A Parallel & Distributed Method for Computing High Dimensional MOLAP*

Kongfa Hu[1,2], Ling Chen [1], Qi Gu [1] , Bin Li [1] , Yisheng Dong [2]
[1] Department of Computer Science and Engineering, Yangzhou University
[2]Department of Computer Science and Engineering, Southeast University

E-mails: kfhu@seu.edu.cn

Abstract. Data cube has been playing an essential role in fast OLAP(on-line analytical processing) in many multidimensional data warehouse. We often execute range queries on aggregate cube computed by pre-aggregate technique in MOLAP. For the cube with d dimensions, it can generate 2^d cuboids. But in a high-dimensional data warehouse (such as the applications of bioinformatics and statistical analysis, etc.), we build all these cuboids and their indices and full materialized the data cube impossibly. In this paper, we propose a multi-dimensional hierarchical fragmentation of the fact table based on dimension hierarchical encoding. This method partition the high dimensional data cube into shell mini-cubes. Using dimension hierarchical encoding and pre-aggregated results, OLAP queries are computed online by dynamically constructing cuboids from the fragment data cubes. Such an approach permits a significant reduction of processing and I/O overhead for many queries by restricting the number of fragments to be processed for both the fact table and bitmap encoding data. This method also supports parallel I/O and parallel processing as well as load balancing for disks and processors. We have compared the methods of our parallel method with the other existed ones such as partial cube by experiment. The analytical and experimental results show that the method of our parallel method proposed in this paper is more efficient than the other existed ones.

1 Introduction

Data warehouses integrate massive amounts of data from multiple sources and are primarily used for decision support purposes. They have to process complex analytical queries for different access forms such as OLAP, data mining, OLAM(on-line analytical mining) etc. Since the advent of data warehousing and online analytical processing (OLAP) [1], data cube has been playing an essential role in the implementation of fast OLAP operations [2]. Materialization of a data cube is a way to pre-

* The research in the paper is supported by the National Natural Science Foundation of China under Grant No. 60473012; the Natural Science Foundation of Jiangsu Province under Grant No. BK2005047, BK2004052 and BK2005046; the National Tenth-Five High Technology Key Project of China under Grant No. 2003BA614A; the Tenth-Five High Technology Key Project of JiangSu Province of China under Grant No. BG2004034.

compute and store multi-dimensional aggregates so that multi-dimensional analysis can be performed on the fly. For this task, there have been many efficient cube computation algorithms proposed, such as ROLAP-based multi-dimensional aggregate computation [3], BUC [4], H-cubing [5], and Star-cubing [6]. Since computing the whole data cube not only requires a substantial amount of time but also generates a huge number of cube cells, there have also been many studies on partial materialization of data cubes [7], computation of condensed[8], dwarf[9], or quotient cubes [10], and computation of approximate cubes [11].

Besides large data warehouse applications, there are other kinds of applications like bioinformatics, statistical analysis, and text processing that need the OLAP data analysis. However, data in such applications usually are high in dimensionality,e.g., over 100 dimensions, and moderate size, e.g., around 10^6 tuples. This kind of datasets behaves rather differently from the datasets in a traditional data warehouse which may have about 10 dimensions but more than 10^9 tuples. Since a data cube grows exponentially with the number of dimensions, it is too costly in both computation time and storage space to materialize a full high-dimensional data cube. For example, a data cube of 100 dimensions, each with 10 distinct values, may contain as many as 11^{100} aggregate cells. If we consider the dimension hierarchies, the aggregate cell will increase by 2^h times. Although the adoption of iceberg cube[5,6], condensed cube[8], or approximate cube[11] delays the explosion, it does not solve the fundamental problem. No feasible data cube can be constructed with such data sets. In this paper we will address the problem of developing an efficient algorithm to perform OLAP on such data sets.

The paper focuses on the design and evaluation of suitable data allocation methods for the fact table and bitmap indices to allow an efficient parallel processing of OLAP queries. We propose a multi-dimensional hierarchical fragmentation of the fact table based on multiple dimension attributes and their dimension hierarchical encoding. Such an approach permits a significant reduction of processing and I/O overhead for many queries by restricting the number of fragments to be processed for both the fact table and bitmap data. Such savings are achieved not only for the fragmentation attributes themselves but also for attributes at different levels of a dimension hierarchy. The proposed data allocation and processing model also supports parallel I/O and parallel processing as well as load balancing for disks and processors.

