Paper Number 1135

Index Selection for OLAP*

Himanshu Gupta   Venky Harinarayan   Anand Rajaraman   Jeffrey D. Ullman

Abstract

On-line analytical processing (OLAP) is a recent and important application of database systems. Typically, OLAP data is presented as a multidimensional "data cube." OLAP queries are complex and can take many hours or even days to run, if executed directly on the raw data. The most common method of reducing execution time is to precompute some of the queries into summary tables (subcubes of the data cube) and then to build indexes on these summary tables. In most commercial OLAP systems today, the summary tables that are to be precomputed are picked first, followed by the selection of the appropriate indexes on them. A trial-and-error approach is used to divide the space available between the summary tables and the indexes. This two-step process can perform very poorly. Since both summary tables and indexes consume the same resource —space — their selection should be done together for the most efficient use of space. In this paper, we give algorithms that automate the selection of summary tables and indexes. In particular, we present a family of algorithms of increasing time complexities, and prove strong performance bounds for them. The algorithms with higher complexities have better performance bounds. However, the increase in the performance bound is diminishing, and we show that an algorithm of moderate complexity can perform fairly close to the optimal.

1 Introduction

In [HRU96], the efficient implementation of "data cubes" was considered. In particular, we investigated the problem of selecting a set of views of the data cube to materialize, in order to minimize the time needed to execute a given population of queries. An algorithm that was both polynomial-time and *competitive* (always gives a solution that is within a constant factor of the optimum) was presented. This "greedy" algorithm is guaranteed to give at least 63% of the benefit of the optimal solution.

While the greedy algorithm can be shown to be competitive (in the formal sense given above) for a variety of materialized view problems, there are also problems for which the greedy approach can be shown arbitrarily bad ([G97]). In this paper, we investigate a simple but important example of a problem where straightforward approaches are neither polynomial nor competitive, yet there is a technique that gives us both of these desirable properties. We study data cubes with indexes on the materialized views permitted. The problem is that when we have the option of putting one or more indexes on a view in order to speed certain queries, then all the benefit of materializing a view might reside in one or more of the indexes that we later build for that view. We are thus faced with two apparently bad choices:

*This work was supported by NSF grant IRI–92–23405, ARO grant DAAH04–95–1–0192, and Air Force Contract F33615–93–1–1339. It was performed at the Department of Computer Science, Stanford University, Stanford, CA 94305-9040. Email: {hgupta, venky, anand, ullman}@db.stanford.edu. Present address of V. Harinarayan and A. Rajaraman: Junglee Corp., Palo Alto, CA.

1. Treat a view and any set of its indexes as one view that we might choose to materialize. Then, the number of different views available grows exponentially. While the greedy algorithm might then be both competitive and polynomial in the size of its data, the fact that its data has grown exponentially makes the approach unlikely to be successful, even for relatively small data cubes.

2. Treat indexes as separate "views" that we might materialize. Here, the problem is that until the underlying view is chosen by the greedy algorithm, there is no benefit in choosing an index. Thus, a simple greedy algorithm might never materialize the underlying view, because it perceives no immediate benefit in doing so. Thus, we could be locked out of the opportunity to make major improvements in the average query time by materializing the view and several of its indexes.

We might imagine that the solution is simple: look at several "objects" (views or their indexes) at a time. Unfortunately, the more objects we consider at a time, the higher the degree of the polynomial that measures the running time. We thus consider both this "k-greedy" approach and another approach that often has a slightly lower performance guarantee but remains close to linear in running time per step of the greedy algorithm. Our conclusion is that we can, with small degradation of performance compared with the no-index case considered in [HRU96], incorporate indexes into the framework of data cube design.

1.1 Motivation

Decision-support systems are an increasingly important application of databases. Corporations are beginning to use the accumulated operational data to help understand and run their business. Towards this purpose, data from the different operations of a corporation are reconciled and stored in a central database commonly called a "data warehouse." Analysts use the data warehouse to extract the business information that enables better decision making. This interactive decision-support process is called OLAP (On-line Analytical Processing) to distinguish it from conventional OLTP (On-line Transaction Processing) applications.

EXAMPLE 1.1 Consider a retailer's data warehouse that has information about every sale in its many stores. Typical OLAP queries on this data warehouse would be:

- What is the total weekly sales of each store?
- What is the total weekly sales of each item?

□

OLAP applications require viewing the data from many different business perspectives (dimensions). In the example above, the dimension attributes are `store` and `item`. The metric of interest is called the *measure attribute*, which is weekly `sales` in the example. *Data cube* [GBLP95] is a multidimensional view of the database where a critical value, `sales` in above example, is organized by several dimensions, `store` and `item` in above example. It is generally accepted that OLAP systems need to present such a multidimensional view of the data to users. Each *cell* of the data cube corresponds to a unique set of values for the different dimensions and contains the value of the measure for this set of values. As mentioned in [GBLP95], the domain of each dimension is augmented with the special value "ALL."

EXAMPLE 1.2 Consider the retailer's warehouse in Example 1.1 with the two dimensions `item` and `store` and the measure `sales`. The data cube in this case consists of the following four subcubes:

- `sales` for each `item`-`store` pair.
- `sales` of each `item` over *all* `store`'s.
- `sales` for each `store` over *all* `item`'s.
- `sales` summed over *all* `store`'s and `item`'s.

□

How to store the data in order to present this multidimensional view is a subject of some controversy. There are two implementation approaches to OLAP:

- MD-OLAP (multidimensional OLAP): This approach stores the data cube in a multidimensional array. In their simplest form, MD-OLAP systems use the given values for the dimensions to find the offset in the array where the corresponding measure value is stored. Thus the value of the measure attribute corresponding to each possible set of values for the dimensions must be precomputed.
- ROLAP (relational OLAP): In this approach, subcubes of the data cube may be precomputed into "summary tables." In Example 1.2, we may choose to precompute any of the four subcubes into their associated summary tables. Queries are answered using these summary tables where possible. If the desired summary tables have not been precomputed, the values of the desired cells of the data cube are determined dynamically using the raw atomic data.

There is a fundamental trade-off between the "load time" — the amount of time spent precomputing parts of the data cube — and the query response time. The space occupied by the precomputed parts is a good indicator of the load time, and we use the terms load time and space interchangeably in this paper. MD-OLAP systems have high load times and low response times. These characteristics make MD-OLAP systems a good choice for the relatively smaller data cubes that are common in applications like financial and executive information systems. For the larger data cube applications, like retailing, the load time of MD-OLAP systems increases exponentially with the number of dimensions. For such data cubes it is usually impossible to precompute everything. It is thus important to use the available load time to do those precomputations that help the query response times the most. Interestingly, it appears that a law of diminishing returns holds: experiments in [HRU96], [MS] show that a little precomputation improves query response times substantially, but further precomputation does not improve query response time very much. It is possible to ensure low query response times even in large data cubes by cleverly selecting the subcubes to be precomputed.

