Abstract

On-line analytical processing (OLAP) requires efficient processing of complex decision support queries over very large databases. It is well accepted that pre-computed data cubes can help reduce the response time of such queries dramatically. A very important design issue of an efficient OLAP system is therefore the choice of the right data cubes to materialize. We call this problem the data cube schema design problem. In this paper we show that the problem of finding an optimal data cube schema for an OLAP system with limited memory is NP-hard. As a more computationally efficient alternative, we propose a greedy approximation algorithm cMP and its variants. Algorithm cMP consists of two phases. In the first phase, an initial schema consisting of all the cubes required to efficiently answer the user queries is formed. In the second phase, cubes in the initial schema are selectively merged to satisfy the memory constraint. We show that cMP is very effective in pruning the search space for an optimal schema. This leads to a highly efficient algorithm. We report the efficiency and the effectiveness of cMP via an empirical study using the TPC-D benchmark. Our results show that the data cube schemas generated by cMP enable very efficient OLAP query processing.

Keywords: Data cubes, Data cube schema design, OLAP, DSS.

1 Introduction

With wide acceptance of the data warehousing technology, corporations are building their decision support systems (DSS) on large data warehouses. Many of these DSS’s have on-line characteristics and are termed On-line Analytical Processing (OLAP) systems. Different from the conventional database applications, a DSS usually needs to analyze accumulative information. Very often, the system needs to scan almost the entire database to compute query answers, resulting in a very poor response time. Conventional database techniques are simply not fast enough for today’s corporate decision process.

The data cube technology has been becoming a core component of many OLAP systems. Data cubes are pre-computed multi-dimensional views of the data in a data warehouse [5]. The advantage of a data cube system is that once the data cubes are built, answers to decision support queries can be retrieved from the 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. The bottom level of an OLAP system is a data warehouse built on top of a DBMS. Data in the warehouse comes from source operational databases. (In the simplest case, the data warehouse could be the DBMS itself.) The warehouse needs to support fast aggregations, for example, by means of different indexing techniques such as bit-map indices and join indices [11, 12].

The middle level of an OLAP system is a set of data cubes, generated from the data warehouse. These cubes are called base data cubes. Each base cube is defined by a set of attributes taken from the warehouse schema. It contains the aggregates over the selected set of attributes. Other aggregates can be computed from the base cubes. The set of base data cubes together define a data cube schema for the OLAP system.

The top level of an OLAP system is a query client. The client, besides supporting DSS queries, allows users to browse through the data it caches from the data cubes. Therefore, a query could be a very complicated DSS query or a simple slicing and dicing request. A query submitted to the query client, after being checked against the data cube schema, will be directed to the
computed results, servicing queries with the cubes is much faster than with the warehouse.

Various developments and research studies have been made on the design of the three levels in an OLAP system. Many commercial products are also now available. Some example query clients include Seagate Info Worksheet [13] and Microsoft PivotTable Services [10]. Currently, these query client products mainly provide browsing and report generation services on cached data. In general, unless the answer is already cached, complex DSS queries submitted to the client module will have to be compiled into accesses to the data cubes or to the warehouse. For the warehouse and the data cube levels, there are products like Microsoft SQL OLAP Services [10], Hyperion Essbase OLAP Server [7], and IBM DB2 OLAP Server [8]. At the warehouse level, many vendors have been enhancing their DBMS products to improve the performance on data aggregation [6]. As for the data cube level, most of the research studies focus on two issues: (1) how to compute aggregates from a base cube efficiently [1], and (2) what data structures should be used to represent the cubes, that is the debate between relational OLAP (ROLAP) and multidimensional OLAP (MOLAP) [1, 16].

As we have mentioned, the OLAP system would be able to support real-time responses if the cube level can intercept (or answer) all the queries. Unfortunately, materializing all possible cubes so that all possible queries can be answered by the cubes is clearly impractical due to the high storage and maintenance costs. Instead, one should carefully choose the right combination of cubes so that query response time is optimized subject to the constraints of the system's capacity (such as storage space). We call the set of materialized base cubes the data cube schema of the OLAP system. We also call the problem of selecting a data cube schema the data cube schema design problem.