2 Parallel Shell mini-Cubes

OLAP Queries tend to be complex and ad hoc, often requiring computationally expensive operations such as joins and aggregation. Those queries must be performed on tables having potentially millions of records. The OLAP query that accesses a large number of fact table tuples that are stored in no particular order might result to much more many I/Os, causing a prohibitive long response time. Due to the huge size of the fact table, such full scans are very costly and must be avoided whenever possible even when parallel scans can be utilized. This is also because for most queries, only a small fraction of the fact data is relevant. To illustrate the method ,a tiny database, Table 1, is used as a running example.

Table 1. A sample database with two measure values

TID	DimProduct			dimRegion			dimTime			Measure	
	Category	*Class*	*Product*	*Country*	*Province*	*City*	*Year*	*Month*	*Day*	*Count*	*SaleNum*
1	Office	OA	Computer	China	Jiangsu	Nanjing	1998	1	1	1	20
2	Office	OA	Computer	China	Jiangsu	Nanjing	1998	1	2	1	60
3	Office	OA	Computer	China	Jiangsu	Yangzhou	1998	1	2	1	40
4	Office	OA	Computer	China	Jiangsu	Yangzhou	1998	1	3	1	20
...	...	...	...	...	...	...	...	...	...	...	...
367	Office	OA	Computer	China	Jiangsu	Nanjing	1999	1	2	1	60
...	...	...	...	...	...	...	...	...	...	...	...

From the RPT Cube, we would compute eight cuboids:{(P,R,T),(P,R,All), (P,All,T),(All,R,T),(P,All,All), (All,R,All), (All,All,T), (All,All,All)}.To the cube of d dimensions, it would create 2^d cuboids (The P dimension in these cuboids would be {P, All},such as the R and T dimension .The aggregate cuboids is $\prod_{i=1}^{d} 2 = 2^d$).

For the cube with *d* dimensions (D_1,D_2,...,D_d) and | D_i| distinct values for each dimension D_i, it can generate 2^d cuboids and $\prod_{i=1}^{d}(|D_i|+1)$ cells. If we consider the dimension hierarchies of each dimension, the cube would generate cuboids $\prod_{i=1}^{d}(h_i+1)$ and $\prod_{i}^{d}\prod_{j=1}^{h_i}(|L_j^i|+1)$ cells. (where h_i is the dimension hierarchy levels of the dimension D_i, $|L_j^i|$ is he max number of the distinct member of the hierarchy L_j^i).

For example, the RPT cube in figure 1 has three dimensions: *DimProduct*, *DimRegion* and *DimTime*. The *DimProduct* dimension has three hierarchies as (*Category*,*Class*,*Product*),the *DimRegion* dimension has three hierarchies as (*Country*,*Province*,*City*),and the *DimTime* dimension has three hierarchies as (*Year*,*Month*,*Day*). Thus this cube would generate $\prod_{i=1}^{d}(h_i+1) = (3+1)*(3+1)*(3+1) = 64$ cuboids such as {(*Product*,*City*,*Day*),(*Product*,*City*,*Month*),(*Product*,*City*,*Year*), (*Product*,*City*,*All*),...,(*All*,*All*,*All*)}.But in a high-dimensional database with many cuboids, it might not be practical to build all these cuboids and their indices. Furthermore, reading via an index implies random access for each row in the cuboid, which could turn out to be more expensive than a sequential scan of the raw data.

A partial solution, which has been implemented in some commercial data warehouse systems is to compute a thin cube shell. For example, one might compute all cuboids with 3 dimensions or less in a 30-dimensional data cube. There are two disadvantages to this approach. First, it still needs to compute $C_{30}^3 + C_{30}^2 + C_{30}^1 = 4525$ cuboids. Second, it does not support OLAP in a large portion of the high-dimensional cube space. If we consider the dimension hierarchies, the cuboids is vary much. So we can use the shell mini-Cubes.

For example, for a database of 30 dimensions, D_1, D_2, ..., D_{30}, we first partition the 30 dimensions into 10 fragments(mini-Cubes) of size 3: (D_1,D_2,D_3), (D_4,D_5,D_6), ... , (D_{28},D_{29},D_{30}). For each fragment, we compute its full data cube while recording the inverted indices. For example, in fragment mini-Cube (D_1,D_2,D_3), we would compute

eight cuboids: {(D_1,D_2,D_3),(D_1,D_2,All),(D_1,All,D_3),(All,D_2,D_3),(D_1,All,All), (All,D_2,All),(All,All,D_3),(All,All,All)}. An inverted encoding index is retained for each cell in the cuboids.