Currently, precomputation in ROLAP systems is a two step process. In the first step, the desired subcubes of the data cube are materialized into summary tables. In the second step, the required indexes on these summary tables are built. Dividing the space (load time) available between these two steps for best performance is a difficult task. In ROLAP systems today, this division is done in an ad hoc manner and can lead to poor performance. Allotting too much space to the indexes may lead to too few subcubes being precomputed, while too much space for the subcubes may result in too few indexes being built. This problem is usually solved in a trial-and-error fashion by varying the indexes and subcubes picked until a satisfactory combination is reached. It is a time consuming and error-prone solution. In this paper, we automate the process and give algorithms that select the subcubes and indexes to precompute.

Harinarayan *et al.* in [HRU96] observed that general strategies for selecting subcubes to precompute can perform badly if they ignored the relative sizes of the subcubes. The same is true when we add indexes to the picture. A general strategy that determines a priori the space to be allocated to the subcubes and that to the indexes can perform very poorly. Our algorithms integrate the selection of subcubes and the selection of indexes into one process. Since both subcubes and indexes consume the same resource — space — this approach leads to an efficient utilization of space. In this paper, we present a family of greedy algorithms of increasing time complexity, and prove strong performance guarantees on the performance of the algorithms. Further, by going to an algorithm of higher complexity, we can improve the performance guarantee. However, we again encounter the law of diminishing returns *i.e.,* increasing the complexity leads to successively smaller improvements in performance. As the problem size is proportional to the factorial of the number of dimensions, algorithms having high complexity are very unlikely to be feasible. Our result is promising as we show that an algorithm of a relatively low time complexity can perform close to optimal.

1.2 Related Work

Gray *et al.* in [GBLP95] introduce the data cube operator as a generalization of the SQL **groupby** operator. The key to this generalization is the introduction of the "ALL" keyword. In many commercial systems, subcubes of this data cube are precomputed to improve performance. In [HRU96], Harinarayan *et al.* look into the problem of determining which subcubes to precompute. They give a simple greedy algorithm to select subcubes, and prove a strong performance guarantee for the algorithm. Johnson and Shasha in [JS96] introduce a new index structure that reduces the number of index accesses in data cube queries.

The issue of what indexes to build has not been investigated before from a research perspective. This problem arises in ROLAP implementations of the data cube. A two-step process — picking subcubes first, followed by the indexes — is typically adopted [MS]. An ad hoc approach is used in dividing the space and in picking indexes. For example, [MS] builds indexes on the most frequently used dimensions. This paper is the first to explore the index-selection problem and automate it with provably near-optimal algorithms.

1.3 Paper Organization

The rest of the paper is organized as follows. In Section 2, we present a motivating example based on the TPC-D benchmark database which illustrates why the two-step process of picking subcubes and indexes can lead to bad choices. In Section 3, we look at the universe of views, queries, and indexes that arise in a data cube. The cost model is introduced in Section 4. We present the algorithms along with the analysis of their performance guarantees in the following section. Experimental results are stated in Section 6. Finally, we present our conclusions in Section 7.

2 Motivating Examples

We use an example taken from TPC-D [TPCD], a decision-support benchmark to motivate the index-selection problem. The example illustrates the complexity of this problem, and the difficulty of doing the selection well in the two-step selection process.

For this example, we use only a subset of the dimensions that exist in the schema of the TPC-D benchmark. TPC-D models a business warehouse with dimensions `part`, `supplier`, and `customer`. The business buys a `part` from a `supplier` and sells it to a `customer`. The measure of interest is the total sales: `sales`. We use the TPC-D database as a running example throughout this paper.

We first discuss the universe of subcubes, queries, and indexes possible. This discussion is formalized in Section 3.

Subcubes

As in Example 1.1, the subcubes considered for precomputation correspond to elements of the power set of the set of dimensions. Each element of this power set corresponds to a subcube. The element {`part`, `customer`}, for example, corresponds to the subcube which has the `sales` for each (`part`, `customer`) pair over all `supplier`'s. In SQL terms, the subcubes differ only in their **groupby** clause, and an element of the power set gives the attributes in the **groupby** clause of the corresponding subcube. We give below a list of the subcubes for TPC-D, with the number of cells (rows) in each.

1. `part`, `supplier`, `customer` (6M, *i.e.*, 6 million rows)
2. `part`, `customer` (6M)
3. `part`, `supplier` (0.8M)
4. `supplier`, `customer` (6M)
5. `part` (0.2M)
6. `supplier` (0.01M)
7. `customer` (0.1M)
8. `none` (1)

`none` indicates the empty set, *i.e.*, there are no attributes in the **groupby** clause.

Throughout the paper, wherever it is clear from the context, we abbreviate `part` to p, `supplier` to s, and `customer` to c. We also use the terms dimensions and attributes interchangeably. In the presence of attribute hierarchies, these terms have different meanings, but for simplicity, in this paper, we do not consider attribute hierarchies.

Queries

The queries we consider can use each dimension as a selection attribute or as an output attribute (in SQL terms, as a **groupby** clause attribute or as a **where** clause attribute respectively). A possible query Q is:

- Find the `sales` to each `customer` of a given `part` "widget" bought from a given `supplier` "Widgets-r-us".

In this query, `sales` and `customer` are output attributes, and `part` and `supplier` are selection attributes. Since all queries have `sales` as an output attribute, we do not use the `sales` dimension in specifying a query.

Using the abbreviations for the dimensions, we write query Q as $\gamma_c\sigma_{ps}$; γ specifies the output or **groupby** attributes, while σ specifies the selection attributes of the query. The order of the dimensions in γ and σ is unimportant, and so, as an example, the query $\gamma_p\sigma_{sc}$ is the same as the query $\gamma_p\sigma_{cs}$. Any subcube that has all the output and the selection attributes of a query can be used to answer the query.

Indexes

We can construct B-Tree indexes (or variants) to speed up query processing. For subcube *ps*, for example, we can construct the following indexes:

- I_{ps}: the search key for this index is a concatenation of the p and s dimensions.
- I_{sp}: the search key for this index is a concatenation of the s and p dimensions.

The order of the dimensions in the indexes matter. Given a value for p, we can use I_{ps} to retrieve those rows in subcube *ps* that have this value for p. Similarly for *ps*, given a value for p and s we can use I_{ps} to retrieve the required row from subcube *ps*. However, consider retrieving those rows in subcube *ps* with a given value of s. If we were to use index I_{ps} to do this retrieval, we would have to scan all the leaf nodes of the index. There is no benefit in terms of the number of rows processed of using the index over doing a simple table scan. In practice too it has been observed that table scans are preferable to doing full index scans. I_{ps} is thus only useful in answering any query with the selection attributes p or *ps*. In general, an index $I_{X_1,\ldots,X_k}$ is only useful in answering any query which has some prefix of $X_1, \ldots, X_k$ in its selection attributes.

