Simultaneous Optimization and Evaluation of Multiple Dimensional Aggregate Queries On Shared-Nothing Multiprocessors

Weifa Liang† Maria E. Orlowska† Hong Shen‡

† Distributed Systems Technology Centre
Dept. of Computer Science
University of Queensland
St. Lucia, QLD 4072
Email: {wliang, maria}@csee.uq.edu.au

‡ School of Computing and Information Technology
Griffith University
Nathan, QLD 4111
Email: hong@cit.gu.edu.au

Abstract

Some significant progress related to multidimensional data analysis has been achieved in the past several years. These include designing fast algorithms for computing datacubes, selecting some precomputed group-bys to materialize, and designing efficient storage structures for multidimensional data. However, little work has been carried out on the multidimensional query optimization issues. As well understood, the response time for answering such queries is very crucial to the performance of OLAP. In doing so, some group-by aggregates in a datacube have been precomputed and stored in disks when it is impossible to store the entire datacube due to limited storage space. Informal speaking, the problem can be described as follows. Given a set of precomputed group-by aggregates $MV = \{mv_1, mv_2, \ldots, mv_s\}$ and a dimensional expression Q (e.g. an MDX in Microsoft's OLE) which can be decomposed into several related subqueries $q_1, q_2, \ldots, q_n$, assume that Q can be answered using the data from MV. The problem is to answer Q with minimizing the response time. Recently, Zhao et al first exploited this problem by introducing a general framework in terms of the evaluation cost (time) and suggesting three heuristic algorithms for finding a better global plan according to the evaluation cost. Following their work, Liang et al re-examined the problem by giving improved results. It must be mentioned that both these algorithms are running in sequential machines.

Since the data involved in OLAP applications is very large, applying parallel processing techniques to speed-up the queries in such applications is inevitable. In this paper we consider how to solve the above problem on a shared-nothing multiprocessor system in which each processor has its own local disk and all processors are connected through a regular-speed/high-speed interconnection network, but no shared memory exists among the processors. Such systems have been existent for many years, for example, clusters of workstations and IBM SP-2 series machines. Based on this computational model, in this paper we devise several heuristic parallel algorithms for optimizing and evaluating multiple dimensional queries simultaneously in terms of the evaluation cost. In our algorithms, we consider how to share the common tasks among the subqueries and to utilize the indexes of tables to reduce the cost. We also consider the trade-off between the communication cost and the computation cost for solving the problem. To the best of our knowledge, the proposed algorithms are the first algorithms for the problem on this computational model.

Keywords: *multiple dimensional query optimization, OLAP, MDDBs, aggregation computation, data warehousing*

1 Introduction

In the past few years, the researchers in database communities have made great progress on the On-Line Analytical Processing (OLAP) issues such as efficient algorithms for computing datacubes [GBLP95, AAD+96, SAG96, DANR96, RS97, ZDN97], techniques for selecting group-bys to materialize [HRU96, Gupta97, GHRU97, YKL97, BPT97], and efficient approaches for incremental maintenance to materialized views [GM95, ZGHW95, AESY97, HZ96, RSS96]. However, there has been little attention devoted to simultaneously optimize and evaluate multiple dimensional aggregate queries. This kind of optimization and evaluation is different from the traditional multiple query optimization [PS88, S88, SS94]. That is, the dimensional aggregate queries involve data aggregation across different dimensions and hierarchies, and the amount of data manipulated is very large, e.g., gigabytes or terabytes of data. In addition to the intrinsic interest of the problem, to optimize and evaluate multiple dimensional aggregate queries simultaneously is likely to be of practical importance due to recent developments in OLAP. For example, Microsoft recently released its proposed "OLE DB for OLAP" standard for interfaces to dimensional data sources [MS], which is likely to become widely supported. One of the most interesting aspects of this standard is the definition of *Multi-Dimensional Expressions* (MDX) which provides a framework that a user can ask several related OLAP queries in a single MDX query. This aspect of MDX reflects the fact that OLAP style analysis often gives rise to simultaneous related queries, and MDX intends to provide a uniform framework for a variety of data sources. Since a single MDX can be decomposed into several related OLAP queries, obviously if all the queries can be evaluated effectively, the MDX query can then be answered efficiently.

In this paper we consider the optimization and evaluation of an MDX in terms of the evaluation cost (time). Recently Zhao et al [ZDNS98] first exploited this problem by giving a general framework and introducing three shared operators: `shared scan for hash-based star join`, `shared index-based join`, and `shared scan for hash-based and index-based star join`. They also suggested three heuristic algorithms for finding a better global plan according to the evaluation cost. Following their work, Liang and Orlowska [LO98] re-examined the problem by giving improved results, It must be mentioned that both these algorithms are running in sequential machines.

Since the data involved in OLAP applications is very large, applying parallel processing techniques to speed-up the queries in such applications is inevitable. In this paper we consider how to solve the above problem on a shared-nothing multiprocessor system in which each processor has its own local disk and all processors are connected through a regular-speed/high-speed interconnection network, but no shared memory exists among the processors. Such systems have been existent for many years, for example, clusters of workstations and IBM SP-2 series machines. Based on this computational model. We devise several heuristic parallel algorithms for optimizing and evaluating multiple dimensional aggregate queries simultaneously in terms of the evaluation cost, assume that the aggregation functions are distributive. In our algorithms, we consider how to share the common tasks among the subqueries and to utilize the indexes of tables to reduce the cost. We also consider the trade-off between the communication cost and the computation cost for solving the problem. To the best of our knowledge, the proposed algorithms are probably the first algorithms for the problem on this computational model.