The benefit of this method can be seen by a simple calculation. For a base cuboid of 30 dimensions, there are only 8*10 = 80 cuboids to be computed according to the above shell fragment partition. Comparing this to 4525 cuboids for the cube shell of size 3, the saving is enormous.

As we will see, our multi-dimensional fragmentation permits eliminating some bitmaps, thus improving storage and access overhead. We propose this novel hierarchical encoding on each dimension table. The encoding is implemented through the assignment of a special surrogate key on each dimension table tuple, called dimension hierarchical encoding.We can create the *dimRegion*, *DimTime* and *dimProduct* dimension hierarchy encoding shown in Table 2, Table 3 and Table 4.

Table 2. *DimTime* dimension hierarchy encoding

TimeID	Year	Month	Day	B^{TimeID}
	yyy	mmmm	ddddd	yyymmmmddddd
1	98	Jan	1	001000100001
2	98	Jan	2	001000100010
3	98	Jan	3	001000100011
…	…	…	…	…

Table 3. The *dimRegion* dimension hierarchy encoding

RegionID	Country	Province	City	$B^{RegionID}$
	uuuuuuu	vvvvv	cccc	uuuuuuuvvvvvcccc
1	China	Jiangsu	Nanjing	0000001000010001
2	China	Jiangsu	Yangzhou	0000001000010010
…	…	…	…	…

Table 4. The *dimProduct* dimension hierarchy encoding

ProductID	Category	Class	Product	$B^{ProductID}$
	gggg	aaaaa	ppppppp	ggggaaaaapppppppp
1	Office	OA	Computer	0001000010000001
2	Office	OA	Printer	0001000010000010
…	…	…	…	…

By using dimension hierarchical encoding, we can register a list of tuples IDs (tids) associated with the dimension members for each dimension. For example, the TID list associated with the *dimProduct* , *dimRegion* and *dimTime* dimension are shown in Table 5, Table 6 and Table 7 in turn. To compute a data cube for this database with the measure avg() (obtained by sum()/count()), we need to have a tid-list for each cell: {tid_1,…, tid_n}. Because each tid is uniquely associated with a particular

set of measure values, all future computations just need to fetch the measure values associated with the tuples in the list. In other words, by keeping an array of the ID-measures in memory for online processing, one can handle any complex measure computation. Table 8 shows what exactly should be kept, which is substantially smaller than the database itself.

Table 5.*dimProduct* dimension TID

$B^{ProductID}$	TID List
0001000010000001	1-2-3-4-367
…	…

Table 6.*dimRegion* dimension TID

$B^{RegionID}$	TID List
0000001000010001	1-2-367
0000001000010010	3-4
…	…

Table 7. *dimTime* dimension TID

B^{TimeID}	TID List
001000100001	1
001000100010	2-3
001000100011	4
…	…

Table 8. TID- measure array of Table 2

tid	Count	SaleNum
1	1	20
2	1	60
3	1	40
4	1	20
…	…	…

In our study , the method can rapidly retrieve the matching dimension member hierarchical encoding and evaluate the set of query ranges for each dimension and improve the efficiency of OLAP queries by using dimension hierarchical path prefix and encoding prefix.

By using encoding prefix, we can register the dimension hierarchy encoding and its TID list for every dimension hierarchy for each dimension. For example, the dimension hierarchy encoding and its TID list associated with the dimension hierarchies *Month* and *Province* are shown in Table 9 , and so on.

Table 9. *Month* hierarchy encoding *Prefix* AND its TID

B^{TimeID}	$Bprefix(B^{TimeID},Month)$	TID List
001000100001	0010001	1-2-3-4
001000100010		
001000100011		
…	…	…
010000100001	0100001	367
…	…	…

For each fragment, we compute the complete data cube by intersecting the TID-lists in the dimension and its hierarchies in a bottom-up depths-first order in the cuboid lattice (as seen in [6]). For example, to compute the cell {0001000010000001, 0000001000010001, 0010001 }, we intersect the TID lists of $B^{ProductID}$ =0001000010000001, $B^{RegionID}$ =0000001000010001, and $Bprefix(B^{TimeID},Month)$= 0010001 to get a new list of {1,2}.

