Multi-Cube Computation

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

Hongjun Lu*
Department of Computer Science
Hong Kong University of Science and Technology
Hong Kong, China
luhj@cs.ust.hk

Abstract

Computing a n-attribute datacube requires the computation of an aggregate function over all groups generated by 2^n interrelated GROUP-BYs. In this paper, we focus on multi-cube computation. We extend the algorithms for single datacube computation to process multiple datacubes simultaneously. The issue we intend to explore is the memory utilization. We propose two multi-cube algorithms, namely, a sort-based algorithm and a hash-based algorithm. Different data skews and sparsities are investigated. Results from our extensive performance studies are reported.

1. Introduction

Todays markets are much more competitive and dynamic than ever. It is highly demanded for information systems to provide ability of analyzing information, and to assist decision makers to make better and faster decisions. To meet this challenge, as a powerful data analysis method for multi-dimensional analysis of data, on-line analytical processing (OLAP) has been successfully deployed in many industries such as manufacturing, retail, financial services, transportation, telecommunications, etc. In relational database systems, the datacube (CUBE) operator [5] generalizes the standard GROUP-BY operator to compute aggregates for every combination of GROUP-BY attributes, and is now supported by commercial database systems like IBM DB2. IBM DB2 extended the traditional GROUP BY functions by adding GROUP BY GROUPING SETS, GROUP BY CUBE, GROUP BY ROLLUP [2]. For example, with a relation Sales(date, product, customer, amount), the following datacube query

```
SELECT date, product, customer, SUM(amount)
FROM Sales
GROUP BY CUBE (date, product, customer)
```

produces the SUM of *amount* for all groups generated by 8 GROUP-BYs, i.e., *(date, product, customer), (date, product), (date, customer), (product, customer), (date), (product), (customer)* and *ALL* (for empty attribute). As such, for a CUBE operator on n attributes, 2^n GROUP-BYs or *cuboids* have to be computed. Sorting-based, hash-based and array-based algorithms were proposed to compute a single datacube [1, 4, 5, 7, 8, 9, 11]. Agarwal et. al. summarized the possible optimization techniques such as smallest-parent, cache-results, amortize-scans, share-partitions and share-sorts, for computing multiple group-bys in datacube computation [1].

Study on multi-cube computation is motivated by two facts. First, due to the rapid growth of information available from both datafeeds and WWW on the Internet, the number of attributes used in tables in data warehouses tend to be very big – easily over 100s. It is impossible for current datacube algorithms to compute a datacube query that involves 2^{100} interrelated cuboids. Consequently, it requests users to specify subsets of attributes or multiple datacubes in their queries. Multi-Dimensional Expressions (MDX) provides a framework in which a user can ask several related OLAP queries in a single MDX expression [3]. Second, in addition, due to the popularity of OLAP techniques, OLAP queries have been used frequently. It requests the system either to process these OLAP queries on the fly, or to process these OLAP queries in a time-window over night. However, due to the globalization of E-commerce, the time-window for processing OLAP queries becomes small. New algorithms are needed to process multiple datacube queries simultaneously.

The issue of processing multiple dimensional queries simultaneously was studied in [10, 6]. Both papers focused on MDX expressions. In [10], three new query evaluation primitives were proposed. The authors considered how to use precomputed aggregates to compute an MDX expression. Algorithms were presented on how to generate a global plan from several related local plans. Along the line of the work in [10], [6] further studied restricted versions of the problem, and proposed approximation and exact algorithms for finding plans within the fixed degree approximation of the optimal cost and optimal costs, respectively. Instead, in this paper, we will study how to extend a single datacube algorithm to process multiple datacubes. The main issue we intend to explore is the memory utilization. Recall that the techniques used in single datacube computation[1] do not consider how to share memory with multiple datacubes simultaneously.

*On leave from The National University of Singapore

0-7695-0996-7/01 $10.00 © 2001 IEEE

The remainder of this paper is organized as follows. Section 2 gives the background information of our study. Related work on single datacube computation is discussed in Section 3. Section 4 and Section 5 discuss a sort-based and a hash-based algorithm, respectively. In Section 6, some interesting results from our extensive experiment studies are given. We conclude the paper in Section 7.

2. Preliminaries

In this section, we provide some notations and background information for datacube computation. Let r be a relation on relation scheme R. Aggregate-by-group is a notion that consists of two things: group-by and aggregate. First, the group-by partitions relation r into groups such that tuples are in the same group if and only if they agree on a given set of attributes, Q. Second, an aggregate function $F(.)$ is applied to an attribute $V \in (R - Q)$ on a group basis. A cuboid is such an aggregate-by-group and is defined as a triple (Q, V, F). In the following discussions, we will identify a cuboid using Q, and focus on distributive aggregates such as SUM, COUNT, AVG, MAX and MIN as discussed in [5].