The remaining of the paper is organized as follows. In Section 2 we introduce the computational model, some basic concepts, and the three shared operators introduced by Zhao et al [ZDNS98]. In Section 3 we review some previously related results to our problem. In Section 4 we first model the problem in terms of the evaluation cost, then present a greedy algorithm for finding a better global plan, and finally give several parallel algorithms to evaluate the plan efficiently on a shard-nothing multiprocessor system. We conclude our discussions and propose some future work along this direction in Section 5.

2 Preliminaries

2.1 Basic notions

A Multidimensional Database (MDDB) is a data repository that provides an integrated environment for decision-support queries that require complex aggregations on very large amounts of historic data. Here we assume that an MDDB is a relational data warehouse in which information is organized as *a star schema* [Kim96]. In other words, an MDDB consists of a single *fact table* $F(D_1, D_2, \ldots, D_n, M_1, M_2, \ldots, M_r)$ and a single table for each dimension D_i. Each *dimension table* D_i contains all the information that is specific only to the dimension itself, while the fact table F correlates all dimensions and contains information on the attributes of interest for intersection of all the dimensions, $1 \le i \le n$. Each M_j is called a *measure* of F, $1 \le j \le r$. Each tuple in F consists of a pointer (foreign key) to each of the dimensions that provides its dimensional coordinations, and stores the numeric measures for those coordinations.

Let $F(D_1, D_2, \ldots, D_n, M)$ be a fact table where D_i is a dimension of F, $1 \le i \le n$, and M is the measure. There are 2^n possibly combinations of the dimensions. The *data cube* of F is to compute all dimensional group-by aggregations. In order to achieve the minimum response time for the cube computation, the objective is to minimize the total of operations for the computation. [HRU96] introduced a *search lattice* $\mathcal{L} = (V, E)$ for such computation, where each group-by is a node in V and there is an directed edge $\langle u, v \rangle \in E$ from group-by u to group-by v if v can generated from u and v has exactly one dimension less than u, the weight associated with the edge is the operation cost (usually this is an estimation value by sampling).

2.2 An MDX example

In order to illustrate the MDX concept, we give the following example, appeared in [MS], which was also used by [ZDNS98]. We assume that the database consists of one fact table `SalesCube` and five dimension tables `Salesperson, Geography, Quarters, Products, Years`, and one `Measure` that is `Sales`, where `Geography` is a hierarchy that has the levels states, countries, continents, and so on. `Quarters` is also a hierarchy that has the levels quarters, months, days. Then, an MDX is defined as follows.

```
NEST ({Venkatrao, Netz}, {USA_North.CHILDREN, USA_South, Japan})
on COLUMNS {Qtr1.CHILDREN, Qtr2, Qtr3, Qtr4.CHILDREN}
on ROWS
CONTEXT SalesCube
FILTER (Sales, [1991], Products.All)
```

where `NEST, CONTEXT, COLUMNS, CHILDREN, and FILTER` are all reserved MDX keywords. This MDX specifies the total sales for salesmen Venkatrao and Netz in all states of USA_North, USA_South, and Japan for all the months of the first quarter, the second quarter, the third quarter, and all the months of the fourth quarter in 1991.

In the relational terms, the join between the fact table and dimension tables are defined implicitly in an MDX expression. Therefore the join attributes and join conditions are not visible in the MDX expression. This is due to the fact that an MDX expression does not define the way in which an OLAP server should process the MDX query.

The above MDX can be decomposed into the following six different group-by queries in terms of SQL statements:

1. the total sales for Venkatrao and Netz in all states of USA_North for the 2nd and 3rd quarters in 1991;
2. the total sales for Venkatrao and Netz in all states of USA_North for the months of the 1st and 4th quarters in 1991;
3. the total sales for Venkatrao and Netz in region USA_South for the 2nd and 3rd quarters in 1991;;
4. the total sales for Venkatrao and Netz in region USA_South for the months of the 1st and 4th quarters in 1991;
5. the total sales for Venkatrao and Netz in Country Japan for the 2nd and 3rd quarters in 1991;
6. the total sales for Venkatrao and Netz in Country Japan for the months of the 1st and 4th quarters in 1991.

More succinctly, we have six group-bys (Salesperson, State, Quarter), (Salesperson, State, Month), (Salesperson, Region, Quarter), (Salesperson, Region, Month), (Salesperson, Country, Quarter), and (Salesperson, Country, Month). In addition, we have disjoint selection predicates for each group-by. This means that we cannot use the finding the common selection predicate techniques which are widely used in multiple query optimization for general SQL queries [S88]. Therefore, from now on, unless otherwise specified we assume that an MDX has been decomposed into a set Q of queries. We will develop algorithms to reduce the total response time for answering all queries in Q.

2.3 Shared Operators

As well understood, the fundamental task of evaluating multiple related dimensional aggregate queries is identifying and exploiting common subtasks among the queries. In star join environment, we try to make the queries share star joins. Such joins are either hash-based star join or index-based star join. For very selective star join queries, the index-based star join is a good alternative [OQ97]. For non-selective star join queries, the hash-based star join is a good solution [Su96]. Based on the star join environment, Zhao et al [ZDNS98] introduce the following three shared operators: `shared scan for hash-based star join`, `shared index-based join`, and `shared scan for hash-based and index-based star join`. Here we briefly introduce these operators which will be used later. We refer to any materialized group-by (node) in the search lattice as a *base table*. The fact table is a base table. Let q be a query and q can be answered using the base table R. Then a hash-based star join for q involves building a hash table on each dimension table, and then probing these hash tables with the tuples of the base table R. we assume that the aggregation afterwards is also done by hashing.

The `shared scan for hash-based star join` is defined as follows. Suppose we have two different queries q_1 and q_2 sharing the same base table. Then q_1 and q_2 can share scanning the base table clearly. Furthermore, if q_1 and q_2 use the same dimension tables, then they can share the hash tables too, instead of redundantly building and probing the hash tables on the same dimension tables.