Cost Model

Suppose that the cost of answering any query is the number of rows processed. Consider answering Q_1: $\gamma_p\sigma_s$. Now Q_1 can be answered using either subcube *ps* at a cost of 0.8M rows, or using subcube *psc* at a cost of 6M rows. Consider now answering Q_1 using the index I_{sp} on subcube *ps*.[1] The average number of rows associated with each value of s in subcube *ps* is $\frac{|ps|}{|s|} = 80$. Thus the cost of answering Q_1 using I_{sp} is the cost of processing 80 rows. Indexes are thus very useful in reducing query costs dramatically and should be considered in any precomputation strategy.

Figure 1 shows the queries and indexes associated with one subcube, *ps*. The subcubes are organized into a lattice as explained in Section 3.4. Queries are associated with the smallest subcube that can be used to answer them.

EXAMPLE 2.1 Returning to our original problem: the precomputation strategy for the TPC-D benchmark database, the question is now: which of the possible subcubes and indexes do we materialize for good query performance? For simplicity, assume that all queries are equiprobable. To materialize all possible subcubes and indexes, we would require space for around 80M rows. In most practical situations there is not enough space (or equivalently load time) to precompute everything. For this example, assume that we have around 25M rows worth of space available.

The Two-Step Approach. The two-step process we mentioned in Section 1, would divide this available space between subcubes and indexes. Subcubes are picked in the first step to fit within the space allotted to them and indexes are picked in the second step again fitting within the space allotted to them. Note that indexes can only be built for those subcubes picked in the previous step. An important problem is dividing the space available between the subcubes and the indexes. One possibility is dividing the space available equally between the subcubes and indexes. In this

[1] In this example, we restrict ourselves to only "fat" ("covering") indexes, i.e., the indexes which correspond to the permutations of the dimensions in the subcube.

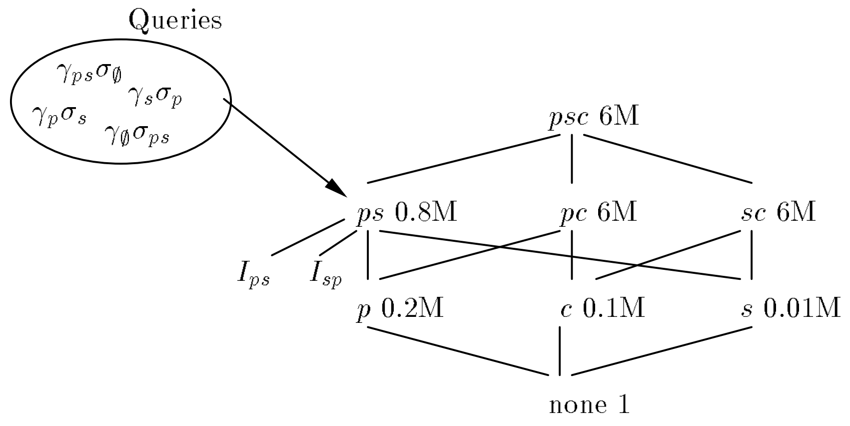

Figure 1: Subcubes, Queries and Indexes in the TPC-D database.

example, we use a greedy algorithm as given in [HRU96] to pick the best subcubes and indexes in each step. On dividing the space equally between the two steps and running this algorithm, we pick the following subcubes in the first step:

$$psc, ps, c, s, p, \texttt{none}, sc,$$

and the following indexes in the next step:

$$I_{csp}, I_{pcs}$$

This selection leads to an average query cost of 1.18M rows. That is, each query requires us to process on an average 1.18M rows .

The 1-Greedy Approach. Now consider integrating the subcube selection and index selection into one step. We use the simplest algorithm we give in this paper: a 1-greedy algorithm that at every stage picks the subcube or the index on a subcube (if the underlying subcube has already been picked) that gives the greatest benefit in terms of the number of rows processed. The 1-greedy algorithm gives us the following selection in order of decreasing benefit:

$$psc, I_{csp}, ps, I_{pcs}, I_{spc}, c, s, p, \texttt{none}$$

The selections made by this algorithm result in an average query cost of 0.74M rows. By integrating the two-steps into one, we have improved the average query cost by almost 40 percent. The reason is the following: while in the two-step process we allocate half of the 25M rows available to the indexes, it turns out that we are best off allocating three-quarters of the available space to the indexes. The actual fraction of space we should allocate to the indexes depends on a number of factors like the sizes of the subcubes and indexes, and it is difficult to determine this fraction a priori without considering the relative benefits of subcubes and indexes at every stage of the selection process.

Interestingly, we see a law of diminishing returns here too. The remaining subcubes and indexes which have a total of around 55M rows provide virtually no benefit and do not impact the query cost at all. □

It may still be argued that one can design better strategies to ensure that the two-step process does not perform so badly. But as we shall show, even the one-step 1-greedy algorithm can perform arbitrarily badly in the general case. In practice moreover, the general strategy used in the two-step case is one of trial-and-error. The algorithms and the associated performance guarantees that we give in this paper are thus a significant step towards automating this complex task of deciding what to precompute for the best query performance.

3 Views, Queries, and Indexes

In this section, we outline the type of views, queries and indexes which we consider in a data cube. Note that, our algorithms are robust and their correctness or performance guarantees do not depend upon the choice of views, queries and indexes.

3.1 Views

In a data cube, we associate a subcube with every set of **groupby** attributes possible. These subcubes are the units of precomputation and form the potential views considered for materialization. Here, we provide a rationale for such a partitioning scheme of the data cube into its subcubes.

There are many ways of partitioning a data cube. The boundary cases illustrate the problems of too many partitions and too few. When every row of the data cube is treated as a subcube, in a ROLAP setting, a separate table is associated with each row. The overhead cost and complexity of the metadata make such a choice very difficult to implement. On the other hand, consider the case when we have only one partition: the entire data cube itself. This approach leads to poor utilization of space, since we do not precompute anything unless we can precompute the entire cube. Thus any available space which is less than the size of the entire data cube is wasted.

It is important that we strike a balance between these two extremes. A commonly used ROLAP partitioning scheme [HRU96], is to associate a subcube with every element of the power set of the set of dimensions, as was mentioned in Section 2. The subcubes when specified in SQL differ only in the **groupby** clause. The element of the power set gives the set of attributes in the **groupby** clause of the corresponding subcube. Thus for example, $\{\mathtt{part}, \mathtt{customer}\}$ corresponds to the subcube given by the SQL query:

```
SELECT Part, Customer, SUM(sales) AS TotalSales
FROM R
GROUP BY Part, Customer;
```

In this paper we denote a subcube by the attributes in the **groupby** clause. The subcube above is denoted by `part`, `customer`. When we abbreviate the dimensions, we just concatenate the abbreviations in specifying the subcubes — the subcube above could also be written as *pc*. Note that the order of the dimensions is irrelevant.

In practice too, such a partitioning scheme is common. The popular "snowflake schema" [ADS] is an example of this partitioning scheme. Since the subcubes are now just aggregate views, some of which are materialized, we refer to them as views.