A datacube on n-attributes, A, is the union of all 2^n cuboids Q_i where $Q_i \subseteq A$. A datacube can be represented as a directed acyclic graph $G = (N, E)$, called *cuboid graph*. Here, N is a set of cuboids such that $\{Q_i \mid Q_i \subseteq A\}$. A directed edge connects cuboid Q_u to cuboid Q_v if cuboid Q_v can be computed from Q_u, that is, $Q_v \subset Q_u$ (where $|Q_v| + 1 = |Q_u|$). For a directed edge $(Q_u, Q_v) \in E$, we call Q_u a parent cuboid of Q_v and call Q_v a child cuboid of Q_u.

Computing a single datacube requires to compute all the cuboids in its cuboid graph. While the root cuboid can only be computed from the original relation, other cuboids can be computed from their parent cuboids. A *cuboid tree* defines the cuboid from which other cuboids should be computed. The smallest-parent optimization technique suggested that a cuboid should be computed from its smallest parent cuboid [1]. In [7], Ross and Srivastava constructed a cuboid tree that minimizes the total number of sorting. Using their `paths` algorithm, the cuboid tree for a 4 attribute datacube is shown in Figure 1. In this study, we construct the same cuboid tree using the `paths` algorithm.

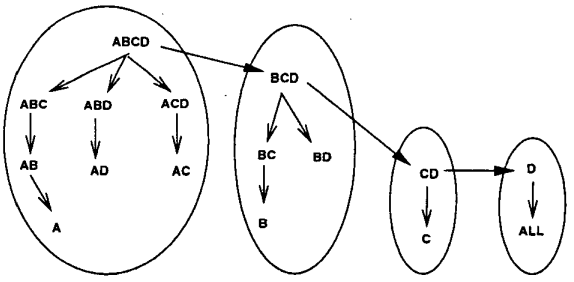


Figure 1. An illustrative cuboid tree example for a datacube on ABCD.

3. Previous Single-Cube Algorithms

The PipeSort algorithm [1, 4] attempts to optimize the overall cost of the computation of a datacube using various ways of cost estimations, in order to determine which cuboid will be used to actually compute other cuboids. Then it converts the resulting tree into a set of paths such that every edge in the tree is in one and only one path. Furthermore, PipeSort performs sorting for the pipelined evaluation of each path. When the number of datacube attributes in the datacube is n, a lower bound on the number of such sorts performed by PipeSort is indicated as $\binom{n}{\lceil n/2 \rceil}$ which is exponential in n as indicated in [7]. The overlap algorithm by Deshpande et al. in [1, 4] attempts to minimize the number of disk accesses by overlapping the computation of the cuboids and making use of partially matching sort orders to reduce the number of sorting steps performed. The I/O costs of the overlap algorithm is at least quadratic in n for sparse data sets, even assuming that partitioning always gives memory-sized partitions. A divide-and-conquer sort-based algorithm was proposed by Ross and Srivastava in [7] which is based on two fundamental ideas: a) partition the large relation into fragments that fit in memory using the `Partition-Cube` algorithm, and b) perform the complex operation over each memory-sized fragment independently using `Memory-Cube` algorithm. When a fragment fits in memory, `Memory-Cube` computes the various cuboids of the datacube using the similar idea of pipelined paths, and picks the minimum number of sorts, which is exactly the largest tier in the lattice, $\binom{n}{\lceil n/2 \rceil}$, where n is the number of attributes appearing in a datacube. When a fragment does not fits in memory, the datacube is broken up into equal-sized $n + 1$ smaller sub-datacube computations recursively. The total I/O cost is $O(n \cdot |R|)$ [7].

As a hash-based algorithm, PipeHash [1, 4] computes a group-by from its smallest parent in the lattice. PipeHash uses a hash table for every simultaneously computed group-by. If all of the hash tables cannot fit in memory, PipeHash partitions the data on some attributes and processes each partition independently. PipeHash suffers two problems. First, it does eager evaluation and attempts to compute group-by on-the-fly. Therefore, it does not overlap as much computation like PipeSort which computes multiple group-bys with one sort. Second, PipeHash requires a significant amount of memory to store the hash tables for the group-bys even after partitioning because it computes cuboids using breadth-first order.

The array-based algorithm proposed by Zhao et al in [11] partitions data into partitions and processes them in an order that requires only fragments of the array

to be present in memory at any one time. Their algorithm performs particularly well because the array representation allows direct access to the needed cells. As pointed in [8, 7], for sparse data, the array cannot fit into memory and so a more costly data structure would be necessary.

Algorithm 1 An Extended Partition-Cube Algorithm, MPC

Input: m datacubes, $C_1, C_2, \cdots, C_m$, an input relation r, an attribute to be aggregated, and an aggregate function.

