



## Optimization in Data Cube System Design

EDWARD HUNG  
DAVID W. CHEUNG\*  
BEN KAO

ehung@cs.umd.edu  
dcheung@csis.hku.hk  
kao@csis.hku.hk

*Department of Computer Science and Information Systems, The University of Hong Kong, Hong Kong*

*Received November 13, 2000; Revised June 25, 2003; Accepted June 30, 2003*

**Abstract.** The design of an OLAP system for supporting real-time queries is one of the major research issues. One approach is to use data cubes, which are materialized precomputed multidimensional views of data in a data warehouse. We can derive a set of data cubes to answer each frequently asked query directly. However, there are two practical problems: (1) the maintenance cost of the data cubes, and (2) the query cost to answer those queries. Maintaining a data cube requires disk storage and CPU computation, so the maintenance cost is related to the total size as well as the total number of data cubes materialized. In most cases, materializing all data cubes is impractical. The maintenance cost may be reduced by merging some data cubes. However, the resulting larger data cubes will increase the query cost of answering some queries. If the bounds on the maintenance cost and the query cost are too strict, we help the user decide which queries to be sacrificed and not taken into consideration. We have defined an optimization problem in data cube system design. Given a maintenance-cost bound, a query-cost bound and a set of frequently asked queries, it is necessary to determine a set of data cubes such that the system can answer a largest subset of the queries without violating the two bounds. This is an NP-hard problem. We propose approximate Greedy algorithms GR, 2GM and 2GMM, which are shown to be both effective and efficient by experiments done on a census data set and a forest-cover-type data set.

**Keywords:** data warehouse, data cube system design, OLAP, optimization, approximate algorithm

### 1. Introduction

#### 1.1. DSS and OLAP

*A data warehouse is a subject-oriented, integrated, non-volatile, and time-varying collection of data that is used primarily in support of organizational decision making (Inmon, 1996). Its aim is to support decision-making based on historical, summarized and consolidated data (Chaudhuri and Dayal, 1997a, 1997b). With the advancement of data warehousing technology, corporations are building their decision support systems (DSS) on large data warehouses. In order to support on-line analytical processing (OLAP) (on-line DSS), the*

The first author is currently in the Department of Computer Science at the University of Maryland, College Park, but the work in this paper was done when he was at the University of Hong Kong. He is supported by the Croucher Foundation Scholarships.

Research of the second and third authors was supported partially by a grant (HKU 7038/99E) from the Research Grants Council of Hong Kong.

\*Author to whom all correspondence should be addressed.

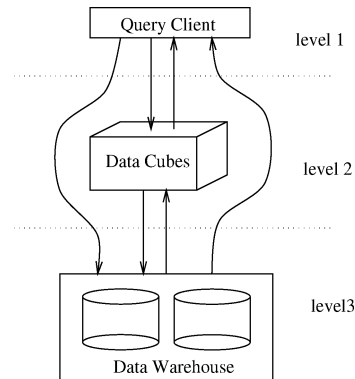


Figure 1. Three-level architecture of a data cube system.

system is required to answer queries with a fast response time. However, queries are usually about summarization information, so the system needs to scan almost the entire database, giving a very long response time. We define the query cost of a query as the response time for a system to answer the query. A set of frequently asked queries can be supplied by the user (the OLAP system designer) or determined by the system from the past history of query submission. The query cost of a set of frequently asked queries of an OLAP system is the sum of the query costs to answer the queries. One efficient approach to reduce the query cost is to translate frequently asked queries to *data cubes*, which are precomputed multi-dimensional views of the data in the data warehouse of an OLAP system (Gray et al., 1996). Once the data cubes are built, answers to the frequently occurring queries can be retrieved from the data cubes in real time.

An OLAP system can be modeled by a *three-level architecture* that consists of: (1) a query client, (2) a data cube engine, and (3) a data warehouse server (figure 1).

The bottom level is a data warehouse built on top of one or more source operational DBMSs. It supports fast aggregations by using different indexing techniques such as bit-map indices and join indices (O'Neil and Graefe, 1995; O'Neil and Quass, 1997).

The middle level contains a set of data cubes for the OLAP system. These data cubes are called *base data cubes*, generated from the data warehouse in the bottom level. A base data cube contains the aggregates computed over a selected set of attributes from the schema of the data warehouse. Some other aggregates can also be computed from the base data cubes. For example, if a base data cube is defined on the attributes *part*, *customer* and *date*, then the data cube containing the aggregates on *part*, *year* can be derived. There are many commercial OLAP products such as Microsoft SQL OLAP Services, Hyperion Essbase OLAP Server and IBM DB2 OLAP Server (Cheung et al., 1999).

The top level is a query client, which supports DSS queries and allows users to browse the data cached from the data cubes, with operations like slicing and dicing.

A query submitted to the client interface level, after being checked against the data cube set, will be directed to the data cube level if it can be answered by the data cubes there; otherwise, the query is passed to the bottom level, from which the results can be derived. Since data cubes store pre-computed results, it is much faster to answer queries with data

cubes than with the data warehouse. Our approach is to identify a set of frequently asked queries and materialize the corresponding initial set of data cubes for those queries, while the answers of the remaining small number of unusual or ad-hoc queries are generated directly from the data in the data warehouse. Materializing data cubes for all frequently asked and unusual queries induces a very high maintenance cost. Answering all above queries directly from the data in the data warehouse imposes a very high query cost. Our approach strikes a balance between the maintenance cost and the query cost. By materializing only the most frequently asked queries, we can service most of the queries with real-time performance, and yet do not have to demand an unreasonable amount of resources for computing and storing all aggregates for all possible queries.

Many commercial products such as Seagate Info Worksheet and Microsoft PivotTable Services are available. Currently these products mainly support browsing and report-generating functions on cached data. If the answers are not cached, it will be necessary for these products to access the data cubes or the data warehouse directly (Cheung et al., 1999).

Various studies have been done on the three levels of an OLAP system. For the data cube level, most research studies focus on two issues: (1) how to compute aggregates from a base data cube efficiently, and (2) how to store a data cube. However, our previous research study (Cheung et al., 1999) has shown that the key to the design of a query-efficient OLAP system lies on the design of a good data cube set. As we have mentioned, the OLAP system would be able to support real-time responses if the data cube level can answer all the queries. Maintenance requires disk storage and CPU computation, and so the maintenance cost grows with the total size and the total number of data cubes materialized. As a result, materializing all possible data cubes so that all possible queries can be answered by the data cubes is clearly impractical due to the high maintenance cost. We define the *maintenance-cost bound* as the maximum total maintenance cost that can be afforded by the system to maintain the materialized data cubes. Given an initial set of data cubes derived from a set of frequently asked queries, we can reduce the total size and the total number of data cubes as well as the maintenance cost by selecting several disjoint sets of data cubes and merging them into some larger data cubes. As a result, a new set of data cubes with a smaller total size and number can be obtained. Merging two or more data cubes means that, instead of materializing those data cubes, we materialize a new data cube with all the attributes of those data cubes to be merged. This reduces the maintenance cost at the expense of increasing the query cost because some queries processed using merged data cubes generally take longer time than using the original smaller data cubes.

Since a typical OLAP system needs to support real-time responses and may have certain time constraints for processing the set of frequently asked queries, it is useful for the user (the OLAP system designer) to define a *query-cost bound*, such that the sum of the query costs to answer a selected set of frequent queries using the resultant set of data cubes should not exceed the bound given by the user. However, sometimes the maintenance-cost bound and the query-cost bound that the user gives are so strict that there does not exist any data cube set that satisfies the bounds even after considering all possible merging of data cubes. In that case, an intelligent system should report to the user that the response time requirement (the query-cost bound) cannot be met. The user then has to consider relaxing the requirement.

This can be done by either imposing a larger (i.e., less stringent) query-cost bound, or by removing certain queries from the initial query set from consideration. In the latter case, the query-cost guarantee would only be applied to the remaining queries. Queries not supported by the middle level (a data cube engine with a set of materialized data cubes) will be answered directly using the data warehouse server, which is slower than the data cube engine but supposed to have a reasonable response time due to the mature development of indexing technique available. In this paper, we will study how a system could make suggestions to a user on which queries to be removed in order to satisfy the timing requirement.

Our problem now is how to choose a minimum number of data cubes to remove, and how to merge the remaining data cubes so that both the maintenance-cost bound and the query-cost bound are satisfied. This is an NP-hard problem. Our approach is to develop some efficient and effective approximate algorithms so that a solution with an acceptable performance can be found in an acceptable execution time.

### 1.2. Optimization problem in data cube system design

Given a set of frequently asked queries, a maintenance-cost bound and a query-cost bound, our goal is to derive a data cube set that satisfies the bounds and that answers a maximum number of queries. Our approach to this data cube system design problem is a two-phase process:

1. Define an Initial Data Cube Set. The first phase is to derive an initial set of data cubes from the set of frequently asked queries. This set is called an *initial data cube set*. The answer of each query can be retrieved directly and efficiently from a data cube in this initial set.
2. Optimize the Data Cube Set. The second phase is an optimization of the initial data cube set: to determine a largest subset of the initial data cube set and merge them so that a maximum number of queries can be answered, satisfying the maintenance-cost bound and the query-cost bound.

We remark that the initial data cube set, with one data cube being tailor-made to answer each query, gives the best query performance. However, since data cubes are derived from each frequently asked query, there may be a lot of redundancy among the data cubes due to overlapping attributes. For example, a data cube containing attributes  $a$ ,  $b$ ,  $c$  and  $d$  is derived to answer a query. Another data cube containing attributes  $b$ ,  $c$ ,  $d$  and  $e$  is derived to answer another query. The two data cubes overlap on the attributes  $b$ ,  $c$  and  $d$ . The maintenance cost can be reduced by merging the data cubes into one.

The data cubes to be merged are called component data cubes of the resulting data cube produced from merging. It is observed that the resulting data cube size is always larger than or equal to the size of the largest one of the component data cubes. Indeed, if we consider a derivation of a data cube from another data cube as a projection of points in a multi-dimensional space on a hyperplane, then the size of the original data cube is the number of points in the multi-dimensional space, and the size of the derived data cube is the number of points of a projection on the hyperplane. Since it is possible that the actual number of

points in the space is larger than the sum of the number of points in the projections on two or more hyperplanes, it is possible that the size of the original data cube is larger than the total size of its component data cubes.