3 Parallel & Distributed MOLAP Aggregation Algorithm

The data cube can be distributed across a set of parallel computers by parallel constructing the segment Cubes. Therefore, for the end-user and other potential applications, we consider this data cube as one large virtual cube, which is distributed across a set of parallel computers, which manage the creation, updates and querying of the associated cube portions. To develop appropriate scheduling mechanisms for these management tasks, we consider that the virtual cube is split into several smaller parts, called mini-Cube segments. But a mini-Cube segment could furthermore also be split into smaller segments and so on, till we achieve the level of chunks. They can then be assigned to parallel computers, having sequential or parallel computing power, which are responsible for their management. The algorithm for shell cube segment parallel computation can be summarized as follows.

Algorithm 1 (Parallel Shell mini-Cube Computation)

Input: A base cuboid BC of n dimensions:(D_1; ... ;D_n).

{ partition the set of dimensions :(D_1; ... ;D_n) into a set of k mini-Cube fragments {P_1;..., P_k};

scan base cuboid BC once and do the following with parallel processing

{ insert each <tid, measure> into ID-measure array;

for each attribute value a_i of each dimension D_i;

build an dimension hierarchy encoding index entry: <*B*; TID list>;}

parallel processing all fragment partition P_i as follows

{build a local fragment mini-cubes MC_i by intersecting their corresponding tid-lists and computing their measures;

build MC_i's aggregate cuboids by the cuboid lattice;}

We can parallel construct the high dimensional cube with the Cube segments parallel construction. The system architecture of these shell mini-Cube segment parallel construction is shown in Figure 2.

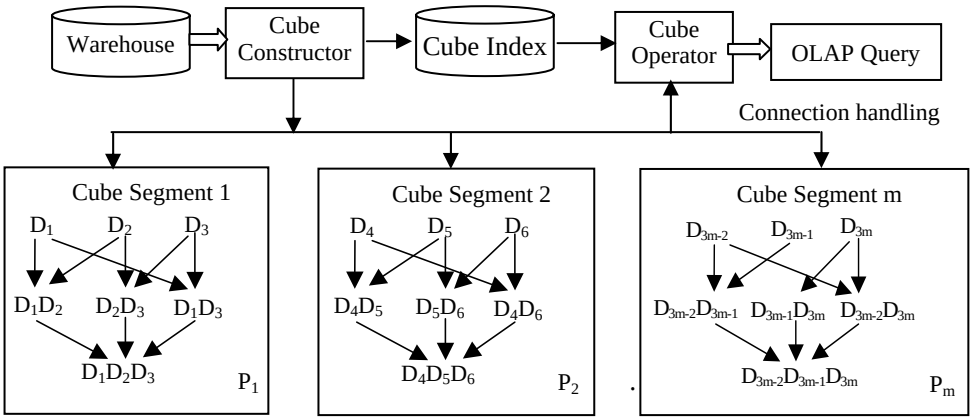

Figure 2. The system architecture of these shell Cube segment parallel construction

4 Performance analysis

The bitmap encoding index table uses the same amount of storage space as the original database. Since we have |T| tuple IDs in total, the entire inverted index will still only need $d \times |T|$ bitmap encoding indices. The amount of memory needed to store the shell mini-cubes of size f is $O(|T|*(2^f * d/f))$,but the amount of storage needed to store partial cube with f dimensions is $O(|T|*(\sum_{i=1}^{f} C_d^i))$,and the full cube's is $O(|T|*(2^d))$.

Based on the above analysis, for a base cuboid of 30 dimensions with 10^6 tuples, our precomputed shell fragments of size 3 will consist of 80 cuboids plus one ID measure array, with the total estimated size of roughly 320 + 12 = 332 MB in total. In comparison, a shell cube of size 3 will consist of 4525 cuboids, with estimated roughly 18 GB in size. A full 30-dimensional cube will have $2^{30} = 10^9$ cuboids, with the total cube size beyond the summation of the capacities of all storage devices. The performance of shell fragment mini-cube method with the partial cube is shown in Figure 3- Figure 5.