Output: the results for multiple datacubes.

```
begin
    sorting m datacubes by size in a descending order.
    foreach C_i do
        let C_i' be an incomplete datacube which is
        the result of removing the common parts of
        datacubes that have already been computed.
        Partition-Cube(C_i');
    endforeach
end
```

4. A Sort-Based Multi-Cube Algorithm: MPC

A simple sort-based multi-cube algorithm, called MPC (for Multiple-Partition-Cube), is illustrated in Algorithm 1. For m datacubes with different sizes (the number of attributes), we first sort them by size into a descending order. We attempt to compute large datacubes first, because there are more possibilities to share the sorting costs. We do not compute any cuboid twice. The cuboid, that has been computed in the previous datacubes, will be removed systematically. For each datacube, we use the fastest sort algorithm, `Partition-Cube` [7], to compute.

Recall the multi-datacube example given in Section 1. Suppose that we need to compute two datacubes $\langle A, B, C, D\rangle$ and $\langle B, C, E\rangle$. With the algorithm MPC, we sort the two datacubes by size. Then, we compute $\langle A, B, C, D\rangle$ using `Partition-Cube` [7]. Next, we compute $\langle B, C, E\rangle$. When we compute $\langle B, C, E\rangle$, we do not need to compute $\langle B, C\rangle$ because it has been computed. Therefore, we only need to compute $\langle B, C, E\rangle$, $\langle B, E\rangle$, $\langle C, E\rangle$ and $\langle E\rangle$.

In fact, for m datacubes, MPC needs to scan the raw data m times. However, the strategy is to use all memory where possible to compute a single datacube individually. The MPC inherits the problems from `Partition-Cube` algorithm. For non-skewed partitions, it cannot utilize the memory space well. For skewed partitions, MPC needs to partition data recursively. Memory may not be used efficiently.

5. A Hash-Based Multi-Cube Algorithm: HIPMS+

In this section, we first, introduce a hash-based single-cube algorithm called HIPMS (for Hash In-Place with Memory Shifting) [9]. Then, we extend our HIMPS algorithm to compute multiple datacubes. The extended HIMPS algorithm is called HIMPS+. Like `Partition-Cube` and `Memory-Cube` algorithms, we also use the divide-conquer techniques. Unlike `Partition-Cube` and `Memory-Cube` algorithms, we do not assume the absence of data skew.

5.1. HIPMS

Given a cuboid tree T and an input relation r. Let Q_v be the root cuboid of the cuboid tree, and let $HP_i(r)$ be a disjoint horizontal partition of the input relation. The cuboid tree T can be computed on a partition basis as $computeCuboidSubtree(T, r) = \bigcup computeCuboidSubtree(T, HP_i(r))$, if it satisfies the following two conditions, namely, a) all the cuboids in the subtree have a common prefix X with Q_v where X is a set of attributes, and b) partitioning Q_v is done by hashing on X. We call the two conditions a T-unique property. The T-unique property allows us to compute any partitions of any subtrees during datacube computation. In addition, we can compute cuboids in a breadth-first fashion or a depth-first fashion. Depth-first strategy significantly reduces the amount of memory that needs to store data in memory for later use. PipeHash uses a breadth-first approach that needs to store hash tables even after partitioning. With our extensive performance studies, we found HIPMS outperformed PipeHash significantly particular when data skews occur. The outline of HIPMS is given below [9].

- For a n attribute datacube, we divide the cuboid tree into n subtrees where all cuboids in the i-th subtree contain the i-th attribute as its first attribute. All subtrees will be executed one-by-one in order.
- A single hash table for the root of the current cuboid subtree is initially constructed and tuples are hashed into this hash table. During computation, three heuristic strategies are used. All of them aim at increasing memory utilization.
  - **Depth-first:** compute any partition of a cuboid subtree as early as possible and then free its memory space, if it satisfies T-unique property. This strategy will reduce possibilities of swapping in/out pages repeatedly.
  - **In-place:** compute any cuboid using the same memory space used by its ancestor cuboid where possible. This strategy is designed for avoiding CPU and I/O costs.

The condition is that the partition of the cuboid must be T-unique.

- **Memory-shifting:** aggressively shift memory space between partitions at run time. This strategy is designed for handling skews, and allows memory sharing even between different cuboid subtrees. For example, when we have finished computation of a partition of the cuboid $\langle A, B, C, D\rangle$, memory space might temporarily shift to $\langle B, C, D\rangle$, because the more memory $\langle B, C, D\rangle$ has the more $\langle B, C, D\rangle$ groups can be computed on the fly when they are hashed into the hash table of $\langle B, C, D\rangle$. Later on, the memory used by $\langle B, C, D\rangle$ will be shifted back to compute other $\langle A, B, C, D\rangle$ partitions if necessary.

5.2. HIPMS+

HIPMS+ is an algorithm for computing multiple datacubes simultaneously. Because the `Partition-Cube` algorithm is the fastest algorithm to compute sparse datacubes [7], we made two changes in HIPMS.