Using the above example, we can merge the two data cubes (containing attributes  $(a, b, c, d)$  and  $(b, c, d, e)$  respectively) to become a new data cube containing attributes  $a, b, c, d$  and  $e$ . However, queries processing using the data cube resulting from merging will in general be slower than using the original data cubes because the resulting data cube is usually larger than any one of the component data cubes. Therefore, there is a trade-off between query performance and data cube maintenance cost.

We call the process of merging some subsets of data cubes in a data cube set in order to satisfy the two bounds and to answer a maximum number of queries an *optimization* of the data cube set.

### 1.3. Related works

Several papers have been published on data cube implementation. Algorithms have been proposed in Harinarayan et al. (1996) and Shukla et al. (1998) to select views to materialize. Since a data cube is a materialized multidimensional view of data, view selection algorithms can be modified to solve the optimization problem we are addressing.

Harinarayan et al. (1996) gave an interesting approach on how to choose a set of views to be materialized in a CUBE (Gray et al., 1996) such that the largest *benefit* (saving in query cost) can be obtained under certain constraint. Recall that a data cube is a materialized multidimensional view of data. As first proposed by Gray et al. (1996), a CUBE is a composite lattice of data cubes constructed from all the subsets of a set of attributes, on which a query can be posted. In other words, a CUBE is a composite lattice of ALL materialized multidimensional views of data. The Greedy algorithm in Harinarayan et al. (1996) chooses a view with the largest benefit. They proved that the total benefit of their algorithm is at least  $\frac{e-1}{e} \approx 0.63$  of the benefit of an optimal algorithm. Shukla et al. (1998) proposed a similar algorithm, *PBS(PickBySize)*, for the same problem in Harinarayan et al. (1996). The only difference of the Greedy algorithm and PBS is that PBS materializes the view with the smallest size. Algorithm PBS gives the same  $(0.63 - f)$  performance bound as that of the Greedy algorithm. However, this bound only holds if all data cubes satisfy certain size constraints. In this paper, we do not make such an assumption.

Cheung et al. (1999) considered a method that applies the Greedy algorithm (Harinarayan et al., 1996) to optimize an initial set of data cubes with the minimum query cost. For an initial set of data cubes that corresponds to a set of frequently asked queries, the method is to apply the algorithm on the *top* data cubes in the initial data cube set. A top data cube in a given data cube set is one that cannot be deduced from any other data cubes in the set. The method includes all the top data cubes in the answer set  $C$ , and then expands  $C$  by applying the Greedy algorithm proposed in Harinarayan et al. (1996) on the top data cubes. The expansion stops when the total maintenance cost exceeds the bound. However, according to Cheung et al., this method is undesirable because it is possible that the total size of only the top data cubes has already exceeded the maintenance-cost bound. Moreover, it is unreasonable to always include the top data cubes if there exist some other data cubes that

are sufficient to answer the queries (to be answered using the top data cubes). Furthermore, there may be many overlapping attributes among the top data cubes, thus, merging some of them could be beneficial. Besides, the union of attribute sets of all queries may include all attributes in the data warehouse, then the number of attributes (for simplicity, we do not consider hierarchy) is often in tens. As a result, the number of data cubes in a CUBE lattice is very large. Since the Greedy algorithm consider the whole CUBE lattice, the execution time grows exponentially with the number of attributes.

Although Harinarayan et al. (1996) and Shukla et al. (1998) state that their algorithms get a very good lower bound of benefit, Karloff and Mihail (1999) argued that the correct way of evaluating the goodness of an algorithm should be based on *performance ratio*, which is the ratio of “the response time to answer queries using the set of views selected by the algorithm” to “the response time to answer queries using an optimal set of views.” The lower bound (0.63 of optimal) of benefit does not give a performance-ratio guarantee. Their proof shows that there do exist some cases where the performance ratio of the Greedy algorithm can be very bad. In this paper, we measure the goodness of our algorithms using the performance ratio. Furthermore, the goal of Harinarayan et al. (1996) and Shukla et al. (1998) is to choose a subset of views in a CUBE, but our goal is to choose a set of data cubes for a given set of queries.

Our previous work is presented in Cheung et al. (1999, 2000). The problem suggested in those papers is exactly the same as our sub-problem at the attribute level (to be discussed in detail in the next section): how to find a set of data cubes to minimize query cost of answering a set of queries with maintenance cost under bound. An algorithm *CMP* (*cube Merging and Pruning*) was proposed for considering how to choose data cubes to merge and how to prune the search space. *rMP* is a *r-Greedy* version of *cMP*, where only at most  $r$  data cubes are chosen to merge into one data cube. If a data cube is an ancestor of another data cube, then the former can be used to derive the latter. If we draw a line to connect two data cubes with *parent/child* relationship (or, to be precise, one is the smallest ancestor of the other, among all other data cubes), then the two data cubes are in the same *path*. Each path may only contain a data cube if and only if the data cube has no proper ancestor or proper descendant. The authors developed some pruning techniques on the search space based on two assumptions. The first assumption is the existence of a small number of paths among the data cubes, i.e., most of the data cubes gather to form a small number of paths. The performance of pruning improves if there are only a small number of paths and the lengths of paths are long. However, in general, this situation does not occur easily. If queries are generated in random, then we can expect that most paths contain only one to two data cubes. The second assumption is that a data cube must have a smaller size than its ancestor. Without taking the assumptions, the effect of pruning degrades seriously and *rMP* becomes a pure *r-Greedy* algorithm of merging data cubes and *cMP* becomes a pure brute-force algorithm.

Besides, *cMP* does not consider to use the smallest possible data cube to answer a query. When data cubes are merged, it is fixed to use the resultant data cube to answer the queries associated to the component data cubes, even though there exist some smaller data cubes to answer some of those queries. In our research, we observe that in practice and in theory, the two assumptions do not hold in general. Therefore, we do not make the assumptions and we use the smallest possible data cube to answer a query.

The work by Theodoratos et al. (1999, 2001) propose (1) a set of transformation rules for Select-Project-Join queries to generate an initial set of materialized views to answer them, (2) algorithms to prune the search space of the following problem: given a set of queries, select a set of materialized views to satisfy a storage constraint and minimize the combination of query cost and maintenance cost. The differences between their goal and our attribute-level optimization are that: (1) they have a storage constraint while our maintenance-cost bound may be either the storage size or the number of data cubes, (2) they minimize the combination of query cost and maintenance cost while we minimize the query cost. They also proposed a Greedy algorithm similar to ours.

While part of the discussion of our attribute-level optimization looks similar to the view merging in Agrawal et al. (2000), there are differences between their paper and our work. First, they used view merging for generating candidate views based on an initial set of views. Pair-wise merging is repeated on the set of newly-formed views and original views until there does not exist any new view generated that is not much larger than the original views it is derived from. Although the algorithm 2GM we proposed also considers pair-wise merging, we use an evaluation function to choose the best merging in each stage. Then we repeat the considering of pair-wise merging based on the new set of views (data cubes) containing the non-chosen old views in the previous step and the new view formed from the chosen merging. Second, the total number of views they generate is exponential to the number of original views as they consider the merging of all possible groupings of views. In the worst case, 2GM does not need to consider all possible cases. While they argued that from the experiments much fewer merged views are explored in practice (the running time is an order of magnitude faster than their exhaustive approach), our experiments show that the execution time of 2GM is a few orders of magnitude faster than the exhaustive approach. This indicates that our search space is reduced much significantly. Third, the whole set of candidate views (and indices which that paper assumed are already generated) are considered by their Greedy( $m, k$ ) algorithm. Our approach is different, as explained above. Fourth, while their work is similar to one of our sub-problems, our work is not only focused on the attribute-level optimization and our main target is not on the attribute-level optimization as they did. In contrast, our main target is very different. We want to optimize the system by choosing a data cube set that satisfy the two constraints and answer a maximum number of queries. Merging the data cube is only a very basic tool in our one sub-problem. Fifth, they stated that other previous work on view selection may be adopted in their architecture by simply substituting Greedy( $m, k$ ), so they view those work to be complementary to their work. Similarly, we think that in principle, their view merging may be adopted in part of (only) our attribute-level optimization, so we view their view merging to be complementary to our work.

The paper of Zaharioudakis et al. (2000) deals with rewriting of a complicated query in order to make use of one or more materialized views. Although their work is not directly related to ours (as we deal with how to decide a set of views to materialize while they deal with how to answer queries using views), their work is useful to the extension of our problem where we are no longer restricted to using exactly one data cube to answer a given query. When a query may be answered by more than one view (data cube), during searching a solution in the search space, we need to not only consider merging the initial set of data

cubes, but also consider dividing them into smaller data cubes since several components from different initial data cubes may be used to answer an original query. This is a much harder but interesting problem, which we may consider as our future work.

#### 1.4. Organization of paper

After introducing the optimization problem in data cube system design and our approach to solve it, we will discuss the problem in detail including the search space and cost model in Section 2. The optimization phase in our approach can be divided into two levels. Section 3 describes the higher level (the query level), where a maximal subset of the initial data cube set is determined. The subset can be refined to satisfy the maintenance-cost bound and the query-cost bound using the attribute-level optimization subroutine. Section 4 describes the lower level (the attribute level), where a given set of data cubes is refined to satisfy the maintenance-cost bound and minimize the query cost. The performance study is given in Section 5. We have a further discussion on this research problem in Section 6. Finally we give a conclusion in Section 7. This paper was written mostly based on Hung (2000), where more details, experimental results and further discussion can be found.

## 2. Data cube system design optimization

### 2.1. Search space of an optimal set

In this paper, we assume that the requirements of an OLAP system is captured in a set of frequently asked queries. We use  $Q$  to denote the initial set of the data cubes derived from the queries. For example, if a query involves the attributes  $a, b, c$ , a three-dimensional data cube on these attributes is included in  $Q$ . Before defining the optimization problem, let us first discuss the search space of the problem.