The key to the design of a query-efficient OLAP system thus lies on the design of a good data cube schema. In particular, two very important questions one needs to address are: on what basis shall we design such a schema? And where should the schema be derived from? We claim that the data cube schema should not be based solely on the database schema in the warehouse. Instead, a practical approach to the cube design problem should be based on the users' query requirements. For example, in the TPC-D benchmark [15], the requirement is to answer the 17 DSS queries that are specified in the benchmark efficiently. This is because these queries presumably are driven from the applications that use the data warehouse most often. Given the user query requirements (i.e., a set of frequently asked queries) and a set of system capacity constraints (e.g., storage limitation), our goal is to derive a data cube schema that optimizes query response time without violating the system capacity constraints.

As we will see in Section 2.4, we prove that the optimization problem is NP-hard. Finding the optimal data cube schema is thus computationally very expensive. As an alternative, we propose an efficient greedy approximation algorithm CMP for the requirement-based data cube schema design problem. Our algorithm consists of two phases:

(1) Define an initial schema

The first phase is to derive an initial set of data cubes (called initial schema) from the application requirements. In this study, we assume that the requirements are captured by a set of frequently-asked queries or FAQs. The initial schema are selected such that all the FAQs can be answered directly and efficiently. In the TPC-D example, we can define a cube to answer each one of the 17 DSS queries. For example, the 7th query of the TPC-D benchmark involve three attributes: suppnation, custnation, and shipdate yr. (2) Schema Optimization

The second phase is to modify the initial schema so that query response time is optimized subject to the system’s capacity constraints. The data cubes derived in an initial schema may have lots of redundancy caused by overlapping attributes. The total size of the cubes may exceed the storage or memory limitation of the system. Too many cubes would always induce a large maintenance cost when the data in the underlying warehouse changes. Therefore, it may be more cost-effective to merge some of cubes. Cube merging may result in fewer but perhaps larger cubes. In terms of query response time, query processed using the merged cubes will in general be slower than using the original cubes. Hence, there is a trade-off between query performance and cube maintenance. Schema optimization is to determine a set of data cubes that replace some of the cubes in the initial schema such that the query performance on the resulted cubes is optimal under the constraint that the total size of the resulted cubes is within an acceptable system limit. The set of data cubes obtained from the optimization process is called the optimal schema for the OLAP system.

The rest of the paper is organized as follows. In Section 2 we present a formal definition of the schema optimization problem. Section 3 introduces the greedy algorithm CMP for schema optimization. A performance study of CMP is presented in Section 4. We use data from the TPC-D benchmark in the study. Finally, we conclude our paper in Section 5. Because lack of space, some details are omitted. Readers can refer to [4] for a

\[1\] It is reasonable to correlate the maintenance cost with the total size of the cubes in the schema.
2 Schema Optimization

2.1 Search space of an optimal schema

In this paper we assume that the requirements of the OLAP system is captured in a set of frequent queries. We use $Q$ to denote the initial schema of the data cubes derived from the queries. The second phase of our cube design process is to refine the set $Q$ so that the maintenance cost (such as storage) of the cube set is within the capacity of the system. We can consider the refinement as an optimization problem with the set of all possible cube sets as the search space.

To simplify the problem, we assume that the database in the data warehouse is represented by a star schema [2]. Attributes in the queries come from the fields of the dimensions and fact tables. Usually, the number of dimensions and fact tables is not large. However, there may be many attributes in one table. For example, the table `part` includes the attributes `partkey`, `brand`, `type`, `size`, `container`, etc. The dimension is in fact a multi-hierarchical dimension as shown in Figure 1. In addition, in the star schema, some attributes are stored directly in the fact table. For example, the attributes `shipdate`, `commitdate`, `receiptdate` in the fact table `lineitem` are such attributes. As a result, the number of attributes (dimensions) needs to be considered in a cube design is much more than the number of dimension tables. In TPC-D, 33 attributes need to be considered.