The `shared scan for index-based star join` is defined as follows. Assume that each dimension table has built a bitmap join indexes mapping the join attributes to the tuples of a base table F'. Now we consider two queries q_1 and q_2 which share the base table F' and both use the bitmaps $BT(q_1, F')$ and $BT(q_2, F')$. The `index-based star join` proceeds as follows. Form a combined bitmap $BM := BT(q_1, F') \vee BT(q_2, F')$, use BM to probe the base table F' to extract the matching tuples. Let F'' be the set of matching tuples in F'. Then each query q_i uses its own bitmap $BT(q_i, F')$ to filter F'' to get its own matching tuples. Finally each q_i do aggregation by hashing, $i = 1, 2$.

The `shared scan for hash-based and index-based star join` operator is an operator which is a mixture of the first two operators. Suppose that a set of queries share scanning the same base table. Among these queries, some of them are hash-based and the remaining are index-based star join. Those hash-based queries also share hash-based star join. For those index-based queries, we first use their combined bitmap to extract a set S of matching tuples from the base table, we then use the bitmap of each query to filter the matching tuples from S. We finally send these matching tuples to the corresponding group-by hash table to do aggregation.

2.4 The computational model

A shared-nothing multiprocessor system consists of P identical processors. Each processor has its own local disk, and there is no shared memory among the processors but all processors are connected through a regular or high-speed interconnection network. Such systems (for example the cluster of workstations, IBM SP-2) have been existent for many years. In the following we make some assumptions regarding the partitioning and storage of materialized views (relational tables) on this machine. We assume that

- every relational table is fragmented horizontally and stored in the local disks of the processors, i.e., each processor holds one pth of the size of the table approximately;
- the bitmap associated with a table is also fragmented horizontally and stored in the local disks of the processors, and each processor only holds the fragment part of the bitmap which is the index of the fragment of the table allocated to that processor;
- each dimension table is available for every processor. We argue that this assumption is reasonable because the size of each of its dimension tables is ignorable, compared with the size of a fact table.

3 Previous Results for Aggregate Queries

In this section we briefly introduce some previous algorithms which deal with a single aggregate query and multiple aggregate queries that share the same base table. We will make use of them as subroutines in some of our algorithms later.

3.1 Algorithms for a single aggregation query

Let R be a relational table. The underlying uniprocessor hash-based algorithm for group-by aggregation on R works as follows:

1. the tuples in R are read and a hash table is built by hashing group-by attribute values of tuples. The first tuple hashing to a new value adds an entry to the hash table, and the subsequent tuples matching an existing entry in the hash table will update the accumulated result as appropriate. It is easy to see the memory requirement for the hash table is proportional to the number of distinct groups seen;
2. if the entire hash table is unable to fit in the allocated memory, the tuples are hash partitioned into multiple buckets and all but the first bucket are residents in disk.
3. The overflow buckets are processed one by one as in Step 1 above.

Based on the above framework, there are two well known parallel algorithms for computing a single aggregation query on the parallel and distributed environment. Recall that the tuples in R have been distributed to every processor equally, i.e, each processor holds a fragment of R which is usually a resident of the local disk of that processor. The aggregation on a processor is performed by hashing.

One is `Two Phase Algorithm` which consists of two phases. In the first phase, each processor in the system computes aggregates on its local fragment of R. In the second phase the partial results of all processors are collected to one of the processors that merges them to produce the final results. Certainly the second phase can be parallelized by hash-partitioning on the `group-by` attributes.

Another is `Repartitioning Algorithm` which partitions the tuples in R into groups by the group-by attribute values, and redistributes the tuples to one of the processors according to their group-by attribute values. Then, each processor performs aggregation on the partitioning data allocated to it in parallel.

As explained in [BBDW83, SN95], both algorithms above have some advantages and drawbacks. For example, `Two Phase Algorithm` may suffer the following two problems when the grouping selectivity is high. First, since the group-by attribute value may be accumulated on every processor, the memory requirement for every processor can be as large as the overall result, which may not fit into the main memory. Second, the duplication of aggregation work in the first phase becomes significant. As for `Repartitioning Algorithm`, it is efficient when the grouping selectivity is high, because it eliminates the duplications by processing each distinct group-by attribute value once. It also reduces the memory requirement, since each group-by value is stored in only a processor. It thus works well for large number of groups. However, when the grouping selectivity is so low that the number of groups is smaller than the number of processors, `Repartitioning Algorithm` cannot fully utilize the all processors, hence it severely affects the entire performance of queries.

In order to overcome the drawbacks of both above algorithms, [SN95] presented two hybrid parallel algorithms by combining the above two algorithms together. That is, each processor runs one of the two algorithms from the very beginning, then it changes to run another automatically when it reaches a threshold. This threshold can be checked during computing aggregation. Their detailed algorithms can be seen in [SN95].

Finally, we introduce a `Broadcasting Algorithm` due to [BBDW83], which broadcasts all local disk pages of R to all processors, and every processor computes aggregations for the groups assigned to it when it receives all tuples in R. Since each group-by attribute value is stored in only one processor, the memory requirement of this algorithm is the same as that of `Repartitioning Algorithm`. But, this algorithm also suffers a severe drawback when the system does not support broadcast efficiently since the amount of data passed through the network is much larger than that of `Repartitioning Algorithm`.

3.2 Algorithms for multiple aggregation queries sharing a base table

Recently, [TM98] extend the work of [SN95] further by considering the following problem. Assume that there is a collection of aggregate queries Q on a data cube, and all the queries share the same materialized view (base table, or source data), which means the queries can be answered using that materialized view. For this problem, they proposed three parallel algorithms: `m2P`, `mRep` and `mBC`, based on the above results of [BBDW83] and [SN95]. For the sake of integrity, we briefly sketch their algorithms, which will be used later, as follows.

