where the dimension number is 9.

RELATED WORK

Efficiently computing and storing data cubes are big challenges for building real OLAP applications, due to their huge sizes. A lot of works have been done on the data cube computation and store in the past recent years.

Since Gray *et al*. first proposed the data cube operator in 1996, many approaches have been suggested for data cube computation. Array-Cube (Zhao

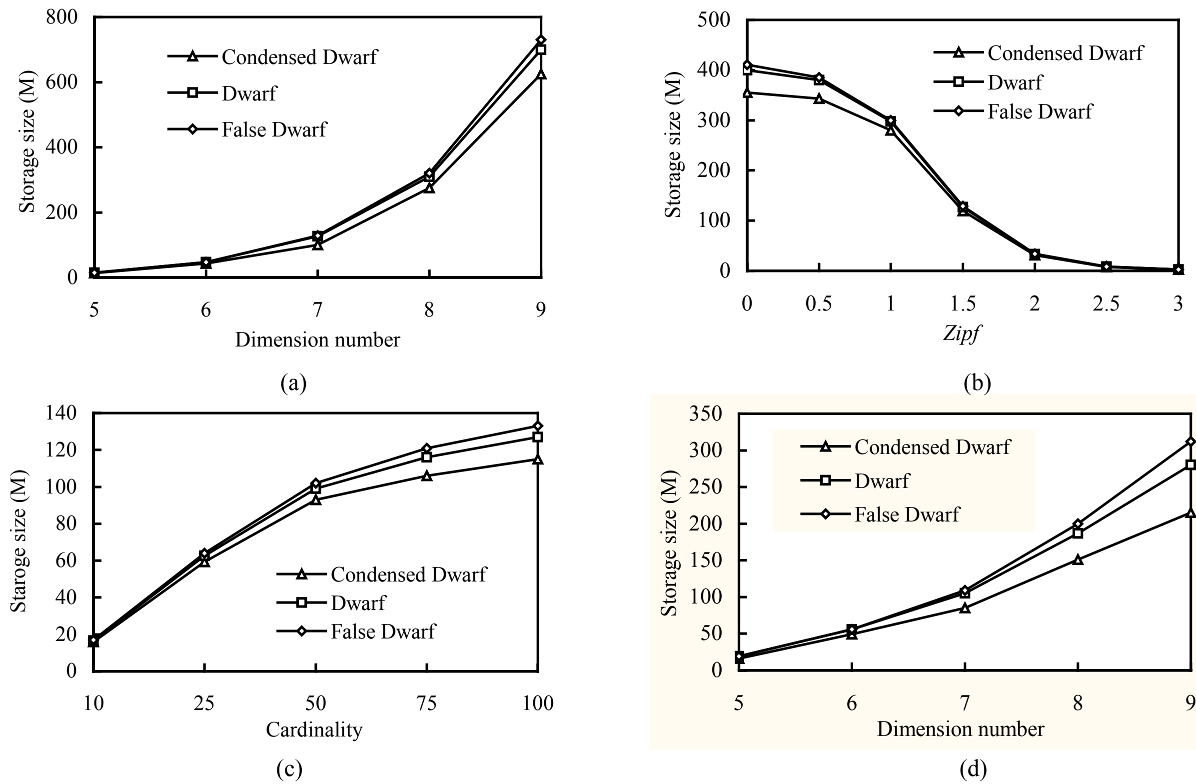

Fig.9 Evaluating the compression of Condensed Dwarf
(a) Synthetic data (Cardinality=100, Tuple=1000 thousand, *Zipf*=1.5); (b) Synthetic data (Dimension=7, Cardinality=100, Tuple=1000 thousand); (c) Synthetic data (Dimension=7, Tuple=1000 thousand, *Zipf*=1.5); (d) Weather data

et al., 1997) is an array-based top-down cubing algorithm. BUC (Beyer and Ramakrishnan, 1999) employs a bottom-up computation and prunes unsatisfied partitions in an Apriori-like manner. Star-Cube (Xin *et al*., 2003) utilizes a Star-Tree structure and extends the simultaneous aggregation method.

