Implementing Data Cubes Efficiently*

Venky Harinarayan **Anand Rajaraman** **Jeffrey D. Ullman**

Stanford University

Abstract

Decision support applications involve complex queries on very large databases. Since response times should be small, query optimization is critical. Users typically view the data as multi-dimensional data cubes. Each cell of the data cube is a view consisting of an aggregation of interest, like total sales. The values of many of these cells are dependent on the values of other cells in the data cube. A common and powerful query optimization technique is to materialize some or all of these cells rather than compute them from raw data each time. Commercial systems differ mainly in their approach to materializing the data cube. In this paper, we investigate the issue of which cells (views) to materialize when it is too expensive to materialize all views. A lattice framework is used to express dependencies among views. We then present greedy algorithms that work off this lattice and determine a good set of views to materialize. The greedy algorithm performs within a small constant factor of optimal under a variety of models. We then consider the most common case of the hypercube lattice and examine the choice of materialized views for hypercubes in detail, giving some good tradeoffs between the space used and the average time to answer a query.

1 Introduction

Decision support systems (DSS) are rapidly becoming a key to gaining competitive advantage for businesses. DSS allow businesses to get at data that is locked away in operational databases and turn that data into useful information. Many corporations have built or are building new unified decision-support databases called *data warehouses* on which users can carry out their analysis.

While operational databases maintain state information, data warehouses typically maintain historical information. As a result, data warehouses tend to be very large and to grow over time. Users of DSS are typically interested in identifying trends rather than looking at individual records in isolation. Decision-support queries thus make heavy use of aggregations and are much more complex than OLTP queries.

The size of the data warehouse and the complexity of queries can cause queries to take very long to complete. This delay is unacceptable in most DSS environments, as it severely limits productivity. The usual requirement is query execution times of a few seconds or a few minutes at the most.

There are many ways to achieve such performance goals. Query optimizers and query evaluation techniques can be enhanced to handle aggregations better [CS94], [GHQ95], [YL95], to use different indexing strategies like bit-mapped indexes and join indexes [OG95], and so on.

*Work was supported by NSF grant IRI–92–23405, by ARO grant DAAH04–95–1–0192, and by Air Force Contract F33615–93–1–1339 Authors' address: Department of Computer Science, Stanford University, Stanford, CA 94305-2140. Email: {venky, anand, ullman}@db.stanford.edu.

A commonly used technique is to materialize (precompute) frequently-asked queries. The data warehouse at the Mervyn's department-store chain, for instance, has a total of 2400 precomputed tables [Rad95] to improve query performance. Picking the right set of queries to materialize is a nontrivial task, since by materializing a query we may be able to answer other queries quickly. For example, we may want to materialize a query that is relatively infrequently asked if it helps us answer many other queries quickly. In this paper, we present a framework and algorithms that enable us to pick a good set of queries to materialize. Our framework also lets us infer in what order these queries are to be materialized.

1.1 The Data Cube

Users of data warehouses work in a graphical environment and data are usually presented to them as a multidimensional "data cube" whose 2-D, 3-D, or even higher-dimensional sub cubes they explore trying to discover interesting information. The values in each cell of this data cube are some "measures" of interest. As an example consider the TPC-D decision-support benchmark.

EXAMPLE 1.1 The TPC-D benchmark models a business warehouse. Parts are bought from suppliers and then sold to customers at a sale price `SP`. The database has information about each such transaction over a period of 6 years.

There are three dimensions we are interested in: `part`, `supplier`, and `customer`. The "measure" of interest is the `total sales`. So for each cell (p, s, c) in this 3-D data cube, we store the `total sales` of `part` p that was bought from `supplier` s, and sold to `customer` c. We use the terms dimension and attribute interchangeably in this section. In the general case, a given dimension may have many attributes as we shall see in Section 2.

Users are also interested in consolidated sales: for example, what is the total sales of a given part p to a given customer c? [GBLP95] suggest adding an additional value "ALL" to the domain of each dimension to achieve this. In the question above we want the total sales of a given part p to a given customer c for "ALL" suppliers. The query is answered by looking up the value in cell (p, ALL, c).