The `m2P` algorithm processes $|Q|$ queries simultaneously. In the first phase each processor reads its local fragment and computes aggregation of $|Q|$ queries. The second phase merges the partial results in parallel. Since only a single scan (from the disks) of source data is required, the scan cost is minimum. However, the memory requirement is multiplied by $|Q|$, and `m2P` requires as much main memory for each processor as the overall result size. Therefore, memory overflow is liable to happen, and extra I/Os is needed for bucketing hash-tables (here each hash table corresponds to a query in Q, there are $|Q|$ hash tables in total). In order to tackle this case, [TM98] suggest the following approach. They divide the $|Q|$ queries into K groups. Each time the system only processes the queries in a group simultaneously. Thus, the scan cost is increased by K times, but the extra I/Os cost will be reduced dramatically. In order to achieve the best performance, the choice of K is important, which actually reflects a trade-off between the extra I/Os cost and the scan cost.

The `mReP` algorithm processes $|Q|$ queries simultaneously, using `Repartitioning Algorithm` for each individual query. It seems to be suitable for multiple queries, since multiple queries require a lot of main memory (for hash tables) and `Repartitioning Algorithm` uses main memory efficiently. However, the source data is needed to be partitioned independently for each query because each query may have different group-by attributes. Therefore, in order to process $|Q|$ queries simultaneously, each processor needs to read a tuple, compute the groups that the tuple belongs to among the $|Q|$ queries, and send the tuple to the corresponding processor. The communication cost of `mRep` is $|Q|$ times than that of `Repartitioning Algorithm`. Certainly one of variant of this algorithm is to divide the queries into K groups, and apply the `mReP` algorithm for each group independently.

The `mBC` algorithm processes $|Q|$ queries simultaneously using the `Broadcasting Algorithm` for each query. As understood, for a single query this algorithm is slower than `m2P` and `mRep` because its communications cost is so high. However, the communication cost of `mBC` is the same as a single query, while the communication costs of `m2P` and `mRep` are proportional to $|Q|$. Therefore, `mBC` will outperform both `m2P` and `mRep` when there are large multiple queries.

Remarks: It must be mentioned that most people still consider that broadcasting is usually much more expensive than simple point-to-point communication in a multiprocessor system. But, with advent of recent optical techniques, this may not be true in the near future.

4 Optimizing and Evaluating Multiple Dimensional Queries Simultaneously

We turn to model our problem in terms of the evaluation cost. Let $\mathcal{L} = (V, E)$ be the lattice of a fact table F. Assume that some group-by aggregates in $\mathcal{L}$ have been precomputed (materialized). Let MV be the set of the precomputed group-by aggregates in $\mathcal{L}$. We further assume that F is also materialized, therefore, it is always included in MV. Let Q be the set of aggregate queries which is the result by decomposing an MDX and the aggregation function in every aggregate query is distributive. *The Multiple Dimensional Aggregate Query Optimization and Evaluation Problem* is to find a materialized view from MV for every query $q \in Q$ and to evaluate q using that view such that (i) all $q \in Q$ have been answered; and (ii) the total response time for answering all queries in Q is minimum. Without loss of generality, assume that every materialized view in MV is *useful* with respect to the queries in Q, which means, at least one query $q \in Q$ can be answered from it for every view in MV. Otherwise, those useless materialized views can be removed from MV for further consideration.

In the sequential environment, a few efficient approaches to solve this problem [ZDNS98, LO98] have been introduced. However, in the parallel and distributed environment, to solve this problem seems much more harder, because both the computation cost including disk I/O cost and the communication cost among processors are needed to be taken into account. As well understood in the parallel community, to solve any non-trivial problem in a multiprocessor system, there always involves computation cost and communication cost, and the computation and communication operations are usually interleaving. Therefore, measuring these two different kinds of costs is very hard. Actually, there always exist a trade-off between them. Now let us consider the above problem in the multiprocessor system. Assume that there is an optimal solution T for it in this system. Clearly,

$$T = T_{comp} + T_{comm} \tag{1}$$

where T_{comp} is the total computation cost by a processor involved (I/O cost, scan cost, and hash cost) and T_{comm} is the total communication cost among the processors.

Given MV and Q, *the global plan* for the queries $q \in Q$ is an assignment scheduling which assigns every $q \in Q$ with a materialized view $mv \in MV$ such that q can be answered using mv. We say that a global plan $\mathcal{A}$ is *better* than a global plan $\mathcal{B}$ if and only if the time for evaluating $\mathcal{A}$ is less than that for evaluating $\mathcal{B}$.

Assume that there is an algorithm which delivers a global plan $\mathcal{A}$. Evaluating $\mathcal{A}$ on the multiprocessor system by the algorithm takes T'_{comp} computation cost and T'_{comm} communication cost. Thus, our objective is to find such an algorithm which generates a plan $\mathcal{A}$ and minimizes $T'_{comp}+T'_{comm}$. As discussed above, this is a very difficult task. Instead, we give a feasible solution for it. Our algorithm basically consists of two phases. In the first phase, we find a "better" global plan such that the cost of evaluating the plan is minimal in terms of the computation cost only. In the second phase, we evaluate the plan to make the communication cost as small as possible.

4.1 Find a better global plan

Now we deal with finding a "better" global plan in the first phase. In this simplified version the communication cost is ignored and does not been taken into consideration. Let $\mathcal{A}'$ be such an optimal plan for the simplified version with the cost $T_{\mathcal{A}'}$. It is easy to see that $T_{\mathcal{A}'} \leq T_{comp}$. But it is easy to show that the communication cost of $\mathcal{A}'$ is no less than T_{comm}. Even for this simplied version, the optimal solution is hard to be found. So, in the following we present a heuristic algorithm to generate a better global plan.