In [9], the notion of a composite lattice is used to integrate multi-hierarchical dimensions with the lattice of aggregates in a data cube. Assume that $A = \{a_1, a_2, \ldots, a_n\}$ is the set of all attributes on which query can be posted. Any subset of $A$ can be used as the dimension attributes to construct a data cube. The composite lattice $L = (P(A), \prec)$ is the lattice of data cubes constructed from all the subsets of $A$. $(P(A)$ is the power set of $A.)$ The cube associated with the set $A$ is the root of the lattice $L$. For two different cubes $c_1, c_2 \in L$, the derived from relationship, $c_1 \prec c_2$, holds if $c_1$ can be derived from $c_2$ by aggregation. For example the 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. As has been mentioned, $n$ is large in general ($n = 33$ in TPC-D). The search space $L$ of the optimization problem is enormous.

2.2 Schema optimization

Given an initial data cube schema $Q$, a search space $L$, and a maintenance cost bound $\text{LIM}$, the schema optimization problem is defined in Table 1.

| Objective: Find $C \subseteq L$ such that $\text{Cost}(Q, C)$ is minimal |
| Constraint: $\forall q \in Q, \exists c \in C$ such that $q \prec c$ and $\text{MC}(C) \leq \text{LIM}$ |

Table 1: Schema Optimization Problem

The objective is to find a cube set $C$ such that the cost of answering the frequent queries, $\text{Cost}(Q, C)$, is the smallest. The constraint states that any frequent query $q$ can be answered by some cube $c$ in $C$ and that the total maintenance cost $\text{MC}(C)$ of the cube set is smaller than the system limit $\text{LIM}$. We will discuss various measures of $\text{Cost}$ and $\text{MC}$ shortly.

For simplicity, we assume that the frequent queries are equally probable. Since each cube in the initial schema $Q$ is derived from one distinct frequent query, we use the same symbol $q$ to denote both a cube in the initial schema and its associated frequent query. Since we do not want to make any assumption on the implementation of the cubes and the structure of the queries, a good measure of $\text{Cost}(Q, C)$ is the linear cost model suggested in [9]. In that model, if $q \prec c$, then the cost of computing the answer for a query $q$ using a cube $c$ is linearly proportional to the number of data points in $c$. We use $S(c)$ to denote the number of data points in $c$. For each query $q \in Q$, we use $F_C(q)$ to denote the smallest cube in $C$ that answers $q$. Formally, $F_C(q)$ is a cube in $C$ such that $q \prec F_C(q)$ and $\forall x \in C$, if $q \prec x$, then $S(F_C(q)) \leq S(x)$. We now define $\text{Cost}(Q, C)$ by:

$$\text{Cost}(Q, C) = \sum_{q \in Q} (S(F_C(q))).$$

Maintaining a data cube requires disk storage and CPU computation. Without assuming any implementation method, two measures can be used to estimate the maintenance cost $\text{MC}(C)$ of a cube set.

- $\text{MC}_1(C) = |C|$, i.e., the number of cubes in $C$.
- $\text{MC}_2(C) = \sum_{c \in C} S(c)$, i.e., the total number of data points in the cubes. This is an estimate of the total disk storage required.
To the best of our knowledge, this paper is the first to explore the data cube schema design problem. Several papers have been published on data cube implementation. Cube selection algorithms have been proposed in [9, 14]. These cube selection algorithms assume that there is one root base cube \( c_0 \) which encompasses all the attributes in the queries. They also assume that some queries are associated with this root base cube \( c_0 \); therefore \( c_0 \in Q \). Very different from the schema optimization problem, their selections always include \( c_0 \) in the answer, i.e., \( c_0 \in C \). However, in a general DSS such as TPC-D, we do not anticipate many frequent queries that involve all the attributes; hence, our cube schema design problem is more general.