To simplify the problem, we assume that the database in the data warehouse is represented by a star schema (Chaudhuri and Dayal, 1997a). Attributes in the queries come from the fields of the dimension and fact tables, which may contain many attributes. For example, in the TPC-D database, the table *part* includes the attributes *p\_partkey*, *p\_brand*, *p\_type*, *p\_size*, *p\_container*, etc. The dimension is in fact a multi-hierarchical dimension as shown in figure 2. Besides, in the star schema, some attributes are stored directly in the fact table, e.g.,

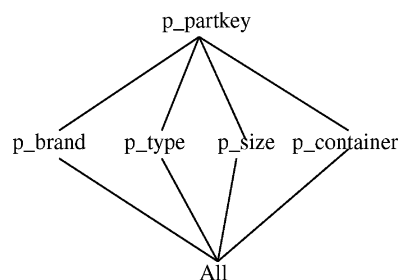


Figure 2. The multi-hierarchical structure of the *part* dimension in TPC-D.

the attributes  $l\_shipdate$ ,  $l\_commitdate$ ,  $l\_receiptdate$  in the fact table  $lineitem$ . As a result, the number of attributes (dimensions) needed to be considered in a data cube system design is much more than the number of dimension tables. In TPC-D (Transaction Processing Performance Council, 1997), 33 attributes need to be considered.

In Harinarayan et al. (1996), the notion of a *composite lattice* is used to integrate multi-hierarchical dimensions with the lattice of aggregates in a data cube. Assume that  $A = \{a_1, a_2, \dots, a_n\}$  is the set of all the attributes on which a query can be posted. Any subset of  $A$  can be used as the dimension attributes to construct a data cube. The composite lattice  $L = (\mathcal{P}(A), \prec)$  is the lattice of data cubes constructed from all the subsets of  $A$ . ( $\mathcal{P}(A)$  is the power set of  $A$ .) The data cube associated with the set  $A$  is the root of the lattice  $L$ . For two different data cubes  $c_1, c_2 \in L$ , the *derived from* relationship,  $c_1 \preceq c_2$ , holds if  $c_1$  can be derived from  $c_2$  directly or by aggregation. For example the data cube  $c_1 = [part, year]$  can be derived from  $c_2 = [part, customer, date]$ . The lattice  $L$  is the search space of the optimization problem. In our optimization problem, we need to search some data cubes in the lattice  $L$  which can be used to answer a maximum number of frequently asked queries such that the query cost and the maintenance cost are under bounds. As we have mentioned,  $n$  is large in general. For example, in the TPC-D benchmark,  $n = 33$ . Thus, the search space  $L$  of the optimization problem is enormous.

## 2.2. Problem definition

Given an initial data cube set  $Q$  (which represents the set of frequently asked queries), a search space  $L$ , a maintenance-cost bound  $MCB$  and a query-cost bound  $QCB$ , the optimization problem in data cube system design is defined in figure 3.

Each data cube in  $P$  is derived from a particular query, so  $P$  contains a set of data cubes that can be used to answer some frequently asked queries.  $QC(P, C)$  is the total query cost of answering the queries associated with  $P$  by using the data cubes in  $C$ . The constraint states that any frequent query  $p$  can be answered by some data cube  $c$  in  $C$ .  $MC(C)$  is the total maintenance cost of  $C$ . Therefore, the problem is to choose a *maximal* set  $P$  of data cubes (and the corresponding queries) from the initial set  $Q$ , and a set of data cubes  $C$  in  $L$ , such that the cost of answering the queries (corresponding to  $P$ ) by using  $C$  is under the query-cost bound  $QCB$  and the cost of maintaining  $C$  does not exceed the maintenance-cost bound  $MCB$ .

For simplicity, we assume that the number of queries associated with each  $q \in Q$  is the same. Then we can use  $q \in Q$  to represent both a data cube in the initial set and the query

---

Objective:
Maximize $ P $
Constraint:
$P \subseteq Q, C \subset L$ , and $\forall p \in P, \exists c \in C$ , such that $p \preceq c$ , $QC(P, C) \leq QCB$ and $MC(C) \leq MCB$

---

Figure 3. Optimization problem in data cube system design.

associated with it. Later in Section 6.1, we will discuss the situation that some queries have different weights or the number of queries associated with each data cube  $q \in Q$  is not the same. Since we do not want to make any assumption on the implementation of the data cubes and the structure of queries, a good measure of  $QC(P, C)$  is the linear cost model suggested in Harinarayan et al. (1996). In that model, if  $p \preceq c$ , then *the cost of deducing the answer for the query  $p$  from the data cube  $c$  is linear to the number of data points in  $c$* . We use  $S(c)$  to denote the number of data points in  $c$ .

With respect to the cost model, many sampling and analytical techniques can be used to estimate the number of data points. Shukla et al. (1996) describe some traditional methods to estimate the size of a data cube or group-by. Shukla et al. (1996) and Deshpande et al. (1997) demonstrate an algorithm based on probabilistic counting to estimate the size of a data cube or even the whole CUBE with only one scan of data. The algorithm performs very well under various degrees of data skew. In this paper, we simply use a function  $S()$  to return the size of a data cube where we give no assumption about the actual method we use to estimate it.

For each query  $p \in P$ , we need to determine a minimum-cost (size) data cube  $c \in C$ , from which the answer of  $p$  can be deduced. We use  $F_C(p)$  to denote the smallest data cube in  $C$  for answering  $p$ , i.e.,  $F_C(p)$  is a data cube in  $C$  such that  $p \preceq F_C(p)$  and  $\forall x \in C$ , if  $p \preceq x$ , then  $S(F_C(p)) \leq S(x)$ . We use the condition  $S(F_C(p)) \leq S(x)$  instead of  $F_C(p) \preceq x$ , because  $F_C(p)$  needs to be the one that has the minimum size. Besides, it is not necessary for  $F_C(p) \preceq x$  to be true because neither of the two data cubes (that can answer the same query) need to be able to be derived from each other. We can now define  $QC(P, C)$  by the following formula:

$$QC(P, C) = \sum_{p \in P} (S(F_C(p))). \quad (2.1)$$

There are mainly two approaches of view maintenance: recomputation and incremental maintenance. In Griffin and Libkin (1995), the authors stated that the cost of incremental maintenance of a view is nearly proportional to the size of updates of the relations involved. Since the updates have to be applied on the data cube for refresh, so the cost of incremental maintenance of a data cube is approximately proportional to the size of the data cube. Therefore the total maintenance cost can be defined as:

$$MC_1(C) = \sum_{c \in C} S(c). \quad (2.2)$$

i.e., the total number of data points in the data cubes. This is also an estimate of the total disk storage required. Following that, the bound  $MCB$  can be interpreted as the maximum maintenance cost allowable or as the maximum amount of storage needed to maintain the data cubes.

In Griffin and Libkin (1995), the authors also stated that the cost of recomputation of a view is nearly proportional to the size of the relations involved. Since data cubes are mainly materialized views on the fact table with grouping operations and aggregations over one or more measures, the maintenance cost of a data cube is approximately proportional to

the size of the fact table. Since each data cube has nearly the same maintenance cost, an alternative measurement of the total maintenance cost is:

$$MC_2(C) = |C|. \quad (2.3)$$

i.e., the number of data cubes in  $C$ .<sup>1</sup> Following that, the bound  $MCB$  becomes the maximum number of data cubes that are materialized in the data cube system. With respect to recomputation (Agarwal et al., 1996) demonstrates fast algorithms for computing a collection of data cubes, especially CUBE operations. Recomputation using different algorithms on different sets of data cubes may take a very different execution time, depending on the properties of the data cube lattice, the relationships among the data cubes to be materialized and the density or closeness of the data cubes in the lattice. However, in general, it is likely that if there are more data cubes, it usually takes more number of scanning on the database to compute aggregates during performing maintenance, so it generally costs more in maintenance. Our this definition of maintenance cost is used to capture this general idea and gives the system designer an alternative. Thus, the definition (Formula 2.3) is also reasonable.

Without assuming any implementation method, either of the above two measures (Formula 2.2, 2.3) can be used to determine the maintenance cost, and we have developed algorithms that work under either definition. Using which maintenance cost definition depends on the data cube system designer's approach of data cube maintenance or other criteria he/she uses to select views to materialize (as data cubes) such as disk storage space. For example, when the data cube system designer is given a constraint on the disk storage space, then the former formula is an obvious choice; however, if there is not such a clear disk storage constraint or the disk storage size is not a concern, and the data cubes are recomputed during maintenance, then it is more intuitive to use the latter formula. After deciding one maintenance cost definition to use and setting the maintenance-cost bound, any algorithms described in later sections may produce solutions different from those using the other maintenance cost definition. The reasons are: (1) the factors considered to remove or merge a data cube in both definitions are different, (2) the bounds of both definitions are not comparable to each other (one is the number of data cube, another is the total size). As a result, we cannot directly compare both definitions by comparing the total query costs of their solutions.

In this paper we do not explicitly discuss the handling of selections, similarly to many other previous work. However, our approach can be generalized by explicitly considering the result on the data cubes due to the selection conditions in each query, and when we merge two data cubes, we also union the selection conditions corresponding to both data cubes which are applied during the generation of the new data cube.

### 2.3. A two-level approach to the optimization problem

Our approach to the data cube optimization problem consists of two phases. The first phase is the design of an initial data cube set. The second phase is an optimization phase, which consists of two levels. The higher level is the *query level*, while the lower one is the *attribute*

*level*. The higher level is called the query level because at that level we need to determine a set of queries to answer. The lower level is called the attribute level because we need to determine the attribute set to be contained in each data cube (to be materialized) at that level. We propose a two-level approach to solve the optimization problem because it is more systematic to break the whole problem into two smaller sub-problems and it is easier to solve each of them one by one. While a brute-force method can solve the whole problem at the same time to find an optimal solution, it is too inefficient. Using the Greedy approach (and its variants) to solve each sub-problem is easier to understand, and easier to obtain good solutions. In fact, our performance study shows that the solutions found by our approximate algorithms are optimal in most cases, and close to optimal in other cases.