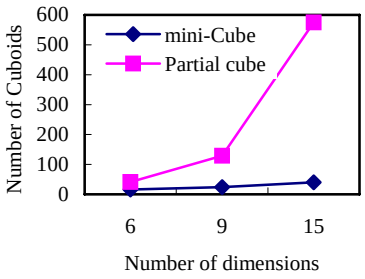

Figure 3. Cuboids of mini-Cube

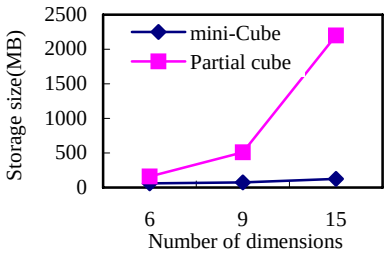

Figure 4. Storage size of mini-Cube

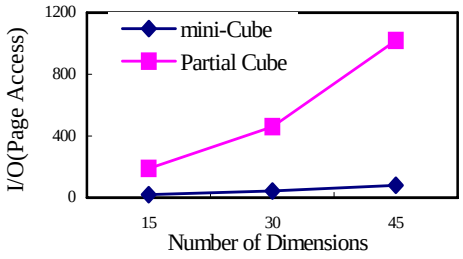

Figure 5. Average I/Os

Figure 3-Figure 5 show the shell fragment mini-cube method has more efficient than other existed ones.

5 Conclusion

Data cube has been playing an essential role in fast OLAP in many multidimensional data warehouse. We often execute range queries on aggregate cube computed by pre-

aggregate technique in MOLAP. For the cube with d dimensions, it can generate 2^d cuboids. But in a high-dimensional data warehouse(such as the applications of bioinformatics and statistical analysis, etc.), while full materialization of the data cube is impossible, we have proposed a reasonable method to partition the high dimensional cube into a set of disjoint low dimensional cubes (i.e., shell fragment mini-cubes). We propose a multi-dimensional hierarchical fragmentation of the fact table based on multiple dimension attributes and their dimension hierarchical encoding. Using inverted hierarchical encoding indices and pre-aggregated results, OLAP queries are computed online by dynamically constructing cuboids from the fragment data cubes. With this method, for high-dimensional OLAP, the total space that needs to store such shell-fragment mini-cubes is negligible in comparison with a high-dimensional cube. Moreover, the query I/O costs for large data sets are reasonable and are comparable with reading answers from a materialized data cube, when such a cube is available. We have compared the methods of parallel shell mini-cubes with the other existed ones such as partial cube by experiment. The analytical and experimental results show that the methods of our parallel shell mini-cubes proposed in this paper are more efficient than the other existed ones.

References

1. Chaudhuri, U., Dayal, U.: Data Warehousing and OLAP for Decision Support. ACM SIGMOD Record 26 (1997) 507-508
2. Gray, J., Chaudhuri, S., Bosworth, A., Layman, A., Reichart, D., Venkatrao, M., Pellow F., Pirahesh, H.: Data Cube: A Relational Aggregation Operator Generalizing Group-by, Cross-tab and Subtotals. Data Mining and Knowledge Discovery 1(1997)29-54
3. Agarwal, S., Agrawal, R., Deshpande, P. M., Gupta, A., Naughton, J. F., Ramakrishnan, R., Sarawagi, S.: On the Computation of Multidimensional Aggregates. VLDB(1996) 506-521
4. Beyer, K., Ramakrishnan, R.: Bottom-up Computation of Sparse and Iceberg Cubes. ACM SIDMOD (1999) 359-370
5. Han, J., Pei, J., Dong, G., Wang, K.: Efficient Computation of Iceberg Cubes with Complex Measures. ACM SIGMOD (2001)1-12
6. Xin, D., Han, J., Li, X., Wah, B. W.: Star-cubing:Computing Iceberg Cubes by Top-down and Bottom-up Integration. VLDB(2003) 476-487
7. Harinarayan, V., Rajaraman, A., Ullman, J. D.: Implementing Data Cubes Efficiently. ACM SIGMOD (1996) 205-216
8. Wang, W., Lu, H., Feng, J., Yu, J. X.: Condensed Cube: An Effective Approach to Reducing Data Cube Size. ICDE(2002) 155-165
9. Sismanis, Y. , deligiannakis, A., Kotidis, Y., Roussopoulos, N.: Hierarchical Dwarfs for the Rollup Cube.VLDB(2004) 540-551
10. Lakshmanan, L. V. S., Pei, J., Zhao, Y.: Q-trees: An Efficient Summary Structure for Semantic OLAP. ACM SIGMOD(2003) 64-75
11. Shanmugasundaram, J., Fayyad, U. M., Bradley, P. S.: Compressed Data Cubes for OLAP Aggregate Query Approximation on Continuous Dimensions. ACM SIGKDD(1999) 223-232