- **Cuboid tree:** As reported in [7], `Memory-Cube` picks the minimum number of sorts, which is exactly the largest tier in the lattice, $\binom{n}{\lceil n/2 \rceil}$, where n is the number of attributes appearing in a datacube. The cuboid tree for a 4-attribute datacube is shown in Figure 1. In this paper, we construct a cuboid tree using the same `paths` algorithm given in [7].
- **In-place:** We also use the `Memory-Cube` algorithm [7] to compute a cuboid subtree if all its data are in memory. In terms of sorting, the difference is that HIPMS+ also computes group-bys on the fly, when data are hashed into hash tables, whereas `Memory-Cube` computes data after all the data reside in memory.

In addition to the above two changes, we also changed the memory-shifting mechanism. In [9], the aggressive shifting algorithm uses an E-tree to control memory-shifting. When computing a partition of a cuboid in a cuboid subtree, with E-tree, we can identify the hot-spot (the partition that needs memory most), and shift memory pages to that hot-spot. The victim partition to be picked up is done with a traversal order on the E-tree originating from the hot-spot. However, because HIMPS+ needs to compute multiple datacubes simultaneously, it is difficult to find a reasonable traversal order to pick up a victim cuboid. Instead of E-tree, we adopt a LRU strategy as follows. A LRU list keeps all hash tables that keep some tuples in memory. When a partition needs a page in memory and there is no memory pages available, from the LRU list, we pick up a hash table that has not been used recently. Each hash table may keep multiple partitions in memory. Then, we will pick up a page from a partition if that partition has already shifted out some pages on disk or has less number of pages in memory.

While computing datacubes, the memory pages are always shifted to the partition which needs the memory space most with our LRU strategy. We ensure 100% of memory utilization. The outline of our algorithm, HIPMS+, is given in Algorithm 2.

Algorithm 2 The algorithm HIPMS+

Input: m datacubes, $C_1, C_2, \cdots, C_m$, an input relation r, an attribute to be aggregated, and an aggregate function.
Output: the results for multiple datacubes.

```
begin
    buildGraph();
    multiCubeHash();
    computeTopInMemoryPartitions();
    foreach datacube C_j do
        suppose C_j is an n_j attribute datacube or
        incomplete datacube.
        for i = 1 to n_j do
            ComputeCuboidTree(T_i, hashT_i);
        endfor
    endforeach
end
```

The HIPMS+ algorithm is illustrated in Algorithm 2. For a given m datacubes, first, we need to construct a cuboid graph using the procedure `buildGraph()`. Second, we hash data into m hash tables simultaneously. While hashing data into multiple hash tables, the LRU mechanism assists memory shifting. Third, after completion of hashing data to the top m hash tables, we use the in-place strategy to compute all in-memory partitions. Note that in our algorithm a partition grows/shrinks on a memory page basis. When a partition is in memory, we exchange memory pages with other partitions and make a big memory chunk, in order to compute cuboids in place. Finally, we compute a datacube, C_j, individually using a "foreach" statement. Suppose the C_j is a n_j attribute datacube. The n_j attribute datacube is divided into n_j cuboid subtrees. As illustrated in Figure 1, for a 4 attribute datacube, the first cuboid subtree, T_1, consists of all cuboids beginning with A. The second cuboid subtree, T_2, consists of all cuboids beginning with B, and so on. The internal "for" statement is to compute those cuboid subtrees. The procedure ComputeCuboidTree takes the cuboid subtree T_i and a hash table $hashT_i$, and computes cuboids in the subtree T_i. There are two types of partitions: fully memory resident partitions and overflow partitions, i.e., partitions with disk pages. When computing a cuboid, we always processes the fully memory resident partitions one by one using the in-pace strategy, followed by computing the fully or partially disk-resident partitions. For those partitions

we cannot use the in-place strategy, we will hash data into the children cuboids like PipeHash. The difference is that we compute a cuboid subtree in a depth-first partition-basis fashion. When a partition of the root cuboid of T_i is finished, tuples are also hashed into the hash table for the root cuboid of the next cuboid subtree. For example, when we finished computing a partition of $\langle A, B, C, D\rangle$, tuples are hashed into the hash table for $\langle B, C, D\rangle$. The same LRU algorithms will be used.

6. A Performance Study

In this section, we present some results of our extensive performance study. Both MPC and HIPMS+ were implemented using g++ 2.8.1. In this study, we only compare MPC with HIPMS+ for several reasons. First, in our early studies, we found that HIPMS, on which HIPMS+ was implemented, outperformed PipeHash. Second, for sparse datacubes, `Partition-Cube` is the fastest single-cube algorithm on which we design MPC. We did not show our results with IBM DB2, because we cannot accurately measure the CPU time and I/O accesses at the server.

6.1. Data Generation

DG-1 (Testing Sparsities)

We used the approach in [1] to generate synthetic datasets. In brief, each dataset is characterized by four parameters.