3.2 Queries

A user query in the TPC-D example, asks for the `sales` grouped by a certain set of attributes, after selecting on another set of attributes. The selection attributes are disjoint from the group-by attributes. For example, a user query could ask only for the `sales` of a single `part`, say "widgets," grouped by the `customer`s it was sold to. We write this query as $\gamma_{\mathtt{customer}}(\sigma_{\mathtt{part}=\mathrm{widget}}(R))$. Here the subscripts of γ denote the group-by attributes and those of σ, the selection attributes.

In general, a query of the form $\gamma_{\texttt{customer}}(\sigma_{\texttt{part}=\text{constant}}(R))$ asks for a *slice* through the subcube `customer`, `part`. We denote a generic query of this form by $\gamma_{\texttt{customer}}\sigma_{\texttt{part}}$ and call it a *slice query* on the subcube `customer`, `part`. We can associate a query $\gamma_{G_1,\ldots,G_k}\sigma_{S_1,\ldots,S_l}$ with the subcube denoted by $G_1,\ldots,G_k,S_1,\ldots,S_l$ which is the smallest subcube that can answer this query. We regard a user query that asks for an entire subcube as a special kind of slice query where the set of selection attributes is empty. Thus all queries are slice queries. An r-dimensional subcube has associated with it 2^r slice queries, because any subset of the dimensions can occur in the select part. An n-dimensional data cube has a total of $\binom{n}{r}$ r-dimensional subcubes. Therefore, the total number of slice queries associated with an n-dimensional data cube is $\sum_{r=0}^{n}\binom{n}{r}2^r$, which equals 3^n.

3.3 Indexes

B-Tree indexes can improve query response times substantially. There can be several indexes on a given view. For subcube ps, for example, we can construct the following four indexes:

$$I_p(ps),\ I_s(ps),\ I_{ps}(ps),\ I_{sp}(ps)$$

In each case we list the search key attributes as the subscript with the subcube, ps, on which the index is built mentioned in parentheses. while the subscript is the base relation. The order of the attributes in the indexes matters as we saw in Section 2.

We can have one index for every subset of the attributes of a view and every ordering of the subset. Thus the number of possible indexes for a view with m attributes is given by $\sum_{r=0}^{m}\binom{m}{r}r!$ which approaches $(e-1)m!$ for large m. Using a similar calculation, the total number of indexes associated with an n-dimensional data cube is close to $(e-1)^2n!$ approximately $3n!$. The number of fat indexes — search key attributes are permutations of the subcube attributes — in an n-dimensional cube is $(e-1)n!$, or approximately $2n!$. In general, an index can help answer slice queries when some prefix of the index attributes correspond to some of the selection attributes in the query.

3.4 The Computability and Dependence Relations

We define the *computability relation* $\ll$ between queries and views as follows. For query Q and view V, we say $Q \ll V$ if the result of query Q can be computed using only the tuples in view V. for example consider the query $Q_1 = \gamma_c\sigma_s$. This query can be computed from the view $V_1 = sc$ and also from the view $V_2 = psc$. $Q_1 \ll V_1$, and $Q_1 \ll V_2$. However, if $V_3 = pc$, Q_1 cannot be computed from V_3.

Define the partial order $\preceq$ on the views as follows: $V_1 \preceq V_2$ iff the set of attributes of V_2 is a superset of the set of attributes of V_1. Thus, `part`$\preceq$ `part` and `part`$\preceq$ `part`, `customer`, but `part` $\not\preceq$ `customer` and `customer` $\not\preceq$ `part`. The different subcubes of a data-cube form a lattice under $\preceq$, which we call the *dependence relation* for views.

There is a relationship between the computability relation $\ll$ and the dependence relation $\preceq$: if $V_1 \preceq V_2$, and $Q_1 \ll V_1$, then $Q_1 \ll V_2$. Therefore, the lattice in Fig. 2 helps us build the the $\ll$ relation as well.

The $\ll$ relation can be represented using a bipartite graph as done in Section 5. If a query Q is computable from a view V, we draw an edge between Q and V. Each edge (Q,V) also has a weight: the cost of answering Q using V. Consider now an index I on a view V that helps answer a query Q more quickly. This index does not affect computability in any way but only speeds up computation. The effect of this index on the graph is an additional edge between Q and V labeled

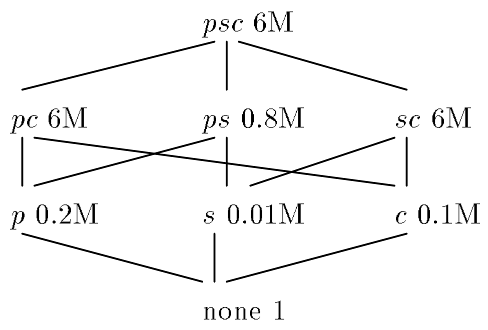

Figure 2: The lattice of subcubes of the TPC-D database.

by a pair consisting of I and the cost of answering Q using V *and* I. The algorithms we outline in Section 5 take as an input this graph for the $\ll$ relationship.

3.5 Summary

An n-dimensional data cube has associated with it:

- 2^n views;
- 3^n slice queries; and
- about $3n!$ possible indexes, about $2n!$ of these being fat indexes.

It is to be noted here that the problem size varies as a factorial of the number of dimensions.

4 A Cost Model

In this section, we present a cost model to estimate the time to answer a query using a view in conjunction with an index on the view. We then consider the problem of estimating the sizes of views, indexes, and query results without actually materializing all of them (as we have seen, the number of queries, views, and indexes can be quite large even for data cubes of small dimension). *Our algorithms do not depend upon any cost model for correctness and performance guarantees.*

4.1 The Linear Cost Model

Suppose we answer a query Q using a view V. We need to process the table corresponding to V to answer Q. Depending on the availability of indexes, we may have to process only some of the rows of the table for V. The cost of answering Q is a function of the number of rows of the table V we must process in order to answer Q. In this paper, we choose the simplest possible cost model:

- The cost of answering Q is the number of rows of the table for V that must be processed to construct the result of Q.

This "linear cost model" was presented in [HRU96] and is also used in the MetaCube product [STG].

Let Q be a slice query such that $Q \ll V$, and consider answering Q using V. For example, suppose Q is a query about the `sales` of a single `part` "widget," and V is the view `part`, `supplier`. That is, $Q = \gamma_{\texttt{none}}\sigma_{\texttt{part}}$. If there are no suitable indexes on V, we must scan almost the entire table for V, and the query cost is given by $|V|$, where $|V|$ is the number of rows in V. Suppose we have available the index $I_{\texttt{part}}(V)$. We can use this index to process exactly the rows of V corresponding to the part Q asks for. On average, V has $|V| \,/\, |\pi_{\texttt{part}}(V)|$ rows corresponding to each part, and so this is the average number of rows we would have to process to answer the slice query Q. Here π is the **distinct** projection, and so $|\pi_{\texttt{part}}(V)|$ gives the number of distinct values of `part` in V. Noticing that the number of distinct part values is the same as the number of rows in the subcube `part`, we get $|\pi_{\texttt{part}}(V)| = |\texttt{part}|$, where $|\texttt{part}|$ is the size of the subcube `part`. We conclude that the average cost of answering the slice query $\gamma_{\texttt{none}}\sigma_{\texttt{part}}$ using the view V and the index $I_{\texttt{part}}(V)$ is $|V| \,/\, |\texttt{part}| = |\texttt{part}, \texttt{customer}| \,/\, |\texttt{part}|$. In computing the cost, we disregard the number of index nodes processed.