Works on compressing the data cube are of clear relevance to us. Iceberg-Cube (Fang *et al*., 1998) is a subset of a data cube containing only those cube tuples whose measure satisfies certain constraints. Condensed-Cube uses the two ideas of “base single tuple” compression and “projected single tuple” compression as a basis for compressing a data cube. Quotient-Cube (Lakshmanan *et al*., 2002) and QC-tree (Lakshmanan *et al*., 2003) creates a summary structure by partitioning the set of cells of a data cube into classes such that cells in a class have the same aggregate measure.

CONCLUSION

In this paper, we explain when the original algorithm for the construction of Dwarf introduces suffix redundancies, and why it is difficult to be corrected from the original algorithm. To solve this problem, we propose a completely new algorithm called PID. Different from the original algorithm, PID is based on partition computing and inserting. PID is efficient, which was illustrated by our experiments on performance analysis.

Another contribution of this paper is to propose Condensed Dwarf. It reduces the Dwarf size by deleting the “ALL” cell from a Dwarf node with two cells. With the increased sparseness and correlation, Condensed Dwarf saves more and more storage space when compared with Dwarf, making it very fit for storing data cubes of the real world. The query proc-

essing over Condensed Dwarf is very straightforward, and the construction of Condensed Dwarf can be implemented by simply modifying the PID algorithm.

Though interesting progress has been made for constructing and condensing Dwarf, incrementally updating Condensed Dwarf based on PID algorithm is still an open question. Our ongoing research will address this issue.

References

Beyer, K., Ramakrishnan, R., 1999. Bottom-up Computation of Sparse and Iceberg CUBEs. Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data, ACM press, p.359-370.

Fang, M., Shivkumar, N., Garcia-Molina, H., Motwani, R., Ullman, J.D., 1998. Computing Iceberg Queries Efficiently. Proceedings of 24th International Conference on Very Large Data Bases, Morgan Kaufmann Press, p.299-310.

Gray, J., Bosworth, B., Layman, A., Pirahesh, H., 1996. Data Cube: A Relational Operator Generalizing Group-by, Cross-tab, and Sub-totals. Proceedings of the 12th International Conference on Data Engineering, IEEE Computer Society Press, p.152-159.

Hahn, C., Warren, S., London, J., 1994. Edited Synoptic Cloud Reports from Ships and Land Stations over the Globe, 1982–1991. http://cidiac.est.ornl.gov/ftp/ndp026b/SEP-85L.z.

Lakshmanan, L.V.S., Pei, J., Han, J.W., 2002. Quotient Cube: How to Summarize the Semantics of A Data Cube. Proceedings of 28th International Conference on Very Large Data Bases, Morgan Kaufmann Press, p.766-777.

Lakshmanan, L.V.S., Pei, J., Zhao, Y., 2003. QC-Trees: An Effective Summary Structure for Semantic OLAP. Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, ACM Press, p.64-75.

Sismanis, Y., Deligiannakis, A., Roussopoulos, N., Kotidis, Y., 2002. Dwarf: Shrinking the PetaCube. Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, ACM Press, p.464-475.

Wang, W., Feng, J.L., Lu, H.J., Yu, J.X., 2002. Condensed Cube: An Effective Approach to Reducing Data Cube Size. Proceedings of the 18th International Conference on Data Engineering, IEEE Computer Society Press, p.155-165.

Xin, D., Han, J.W., Li, X.L., Wah, B.W., 2003. Star-cubing: Computing Iceberg Cubes by Top-down and Bottom-up Integration. Proceedings of 29th International Conference on Very Large Data Bases, Morgan Kaufmann Press, p.476-487.

Zhao, Y., Deshpande, P., Naughton, J.F., 1997. An Array-based Algorithm for Simultaneous Multidimensional Aggregates. Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data, ACM Press, p.159-170.

Welcome visiting our journal website: ***http://www.zju.edu.cn/jzus***
Welcome contributions & subscription from all over the world
The editor would welcome your view or comments on any item in the journal, or related matters
Please write to: Helen Zhang, Managing Editor of JZUS
E-mail: **jzus@zju.edu.cn** Tel/Fax: 86-571-87952276