Given an initial data cube set  $Q$ , optimization at the query level is to find a largest subset  $P$  of  $Q$  so that the set  $C$  obtained by merging data cubes in  $P$  satisfies the maintenance-cost bound and the query-cost bound, i.e., to select a maximum number of queries to answer so that the bounds can be satisfied. We want to reach our goal with the removal of as few queries as possible. In order to achieve this, we need to perform the attribute-level optimization defined as follows. Given a data cube set  $P$ , optimization at the attribute level is to refine the set  $P$  to  $C$  so that the sum of the query cost of using  $C$  to answer all queries associated to  $P$  is minimum and the maintenance cost is within the maintenance-cost bound. In other words, it is to choose a data cube set to answer a set of queries selected by the query level optimization so that the cost of answering that set of frequently asked queries can be minimized while the maintenance cost is under bound. The attribute-level optimization is used as a subroutine in the query-level optimization.

The attribute-level optimization problem can be formulated as below. Given an initial data cube set  $P$ , a search space  $L$ , a maintenance-cost bound  $MCB$ , the attribute-level optimization problem in data cube system design is defined in figure 4.

The data cube system design optimization problem (as well as the attribute-level optimization problem) is NP-hard (details in Hung (2000)), so it is computationally very expensive to find an optimal solution. Therefore, what we can do is to develop some efficient and effective approximate algorithms so that a solution with acceptable performance can be found in an acceptable execution time. In this paper, two algorithms *Optimal Removing* and *Greedy Removing* (of queries or data cubes) are proposed for the query-level optimization. Three algorithms *Optimal Merging*, *2-Greedy Merging* and *2-Greedy Merging with Multiple paths* are proposed for the attribute-level optimization. Our suggestion is to use the following two approximation algorithms: Greedy-Removing at the query level and

---

Objective:

Find  $C \subset L$  such that  $QC(P, C) \leq QC(P, C')$

for all possible  $C'$  (as  $C$ ) satisfying the following constraint.

Constraint:

$\forall p \in P, \exists c \in C$ , such that  $p \preceq c$  and  $MC(C) \leq MCB$

---

Figure 4. Attribute-level optimization problem in data cube system design.

2-Greedy Merging with Multiple paths at the attribute level. The details will be discussed in the following sections.

### 3. Query-level optimization

In this section, we discuss the query-level optimization. Given an initial data cube set  $Q$ , optimization at the query level is to find a subset  $P$  of  $Q$  such that: (1) the set  $C$  obtained by merging some data cubes in  $P$  satisfies the bounds on the maintenance cost and the query cost, and (2) the number of data cubes in  $P$  is maximum, i.e., the number of queries removed is minimum.

We consider two ways of searching for a solution: *Optimal Removing (OR)* and *Greedy Removing (GR)* (of queries or data cubes). For OR, we first check whether the initial data cube set  $Q$  can be refined to satisfy the maintenance-cost bound and the query-cost bound. If not, we consider the results of all possible ways of removing one data cube from the initial set. If a solution is still not found, we continue to consider the results of all possible ways of removing two data cubes from the initial set and so on until a solution is found.

For GR, first we have a set  $P$  containing all the data cubes of the initial set  $Q$  and check whether it is a solution. If not, we consider removing one data cube from the set  $P$  such that it gives a minimum query cost among all others. If it is not a solution, we continue to remove one data cube from the set  $P$  until we find a solution.

Comparing the two algorithms, OR is very time consuming. On the other hand, GR is much more efficient. Moreover, as we will show later, the solution obtained from GR is very close to that of OR.

#### 3.1. Optimal Removing

Optimal Removing (OR) finds an optimal removal of data cubes from the initial set  $Q$  of data cubes, so that the resulting set  $P$  has a maximum size and can be refined to give a final data cube set  $C$  that satisfies the maintenance-cost bound  $MCB$  and the query-cost bound  $QCB$ . OR takes a brute-force approach. The outline of OR is shown in figure 5.

```

/* input: L, Q, MCB, QCB; output: P, C */
1)  j = |Q|;
2)  while(j ≥ 1) {
3)    S = AllSubsets(Q, j);
4)    for all P ∈ S do {
5)      C = AO(L, P, MCB)
6)      if (C ≠ ∅)
7)        if (QC(P, C) ≤ QCB) return P, C;
8)    }
9)    j = j - 1;
10) }
11) print "The bounds are too strict."; return ∅, ∅;

```

Figure 5. Algorithm Optimal Removing.

The algorithm starts with the initial set  $Q$ . The loop (lines 2–10) tries all possible combinations of queries starting from keeping all data cubes in the initial set  $Q$ . In the next iteration, the number of data cubes decreases by one. In line 3,  $S = AllSubsets(Q, j)$  assigns  $S$  with all subsets of  $Q$  with the number of queries equal to  $j$ . In the loop in lines 4–8, for each set  $P$  in  $S$ , the attribute-level optimization ( $AO(L, P, MB)$ ) (which will be discussed in Section 4) finds a data cube set  $C$  (in lattice  $L$ ) that can answer  $P$  and satisfy the maintenance-cost bound (line 5). It is possible that there is no solution from the attribute-level optimization if the maintenance-cost bound is too strict, which is checked in line 6. If a non-empty set  $C$  is returned, then it is checked whether its query cost (the cost to answer all queries in  $P$  using  $C$ , i.e.,  $QC(P, C)$  by Formula 2.1) satisfies the query-cost bound  $QCB$  or not. If yes, OR returns  $P$  and  $C$ . If none of them is a solution, we decrease the number of data cubes by one. The iteration of loop in lines 2–10 goes on until a solution is found or until no solution is found even after removing all data cubes (line 11).

For example, suppose the initial set  $Q$  contains data cubes  $T, U, V$  and  $W$ . In the beginning, we consider whether keeping all the data cubes  $\{T, U, V, W\}$  can give a solution. If not, we remove one data cube from the initial set and check whether there is a solution from any of the following combinations:  $\{U, V, W\}$ ,  $\{T, V, W\}$ ,  $\{T, U, W\}$ ,  $\{T, U, V\}$ . If a solution is still not found, we try all possible combination of two cubes:  $\{T, U\}$ ,  $\{T, V\}$ ,  $\{T, W\}$ ,  $\{U, V\}$ ,  $\{U, W\}$ ,  $\{V, W\}$ , and so on.

If the number of data cubes in the initial data cube set is  $n$ , then the number of ways of keeping  $n$  data cubes is  ${}_n C_n$ , where  ${}_n C_r$  is defined as  $\frac{n!}{r!(n-r)!} = \frac{n(n-1)\dots(n-r+1)}{r(r-1)\dots 1}$ . In the next iteration, the number of ways of keeping  $(n-1)$  data cubes is  ${}_n C_{n-1}$ , etc. With respect to the number of attribute-level optimization done, in the worst case, only one or no data cube is kept in the end, and the time complexity of Optimal Removing is  ${}_n C_n + {}_n C_{n-1} + \dots + {}_n C_1 = O(2^n)$ . If a solution is found after  $m$  data cubes are removed, then the number of attribute-level optimization done is  ${}_n C_n + {}_n C_{n-1} + \dots + {}_n C_{n-m} = O(n^{\lceil \frac{n}{2} \rceil})$  if  $m \geq \frac{n}{2}$  or  $O(n^m)$  if  $m < \frac{n}{2}$ . The execution time grows exponentially with increasing  $m$  when  $m < \frac{n}{2}$ . OR is too time-consuming in practice, so we propose another method, Greedy Removing, as described in Section 3.2.

### 3.2. Greedy Removing

Greedy Removing (GR) is a simple, efficient, but still very effective method. The outline of GR is shown in figure 6.

$P$  is the set of data cubes kept. First we check whether keeping all data cubes ( $P = Q$ ) can give us a solution (lines 2–4). If not, we consider all possible ways of removing one data cube from  $P$  and check whether there is a solution there (lines 5–13). If there is no solution there, then we choose to remove a data cube  $p$  from  $P$  (i.e.,  $P = P - \{p\}$ ) if the resulting set ( $P - \{p\}$ ) of data cubes gives us a minimum query cost among all others (line 11). The above is repeated until either the query-cost bound is satisfied (line 9) or all data cubes are removed, indicating that the query-cost bound is too strict (line 14). The iteration of loop in lines 5–13 goes on until a solution is found or until no solution is found even after removing all data cubes.

```

/* input: L, Q, MCB, QCB; output: P, C */
1) j = |Q|;
2) P = Q;
3) C = AO(L, P, MCB)
4) if (QC(P, C) ≤ QCB) return P, C;
5) while(j ≥ 1) {
6)   for all (p ∈ P) do {
7)     C = AO(L, P - {p}, MCB)
8)     if (C ≠ ∅)
9)       if (QC(P - {p}, C) ≤ QCB) P = P - {p}, return P, C;
10)   }
11)   P = P - {p} which gives the minimum QC(P - {p}, C);
12)   j = j - 1;
13) }
14) print "The bounds are too strict."; return ∅, ∅;

```

Figure 6. Algorithm Greedy Removing.

For example, suppose the initial set  $Q$  contains data cubes  $T, U, V$  and  $W$ . In the beginning, we consider whether keeping all the data cubes  $\{T, U, V, W\}$  can give a solution. If not, we remove one data cube from the initial set and check whether there is a solution from any of the following combinations:  $\{U, V, W\}, \{T, V, W\}, \{T, U, W\}, \{T, U, V\}$ . If a solution is still not found and, say  $\{U, V, W\}$  gives a minimum query cost, then we can try  $\{V, W\}, \{U, W\}, \{U, V\}$ , and so on.

If the number of data cubes in the initial data cube set is  $n$ , the number of ways of removing one data cube from them is  $n$ . After we have removed one data cube, the number of ways of removing another data cube is  $n - 1$ , etc. With respect to the number of attribute-level optimization done, in the worst case that only one or no data cube remains, the time complexity of GR is  $1 + n + (n - 1) + \dots + 3 + 2 = O(n^2)$ . If a solution is found after  $m$  data cubes are removed, then the number of attribute-level optimization done is  $1 + n + (n - 1) + \dots + (n - m + 1)$ , i.e., it is sub-quadratic to  $n$ . We will see later that the execution time of one execution of attribute-level optimization decreases when there are fewer data cubes. Thus, when more data cubes are removed, the execution time of one execution of attribute-level optimization becomes less.