We can follow exactly the same process and arrive at the same cost if we had used the index $I_{\texttt{part,supplier}}(V)$ instead. However, suppose we had available only either the index $I_{\texttt{supplier}}(V)$ or the index $I_{\texttt{supplier,part}}(V)$. We cannot use either of these indexes to reduce the number of rows of V we need to process in order to answer Q. Therefore, the cost of answering Q is just $|V|$, exactly as if there were no indexes.

4.1.1 Cost Formula

We generalize these observations to obtain a formula for the cost of answering query Q using view V and index J.

- Let Q be the query $\gamma_{\bar{A}}\sigma_{\bar{B}}$, where $\bar{A}$ and $\bar{B}$ are sets of dimensions. We have $\bar{B} = \emptyset$ iff Q is a subcube query, and $\bar{A} = \emptyset$ denotes aggregation over all dimensions.
- Let V be the view $\bar{C}$ and Q be the query $\gamma_{\bar{A}}\sigma_{\bar{B}}$. If $Q \ll V$, we must have $\bar{A} \cup \bar{B} \subseteq \bar{C}$. $\bar{C} = \emptyset$ denotes the view `none` at the bottom of the lattice in Fig. 2.
- Let J be the index $I_{\vec{D}}(V)$. We use $\vec{D}$ to emphasize that the order of attributes matters; $\vec{D}$ is a sequence of attributes rather than a set. In particular, $D = \langle\rangle$ (the empty sequence) denotes the case where we are not using an index.

Let $\bar{E}$ denote the largest subset of $\bar{B}$ such that the attributes in $\bar{E}$ form a prefix (not necessarily proper) of $\vec{D}$. The cost of answering Q using the view V in conjunction with the index J is given by:

$$c(Q, V, J) = \frac{|(\bar{C})|}{|(\bar{E})|}$$

Recall that $|(\bar{C})|$ and $|(\bar{E})|$ denote the number of rows in the tables corresponding to views $(\bar{C})$ and $(\bar{E})$ respectively.

As an example consider the TPC-D database, with the view $V = psc$ of size 6 million rows, query $Q = \gamma_c, \sigma_{ps}$ and the index $J = I_{scp}$ on subcube psc. In this case, $\bar{C} = psc$ and $\bar{E} = s$, since the largest subset of attributes of ps that forms a prefix in scp is s. From Section 2 we know the desired cardinalities: $|(\bar{psc})| = 6M$, $|(\bar{s})| = 0.01M$. The cost is therefore

$$c(Q, V, J) = \frac{6M}{0.01M} = 600$$

In other words, 600 rows have to be accessed in answering query Q using index J on view V.

The above formula works in all cases. The case when $\bar{E} = \emptyset$ deserves some discussion. This case might occur either because no index is available on V, or because Q is a subcube query, or because the index used has no prefix composed only of select attributes of Q. In all these cases, we must process all the rows in V to answer Q, and so $c(Q, V, J) = |V|$. The formula gives exactly the same result, because $|(\emptyset)| = 1$ (recall that $(\emptyset)$ denotes the view `none`, which has 1 row).

4.2 Determining View and Index Sizes

Our algorithms require the following information:

1. The size of each view.

2. The size of each index.

3. For each (query, view, index) triple, the cost of answering the query using the view in conjunction with the index.

Section 4.1.1 shows how to obtain item (3) given item (1). But we still need to know the sizes of each view and each index. The problem is nontrivial because the number of views and indexes is very large even for cubes of moderate dimension (Section 3 shows that the number of views and indexes is exponential in the cube dimension).

4.2.1 Estimating View Sizes

There are many ways of estimating the view sizes that avoid materializing all the views. We can use sampling and analytical methods to compute the sizes of the different views if we only materialize the largest element V_l in the lattice (the view that groups by all the dimensions). For a view, if we know that the grouping attributes are statistically independent, we can estimate the size of the view analytically, given the size of V_l. Otherwise we can sample V_l (or the raw data) to estimate the size of the other views. The size of a given view is the number of distinct values of the attributes it groups by. Thus for example, the size of the view that groups by `part` and `supplier` is the number of distinct values of `part,supplier` in the raw data. There are many well-known sampling techniques that we can use to determine the number of distinct values of attributes in a relation [HNSS95].

4.2.2 Estimating Index Sizes

Given the view sizes, we can estimate index sizes. The size of each view in our cost model is the number of rows in the view. For indexes too, we follow a similar model to estimate the space cost. The size of each index (B-Tree) is the number of leaf nodes in the index. The number of leaf nodes of an index is approximately the number of rows in the underlying view. Thus,

- The size of any index on a view V is the same as the size of view V.

Our model of index sizes has the important consequence of pruning our space of possible indexes. Consider two indexes $J_1 = I_{\vec{A}}(V)$ and $J_1 = I_{\vec{B}}(V)$ on the same view V. If $\vec{B}$ is a proper prefix of $\vec{A}$, then surely $c(Q, V, J_1) \leq c(Q, V, J_2)$ for any query Q, using our cost formula. Moreover, the sizes of J_1 and J_2 are approximately the same under our index size model. Therefore, in any reasonable scheme of materializing views and indexes, we can ignore the index J_2 in favor of the index J_1. Thus, for each view, we need to consider only the fat indexes: those indexes whose search attributes are not a proper prefix of the search attributes of any other index on the same view. If the view V is $(\bar{C})$, this is exactly the set of indexes

$$\{I_{\vec{D}}(V) \mid \vec{D} \text{ is a permutation of } \bar{C}\}$$

This result is similar to that in [JS96], where they consider only fat indexes. It can be shown that this pruning reduces the number of indexes of interest by approximately a factor of $e - 1$, where e is the base of the natural logarithms.

5 Materializing Views with Indexes

In this section, we develop algorithms for selecting views and indexes to be materialized in the data cube. Informally, we are given a set of views, each of which has a set of indexes, and a set of queries that are to be supported by the system. A view with one of its indexes can be used to answer a query at some specified cost. The goal is to select a set of views and indexes which will minimize the total cost to answer the queries, under the constraint that the set of views and indexes selected do not occupy more than a given amount of space, S.

The above problem is NP-complete, even in the absence of indexes and even when each view occupies a unit space; there is a straightforward reduction from Set-Cover. We develop heuristic algorithms which provably deviate from the optimum selection of views and indexes by only a small amount.

First, we state the above problem formally. Then, we present a class of algorithms which have different guarantees of performance ratios and time complexities. We also rigorously analyze the performance of the algorithms presented.

5.1 Problem Definition

Consider a bipartite multigraph, $G = (V \cup Q, E)$, called a *query-view graph*. V contains the set of views and Q contains the set of queries.