- Number of tuples, T.
- Number of grouping attributes, N.
- Ratio amongst the number of distinct values of each attributes $d_1 : d_2 : \cdots : d_N$.
- A parameter, p, denoting the degree of sparsity of the data. It is defined as the ratio of T to the total number of possible attribute value combinations. Thus, if D_i denotes the number of distinct values of attribute A_i, then p is defined as $T/(D_1 \times D_2 \times \cdots \times D_N)$. The smaller the sparsity value is, the higher group skews is (the lower the reduction in the number of tuples after aggregate).

Given these four parameters, a dataset is generated as follows. The total number of D_i is as follows.

$$D_i = \left(\frac{T}{p}\right)^{\frac{1}{N}} \frac{d_i}{(d_1 \times d_2 \times \cdots \times d_N)^{\frac{1}{N}}}$$

Then, for each of the T tuples, a value is chosen for each attribute A_i randomly between 1 and D_i.

DG-2 (Data Generation using Zipf Distributions)

Based on DG-1, we propose a different way to generate data regarding both the groupZipf factor, θ_g, and the tupleZipf factor, θ_t, for a dataset with P partitions. The number of tuples in the i-th partition, T_i, is determined as follows.

$$T_i = \frac{T}{i^{\theta_t} \cdot \sum_{j=1}^{P} \frac{1}{j^{\theta_t}}}$$

In addition, the number of groups in a partition is determined by two sparsity factors, $\hat{p}_1$ and $\hat{p}_2 (> \hat{p}_1)$ in addition to θ_g. Let $\hat{p} = \lceil \log_{10}(\hat{p}_2/\hat{p}_1) \rceil$. The sparsity for the i-th partition, p_i, is determined as $p_i = \hat{p}_1 \cdot 10^{\hat{p}-p'_i}$ where p'_i is given below.

$$p'_i = \frac{k \cdot \hat{p}}{i^{\theta_g} \cdot \sum_{j=1}^{P} \frac{1}{j^{\theta_g}}}$$

where k is the minimum positive number for $p_1 \geq \hat{p}$. For example, let $\hat{p}_1 = 0.0001$ and $\hat{p}_2 = 100$. The sparsities for $P = 6$ are shown in Table 1. Note that the smaller the sparsity value is the greater the group skew is.

θ_g	p_1	p_2	p_3	p_4	p_5	p_6
0.0	0.0001	0.0001	0.0001	0.0001	0.0001	0.0001
0.2	0.0001	0.001	0.01	0.01	0.01	0.01
0.4	0.0001	0.01	0.1	0.1	1.0	1.0
0.6	0.0001	0.1	1.0	10.0	10.0	100.0
0.8	0.0001	0.1	1.0	10.0	10.0	100.0
1.0	0.0001	1.0	100.0	100.0	1000.0	1000.0

Table 1. Sparsities for 6 partitions where $P = 6$, $\hat{p}_1 = 0.0001$ and $\hat{p}_2 = 100$.

As can be seen from Table 1, when groupZipf factor (θ_g) is 0.0, all partitions will generate a large number of groups. When the groupZipf changes to a larger number, the number of groups generated by the datacube will be reduced.

After we have obtained T_i and p_i for the i-the partition, we use the DG-1 approach to generate data for the i-the partition on the condition that all values generated for the i-the partition will be multiplied by P and then plus i. The whole data set is the union of the P partitions.

6.2. System Parameters

These experiments were done on a Sun UltraSPARC-II/400 workstation running Solaris 2.6. The workstation has a total physical memory of 192 MB. Like [7], we did not use a raw file system and we assume a disk transfer rate of 1.5 MB/sec as in [1, 8]. A disk-page is a format page for handling variable length tuples. Each page is 8-Kbytes long in which a header will take 26 bytes. For each tuple in a page, a 8 byte slot is used. The notations and definitions, together with the default values, for all the parameters are summarized in Table 2.

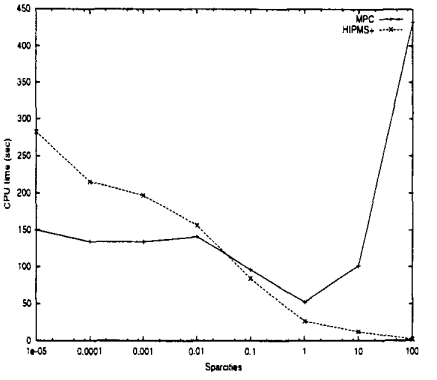
6.3. Exp-1: Testing sparsities

In this experimental study, using DG-1, we generate a 6-attribute relation with 500,000 tuples. The first 5

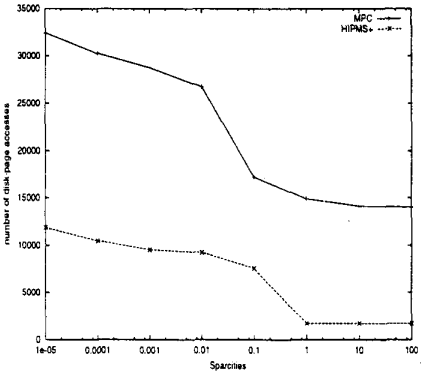
Notation	Definition (Default Values)
N	the number of datacube attributes (5)
T	the number of tuples in a relation (500,000)
M	the memory used for hash tables (5 Mbytes)
θ_g	group-skew by Zipf distribution factor (0.0)
θ_t	tuple-skew by Zipf distribution factor (0.0)
P	the number of partitions used in DG-2 (6)