Greedy Removing is very efficient compared with Optimal Removing. It is also very effective, as we will see in the performance study section.

#### 4. Attribute-level optimization

The query-level optimization was described in the previous section. In this section we discuss how we optimize a given set of data cubes at the attribute level. Given a data cube set  $P$ , optimization at the attribute level is to refine the set  $P$  to  $C$  so that the sum of the query cost of using  $C$  to answer all queries associated to  $P$  is minimum while the maintenance cost is within the maintenance-cost bound. In other words, given a lattice  $L$  of data cubes constructed from all the subsets of attributes, the objective at the attribute level is to find a set of data cubes in  $L$  so that each query can be answered by one of the data cubes in the set, the maintenance cost does not exceed the limit  $MCB$ , and the total query cost is minimum.

Essentially, this means that we need to divide the data cubes in  $P$  into a number of groups. Data cubes in each group are then merged into one single data cube.

A straight-forward brute-force approach would try all possible groupings such that the total maintenance cost of data cubes resulting from the merging of data cubes within the groups does not exceed the maintenance-cost bound. We call this approach *Optimal Merging (OM)* (Section 4.1). However, this approach is too time-consuming.

Another approach is *2-Greedy Merging (2GM)* (Section 4.2). Each time, we choose two data cubes to merge. The selection criteria is that the decrease in maintenance cost per unit increase of query cost caused by merging two data cubes is maximum. The merging process is repeated until the total maintenance cost of data cubes is within the maintenance-cost bound. This approach is very efficient, but the solution is not always optimal.

The third approach is *2-Greedy Merging with Multiple paths (2GMM)* (Section 4.3). It is a multiple-path variation of 2GM. In 2GM, only one pair of data cubes are chosen, and the following merging will be based on that result. However, we do not know whether that pair is really the correct choice for reaching an optimal solution. Therefore, we could keep track on more than one way of selection of a pair of data cubes for merging at the same time. Based on each of the several ways of selection, we repeat recording of more than one way of selection and merging until we find a solution with the lowest query cost. If we consider the process of 2GM as a *path* from a given data cube set to obtain a solution data cube set, then many paths are created for our consideration in 2GMM. The exhaustive way to try all possible paths can get an optimal solution, but it is too time-consuming, so we will restrict the number of paths so that the execution time is acceptable.

#### 4.1. Optimal Merging

In the beginning, given a set  $P$  of data cubes, there are many different ways of merging in the process of finding the final set  $C$  of data cubes that satisfies the maintenance-cost bound. In order to find an optimal solution (with the minimum query cost), instead of trying all different ways of merging, what we need is actually the answer to the following question: in an optimal solution, how are the data cubes grouped into groups? The process of merging and formation of intermediate merged data cubes is not important. Optimal Merging (OM) considers all ways of groupings in a brute-force manner. We outline OM in figure 7.

In line 1,  $TOP(L)$  returns the top node of the lattice  $L$ , which contains all attributes, and so this data cube has the largest size.  $C$  is initialized to contain the top node first. In line 2,

```

/* input: L, P, MCB; output: C */
1) C = TOP(L);
2) G = AllGroupings(P);
3) for all H ∈ G, do {
4)   if MC(C) ≤ MCB
5)     if (QC(P, H) < QC(P, C)) C = H;
6)   }
7) if (MC(C) ≤ MCB) return C;
8) return ∅;

```

Figure 7. Algorithm Optimal Merging.

the function  $AllGroupings(P)$  considers all possible groupings of data cubes in  $P$ . For each grouping, data cubes in every group are merged into one data cube. The resultant sets of data cubes of all groupings are returned to  $G$  (i.e.,  $G$  is a set of data cube sets). Therefore, each element  $H \in G$  contains a set of data cubes. In lines 3–6, we run the loop to consider every data cube set  $H$  in  $G$ . If its maintenance cost is within the maintenance-cost bound and its query cost (the cost to answer all queries using  $H$ , i.e.,  $QC(P, H)$  by Formula 2.1) is smaller than that of  $C$  ( $QC(P, C)$ ), then we assign  $H$  to  $C$  (line 4–5). Finally we return the set of data cubes with the minimum query cost (line 7). If the maintenance-cost bound is so strict that there is no data cube set that satisfies the bound, an empty set is returned (line 8).

If there are initially  $n$  data cubes, then in the worst case, the sum of the total number of ways of grouping the  $n$  data cubes into  $m$  groups (where  $m = 1, 2, \dots, n$ ) is  $B_n$ , which is called *Bell number*. A coarse approximation of  $B_n$  is  $O(n^n)$ . More details can be found in Hung (2000).

It is obvious that the time complexity is very large. Therefore, two more efficient methods, 2-Greedy Merging and 2-Greedy Merging with Multiple paths, should be considered. As we will see in the performance study, the goodness of the solutions of the two new algorithms are very close to that of an optimal solution.

#### 4.2. 2-Greedy Merging

In 2-Greedy Merging (2GM), starting from a given set  $P$  of data cubes, we try to merge data cubes step by step in a bottom-up 2-Greedy approach, i.e., every time we only choose two data cubes to merge. The outline of 2GM is shown in figure 8.

In each loop (lines 2–7) of the algorithm, we select two data cube sets:  $D$  with two data cubes and  $A$  with one data cube. The data cube sets are chosen to have a maximum value of an evaluation function  $\alpha$ . The evaluation function will be discussed later. The data cube in  $A$  is obtained from merging the two cubes in  $D$ . The data cubes in  $D$  are removed from  $C$ , and the data cube in  $A$  is added into  $C$  such that the data cubes in  $P$  can be derived from the new  $C$ . We call the updated set  $C^+$  ( $C^+ = (C - D) \cup A$ ). Also, each data cube  $p$  in the input set  $P$  can be derived from a data cube in  $C^+$ . Therefore, the new maintenance cost  $MC(C^+)$  might be less than the old one ( $MC(C)$ ) and the query cost would increase. The

```

/* input: L, P, MCB; output: C */
1) C = P;
2) while MC(C) > MCB do {
3)   if (|C| == 1) return  $\emptyset$ ;
4)   SelectCubes( $D \subseteq C, A \subseteq L$ ) such that
         $\alpha(C, D, A)$  is maximal,  $|D| = 2$  and  $|A| = 1$ ;
5)   /*  $\alpha$  is an evaluation function */
6)    $C = (C - D) \cup A$ ;
7) }
8) return C;
```

Figure 8. Algorithm 2-Greedy Merging.

algorithm terminates when the maintenance-cost bound is satisfied. If there is no solution when only one data cube remains, then an empty set is returned (line 3).

Readers may find that it is possible to choose a smaller data cube to add as  $A$ . If all queries can be answered by  $(C - D) \cup E$ , where  $E$  contains a data cube containing a subset of attributes of the data cube in  $A$ , then taking  $C^+$  as  $(C - D) \cup E$  rather than  $(C - D) \cup A$  could probably give a lower maintenance cost as well as a lower query cost. For example, consider merging two data cubes  $abc$  and  $cd$  to form a data cube  $abcd$  (i.e.,  $D = \{abc, cd\}$ ,  $A = \{abcd\}$ ). If  $ab$  is the only query that involves attribute  $a$ , assume that it was previously answered by  $abc$  and can be now answered by another *smaller* data cube  $abg$  rather than the data cube  $abcd$  (i.e.,  $S(abc) < S(abg) < S(abcd)$ ). As a result, we could remove attribute  $a$  from  $abcd$  and replace  $abcd$  by  $bcd$ , i.e.,  $E = \{bcd\}$ , and we could have a better  $C^+$ . There may be more than one attribute removable. In considering whether to remove an attribute from  $A$  (and if so, which attribute to remove), we need to compute the evaluation function  $\alpha$  once for every attribute considered. The effect will be the increase of a multiple times of the original execution time. The solution may be better, but it still suffers from the disadvantage of Greedy method: a data cube selected to merge with some other data cubes will be kept merged with those (plus some other data cubes) forever in later stages. On the other hand, another approach 2GMM (to be described in the next section) does not suffer from this problem and can more likely find better solutions while its execution time is still comparable with that of 2GM. Therefore, in order to make things simpler, we simply merge the data cubes in  $D$  and add the resulting data cube into  $C$ .

The algorithm begins with  $n$  data cubes. It first considers  ${}_n C_2$  ways of choosing a pair of data cubes for merging. There are  $n - 1$  data cubes left after the first iteration, and the algorithm considers  ${}_{n-1} C_2$  ways of merging two data cubes. It stops after the maintenance-cost bound  $MCB$  is satisfied. Assume that  $m$  data cubes are left, then the number of considerations of merging is  ${}_n C_2 + {}_{n-1} C_2 + \dots + {}_{m+1} C_2$ . In the worst case,  $m = 1$ , the time complexity of the algorithm is  ${}_n C_2 + {}_{n-1} C_2 + \dots + {}_2 C_2 = {}_{n+1} C_3 = O(n^3)$ , which is obviously much lower than the time complexity of Optimal Merging. Derivation can be found in Hung (2000).

The execution time of 2GM is very good, but it does not guarantee an optimal solution. The third algorithm 2-Greedy Merging with Multiple paths, which will be discussed in Section 4.3, gives a solution that is closer to an optimal solution, and gives an acceptable execution time.

**4.2.1. Evaluation function  $\alpha$ .** We define the evaluation function  $\alpha$  based on the goal of the attribute-level optimization: to achieve a minimum query cost within the maintenance-cost bound. Initially, using the input set  $P$  of data cubes ( $C = P$ ) to answer queries associated to  $P$ , the query cost is minimum. In each iteration, we want to choose a new  $C$  such that the decrease in maintenance cost per unit increase of query cost caused by merging two data cubes is maximum. Thus, we define our evaluation function  $\alpha$  by the following formula:

$$\alpha(C, D, A) = \frac{\text{DecreaseInMaintenanceCost}}{\text{IncreaseInQueryCost}} = \frac{MC(C) - MC(C^+)}{QC(P, C^+) - QC(P, C)}, \quad (4.4)$$

where  $C^+ = (C - D) \cup A$  is the new  $C$ . The numerator of Formula 4.4 is the saving in maintenance cost.