- With each view $v_i \in V$ is associated a tuple (S_i, I_i), where

  S_i is the space occupied by the view, and

  I_i is the set of indexes on the view. I_{ik} is used to denote the k^{th} index of v_i.

- With each query $q_i \in Q$ is associated a default cost T_i of answering the query q_i, even without using any other view or index in G. (In data cube, the default cost of answering any query is the cost incurred in answering the query using the raw data table(s)).

- Every edge (q_i, v_j) has a label (k, t_{ijk}) associated with it, where t_{ijk} is the cost of answering the query q_i using the view v_j and its k^{th} index. When $k = 0$, t_{ijk} is the cost of answering q_i using just v_j.

Goal: Given a set of views V and a set of queries Q, we must select $M \subseteq V$, a set of views and indexes to be materialized, under the constraint that the views and indexes in M can be accommodated in S (a given constant) units of space. M must minimize the total cost incurred answering each query in Q from one of the views in M. More formally, we wish to minimize the following quantity

$$\tau(G, M) = \sum_{i=1}^{|Q|} \mathbf{min}(T_i, \min_{v_j, I_{j,k} \in M,} t_{ijk})$$

under the constraint that the total space occupied by the structures[2] in M is less than S.

The above problem is a simple formalization of the problem of selecting views and indexes in data cube. We have assumed that the queries supported by the system are uniformly distributed across the queries in Q. Our algorithms generalize easily to the case when there is a frequency f_i associated with each query q_i (by including the factor f_i with the term associated with q_i in the summation used to define τ).

[2] A *structure* is a view or an index.

Algorithm 5.1

<u>r**-Greedy Algorithm**</u>

Given: G, a query-view graph, and S, the space constraint.
BEGIN
$M = \phi$; /* M = set of structures selected so far. */
while $(S(M) < S)$
Look at all possible sets of views and indexes of one of the following forms:
- $\{v_i, I_{ij_1}, I_{ij_2}, \ldots, I_{ij_p}\}$, such that $v_i \notin M, I_{ij_l} \notin M$ for $1 \leq l \leq p$, and $0 \leq p < r$, **or**
- $\{I_{ij}\}$, such that the view v_i of the index I_{ij} is in M, and $I_{ij} \notin M$.

Among these sets, let C be the set which has the maximum benefit per unit space with respect to M.
$M = M \cup C$;
end while;
return M;
END.

$\diamond$

5.2 The Benefit of a Choice of Structures

Let C be an arbitrary set of views and indexes in a query-view graph G. We use $S(C)$ to denote the total space occupied by the structures in C. The *benefit* of C with respect to M, an already selected set of structures, is denoted by $B(C, M)$ and is defined as $(\tau(G, M) - \tau(G, M \cup C))$, where τ is the function defined above. Benefit of C per unit space with respect to M is $B(C, M)/S(C)$. Also, $B(C, \phi)$ is called the *absolute benefit* of the set C.

5.3 The r-Greedy Algorithm

The r-greedy algorithm executes in a number of stages, selecting at each stage a subset C having at most r structures. The set C consists either of

1. A view and some of its indexes, or

2. A single index whose view has already been selected in one of the previous stages.

At any given stage, the set C that has the maximum benefit per unit space with respect to M, the set of structures selected prior to this stage, is selected. See Algorithm 5.1.

Suppose there are v views and each view has at most i indexes. Then at each stage, the r-greedy algorithm must consider and calculate the benefit of at most $vi + v\binom{i}{r-1}$ possible sets. Hence an upper bound on the running time of the algorithm is $O(km^r)$, where m is the number of structures in the given query-view graph and k is the number of structures selected by the algorithm, which is S in the worst case.

EXAMPLE 5.1 We illustrate the working of r-greedy algorithm through a simple example.

Consider the query-view shown in Figure 3. For simplicity, we have assigned a space cost of 1 unit to each of the indexes and views. Let the value of S be 7 units.

We now see how the r-greedy algorithm works on the example for different values of r.

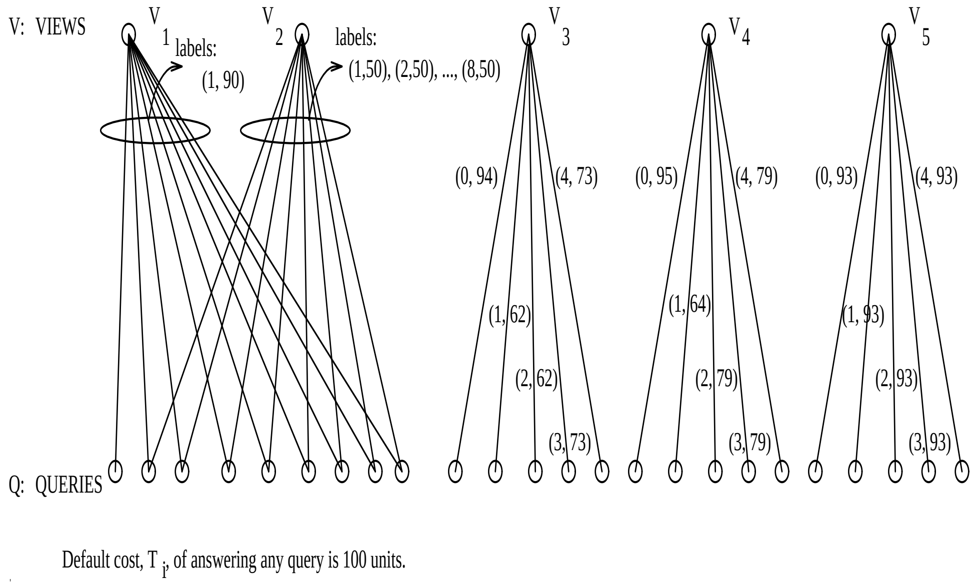

Figure 3: A query-view graph

1. **1-greedy:** Initially, absolute benefit of every index is zero. Absolute benefits of the views in order of their subscripts are viz. 0, 0, 6, 5, and 7. Hence, at the first stage the 1-greedy algorithm selects V_5. Except for the indexes of V_5, the benefits of all views and indexes with respect to $M = \{V_5\}$ remain the same as their absolute benefits. The benefits of $I_{5,i}, 1 \leq i \leq 4$, relative to M become 7 each. Hence, the 1-greedy algorithm choses one by one all the indexes of V_5 in the later stages, followed by V_3 and V_4. Thus, the solution returned by 1-greedy is $\{V_5, I_{5,1}, I_{5,2}, I_{5,3}, I_{5,4}, V_3, V_4\}$, with an absolute benefit of 46.

2. **2-greedy:** In the first stage, the 2-greedy algorithm selects $C = \{V_1, I_{1,1}\}$ which has an absolute benefit of $10 \times 9 = 90$. Benefit of $\{V_2, I_{2,i}\}$ for any $i \leq 8$ with respect to C is 40 (i.e., 20 per unit space). Hence, $\{V_4, I_{4,1}\}$, whose benefit with respect to C is 41 (i.e., 20.5 per unit space), gets selected in the second stage In the later stages, the other indexes of V_4 get selected one by one. Thus, the solution returned by 2-greedy is $\{V_1, I_{1,1}, V_4, I_{4,1}, I_{4,2}, I_{4,3}, I_{4,4}\}$ with an absolute benefit of 194.