Table 2. System parameters.

attributes are used as grouping attributes. The ratios are $(d_1, d_2, d_3, d_4, d_5) = (20, 4, 2, 1, 1)$. The sparsity, p, varies from 0.00001 to 100. When the sparsity is small, the number of results of datacube will be big. We run three datacubes simultaneously, $\langle A, B, C, D\rangle$, $\langle A, C, D, E\rangle$ and $\langle A, B, D, E\rangle$. The results are shown in Figure 2. Figure 2(a) shows the CPU time. The CPU time for HIPMS+ decreases while the sparsity increases. It is expected because HIPMS+ computes aggregate functions on-the-fly when data are hashed into the tables. On the other hand, the CPU time for MPU increases while the sparsity increases. It is because MPC cannot partition data well when many tuples have the same patterns. The cross point of the two curves is about $p = 0.01$ where the number of resulting groups is 2,070,860. Figure 2(b) shows the number of 8-KB disk-page accesses. The number of disk-accesses of MPC is about 3 time more than that of HIPMS+. We used different ratios, $(d_1, d_2, d_3, d_4, d_5)$. The results were similar. It suggests that HIPMS+ will outperform MPC when the possibility of group reduction is high. It is worth noting that the possibility of reduction is considerably high when attributes are somehow correlated.



(a) CPU



(b) The number of disk-page accesses

Figure 2. Testing Sparsities.

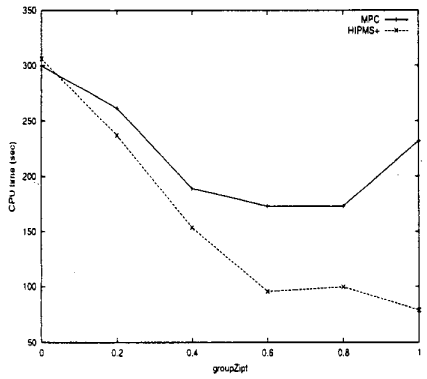
6.4. Exp-2: Sparse Datacubes with Data skews

In this experimental study, using DG-2, we generate a 6-attribute relation with 500,000 tuples. The first 5 attributes are used as grouping attributes. The ratios are $(d_1, d_2, d_3, d_4, d_5) = (20, 4, 2, 1, 1)$. The number of partitions is 6. In HIPMS+, the number of buckets in a hash table is chosen as 6, in order to test data skews. MPC calculates a partition number if it can not fit data in memory. The calculation is done by dividing the size of all tuples by the memory size. In this experimental study, the result of such calculation is 6. The groupZipf information is summarized in Table 1.

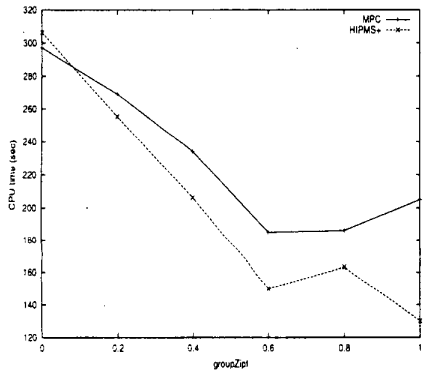
Like Exp-1, we run three datacubes simultaneously, $\langle A, B, C, D\rangle$, $\langle A, C, D, E\rangle$ and $\langle A, B, D, E\rangle$. The results are shown in Figure 3 and Figure 4. In Figure 3, we fixed the tupleZipf (θ_t) and varied the groupZipf (θ_g) from 0.0 to 1.0. Figure 3 (a) shows the combined CPU and I/O times, when $\theta_t = 0.2$. Figure 3 (b) shows the combined CPU and I/O times, when $\theta_t = 1.0$. When $\theta_t = 0.2$, the i-th partition has more tuples than j-th partition if $i < j$. But, the differences are considerably small. However, when $\theta_t = 1.0$, the first few partitions have much more tuples than the other partitions. Note the first partition has the smallest sparsity, 0.0001, and the largest number of tuples (about 40% of tuples). The Figure 3 confirms that memory shifting performs well, in particular, when reduction of groups can be achieved in some partitions. Those small partitions can be computed at early stage. The changes of swapping in/out pages will be reduced. The memory being released can be fully used for other partitions.