When the maintenance cost is defined using Formula 2.2, i.e., the total size of data cubes, then using Formulae 2.1 and 2.2,

$$\alpha(C, D, A) = \frac{\sum_{t \in D} S(t) - \sum_{t \in A} S(t)}{\sum_{p \in P} [S(F_{C^+}(p)) - S(F_C(p))]} \quad (4.5)$$

When the maintenance cost is defined using Formula 2.3, i.e., the number of data cubes, then using Formulae 2.1 and 2.3,

$$\alpha(C, D, A) = \frac{1}{\sum_{p \in P} [S(F_{C^+}(p)) - S(F_C(p))]} \quad (4.6)$$

The denominator of Formula 4.5 and 4.6 is the increment in the query cost.  $F_{C^+}(p)$  is the smallest data cube in  $C^+$ , which can answer  $p$ . Hence, the increment in the query cost for every  $p$  is  $[S(F_{C^+}(p)) - S(F_C(p))]$ . The corresponding Formula 4.5 or 4.6 is used when the maintenance cost is defined by Formula 2.2 or 2.3.

When the number of data cubes is taken as the maintenance cost, then Formula 4.6 is used as  $\alpha$ . The numerator is always 1 and the denominator (increase in query cost) can be positive or zero. The case we like most is that the increase of query cost is zero, then  $\alpha$  will become positive infinity. Otherwise, we would like to choose the case that the increase of query cost is small, i.e.,  $\alpha$  is large. Therefore, choosing the case with a maximum  $\alpha$  matches with the order of our priority of choices.

When the total size of data cubes is taken as the maintenance cost, then Formula 4.5 is used as  $\alpha$ . Since merging two data cubes may produce a data cube whose size is larger than the total size of its component data cubes, so the maintenance cost after merging two data cubes may increase rather than decrease as we expected. If we consider merging two data cubes  $u$  and  $v$  to produce a new data cube  $w$ , let their sizes be  $S(u)$ ,  $S(v)$ ,  $S(w)$  respectively, then,

$$\alpha(C, \{u, v\}, \{w\}) = \frac{S(u) + S(v) - S(w)}{\sum_{p \in P} [S(F_{C^+}(p)) - S(F_C(p))]} \quad (4.7)$$

In general, the numerator can be positive, zero or negative while the denominator can be positive or zero. All the possible cases are listed below, in the order of priority with which we would like to choose for merging.

1. If there is no increase in the query cost, then a query answered by  $w$  must have the same query cost as it was previously answered by  $u$  or  $v$ .<sup>2</sup> Assume that at least one query previously answered by  $u$  and at least one by  $v$  are now answered by  $w$ ,<sup>3</sup> then the size of  $u$ ,  $v$  and  $w$  must be the same and there must be a decrease in the maintenance cost ( $S(u) = S(v) = S(w)$ , so  $S(u) + S(v) - S(w) > 0$ ), so  $\alpha$  is positive infinity.
2. There are an increase in the query cost and a decrease in the maintenance cost, so  $\alpha$  is positive.
3. If there is zero decrease in the maintenance cost, then there must be an increase in the query cost, so  $\alpha$  is zero.

4. If there is an increase in the maintenance cost, then there must be an increase in the query cost, negative.

We can see that the order of priority of our choices matches with our evaluation function  $\alpha$ .

#### 4.3. 2-Greedy Merging with Multiple paths

The third approach we consider here is 2-Greedy Merging with Multiple paths (2GMM). It is a multiple-path variation of 2-Greedy Merging (2GM). If we consider that 2GM is an algorithm to find a path from the state of a given data cube set  $P$  to a state of a data cube set  $C$  with maintenance-cost bound satisfied, and every intermediate state (of a data cube set produced from merging two data cubes) is an intermediate node in the path connecting the two states ( $P$  and  $C$ ), then 2GMM is an algorithm which gives two or more branches of the path at every intermediate node. Each branch continues to “grow” and branches at every later intermediate node. Meanwhile, at an intermediate node, it will check whether there is any solution (i.e., a data cube set satisfying the maintenance-cost bound), which can be produced by one more merging. If a solution is found, its query cost is recorded. A branch will stop growing if its query cost has already exceeded the temporarily recorded minimum. The solution with the minimum query cost is returned as the final solution. By controlling the number of paths carefully, this algorithm can produce a solution that is better than 2GM at the cost of a bit more execution time. The brief outline of 2GMM is shown in figure 9.

The algorithm is recursive. In the algorithm,  $P$  is the data cube set first input to the algorithm as well as the queries necessary to answer. The query cost is calculated as the cost of answering queries in  $P$ .  $R$  is the data cube set passed recursively for merging. In the beginning, we set  $R$  as  $P$  and execute the algorithm by calling  $2GMM(L, P, P, MCB)$ . First, it will check whether the maintenance-cost bound of  $R$  is satisfied. Besides, it will also check whether there remains one data cube only. If it is neither of the above cases, then it will execute the following iterations. In each loop (lines 4–11) of the algorithm,

```

2GMM (L, P, R, MCB) /* input: L, P, R, MCB; output: C */
1) if (MC(R) ≤ MCB) do return R;
2) if (|R| == 1) do return ∅;
3) C = ∅;
4) for (i = 1; i ≤ number of paths allowed; i++) do {
5)   SelectCubes(D ⊆ R, A ⊆ L) such that
        α(R, D, A) is i-th maximal, |D| = 2 and |A| = 1;
6)   /* α is an evaluation function */
7)   E = 2GMM(L, P, (R - D) ∪ A, MCB);
8)   if (C == ∅) do C = E;
9)   else if (E ≠ ∅)
10)      if (QC(P, E) < QC(P, C)) C = E;
11) }
12) return C;

```

Figure 9. Algorithm 2-Greedy Merging with multiple paths.

we select two data cube sets  $D$  and  $A$  with the  $i$ th maximum evaluation function value. The data cube in  $A$  is obtained from merging the two data cubes in  $D$ . The new data cube set is input into the algorithm recursively, which will return either a solution set of data cubes with the minimum query cost and the maintenance-cost bound satisfied or an empty set if a solution cannot be found under this path (and its subpaths). Then the algorithm will choose the solution with the minimal query cost among all solutions as our final solution.

The same evaluation function  $\alpha$  defined in 2GM (Formula 4.5 or 4.6) is also used in 2GMM.

We assume that the number of paths to be 2. The algorithm begins with  $n$  data cubes. It first considers  ${}_nC_2$  ways of choosing a pair of data cubes for merging. Two paths are created. There are  $n - 1$  data cubes left while calling the algorithm recursively, and each path considers  ${}_{n-1}C_2$  ways of merging two cubes. Four paths are created now. This repeats until the maintenance-cost bound  $MCB$  is satisfied. Assume that the maximum number of data cubes left among all paths is  $m$ , then the number of considerations of merging is  ${}_nC_2 + 2{}_{n-1}C_2 + \dots + 2^{n-m-1}{}_{m+1}C_2$ . The time complexity of the algorithm is  $O(2^{n-m}(m)^2)$ . In the worst case,  $m = 1$ , the time complexity is  ${}_nC_2 + 2{}_{n-1}C_2 + \dots + 2^{n-2}{}_2C_2 = 2^{n+1} - 1 - (n + 1) - {}_{n+1}C_2 = O(2^n)$ . Derivation can be found in Hung (2000). The time complexity is still smaller than that of OM. Usually,  $m$  is between  $n$  and  $\frac{n}{2}$ . If  $m$  is too small, the number of paths for greater depth should be restricted to be 1 for reducing the execution time. An important point is that the number of paths is flexible and could be adjusted in order to achieve acceptable execution time.

Theoretically, 2GMM provides better results than 2GM does because the data cube sets considered by 2GMM cover those of 2GM. Furthermore the number of data cube sets considered by 2GMM is also much more than that of 2GM. Experiment results indicates that optimal solutions may be produced within a few seconds of execution time.

## 5. Performance study

### 5.1. Experimental setup

Experiments were done to test the effectiveness and the efficiency of the different optimization algorithms proposed at the attribute level and the query level. We want to show that 2-Greedy Merging with Multiple paths (2GMM) and Greedy Removing (GR) produce a near-optimal solution efficiently.

Two data sets were used separately for our experiments. The first data set, *Hong Kong Census Data*, contains census data in Hong Kong with 62010 tuples. Each tuple originally has 56 attributes, among which 15 attributes were chosen with the larger ranges (tens to tens of thousands) and variations of values. We call the first data set *DS1*.

The second data set, *Forest Covertype Data*, contains information of forest cover type in four wilderness areas located in the Roosevelt National Forest of northern Colorado. The raw data were determined from *US Forest Service (USFS) Region 2 Resource Information System (RIS)* data and *US Geological Survey (USGS)* data. There are 581012 tuples, with 55 attributes. The queries were set to contain a subset of the first 10 attributes with the

largest ranges (tens to thousands). We call the second data set *DS2*. More details of the data sets can be found in Hung (2000).

The experiments were done on a Sun Enterprise Ultra 450 server with 4 UltraSPARC-II CPU running at 250 MHz, with 1GB RAM and four 4.1GB hard disks.

## 5.2. Attribute level

In our experiments of attribute-level optimization using DS1, experiments were done on 70 sets of queries, each for both of the definitions of total maintenance cost. The number of queries ranges from 5 to 14, while the total number of attributes involved in a query set ranges from 10 to 15. For DS2, the total number of attributes was fixed at 10. An experiment was done on each combination of query number and attribute number. The queries were generated in random so that the probability of a particular attribute appearing in a query is  $\frac{1}{3}$ .