Cube selection algorithms start from a base cube and determine what cubes deducible from it should be implemented so that queries on the aggregates in the base cube can be answered efficiently. Tackling a very different problem, cube schema design attempts to merge the cubes in an initial schema bottom-up to generate a set of cubes which provide an optimal query performance, while system capacity constraints are satisfied. The search space of the design problem is in general much larger because of the large numbers of attributes present in the initial schema. In short, cube selection algorithms are for cube implementation but not for cube schema design.

An interesting question is whether it is possible to modify the selection algorithm [9] to solve the schema design problem. One solution is to apply the selection algorithm on the maximal cubes of \( Q \), i.e., those that cannot be deduced from any other cube in \( Q \). In general, there are more than one maximal cubes in \( Q \). In order to adopt the selection algorithm, we can include all the maximal cubes in the answer set \( C \) as the initial members, and then expand \( C \) by applying the selection algorithms on them. The expansion stops when the total size exceeds the storage bound \( LIM \).

However, the above solution has a few undesirable problems. First, if the maximal cubes alone have already exceeded the maintenance bound, then the cube selection algorithm fails. Second, if some of the maximal cubes are highly correlated, e.g., with many overlapping attributes, then merging some of them could be beneficial and is sometimes even necessary. Selection algorithms, however, never merge cubes. For example, given a lattice \( L = (\{A,B,C,\cdot,\cdot,\cdot\}) \), suppose both cubes \( ABCD \) and \( BCDE \) are maximal, and \( S(ABCD)=S(ABCD) \), then using a selection algorithm, both \( AB \) and \( BCDE \) are selected. However, replacing them by the cube \( ABCDE \) decreases the maintenance cost without increasing the query cost. Hence, the selection algorithm is not always applicable to the cube schema design problem.

### 2.4 Complexity of the optimization problem

The schema optimization problem is computationally difficult. Here, we summarize its complexity in the following theorem. The proof of the theorem can be found in [4].

**Theorem 1** (1) Given an initial schema \( Q \), a search space \( L \), and a bound \( LIM \), the problem of finding a subset \( C \subseteq L \), such that \( C \) does not contain the root of \( L \), every \( q \in Q \) can be derived from some \( c \in C \) and \( |C| \leq LIM \) is NP-complete.

(2) Given a performance ratio \( r \), \( r > 1 \), to find an algorithm \( A \) for the Schema Optimization Problem defined in Table 1 whose performance is bounded by \( r \) times the optimal performance is NP-hard.

Theorem 1 tells us that the optimization problem is a very difficult one. In theory, it is impossible to find even an efficient approximation algorithm that can give a performance guarantee. In the next section, we will discuss a greedy approximation algorithm and discuss the heuristics it uses to prune the vast search space looking for a “good” solution. The efficiency and the effectiveness of the algorithm are studied in Section 4.

### 3 The Algorithm cMP

We have developed a greedy algorithm called cMP (cube Merging and Pruning). The outline of cMP is shown in Figure 2.

**Figure 2**: The algorithm cMP

```
/* input: L, Q, LIM; output: C */
1) C = Q;
2) while \( MC(C) > LIM \) do {
   3) SelectCubes(\( D \subseteq C, A \subseteq L \) - C)
      such that \( \alpha(C, D, A) \) is maximum;
   4) \( C = C \cup A - D \);
5) } return C;
```

During each iteration of the loop (lines 2 to 6) of algorithm cMP, we select two cube sets \( D \) and \( A \). The cubes in \( D \) are removed from \( C \), and the cubes in \( A \) are added into \( C \). The cube sets \( D \) and \( A \) are selected such that the cubes in \( Q \) can still be answered by the new \( C \). The algorithm terminates when the maintenance cost of the cube set no longer exceeds the limit \( LIM \).

The selection of the cube sets \( D \) and \( A \) is governed by the evaluation function \( \alpha \). The evaluation function is defined such that the reduction in the maintenance cost is large while the increment in the query cost is small.