3. **3-greedy:** As in the 2-greedy case, the first stage of the 3-greedy algorithm selects $C = \{V_1, I_{1,1}\}$, with the absolute benefit of 90. The second stage selects $\{V_3, I_{3,1}, I_{3,2}\}$, having a benefit of 82 with respect to C (i.e., 27.3 per unit space), as the benefit of V_2 with any two of its indexes is at most 80 with respect to C (i.e., 36.7 per unit space). The structures selected in the later stages are $I_{3,3}$, and $I_{3,4}$. Thus, the solution returned by 3-greedy is $\{V_1, I_{1,1}, V_3, I_{3,1}, I_{3,2}, I_{3,3}, I_{3,4}\}$, which has an absolute benefit of 226.

4. **Optimal Solution:** It is not difficult to see that the optimal solution for the given example is $\{V_2, I_{2,1}, I_{2,2}, I_{2,3}, I_{2,4}, I_{2,5}, I_{2,6}\}$, having an absolute benefit of 300.

□

Theorem 5.1 *In the case when each structure occupies a unit space, the r-greedy algorithm produces a solution M that uses at most $S + r - 1$ units of space. Also, the absolute benefit of M is at least $(1 - 1/e^{(r-1)/r})$ times the optimal benefit achievable using as much space as that used by M.*

Proof: It is easy to see that the solution M produced by the r-greedy algorithm has at most $S+r-1$ structures. Let $k=|M|$. Let the optimal solution containing k structures be O and the absolute benefit of O be B.

Consider a stage at which the r-greedy algorithm has already chosen a set G_l having l structures with "incremental" benefits $a_1, a_2, a_3, \ldots, a_l$. The absolute benefit of G_l is thus $\sum_{i=1}^{l} a_i$. Surely the absolute benefit of the set $O \cup G_l$ is at least B. Therefore, the benefit of the set O with respect to G_l, $B(O, G_l)$, is at least $B - \sum_{i=1}^{l} a_i$.

Without loss of generality, we can assume that the optimal set O doesn't contain any index whose corresponding view is not in O. Hence, if O contains m views, it can be split into m disjoint sets $O_1, O_2, \ldots, O_m$, such that each O_i consists of a view and its indexes in O. Then, $B(O, G_l) \leq \sum_{i=1}^{m} B(O_i, G_l)$. Therefore, by pigeon hole principle, there exists an O_i such that $B(O_i, G_l)/|O_i| \geq B(O, G_l)/k$. Now, consider the best r-subset[3] O_c of such an O_i. Its benefit per unit space with respect to G_l is at least $(\frac{r-1}{r})(\frac{k}{k-1})(B(O_i, G_l)/|O_i|)$, which happens when the benefit of the view in O_i is zero and $|O_i| = k$. Let, $k' = (\frac{r-1}{r})(\frac{k}{k-1})$. As O_c (or its best subset) is also considered for selection at this stage of the r-greedy algorithm, the benefit per unit space with respect to G_l of the set C selected by the algorithm is at least $k' B(O_i, G_l)/|O_i|$, which is at least $k'(B - \sum_{i=1}^{l} a_i)/k$. Note that O_c may contain some structures from G_l, but the argument still holds. Distributing the benefit of C over each of its structures equally (for the purpose of analysis), we get $a_{l+j} \geq k'(B - \sum_{i=1}^{l} a_i)/k$, for $0 < j \leq |C|$. As this is true for each set C selected at any stage, we have

$$B \leq \frac{k}{k'} a_j + \sum_{i=1}^{j-1} a_i, \qquad \text{for} \;\; 0 < j \leq k.$$

Let $k'' = k/k'$. Multiplying the j^{th} equation by $(\frac{k''-1}{k''})^{k-j}$, and after adding all the equations and some minor manipulations we get

$$A/B \geq 1 - (\frac{k''-1}{k''})^{k}$$

where $A = \sum_{i=1}^{k} a_i$, the absolute benefit of M.

$$\Rightarrow A/B \geq 1 - (\frac{k''-1}{k''})^{k''k'}$$

The quantity on the right hand side is at least $1 - 1/e^{k'}$, which is greater than $1 - 1/e^{(r-1)/r}$. ∎

We have come up with instances of the problem for which the r-greedy algorithm performs as bad as the worst case bound of $1 - 1/e^{(r-1)/r}$.

5.4 Inner-Level Greedy Algorithm

The Inner-level greedy algorithm works in stages. At each stage, it selects a subset C, which consists either of

1. A view and some of its indexes selected in a greedy manner, or

2. A single index whose view has already been selected in one of the previous stages.

[3] i.e., set of size at most r.

Algorithm 5.2

Inner-Level Greedy Algorithm

Given: G, a query-view graph, and S, the space constraint.
BEGIN
  $M = \phi$; /* M = Set of structures selected so far */
  while $(S(M) < S)$
    $C = \phi$; /* Best set containing a view and some of its indexes found so far */
    for each view v_i in $V \cap M$
      $IG = \{v_i\}$; /* IG = Set of v_i and some of its indexes selected in a greedy manner. */
      while $(S(IG) < S)$ /* Construct IG */
        Let I_{ic} be the index of v_i whose benefit per unit space w.r.t $(M \cup IG)$ is maximum.
        $IG = IG \cup I_{ic}$;
      end while;
      if $(B(IG, M)/S(IG) > B(C, M)/|C|)$ **or** $C = \phi$
        $C = IG$;
    end for;
    for each index I_{ij} such that its view $v_i \in M$
      if $B(I_{ij}, M)/S(I_{ij}) > B(C, M)/S(C)$
        $C = \{I_{ij}\}$;
    end for;
    $M = M \cup C$;
  end while;
  return M;
END.

$\diamond$

Note that there is no constraint on the size of set C. Each stage can be thought of as consisting of two phases. In the first phase, for each view v_i we construct a set IG_i which initially contains only the view. Then, one by one its indexes are added to IG_i in the order of their incremental benefits until the benefit per unit space of IG_i with respect to M, the set of structures selected till this stage, reaches its maximum. That IG_i having the maximum benefit per unit space with respect to M is chosen as C. In the second phase, an index whose benefit per unit space is the maximum with respect to M is selected. The benefit per unit space of the selected index is compared with that of C, and the better one is selected for addition to M. See Algorithm 5.2.

The running time of the Inner-level greedy algorithm is $O(k^2m^2)$, where m is the total number of structures in the given query-view graph and k is the maximum number of structures that can fit in S units of space, which in the worst case is S.

EXAMPLE 5.2 We illustrate the working of the Inner-level greedy algorithm for the example in Figure 3.