In Figure 4, we fixed the groupZipf (θ_g) and varied the tupleZipf (θ_t) from 0.0 to 1.0. Figure 4 (a) shows the combined CPU and I/O times, when $\theta_g = 0.2$. Figure 4 (b) shows the combined CPU and I/O times, when $\theta_g = 1.0$. When $\theta_g = 0.2$, all the sparsities for all the six partitions are in the range between 0.0001 and 0.01. In this range, as also shown in Figure 2, MPC outperforms HIPMS+ in terms of CPU time. However, if we use 1.5MB/sec as data transfer rate, and consider both CPU time and the time for processing I/O accesses, HIPMS+ outperforms MPC. It also suggests that the memory shifting mechanism work well up to a point that can cancel the effectiveness of MPC, a fast sort-based algorithm. On the other hand, when $\theta_g = 1.0$, the sparsities are in a range from 0.0001 to 1000. When $\theta_g = 1.0$, the CPU times for MPC decreases but the

CPU time for HIPMS+ increases while the tupleZipf increases. At the point $\theta_t = 1$, the two CPU times are very closed. It is because that despite the range of sparsities is wider, more tuples will go to the first few partitions that have smaller sparsities, while θ_t increases. Recall when $\theta_t = 1.0$, over 40% of tuples are in the first partition which has the smallest sparsity 0.0001.



(a) The combined CPU and I/O (1.5MB/sec) ($\theta_t = 0.2$)

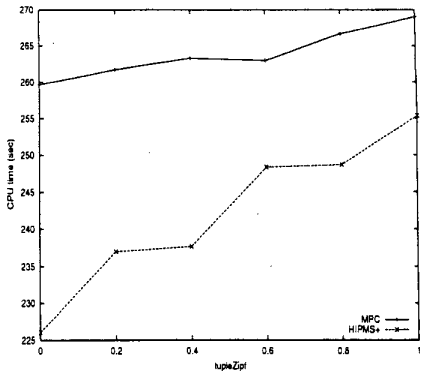


(b) The combined CPU and I/O (1.5MB/sec) ($\theta_t = 1.0$)

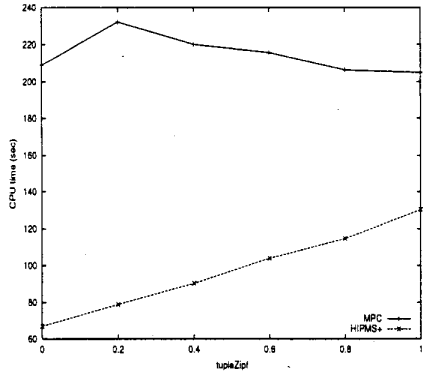
Figure 3. Testing group skews.

6.5. Exp-3: Case Studies

In this section, we study several different cases using uniform distributions. We assume that data values are randomly generated in all attributes. In Figure 5(a), we run three 6-attribute datacubes for a 10-attribute relation. The last attribute is used as the measure. For the case-A, all values in the first 9 attributes are in the range between 0 and 100. For the case-B and case-C, values in the first three attributes are in the range between 0 and 1000. Values in the next three attributes are in the range of 0 and 100. Values in the next three attributes are between 0 and 10. The three 6-attribute datacubes for the case-A are $\langle A, B, C, D, E, F \rangle$,



(a) The combined CPU and I/O (1.5MB/sec) ($\theta_g = 0.2$)



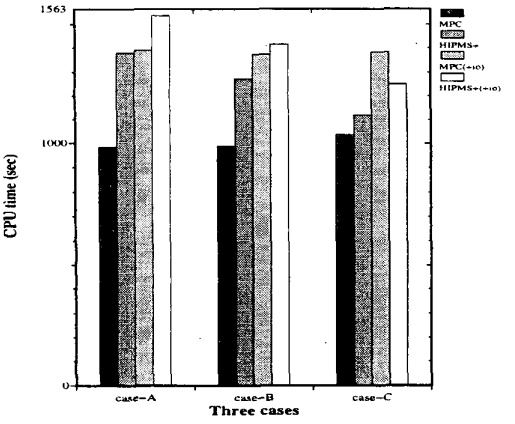
(b) The combined CPU and I/O (1.5MB/sec) ($\theta_g = 1.0$)

Figure 4. Testing tuple skews.

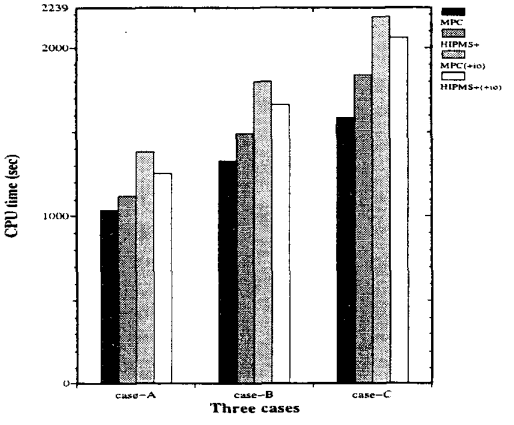
$\langle A, C, D, E, F, G \rangle$ and $\langle A, D, E, F, G, H \rangle$. The three 6-attribute datacubes for the case-B and case-C are $\langle A, B, C, D, E, F \rangle$, $\langle A, C, D, E, F, G \rangle$ and $\langle A, D, E, F, G, H \rangle$, and $\langle A, B, C, G, H, I \rangle$, $\langle B, C, D, E, G, H \rangle$ and $\langle A, D, F, G, H, I \rangle$, respectively. For each case, the first and the second bar is CPU time for MPC and HIPMS+, respectively. The third and the fourth bar is combined CPU time and I/O time (1.5MB/s) for MPC and HIPMS+, respectively.