In order to reduce the computation cost, we must make as many queries as possible to share common tasks such as scan-shared hash/index based star-join operation and the materialized views, and to utilize the indexes built for the materialized views in MV to speed-up the total response time. Since we assume that every processor in the multiprocessor system is identical, so, we only consider the time spent at one processor for evaluating an MDX. We formalize this simplified version of the problem at every processor as an optimization problem in terms of the computation cost including CPU computation and I/O cost. Let

$$T_{qucpu} = \sum_{i=1,\ldots,|Q|,j=1,\ldots,|MV|} t'_{ij}x_{ij} \tag{2}$$

be the total response time involving CPU computation, assuming that all materialized views needed are in main memory already.

Associated with every query $q_i \in Q$, there are $|MV|$ variables $x_{i1}, x_{i2}, \ldots, x_{i|MV|}$ in which there is a unique $x_{ij} = 1$, and all other $x_{ij'} = 0$ if $j' \neq j$ for any j, $1 \leq j \leq |MV|$, i.e., q_i chooses mv_j as its base table. For these variables x_{ij} with $1 \leq j \leq |MV|$, there are r_i different assignments, i.e, $(0, 0, \ldots, 0, 1, 0, \ldots, 0)$ with $x_{ij} = 1$ is a valid assignment if q_i uses mv_j as its base table, where r_i is the number of materialized views from which q_i can be answered. Now we explain the variable t'_{ij} which is the time for answering q_i, defined as follows.

If q_i is index-based join with mv_j, then

$$t'_{i,j} = \sigma_{ij} \cdot Size(mv_j)/P \cdot t_{cpu}(q_i, mv_j) \tag{3}$$

where $Size(v)$ is the number of tuples of table v, σ_{ij} is the selectivity. Let F_{ij} be the bitmap of q_i to mv_j and $|F_{ij}|$ the number of 1s in F_{ij}, then $\sigma_{ij} = \frac{|F_{ij}|}{Size(mv_j)}$, i.e, $0 < \sigma_{ij} < 1$. $t_{cpu}(q, u)$ is the CPU time needed to match the predicates in query q to a tuple in u and to do hashing on the group-by attributes for aggregation.

Otherwise (hash-based join),

$$t'_{i,j} = Size(mv_j)/P \cdot t_{cpu}(q_i, mv_j). \tag{4}$$

Let $M' = \{mv_j \mid \text{if there is } x_{ij} = 1 \text{ in expression 2 }\}$. We now calculate the scan cost $C_{load}(mv_j)$ for every $mv_j \in M'$. Assume that $q_{i_1}, q_{i_2}, \ldots, q_{i_k}$ are associated with mv_j, i.e, $x_{i_l j} = 1$ with $1 \leq l \leq k$. Then

(i) if all q_{i_l} are hash-based join, then

$$C_{load}(mv_j) = \frac{Size(mv_j)}{P \cdot Page\_Size} \cdot t_{I/O} + t_{seek} + t_{hash-join}(mv_j)/P \tag{5}$$

where $Page\_Size$ is the size of the unit block (e.g. 4K bytes) between memory and the disk, $t_{I/O}$ is the transfer time for a single block, t_{seek} is the time for initial seeking v's location on disk, and $t_{hash-join}(v)$ is the sum of the star-join time of the table v with its dimension tables and the time for generating the hash tables for the dimensions involved, assume that the dimension tables are stored in each processor.

(ii) if all q_{i_l} are index-based join, then

$$C_{load}(mv_j) \quad = \quad \sigma_j \cdot \frac{Size(mv_j)}{P \cdot Page\_Size} \cdot t_{I/O} + \sum_{l=1}^{k} t_{index}(q_{il}, mv_j)/P \tag{6}$$

where $t_{index}(q, v)$ is searching a bitmap time and the bitmap was built for q with respect to v, and σ_j is defined as follows.

Let R_{il} be the set of the tuples in mv_j satisfying the query q_{i_l}. Then, $|R_{il}| = \sigma_{il} \cdot Size(mv_j)$. Then the union R' of the sets by the all queries using mv_j will be moved to memory, i.e, $R' = \cup_{i=1}^{k} R_{il}$. Therefore

$$\sigma_j \quad = \quad \frac{|R'|}{Size(mv_j)} \tag{7}$$

Remarks. In practice before running the query, we do not know R_{il}, so, there is no way to give an exact value for σ_j. Instead we assign σ_j an approximate value. Clearly, $\max\{\sigma_{lj} \mid 1 \leq l \leq k\} \leq \sigma_j \leq \sum_{l=1}^{k} \sigma_{lj}$.

$$\sigma_j \quad \approx \quad \frac{\sigma_{i1} + \sigma_{i2} + \ldots + \sigma_{ik}}{k} \tag{8}$$

(iii) otherwise, assume that the k queries are associated with mv_j in which the first r queries $q_{i_1}, q_{i_2}, \ldots, q_{i_r}$ are index-based join and the remaining $k-r$ queries $q_{i_{r+1}}, \ldots, q_{i_k}$ are hash-based join. Let $R'' = \cup_{l=1}^{r} R_{il}$. Then

$$C_{load}(mv_j) = (1 + \sigma'_j) \cdot \frac{Size(mv_j)}{P \cdot Page\_Size} \cdot t_{I/O} + t_{seek} + \sum_{l=1}^{r} t_{index}(q_{il}, mv_j)/P \tag{9}$$

and

$$\sigma'_j \quad = \quad \frac{|R''|}{Size(mv_j)} \tag{10}$$

Therefore, our objective now is to minimize

$$\sum_{mv \in M'} C_{load}(mv) + T_{qucpu} \tag{11}$$

The minimum value of expression 11 is a lower bound of the optimal cost (the minimum response time) for answering all queries in Q in a multiprocessor system because the communication cost among the processors does not been taken into account. Even so, it seems too hard to find an exact solution for it in polynomial time.