In our algorithm, we use the following \( \alpha \) function:

\[
\alpha(C, D, A) = \frac{\text{SavingInMC}(C)}{\text{IncreaseInQueryCost}} = \frac{\sum_{t \in D} S(t) - \sum_{t \in A} S(t)}{\sum_{t \in D} n(t) - [S(FC(t)) - S(t)]}
\]
where $C^+ = C \cup A - D$ is the new $C$. The numerator of formula 2 is the saving in maintenance cost. (We have used $M_C$ in this formula. The results in the rest of the paper, unless stated explicitly, are also valid for $M_C$.) The denominator is the increment in the query cost. $F_C(t)$ is the smallest ancestor of $t$ in $C^+$.

3.1 Properties of the evaluation function $\alpha$

In cMP, the search space for $D$ and $A$ is enormous. In this subsection, we show some properties of the evaluation function which can be used to prune the search space effectively.

**Theorem 2** Suppose $L$ and $\alpha$ are defined as above, and $C$ is the set of cubes when cMP enters an iteration. Suppose that $D_s$ and $A_s$ are selected based on $C$ which maximizes the value of $\alpha$ over all $D \subseteq C$ and $A \subseteq L - C$. (If there are more than one combinations that give the maximum value, $D_s$ is the combination with the fewest cubes.) Then $D_s$ and $A_s$ must have the following properties.

1. If $|D_s| = 1$, then $A_s = \emptyset$.
2. If $D_s = \{b_1, b_2, \ldots, b_k\}$, $k > 1$, then the following is true:
   (a) $A_s = \{a\}$, and $a = F_L(\{b_1, b_2, \ldots, b_k\})$, which is the smallest common ancestor of $b_1, b_2, \ldots, b_k$ in $L$;
   (b) $\forall b_i, i = 1, 2, \ldots, k$, $S(F_C(b_i)) > S(a)$.

According to the above theorem, $A$ is determined by $D$ in case $\alpha$ attains its maximum value. Also, $A$ contains only the smallest common ancestor of the removed cubes — this significantly reduces the search space. Furthermore, item 2b of Theorem 2 makes the evaluation of many combinations of $D$ unnecessary.

**Corollary 1** If $D_s = \{b_1, b_2, \ldots, b_k\}$, $k > 1$, then $\forall i, j \leq k, i \neq j$, $b_i \prec b_j$ is not true.

Corollary 1 tells us that we do not need to consider a $D$ which contains a cube that can be derived from another cube in $D$. For such $D$, the corollary implies that the $\alpha$ value is not the maximum. We develop the procedure `SelectCubes` which uses this result to prune candidates in the search space of cMP:

1. Build a directed acyclic graph (DAG) of all the cubes in $C$ in which the edges are the derived from relationship $\prec$ among the cubes in $C$.
2. Partition the graph into disjoint paths. We partition the DAG by traversing the graph from a maximal node which has no ancestor towards a bottom node which has no descendant. The visited nodes (and their associated edges) are removed. We repeat the same procedure on the remaining nodes until all nodes are removed.

3. The nodes on the same path have a derived-from relationship. According to Corollary 1, no two nodes from the same path should be picked together for $D$. Hence, we pick at most one node from each path. In practice, the number of paths should not be large. This pruning significantly reduces the number of possible candidates of $D$ and hence $A$ from all possible combinations.

Following Theorem 2 and the path-based processing, we derive another pruning technique for further reduction of the search space.

**Corollary 2** Assume $C$ is partitioned into $p$ paths: $P_1, P_2, \ldots, P_p$, $(p > 1)$, where $\forall i = 1, 2, \ldots, p$, $P_i = \langle b_{i,1}, b_{i,2}, \ldots, b_{i,n_i} \rangle$, and $b_{i,s} \prec b_{i,s+1}$, $s = 1, \ldots, n_i - 1$.