As shown in Figure 5 (a), in terms of pure user CPU time, MPC outperforms HIPMS+ in all the three cases. However, the I/O costs are a concern. The three cases show, with combined CPU time and I/O time, three different cases. For the case-A, MPC outperforms HIPMS+. For the case-B, both perform in a similar way. For the case-C, HIPMS+ outperforms MPC. All the results strongly rely on the datacubes being issued and the data.

In Figure 5 (b), we show another three different cases for a 10-attribute relation with 500,000 tuples. Values in the first three attributes are in the range between 0 and 1000. Values in the next three attributes



(a) Three 6-attribute cubes.



(b) Increasing the number of datacubes

Figure 5. Case studies

are in the range of 0 and 100. Values in the next three attributes are between 0 and 10. The last attribute is of the measure. The case-A runs three 6-attribute datacubes, namely, $\langle A, B, C, G, H, I\rangle$, $\langle B, C, D, E, G, H\rangle$, $\langle A, D, F, G, H, I\rangle$. The case-B adds another 6-attribute datacube to the case-A, $\langle B, C, D, E, F, G\rangle$. The case-C adds another 6-attribute datacube to the case-B, $\langle C, D, E, F, H, I\rangle$. Even though, from Figure 5(b), it shows that HIPMS+ outperforms MPC when the combined CPU time and I/O time is used, we cannot easily conclude that it is always the case. We plan to further study the effectiveness of the two algorithms in the future.

7. Conclusion

Datacube computation is an expensive computation. One of the key issue is to make efficient use of memory available to compute a large number of cuboids. In this paper, we explore two algorithms for computing multiple datacubes, namely, MPC and HIPMS+. Both algorithms adopt the divide-and-conquer strategy. MPC assumes the absence of data skews, while HIPMS does not. MPC partitions data evenly, and computes the memory-resident partition efficiently. In other words, MPC can only compute a cuboid subtree after all the data are in memory. HIPMS+ computes the aggregate function on the fly when data are hashed into hash tables. Also, HIPMS+ uses a unique memory shifting mechanism. We conducted extensive performance studies, and showed some of the results in this paper. When the sparsity is small, HIPMS+ outperforms MPC, even with data skews. However, as shown in other experimental results, at this stage, it is difficult for us to conclude that one definitely outperforms the other. It heavily relies on the datacubes and the data distribution. As our future work, we are planning to further study the two algorithms, and investigate adaptive algorithms to combine the two algorithms.

Acknowledgment

The work described in this paper was substantially supported by the Direct Grant for Research, CUHK (Project No. 2050256).

References

[1] S. Agarwal, R. Agrawal, P. M. Deshpande, A. Gupta, J. F. Naughton, R. Ramakrishnan, and S. Sarawagi. On the computation of multidimensional aggregates. In *Proceedings of the 22nd International Conference on Very Large Data Bases*, January 1996.

[2] D. Chamberlin. *A Complete Guide To DB2 Universal Database*. Morgan Kaufmann, 1998.

[3] M. Corp. *OLE DB for OLAP Design Specification*. http://www/microsoft.com/data/oledb/olap.

[4] P. Deshpande and et al. Computation of multidimensional aggregates. Technical Report Technical Report 1314, University of Wisconsin-Madison, 1996.

[5] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. In *Proceedings of the 12th International Conference on Data Engineering*, pages 152–159, 1996.

[6] W. Liang, M. E. Orlowska, and J. X. Yu. Optimizing multiple dimensional queries simultaneously in multidimensional databases. *VLDB Journal*, 8(4), 2000.

[7] K. A. Ross and D. Srivastava. Fast computation of sparse datacubes. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, pages 116–125, August 1997.

[8] S. Sarawagi, R. Agrawal, and A. Gupta. On computing the data cube. Technical Report Research Report RJ 10026, IBM Almaden Research Center, 1996.

[9] J. X. Yu and H. Lu. Hash in place with memory shifting: Datacube computatino revisited. In *ICDE'99*, 1999.

[10] Y. Zhao, P. Deshpande, J. Naughton, and A. Shukla. Simultaneous optimization and evaluation of multiple dimensional queries. In *Prof. of the 1998 ACM-SIGMOD*, 1998.

[11] Y. Zhao, P. M. Deshpande, and J. F. Naughton. An array-based algorithm for simultaneous multidimensional aggregations. In *Proceedings of the 1997 ACM-SIGMOD International Conference on Management of Data*, pages 159–170, June 1997.