In the sequential environment, [ZDNS98] present three greedy algorithms for finding a feasible solution by using so called *GroupbyLevel* as the guideline, whereas *GroupbyLevel* does not carry much information about the evaluation cost as recently shown by an example in [LO98]. [LO98] also present a better algorithm for this simplified version. Unlike Zhao et al's algorithm, [LO98]'s algorithm does not set the order for queries to be added. General speaking, [LO98]'s algorithm proceeds by adding the queries one by one to the global plan. So, their algorithm takes $|Q|$ times iterations. Suppose the first $i-1$ queries have been assigned to their base tables. Let AQ be the set of queries which have not been assigned the base tables. In the very beginning of the i-th iteration, [LO98]'s algorithm does not know which query in AQ to be added. It chooses a query which will lead to the minimum cost increase. Here we present an algorithm for our purpose through a minor modification to [LO98]'s algorithm in a uniprocessor as follows.

Algorithm 1:
$MV := \{mv_1, mv_2, \ldots, mv_{|MV|}\}$;
$Q := \{q_1, q_2, \ldots, q_{|Q|}\}$;
$AQ := Q$;/* the set of queries unassigned yet */
$Cost := 0$; /* the total cost for all queries */
for $i := 1$ **to** $|MV|$ **do**
    $IX(mv_i) := \emptyset$; /* the set of index-based queries used mv_i as the base table */
endfor;
repeat
    $\Delta\_cost := \infty$; /* the cost incurred by adding the i-th query to the solution */
    $q\_candidate := \emptyset$; /* the i-th query will be q_candidate */
    for every $q_x \in AQ$ **do**
        for every $mv_y \in MV$ **do**
            if q_x can be answered from mv_y **then**
                if mv_y is chosen in the first time **then**
                    if q_x is hash-based scan join with mv_y **then**
                    $temp := \frac{Size(mv_y)}{P \cdot Page\_Size} \cdot t_{I/O} + t_{seek} + t_{hash-join}(mv_y)/P$
                        $+Size(mv_y)/P \cdot t_{cpu}(q_x, mv_y)$
                    else /* q_x is index-based join */
                    $temp := \sigma_{xy} \cdot \frac{Size(mv_y)}{P \cdot Page\_Size} \cdot t_{I/O} + t_{index}(q_x, mv_y)/P$
                        $+\sigma_{xy} \cdot Size(mv_y)/P \cdot t_{cpu}(q_x, mv_y)$
                    endif
                else /* mv_y is not chosen in the first time */
                    if mv_y has been used by other hash-based queries **then**
                        if q_x is hash-based scan join **then**
                        $temp := Size(mv_y)/P \cdot t_{cpu}(q_x, mv_y)$
                        else $temp := Delta\_Cost(q_x, IX(mv_y))$
                        endif
                    else if mv_y has been used by only index-based queries **then**
                            if q_x is hash-based scan join **then**
                            $temp := \frac{Size(mv_y)}{P \cdot Page\_Size} \cdot t_{I/O} + t_{seek} + t_{hash-join}(mv_y)/P$
                            $+Size(mv_y)/P \cdot t_{cpu}(q_x, mv_y)$
                            else $temp := Delta\_Cost(q_x, IX(mv_y))$
                            endif
                        endif
                    endif
                endif
            endif;
            if $\Delta\_cost > temp$ **then**
                $\Delta\_cost := temp$;
                $q\_candidate := q_x$;
                $BT(q_x) := mv_y$; /* q_x uses mv_y as its base table */
            endif
        endfor
    endfor;
    if $q\_candidate$ is index-based **then**
        $IX(BT(q\_candidate)) := IX(BT(q\_candidate)) \cup \{q\_candidate\}$
    endif;
    $Cost := Cost + \Delta\_cost$;
    $AQ := AQ - \{q\_candidate\}$
until $AQ = \emptyset$.

The function `Delta_Cost` is defined as follows.

Function `Delta_Cost`$(q, IX(v))$;
begin

/* $IX(v)$ is a set of index-based queries which use v as their base table */
$\sigma_{xy}^{old} := \frac{|\vee_{q' \in IX(v)} F(q',v)|}{Size(v)}$;
$\sigma_{xy}^{new} := \frac{|\vee_{q' \in IX(v)} F(q',v) \vee F(q,v)|}{Size(v)}$;
$\Delta\sigma_{xy} := \sigma_{xy}^{new} - \sigma_{xy}^{old} \quad (\leq \sigma_{xy})$;
$Delta\_Cost := \Delta\sigma_{xy} \cdot \frac{Size(v)}{P \cdot Page\_Size} \cdot t_{I/O} + t_{index}(q,v) + \sigma_{xy}^{new} \frac{Size(v)}{P \cdot t_{cpu}(q,v)}$
$\quad + \Delta\sigma_{xy} \sum_{q' \in IX(v)} \frac{Size(v)}{P \cdot t_{cpu}(q',v)}$
end

4.2 Parallel algorithm for evaluating the global plan

After running `Algorithm 1`, a global plan has been obtained. In the second phase, we need to evaluate the plan efficiently. As understood, implementing the second phase is so obvious in the sequential environment. However, it seems not to be trivial in the parallel and distributed environment because the communications are involved. From now on, assume that the global plan for a set of queries Q has been generated, we will focus on how to evaluate the plan in the multiprocessor system efficiently. In other words, we need to answer all queries in Q, thus, the MDX query efficiently.