For the case that the total maintenance cost was defined using Formula 2.2, i.e., the total size of data cubes, the maintenance-cost bound was defined to be 75% of the total maintenance cost of the input data cubes. In the case that the total maintenance cost was defined using Formula 2.3, i.e., the total number of data cubes, the maintenance-cost bound was generated randomly around three-tenths to eight-tenths of the total maintenance cost of the input data cubes (i.e., the query number). The same sets of queries have been used for both sets of experiments.

Three programs, Optimal Merging (OM), 2-Greedy Merging (2GM) and 2-Greedy Merging with Multiple paths (2GMM) were implemented. For 2GMM, the number of paths at each immediate state was defined as  $Max(2, (13 - QueryNumber - depth))$ , where *depth* was the depth of the path or recursion, i.e., the depth of the first level call of 2GMM was 0, the depth of the second level call of 2GMM (first recursion) was 1, etc. The number of paths was adjusted in that manner so that the execution time of 2GMM was within seconds and that the execution time of the program combining 2GMM and Greedy Removing (GR) was within minutes.

OM, 2GM and 2GMM were run 140 times in total: 70 sets of queries on DS1 and DS2, with each of the definition of total maintenance cost. Two query sets (one with 12 attributes and 7 queries, the other with 13 attributes and 7 queries) in DS1 with the total maintenance cost defined by Formula 2.2 do not have solutions due to the too strict maintenance-cost bound.

By analyzing the results of the remaining 138 experiments details in Hung (2000), we have the following observation on OM, 2GM and 2GMM at the attribute-level optimization.

If the query cost of answering the given queries using the data cubes refined from a program (other than OM) is  $a$ , and that from OM is  $b$ , then the *performance ratio (PR)* of that program is defined as  $\frac{a}{b}$ . Solutions obtained from 2GM and 2GMM were very close to those of OM (the performance ratios are very close to 1). For 2GM, over 78% of all cases gave optimal solutions. The remaining cases have PR within 1.2. For 2GMM, over 95% of all cases gave optimal solutions. The remaining cases have PR within 1.02. Therefore, we conclude that 2GMM and 2GM are very effective approximate algorithms.

Table 1. Table of execution time of 2GM, 2GMM and OM (in sec.) ( $MC_1$  by Formula 2.2) in the data set DS1.

Program	Query number									
	5	6	7	8	9	10	11	12	13	14
2GM	0.007	0.012	0.013	0.028	0.089	0.067	0.143	0.285	0.200	0.245
2GMM	0.040	0.048	0.255	0.155	0.677	0.200	0.403	0.688	0.357	0.837
OM	0.01	0.01	0.15	0.70	5.10	27.8	185.7	1444	9100	159019

On the other hand, 2GM and 2GMM are very efficient. The average values of execution time of 2GM and 2GMM are 0.1102s and 0.5852s respectively. For example, Table 1 shows the trends of the actual execution time of the three programs with the total maintenance cost defined using  $MC_1$  by Formula 2.2 and the data set DS1. The average value of execution time for each query number is shown. The execution time of the three programs with other combinations of total maintenance cost definition and data set is very similar to that one. The experimental results indicate that the execution time of OM, 2GM and 2GMM are not directly related to the number of attributes. The execution time can be calculated approximately by the formulae in Tables 2 and 3, where  $q$  is the number of queries. The experimental results verify the theoretical analysis of time complexity of the three algorithms discussed previously. The execution time of OM is exponential to the query number. The execution time of 2GM and 2GMM depends on the number of mergings that were done to find a solution. Besides, the execution time of calculating the query cost during merging also increases with the query number since there are more queries to calculate the query cost and it takes longer for each query  $p$  to find the smallest data cube ( $F_C(p)$ ) that can answer it. It is nearly quadratic to the query number. Eliminating the effect of increasing execution time of calculating query cost, the execution time of 2GM is sub-cubic to the query number. The execution time of 2GMM is also exponential to the query number. However, 2GMM grows much more slowly than OM and the execution time is still acceptable. For example, when the query number was 20, which is the condition used in experiments of query-level optimization, the execution time of 2GMM and 2GM were within seconds.

Table 2. Table of approximation formulae of execution time of OM, 2GM, 2GMM (in sec.) in the data set DS1.

Program	OM	2GM	2GMM
$MC_1$ (Formula 2.2)	$(3.9 \times 10^{-7}) \times 6.3^q$	$(2.9 \times 10^{-5}) \times q^{3.5}$	$0.012 \times 1.4^q$
$MC_2$ (Formula 2.3)	$(1.7 \times 10^{-6}) \times 4.8^q$	$(3.7 \times 10^{-5}) \times q^{3.4}$	$0.014 \times 1.5^q$

Table 3. Table of approximation formulae of execution time of OM, 2GM, 2GMM (in sec.) in the data set DS2.

Program	OM	2GM	2GMM
$MC_1$ (Formula 2.2)	$(6.1 \times 10^{-8}) \times 7.4^q$	$(1.8 \times 10^{-5}) \times q^{3.6}$	$0.0014 \times 1.6^q$
$MC_2$ (Formula 2.3)	$(2.7 \times 10^{-6}) \times 5.9^q$	$(7.0 \times 10^{-5}) \times q^{3.0}$	$0.0041 \times 1.6^q$

Therefore, 2GMM (and 2GM) is thus a very efficient and effective algorithm for attribute-level optimization with the total maintenance cost defined using  $MC_1$  by Formula 2.2 or  $MC_2$  by Formula 2.3.

We remark that when the total maintenance cost is defined as the total data cube size ( $MC_1$  by Formula 2.2), it is possible that solutions do not exist for some queries and maintenance-cost bounds because the bounds are too strict.

### 5.3. Query level

In the experiments of query-level optimization, the total number of attributes involved was set to 15 for DS1 and 10 for DS2. The number of queries was set to 20. For the case that the total maintenance cost was defined using Formula 2.2, (i.e., the total size of data cubes), the maintenance-cost bound was set to 75% of the total size of the initial set of data cubes. For the case that the total maintenance cost was defined using Formula 2.3, (i.e., the total number of data cubes), the maintenance-cost bound was fixed to be 9 (DS1) or 10 (DS2). The same set of randomly generated queries have been used for all experiments.

In this section, we consider query-level optimization using Greedy Removing (GR) and Optimal Removing (OR). It is possible that both approaches remove the same queries. However, since our goal is to satisfy the query-cost bound and the maintenance-cost bound by removing a minimal subset of queries, it is possible (and likely) that there exist more than one minimal subset (with the same number of queries) whose removal can produce a solution satisfying the bounds.

The previous section shows that 2-Greedy Merging with Multiple paths (2GMM) performs very well for the attribute-level optimization in terms of the performance ratio as well as the execution time. Thus, we chose it as the attribute-level optimization subroutine in our experiments of query-level optimization.

For DS1, 21 experiments were done with a range of query-cost bound on each definition of total maintenance cost respectively. The query-cost bounds range from 600000 to 1000000 with intervals of 250000. For DS2, 24 experiments were done with a range of query-cost bound on each definition of total maintenance cost respectively. The query-cost bounds range from 8250000 to 25000000 with intervals of 2500000.

Among 89 out of all the 90 experiments (details in Hung (2000)), the number of queries removed by GR and OR is the same for each query-cost bound. In the remaining case, the difference between the number of queries removed by GR and that of OR is only one. Therefore, we see that Greedy Removing is very effective, using either definition of total maintenance cost.

When the query-cost bound is lowered, the number of queries kept decreases (more queries are removed) and the execution time grows with the number of queries removed. The execution time of GR is within several minutes and is sub-linear to the number of queries removed. On the other hand, the execution time of OR is exponential to the number of queries removed (from several minutes to several hours).

From the previous discussion, we see that the combination of 2GMM (for attribute-level optimization) and GR (for query-level optimization) is a very good choice for solving the data cube system design optimization problem.

## 6. Discussion

### 6.1. Extension to queries with different weights

Previously we have assumed that the frequencies or the weights of all queries are identical. Now we consider the case that queries have different weights. We can normalize the weights of all queries  $p$  in a query set  $P$  so that the sum of the normalized weights  $w_p$  of all queries is 1. i.e.,  $\sum_{p \in P} w_p = 1$ .

We can modify the definition of  $QC(P, C)$  (Formula 2.1) to the following:

$$QC(P, C) = \sum_{p \in P} (w_p S(F_C(p))). \quad (6.8)$$

$QC(P, C)$  is the total weighted query cost of answering the queries associated with  $P$  by using the data cubes in  $C$ . If a query has a higher occurrence frequency, we assign it a larger weight. Our new query-cost formula will let the query with larger weight to contribute more to the total weighted query cost. Trying to lower the query cost of that query will lower the total weighted query cost more. The same evaluation function (Formulae 4.4–4.6) can be used in the attribute-level optimization. Further research work can be done on experiments using the above definition of query cost.

Recall that the aim of the query-level optimization is to remove a minimum number of queries so that the maintenance cost and the query cost are within bounds. Now, since each query has its weight, it may be more reasonable to change the aim to remove a number of queries such that the sum of weights of those queries removed is minimum and that the maintenance cost and the query cost are within bounds. It is a much harder problem than the previous one. Simply using GR may not be a good choice. Further research can be done on pruning the search space and finding heuristics to minimize the number of execution of the attribute-level optimization subroutine.

### 6.2. Monotonicity of Greedy Removing

In this section, we consider whether the query cost is decreasing whenever GR is removing a query from a query set. When we say *the query cost of a query set*, we mean the cost to answer that query set using a data cube set obtained from the attribute-level optimization (subroutine).

In GR, when the solution obtained from the attribute-level optimization subroutine for a query set does not satisfy the query-cost bound, a query is chosen from the query set to be removed so that the new query set has the lowest query-cost among other query sets (obtained from removing one of other queries). If the *lowest* query cost (after removing a query, among all ways of removing a query) is always smaller than the previous query cost (before removing that query), then the query cost will continue to decrease in the iterations of GR and finally the query-cost bound can be satisfied (or in the extreme case, all queries have been removed). In other words, if there exists a query in any query set such that removing that query can result in a query cost smaller than that before, then the query cost is

strictly decreasing. From now on, we will borrow the term *monotonicity* to mean “*strictly decreasing of query cost*.”