Let $a_{k_1,k_2,\ldots,k_m} = F_L(\{b_{1,k_1}, b_{2,k_2}, \ldots, b_{m,k_m}\})$, $m \leq p$. If $\exists t \leq m$, such that $S(a_{k_1,k_2,\ldots,k_m}) \geq S(F_C(b_{t,k_t}))$, then for any set of cubes $D = \{b_{1,x_1}, b_{2,x_2}, \ldots, b_{t,x_t}, \ldots, b_{n,x_n}\}$, where $m \leq n \leq p$, and $x_i \geq k_i$, $i \leq m$ and $i \neq t$, $\alpha$ cannot attain the maximum value using $D$.

The corollary suggests that we can organize the `SelectCubes` procedure starting from the bottom of each path to compose candidates $D$ from the nodes. When the select procedure reaches a candidate (combination) that satisfies the condition of Corollary 2, then those yet-to-be-evaluated candidates of $D$ “above” the current combination in the lattice hierarchy can be ignored. We illustrate the pruning process with an example shown in Figure 3. The cube set $C$ is partitioned into 3 paths containing 3, 4, and 3 nodes respectively. We select the combination for $D$ from the bottom nodes of the paths: $b_{1,1}, b_{2,1}, b_{3,1}$. Suppose that when we evaluate the combination $D = \{b_{1,2}, b_{2,2}\}$, $S(a_{2,2}) \geq S(F_C(b_{1,2}))$ is true. According to Corollary 2, all the remaining combinations for $D$ which include $b_{1,2}$ do not need to be evaluated. These pruned combinations are: $\{b_{1,2}, b_{2,3}\}, \{b_{1,2}, b_{2,4}\}$, and all the 9 combinations of 3 cubes: $\{b_{1,2}, b_{2,3}, b_{3,y}\}$, where $x = 2, 3, 4, y = 1, 2, 3$. Eleven combinations are pruned in this case.

![Figure 3: An example of pruning in cMP](image-url)
Although the pruning methods introduced above is very effective for rMP, its complexity is still high when r is large. It is thus interesting to see how 2MP performs comparing with the more general rMP.

**Theorem 3** Suppose C is in a state when rMP(r > 2) is entering an iteration to identify new sets D and A. Assume that the maintenance cost function is $MC_2$. If, for all possible D’s which cannot be pruned away by either Corollaries 1 or 2, D satisfies the following condition:

$$S_a \geq \frac{1}{2(k-1)} \sum_{i=1}^{k} \sum_{j=1,j \neq i}^{k} S_{i,j}$$

(3)

where \(D = \{b_1, b_2, \ldots, b_k\}, (2 < k \leq r), S_a = S(F_L(D)), \text{ and } \forall i, j = 1, 2, \ldots, k, i \neq j, S_{i,j} = S(F_L(\{b_i, b_j\})), \text{ then the (D,A) pair selected by } rMP(r > 2), \text{ and 2MP are identical.}$$

Theorem 3 shows a sufficient condition that rMP(r > 2) and 2MP are equivalent. When we try our algorithm on some real data sets, the results obtained by 2MP is very close to rMP, even for some large values of r. The restricted but more efficient rMP algorithm (with a small r) is thus a viable choice in many occasions.

4 Performance study

We have carried out a performance study of the algorithms on a Sun Enterprise 4000 running Solaris 2.6. Our first goal is to study the “goodness” of the schemas generated by cMP and 2MP. The second goal is to study the efficiency of the two algorithms, and their pruning effectiveness.

We use the TPC-D benchmark data for the study. The database is generated with a scale factor of 0.5 [13]. The size of the database is about 0.5 GB. All the
17 DSS queries in the benchmark are frequent queries in our model.

4.1 Goodness of the schema generated

In the first experiment, we compare \( \text{Cost}(Q, C) \) of the outputs, \( C \), from both cMP and 2MP. The results are also compared with a random selection algorithm (labeled 2Rand) that randomly merges pairs of cubes iteratively until the maintenance cost limit is not exceeded. We use \( M C_1(C) \), the number of cubes in \( C \), to compute the maintenance cost for simplicity.