The implementation of the second phase basically consists of several rounds. At each round, the algorithm evaluates all queries that share the same base table simultaneously. That is, assume that MV' is the set of the materialized views chosen by the queries in Q, $(MV' \subseteq MV)$. The algorithm will run $|MV'|$ rounds, and at round i it proceeds as follows. Let $mv_i \in MV'$ be the materialized view being considered and mv_i have been chosen by a group of queries $q_{i_1}, q_{i_2}, \ldots, q_{i_k}$ according to the plan generated, $1 \leq k \leq m$. We evaluate these queries simultaneously at this round. In the following we first present three algorithms based on the three corresponding algorithms in [TM98], then present an algorithm using pipelining.

4.2.1 Efficient parallel algorithms

We now consider how to evaluate the queries $q_{i_1}, q_{i_2}, \ldots, q_{i_k}$ which share a materialized view mv_i efficiently. We make use of the algorithms due to [TM98] as a subroutine to evaluate these queries simultaneously through a modification to [TM98]'s algorithms. That is, in their algorithm they only consider the scan cost of the materialized view when loading that materialized view into main memory, whereas in our case we need to take the scan cost, hash star-join cost, and index-star join cost together into consideration. We now present the details of the modification to the [TM98] algorithms.

We are now ready to give the detailed procedure for the second phase is as follows. Assume that there is a global plan already and MV' is the set of materialized views chosen by the queries in Q.

Procedure `Evalu_Quer`(MV', Q);
while $MV' \neq \emptyset$ **do**
choose a mv_i from MV',
let Q' be the set of queries using mv_i as the base table.
$MV' := MV' - \{mv_i\}$;
$Q := Q - Q'$
apply Algorithm X in [TM98] to evaluate all queries in Q'
endwhile

where X is either `m2P`, or `mReP`, or `mBC`. Thus, three parallel algorithms for our problems have been obtained depending on which X will be used.

Following the idea in [TM98], we may need to further partition a group of queries that share a base table into K sub-groups in order to achieve a better trade-off between the extra I/Os cost and the combined cost consisting of the scan cost and hash and/or index star-join cost. However, we wish to make as many queries as possible to share the common tasks. From this point, we expect that K is not too large because the combined cost is K times of the cost of the case with $K = 1$.

Here we must mention that our problem is much harder than the problem discussed in [TM98]. It is easy to see that their problem is a special case of ours in which (1) all queries share the same base table, therefore, no choice for finding a base table for each query is needed; and (2) to answer the queries, there is neither scan hash star-join nor index star-join involved.

4.2.2 A parallel algorithm based on pipelining

Through the previous analysis to algorithms `m2P, mReP, mBC`, we know that for our problem, the algorithm based on `m2P` is effective only the grouping selectivity is low, otherwise, it requires lots main memory and redundant grouping computation. The algorithm for our problem based on `mReP` is generally better than the first one in terms of the memory utilization and redundant grouping computation. However, its communication cost is not good because for each individual query there is a different partitioning to the resource data. Compared with the algorithms based on `m2P` and `mReP`, the communication cost of the algorithm based on `mBC` is the same as the communication cost of `Broadcasting Algorithm`. Therefore, if a multiprocessor system supports broadcasting efficiently, the algorithms using broadcasting has a better performance which has been proven by Takeshi et al through experiments. Otherwise, the expense for that is so expensive because the communication overhead overshadows the system performance due to the communication conflicts. In this paper we suggest a new approach which not only reduces the communication overhead among the processors but also has the same broadcast capability as the broadcasting algorithms. The approach is presented as follows.

We first establish a *virtual* ring among the processors such that each processor in the system has a unique predecessor and descendent. As a result, each processor PE in the virtual ring has a unique predecessor $L(PE)$ and descendent $R(PE)$. Note that most modern multiprocessor systems have such ring structures (Hamiltonian circuits). The basic idea behind the proposed algorithm is that each processor processes, sends or forwards its messages to $R(PE)$, and receives messages from $L(PE)$. Assume that each processor holds a fragment of a table which consists of r local disk pages, i.e., the entire table consists of no more than rP disk pages. The pipelining algorithm is described as following.

Procedure `Pipe_Ring`
for $i := 1$ **to** r **do**
    for each processor PE_j: $0 \leq j \leq P-1$ **do**
        do hash/index star-join and aggregation using its local disk page i;
        send its local disk page i to $R(PE_j)$;
    endfor;
    $T := true$; /* test whether finishing transfer disk page i through all processors */
    while T **do**
        /* assume that PE_j received a disk page $Page$ from $L(PE_j)$ */
        if $Page$ is not the local disk page i **then**
        then PE_j executes hash/index star-join and aggregation using $Page$;
            PE_j sends $Page$ to $R(PE_j)$;
        endif;
        if all processors received their own local disk page i from the net
        then $T := false$;
        endif
    endwhile
endfor.

In Algorithm `Pipe_Ring`, each processor sends its disk pages or forwards those disk pages resided in other processors to the net, and processes the pages in the pipeline fashion. Thus, the communication conflicts among the processors is not significant, therefore, the communication cost is not as large as the broadcasting algorithm.

5 Conclusions and Future Work

In this paper we considered optimizing and evaluating multiple dimensional aggregate queries simultaneously on a shred-nothing multiprocessor system. We have shown how to generate a good global plan and how to evaluate the plan efficiently in such a system. As mentioned, our approach is decomposed the original optimization problem into two sub-optimization problems with objectives to minimize the commutation cost and communication respectively. However, how to put the communication cost and computation cost together as an optimization objective by giving efficient solutions on the shared-nothing multiprocessor system is still an open problem, which we will study in our future work.

References

[AAD+96] S. Agawal, R. Agrawal, P. M. Deshpande, A. Gupta, J. F. Naughton, R. Ramakrishnan and S. Sarawagi. On the computation of multidimensional aggregates. *Proc. of the 22nd VLDB Conf.*, Mumbai, India, 1996, pp.506-521.