We have proved the monotonicity of GR using OM as the attribute-level optimization subroutine. Besides, when 2GM is used as the attribute-level optimization subroutine and the total maintenance cost is defined as the total number of data cubes (Formula 2.3), we have also proved that the monotonicity of GR is guaranteed in the following conditions:  $m = 1$  or  $n \leq 7$  or  $m \geq \frac{n-3}{2}$ , where  $n$  is the number of data cubes and  $m$  is the maintenance-cost bound. In practice, our experimental results show that GR performs very well in general. Before we execute 2GM, given the maintenance cost cannot tell us the number of data cubes remaining after the execution of 2GM. Thus, we cannot use the proof used for Formula 2.3. The details including the theorems and proofs can be found in Hung (2000). The property of 2GMM is between those of OM and 2GM.

### 6.3. Other approaches

We have studied and implemented other approaches including: other evaluation functions, set-partitioning algorithms, and genetic algorithms.

For other evaluation functions, we have considered (1) the number of common attributes of data cubes to be merged, and (2) the increase in query cost. However, since there is no direct relationship between the number of common attributes and the size of data cubes, the performance of using the number of common attributes generally will not be better than that of our evaluation function. Furthermore, we found in our experiments that the performance of the new evaluation functions was not as good as that of our evaluation function. In short, our original evaluation function is a good and simple one, compared with some others.

For the attribute-level optimization, another totally different approach is set-partitioning algorithm. Initial data cubes are partitioned into a number of sets. Data cubes in each set are merged into a data cube. A complete set-partitioning algorithm, for example, “implicit enumeration algorithm” is equivalent to our Optimal Merging since they search through all possible data cube sets. The algorithm can be modified to perform a partial search, for example by Greedy method using a cost function.<sup>4</sup> A multiple-path version of the partial search algorithm can also be considered. However, we have done experiments, which show that the results are much worse than that of 2GMM in execution time as well as the goodness of solutions produced. It is important to find a good cost function so that a good solution can be obtained within a reasonable execution time. Another approach for the attribute-level optimization is to use genetic algorithm. However, it suffers from the lack of an execution time bound and too long execution time. Besides, further research is needed on the use of crossover and mutation operators in our problem.

Details can be found in Hung (2000).

### 6.4. Changes in the set of frequently-asked queries

Although users of an OLAP system or a decision support system usually submit typical queries, which form a set of frequently-asked queries that we can use to produce a data cube

set to answer them efficiently, it is likely that the need of users will change gradually. However, it's unlikely that the set of frequently-asked queries are changed enormously within a short period time. It is more likely that the frequency of data cube maintenance is higher than the frequency of major changes in the set of frequently-asked queries. Our assumption is also taken by researchers of most other related work. Therefore, one approach to respond to the gradual change of users' need is to use our algorithms to obtain a new set of data cubes according to the new set of frequently-asked queries whenever it is the time to update the data in data cubes. On the other hand, when there is a slight change in the users' need, for example, an original query is replaced by a new query, first we can check whether the total query cost of answering this new query set using the original data cube set exceeds the query-cost bound or not. If not, no change is needed. Otherwise, depending on how much the bound is exceeded, the system designer can try to relax the bounds (depending on the available resources). When the system designer decides that the data cube set needs changes, he/she can run the algorithms with the new set of queries and bounds, or just a subset of queries (containing the queries affected or answered by the data cube which is used to answer the replaced query, etc.) and bounds (that only include the queries in the above subset). If the result of the algorithms indicates that the number of queries that can be answered by the new data cube set is the same as the old set (i.e., the old set is already a solution), then we can simply keep the old set. Otherwise, only new data cubes need to be computed. All the above are usually expected to be done off-line or at night without affecting the users.

## 7. Conclusion

In this paper, we discussed the optimization problem in requirement-based data cube system design subject to a maintenance-cost bound and a query-cost bound. We proposed a two-phase approach to the problem. In the first phase, we derive an initial data cube set from a set of frequently asked queries. In the second phase, we answer a maximum number of queries and optimize the initial data cube set to get a new set of data cubes that can satisfy both of the maintenance-cost bound and the query-cost bound. The second phase can be divided into two levels. At the query level, we remove as few data cubes from the initial set and answer as many queries as possible. At the attribute level, we refine the data cube set (obtained from the query level) to satisfy the maintenance-cost bound and the query-cost bound. Experiments have been done on two data sets and the results show that the combination of 2-Greedy Merging with Multiple paths (at the attribute level) and Greedy Removing (at the query level) gives a near-optimal solution. These algorithms are also very efficient.

## Notes

1. Studies and results for this definition have been reported in another paper (Hung et al., 2000). This paper considers both definitions.
2. A query answered by a data cube  $t \in C - \{u\} - \{v\}$  will not need to be answered by  $w$  because the size of  $w$  must not be smaller than  $t$ .

3. An extreme rare case is that all queries previously answered by  $u$  (similar for  $v$ ) can also be answered by data cubes in  $C - \{u\} - \{v\}$  with the same query cost so that after merging  $u$  and  $v$  to form  $w$ , all queries will not be answered by  $w$ , and so even if the size of  $w$  is not equal to that of  $u$ , the query cost still can be kept the same. In this case, we will not add  $w$  and simply remove  $u$  and  $v$ .
4. This cost function is used for calculating the cost of a partition while choosing a way of partitioning. It is not related to our query cost (Formula 2.1).

## References

- Agarwal, S., Agrawal, R., Deshpande, P.M., Gupta, A., Naughton, J.F., Ramakrishnan, R., and Sarawagi, S. (1996). On the Computation of Multidimensional Aggregates. In *Proc. of the 22th Very Large Database (VLDB) Conference*. Mumbai (Bombay), India.
- Agarwal, S., Chaudhuri, S., and Narasayya, V.R. (2000). Automated Selection of Materialized Views and Indexes in SQL Databases. In *Proc. VLDB* (pp. 496–505).
- Chaudhuri, S. and Dayal, U. (1997a). An Overview of Data Warehousing and OLAP Technology. *ACM SIGMOD Record*, 26(1).
- Chaudhuri, S. and Dayal, U. (1997b). Data Warehousing and OLAP for Decision Support. In *Proc. of ACM SIGMOD International Conference on Management of Data*. Tucson, Arizona, USA.
- Cheung, D.W., Zhou, B., Kao, B., Lu, H., Lam, T.W., and Ting, H.F. (1999). Requirement-Based Data Cube Schema Design. In *Proc. of Eighth International Conference on Information and Knowledge Management (CIKM)*. Kanas City, Missouri, USA.
- Cheung, D.W., Zhou, B., Kao, B., Lu, H., Lam, T.W., Ting, H.F., and Hung, E. (2000). Requirement-Based Data Cube Optimization. Under review.
- Deshpande, P.M., Naughton, J.F., Ramasamy, K., Shukla, A., Tufte, K., and Zhao, Y. (1997). Cubing Algorithms, Storage Estimation, and Storage and Processing Alternatives for OLAP. *IEEE Data Engineering Bulletin*, 20(1).
- Gray, J., Bosworth, A., Layman, A., and Pirahesh, H. (1996). Data Cube: A Relational Aggregation Operator Generalizing Group-by, Cross-Tab, and Sub-Total. In *Proc. of the 12th Intl. Conference on Data Engineering* (pp. 152–159). New Orleans, USA.
- Griffin, T. and Libkin, L. (1995). Incremental Maintenance of Views with Duplicates. In *Proc. of ACM SIGMOD International Conference on Management of Data*. San Jose, California.
- Harinarayan, V., Rajaraman, A., and Ullman, J.D. (1996). Implementing Data Cubes Efficiently. In *Proc. of the ACM SIGMOD Conference on Management of Data* (pp. 205–216). Montreal, Quebec.
- Hung, E. (2000). Data Cube System Design: An Optimization Problem. M.Phil. Thesis, the University of Hong Kong. “<http://www.cs.umd.edu/~ehung/paper/thesisabstract.ps>”; “<http://www.cs.umd.edu/~ehung/paper/thesis.ps>”.
- Hung, E., Cheung, D.W., Kao, B., and Liang, Y.L. (2000). An Optimization Problem in Data Cube System Design. In *Proc. of the Fourth Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD 2000)*. Kyoto, Japan.
- Inmon, W.H. (1996). *Building the Data Warehouse*, 2nd edn. John Wiley & Sons.
- Karloff, H. and Mihail, M. (1999). On the Complexity of the View-Selection Problem. In *Proc. of the Eighteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'99)*. Philadelphia, Pennsylvania, USA.
- O’Neil, P. and Graefe, G. (1995). Multi-Table Joins Through Bitmapped Join Indexes. In *SIGMOD Record* (pp. 8–11).
- O’Neil, P. and Quass, D. (1997). Improved Query Performance with Variant Indexes. In *Proc. of the ACM SIGMOD Conference on Management of Data* (pp. 38–49). Tucson, Arizona.
- Shukla, A., Deshpande, P.M., and Naughton, J.F. (1998). Materialized View Selection for Multidimensional Datasets. In *Proc. of the International Conference on Very Large Databases* (pp. 488–499). New York, USA.
- Shukla, A., Deshpande, P.M., Naughton, J.F., and Ramasamy, K. (1996). Storage Estimation for Multidimensional Aggregates in the Presence of Hierarchies. In *Proc. of the 22th Very Large Database (VLDB) Conference*. Mumbai (Bombay), India.

- Theodoratos, D., Ligoudistianos, S., and Sellis, T.K. (1999). Designing the Global Data Warehouse with SPJ Views. *CAiSE* (pp. 180–194).
- Theodoratos, D., Ligoudistianos, S., and Sellis, T.K. (2001). View Selection for Designing the Global Data Warehouse. *DKE*, 39(3), 219–240.
- Transaction Processing Performance Council. (1997). TPC Benchmark D(Decision Support), Standard Specification, Revision 1.2.3. San Jose, CA, USA.
- Zaharioudakis, M., Cochrane, R., Lapis, G., Pirahesh, H., and Urata, M. (2000). Answering Complex SQL Queries Using Automatic Summary Tables. In *Proc. of ACM SIGMOD International Conference on Management of Data*. Dallas, Texas, USA.