As the absolute benefit per unit space of V_2 with at most six of its indexes is less than 43, the algorithm selects $\{V_1, I_{1,1}\}$, whose absolute benefit is 90 (i.e., 45 per unit space) in the first stage. In the next stage, the algorithm selects V_2 and six of its indexes with an "incremental"

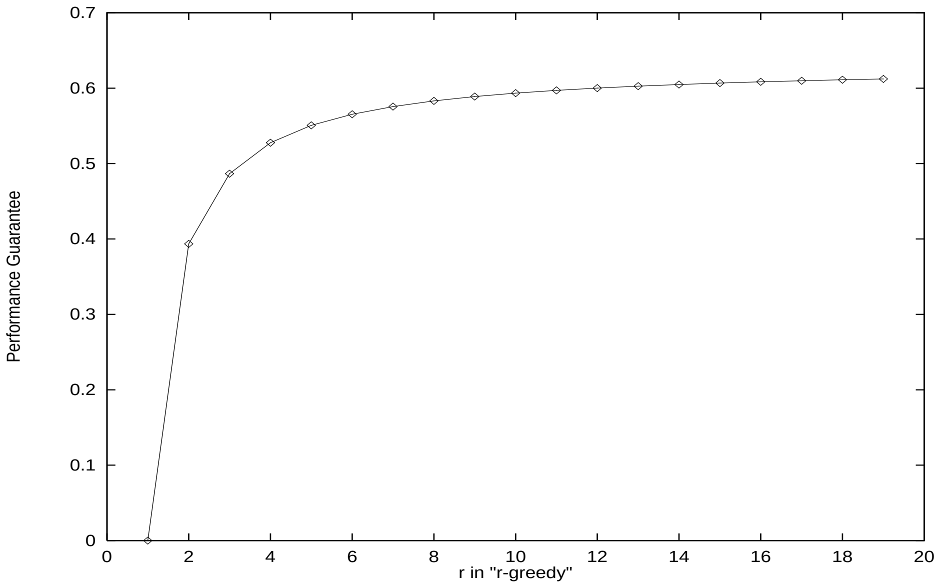

Figure 4: Performance guarantees of various algorithms

benefit of 240 (i.e., 34.3 per unit space). Thus, the solution returned by the Inner-level greedy is $\{V_1, I_{1,1}, V_2, I_{2,1}, I_{2,2}, I_{2,3}, I_{2,4}, I_{2,5}, I_{2,6}\}$ with an absolute benefit of 330. Note that the size of the solution returned is 9 units, slightly more than the given space limit. The optimal solution using 9 units of space is V_2 with its eight indexes, having an optimal benefit of 400. □

Theorem 5.2 *The Inner-level greedy algorithm produces a solution M that uses at most $2 * S$ units of space. Also, the absolute benefit of M is at least $(1 - 1/e^{0.63}) = 0.467$ of the optimal benefit achievable using as much space as that used by M, assuming that no structure occupies more than S units of space.*

Proof: The proof is almost identical to the proof of Theorem 5.1. The only difference is that in the case of Inner-level greedy algorithm the value of k' is 0.63 and is independent of the sizes of the views and indexes. This follows from the result in [HRU96] on the performance guarantee of the simple greedy algorithm under space constraint. As the value of k' is independent of the relative sizes of structures, the result holds for arbitrary sizes of views and indexes. ∎

6 Performance of the Algorithms

Figure 4 plots the performance guarantees of the family of r-greedy algorithms against r. Observe that the performance guarantee of the 1-greedy is 0: it is possible to construct examples where the the ratio of the benefit of the 1-greedy choice to that of the optimal choice arbitrarily small. The performance guarantee initially rises rapidly as we increase r beyond 1, but the increases are exponentially diminishing. The actual performance guarantees for 2-greedy, 3-greedy, and 4-greedy are 0.39, 0.49, and 0.53 respectively. The graph has a “knee” at $r = 4$, and the increments for greater r are vanishingly small. The performance guarantee approaches 0.63 as r approaches infinity (more precisely, as r approaches the number of structures we are willing to materialize). Recall that the running time is $O(m^r)$ where m is the total number of structures, and that m is large even for small cube dimensions. Therefore, it seems that in practice, using r-greedy algorithms is not worth the additional complexity for $r > 4$.

For comparison, the Inner-greedy algorithm runs in time $O(m^2)$ and has a performance guarantee of 0.47, which is between the ratios for 2-greedy and 3-greedy. Thus Inner-greedy is preferable to 2-greedy because it gives a better guarantee than 2-greedy for approximately the same running time.

We experimented with the r-greedy family of algorithms on cubes of dimension up to 6, for $r = 1, 2, 3$. We generated cubes using the analytical model in [HRU96], extended to incorporate indexes and slice queries. We varied different parameters: the cardinality of each dimension, the *sparsity* of the cube, and the query frequencies. (The sparsity of a data cube is the ratio of the number of rows in the raw data relation to the product of the cardinalities of the individual dimensions.) For dimensions up to 6, we observed that the algorithms in the r-greedy family produced solutions that were extremely close to the optimal. The results are encouraging because they indicate that in practice, we can obtain near-optimal solutions using an algorithm of low complexity.

7 Conclusions

In this paper we investigate the problem of what indexes to build to improve OLAP query performance. While this problem is very important to the success of ROLAP systems, commercial systems today usually use ad hoc solutions. In this paper we show that the precomputation of subcubes and indexes should be integrated into one step. ROLAP systems currently use a two-step process which can adversely affect query performance. We give an example of the poor performance of the two-step process using an example based on the TPC-D benchmark database.

We provide a family of one-step algorithms that select which subcubes and indexes should be precomputed for improved query performance, given the space constraint. We give strong performance bounds for our algorithms and show the trade-off between the performance bounds and the complexity of the algorithm. Our results indicate that an algorithm of moderate complexity performs almost as well as that of high complexity. We also present the experimental results which validate our analysis.

References

[ADS] Archer Decision Sciences. Star Schema 101. White Paper. At `http://members.aol.com/nraden/str101.htm`.

[GBLP95] J. Gray, A. Bosworth, A. Layman, H. Pirahesh. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals. Microsoft Technical Report No. MSR-TR-95-22.

[G97] H. Gupta. Selection of Views to Materialize in a Data Warehouse. To appear in *ICDT*, January, 1997, Delphi, Greece.

[HNSS95] P. J. Haas, J. F. Naughton, S. Seshadri, L. Stokes. Sampling-Based Estimation of the Number of Distinct Values of an Attribute. In *Proceedings of the 21st International VLDB Conference*, pages 311-320, 1995.

[HRU96] V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing Data Cubes Efficiently. *ACM SIGMOD 1996*, pages 205-216, 1996.

[JS96] T. Johnson and D. Shasha. Hierarchically Split Cube Forests for Decision Support: description and tuned design. Personal Communication.

[MS] Microstrategy Inc. The Case for Relational OLAP. White Paper. At `http://www.strategy.com`.

[TPCD] F. Raab, editor. TPC Benchmark(tm) D (Decision Support), Proposed Revision 1.0. Transaction Processing Performance Council, San Jose, CA 95112, 4 April 1995.

[STG] Stanford Technology Group, Inc. Designing the Data Warehouse On Relational Databases. White Paper.