[AESY97] D. Agrawal, A. El Abbadi, A. Singh and T. Yurek. Efficient View Maintenance at Data Warehouse. *Proc. of the 1997 ACM-SIGMOD Conf.*, 1997, 417–427.

[BPT97] E. Baralis, S. Paraboschi, and E. Teniente. Materialized view selection in a multidimensional database. *Proc. of the 23rd VLDB Conf.*, Athens, Greece, 1997.

[BBDW83] D. Bitton, H. Boral, et al. Parallel algorithms for the execution of relational database operations. *ACM Trans. on Database Systems*, 8(3), 1983.

[DANR96] P. M. Deshpande, S. Agarwal, J. F. Naughton, R. Ramakrishnan. Computation of multidimensional aggregates. *Technical Report*, No: 1314, Dept. of CS, Univ. of Wisconsin-Madison, 1996.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability.* W. H. Freeman, San Francisco, 1979.

[GBLP95] J. Gray, A. Bosworth, A. Layman and H. Prahesh. Data cube: a relational aggregation operator generalizing group-by, cross-tab, and sub-total. *Microsoft Technical Report*, MSR-TR-95-22, 1995.

[GM95] A. Gupta and I. Mumick. Maintenance of materialized views: problems, techniques, and applications. *IEEE Data Engineering Bulletin, Special Issue on Materialized Views and Data Warehousing*, 18(2), 1995, 3-19.

[GHRU97] H. Gupta, V. Harinarayan, A. Rajaraman and J. D. Ullman. Index selection for OLAP. *Proc. of Intern. Conf. on Data Engineering*, Birmingham, UK, 1997.

[Gupta97] H. Gupta. Selection of views to materialize in a data warehouse. *Proc. of Intern. Conf. on Database Theory*, LNCS , 1997.

[HRU96] V. Harinarayan, A. Rajaraman and J. D. Ullman. Implementing data cubes efficiently. *Proc. of the 1996 ACM-SIGMOD Conf.*, 1996, 205–216.

[HZ96] R. Hull and G. Zhou. A framework for supporting data integration using the materialized and virtual approaches. *Proc. of the 1996 ACM-SIGMOD Conf.*, 1996, 481–492.

[Kim96] R. Kimball. *The Data Warehouse Toolkit.* John Wiley Sons, 1996.

[LO98] W. Liang and M. E. Orlowska. Optimizing multiple dimensional queries simultaneously in multidimensional databases. *manuscript*, submitted for publication, May, 1998.

[MS] Microsoft Corp. " OLE DB for OLAP Design Specification– Beta 2". **http://www.microsoft.com/data/oledb/olap/prodinfo.html**.

[OQ97] P. O'Neil and D. Quass. Improved query performance with variant indexes. *Proc. of the 1997 ACM-SIGMOD Conf.*, May, 1997.

[PS88] J. Park and A. Segev. Using common subexpressions to optimize multiple queries. *Proc. 4th Intern. Conf. on Data Engineering*, 1988, 311–319.

[S88] T. K. Sellis. Multi-query optimization. *ACM Trans. on Database Systems*, Vol. 13, No.1, March, 1988, 23–52.

[SN95] A. Shatdal and J. F. Naughton. Adaptive parallel aggregation algorithms. *Proc. of the 1995 ACM-SIGMOD Conf.*, 1995, 104–114.

[SS94] K. Shim and T. K. Sellis. Improvements on heuristic algorithm for multi-query optimization. *Data and Knowledge Engineering*, Vol.12, No.2, March, 1994.

[RSC98] K. A. Ross, D. Srivastava, and D. Chatziantoniou. Complex aggregation at multiple granularities. *Proc. of Intern. Conf. on Extending Databases Technology*, March, 1998.

[RSS96] K. A. Ross, D. Srivastava, and S. Sudarshan. Materialized view maintenance and integrity constraint checking: trading space for time. *Proc. of the 1996 ACM-SIGMOD Conf.*, 1996, 447–458.

[RS97] K. A. Ross and D. Srivastava. Fast computation of sparse datacubes. *Proc. of the 23rd VLDB Conf.*, Athens, Greece, 1997.

[TM98] F. Takeshi and H. Matsuzawa. Parallel processing of multiple aggregate queris on shared-nothing multiprocessors. *Proc. of 6th Intern. Conf. on Extending Database Technology*, Valencia, Spain, 1998, 278–292.

[SAG96] S. Sarawagi, R. Agrawal and A. Gupta. On the computing the data cube. *Research Report*, IBM Almaden Research Center, San Jose, CA, 1996.

[Su96] P. Sundaresan. Data warehousing features in Informix Online XPS. *Proc. of the 4th PDIS Conf.*, Miami Beach, FL, USA, 1996.

[YKL97] J. Yang, K. K. Karlapalem, and Q. Li. Fast computation of sparse databases algorithms for materialized view design in data warehousing. *Proc. of the 23rd VLDB Conf.*, Athens, Greece, 1997.

[Zeli97] A. Zelikovsky. A series of approximation algorithms for the acyclic directed Steiner tree problem. *Algorithmica*, 18:99-110, 1997.

[ZDN97] Y. Zhao, P. M. Deshpande, and J. F. Naughton. An array-based algorithm for simultaneous multidimensional aggregates. *Proc. of the 1997 ACM-SIGMOD Conf.*, 1997, 159–170.

[ZDNS98] Y. Zhao, P. M. Deshpande, J. F. Naughton, and A. Shukla. Simultaneous optimization and evaluation of multiple dimensional queries. *Proc. of the 1998 ACM-SIGMOD Conf.*, June, 1998.

[ZGHW95] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View maintenance in a warehousing environment. *Proc. of the 1995 ACM-SIGMOD Conf.*, 1995, 316–327.