![Graph showing query cost of the schemas generated](image)

Figure 5: Query cost of the schemas generated

Figure 5 shows the result. The graph shows that cMP and 2MP are significantly better than random selection, in particular, when the cost limit is not too small so that there are more combinations for \( D \) for the algorithm to make a wise pick.

In our experiment, the query costs of cMP and 2MP are the same except that the curve for cMP has a gap at \( \text{LIM} = 10 \). When \( \text{LIM} = 10 \), cMP at one point has reduced the schema to 11 cubes. In the next reduction, 3 cubes are selected to be replaced by one cube; hence, the size of \( C \) becomes 9. In contrast, in each step, 2MP replaces no more than 2 cubes by another.

4.2 Efficiency of cMP

Our second goal is to study the efficiency of cMP. In particular, we are interested in studying their effectiveness in pruning the search space of the optimization problem.

The effect of the first pruning method that the newly added cubes \( A \) can be determined from the combination \( D \) is evident. Therefore, we only consider the other two pruning methods which are based on Corollaries 1 and 2. We measure the effectiveness by the average pruning rate, defined as the percentage of the pruned combinations of \( D \) over the number of all possible combinations.

\(^2\)The closeness of the two curves from cMP and 2MP is due to the extremely low correlations between the queries (cubes) in TPC-D.

The results are shown in Figure 9. From the figure, we see that the average pruning rate is higher than 80% in all the cases. The pruning rate becomes smaller while the maintenance cost limit decreases. This is because, with a small limit, cMP is forced to merge cubes that are at a higher level of the lattice hierarchy (Figure 3). Hence, the chance of pruning becomes smaller. In fact, due to the design purpose of the benchmark, the correlations among the initial set of cubes in TPC-D is quite low. The high pruning rate in our experiment shows that cMP is effective even in such a not-so-favorable situation. In many general applications, we expect that the frequently asked DSS queries will have high correlation; and the pruning rate of cMP will even be better than what is shown in our experiment.

![Graph showing average pruning rate of cMP](image)

Figure 6: Average pruning rate of cMP

Finally, we compared the efficiency of cMP and 2MP by measuring their response times. Figure 4.2 shows that 2MP is at least two order of magnitude faster than cMP. Considering also the effectiveness of 2MP (Figure 5), our results show that 2MP is an effective and efficient approximation solution to the data cube schema design problem.

![Graph showing response time of cMP and 2MP](image)

Figure 7: Response time of cMP and 2MP
Discussion and Conclusion

The basis of our 2-phase schema design approach is a set of cubes extracted from the query requirements. How valid is this approach? We have observed that some vendors have already been doing something similar. For example, Microsoft SQL OLAP server allows the users to optionally log queries submitted to it to fine tune the set of cubes [10]. From these logs, frequent queries can be identified and grouped into similar types. It is thus feasible to identify the cubes in the initial schema from the frequent queries. Currently, general practitioners design cube schema in an ad-hoc way, which is very likely far from optimal. This problem will become very serious when data cubes are required to be built on large data warehouses such as those from retail giants or Internet e-commerce shops, as their databases contain large numbers of attributes.

We have formulated the second phase of the design problem as an optimization problem, and have developed an efficient greedy algorithm to solve it.

Once a data cube schema is defined, the most imminent problem that follows is query processing. Given a DSS query submitted to the query client, the query client module needs to determine whether the query should be processed at the data cube level or at the warehouse level. If a query can be answered by the cubes, one needs to determine which cube should be used. If multiple solutions exist, one needs to determine the best choice of a cube.

We have proposed a two-phase approach to deal with the design problem in a data cube system: (1) an initial schema is derived from the user's query requirements; (2) the final schema is derived from the initial schema through an optimization process. The greedy algorithm cMP proposed for the optimization is very effective in pruning the search space of the optimal solution. Variants of cMP have been studied to reduce the search cost. Experiments on real data (TPC-D) have been performed to investigate the behavior of cMP. Results observed from the performance study confirm that cMP is an efficient algorithm.

References