□

We use the TPC-D database of size 1GB as a running example throughout this paper. For more details on this benchmark refer to [TPCD].

We have only discussed the presentation of the data set as a multi-dimensional data cube to the user. The following implementation alternatives are possible:

1. Physically materialize the whole data cube. This approach gives the best query response time. However, precomputing and storing every cell is not a feasible alternative for large data cubes, as the space consumed becomes excessive. It should be noted that the space consumed by the data cube is also a good indicator of the time it takes to create the data cube, which is important in many applications. The space consumed also impacts indexing and so adds to the overall cost.

2. Materialize nothing. In this case we need to go to the raw data and compute every cell on request. This approach punts the problem of quick query response to the database system where the raw data is stored. No extra space beyond that for the raw data is required.

3. Materialize only part of the data cube. We consider this approach in this paper. In a data cube, the values of many cells are computable from those of other cells in the data cube. This dependency is similar to a spreadsheet where the value of cells can be expressed as a

function of the values of other cells. We call such cells "dependent" cells. For instance, in Example 1.1, we can compute the value of cell (p, ALL, c) as the sum of the values of cells of $(p, s_1, c), \ldots, (p, s_{N_{\texttt{supplier}}}, c)$, where $N_{\texttt{supplier}}$ is the number of suppliers. The more cells we materialize, the better query performance is. For large data cubes however, we may be able to materialize only a small fraction of the cells of the data cube, due to space and other constraints. It is thus important that we pick the right cells to materialize. Our approach is very scalable and can handle large data cubes well.

Any cell that has an "ALL" value as one of the components of its address is a dependent cell. The value of this cell is computable from those of other cells in the data cube. If a cell has no "ALL"s in its components, its value cannot be computed from those of other cells, and we must query the raw data to compute its value. The number of cells with "ALL" as one of their components is usually a large fraction of the total number of cells in the data cube. In the TPC-D database with the dimensions as in Example 1.1, seventy percent of all the cells in the data cube are dependent.

The problem of what cells of the data cube to materialize, is a very real one. There are different commercial systems which pick one of the different strategies given above. Clearly, each strategy has its benefits. For example, for applications where performance is of paramount importance and scalability is not important we can go with the materialize-everything strategy. The Essbase system [ESS], for example, materializes the whole data cube, while BusinessObjects [X94] materializes nothing, and the MetaCube system [STG] materializes part of the cube.

There is also the issue of where the materialized data cube is stored: in a relational system or a proprietary MDDB (multi-dimensional database) system. In this paper, we assume that the data cube is stored in "summary" tables in a relational system. Sets of cells of the data cube are assigned to different tables.

The cells of the data cube are organized into different sets based on the positions of "ALL" in their addresses. Thus, for example, all cells whose addresses match the address (_,ALL,_) are placed in the same set. Here, "_" is a placeholder that matches any value. Each of these sets corresponds to a different SQL query. The values in the set of cells (_,ALL,_) is output by the SQL query:

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

Here, R refers to the raw-data relation. The queries corresponding to the different sets of cells, differ only in the `GROUP-BY` clause. In general, attributes with "ALL" values in the description of the set of cells, do not appear in the `GROUP-BY` clause of the SQL query above. For example, `supplier` has an "ALL" value in the set description (_,ALL,_). Hence it does not appear in the `GROUP-BY` clause of the SQL query. Since the SQL queries of the various sets of cells differ only in the grouping attributes, we use the grouping attributes to identify queries uniquely.

Deciding which sets of cells to materialize is equivalent to deciding which of the corresponding SQL queries (views) to materialize. In the rest of this paper we thus work with views rather than with sets of cells.

1.2 Motivating Example

The TPC-D database we considered in Example 1.1 has 3 attributes: `part`, `supplier`, `customer`. We thus have 8 possible groupings of the attributes. We list all the queries (views) possible below with the number of rows in their result. Note again it suffices to only mention the attributes in the `GROUP-BY` clause of the view.

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 that there are are no attributes in the `GROUP-BY` clause. Figure 1 shows these eight views organized as a lattice of the type we shall discuss in Section 2. In naming the views in this diagram, we use the abbreviation p for `part`, s for `supplier`, and c for `customer`.

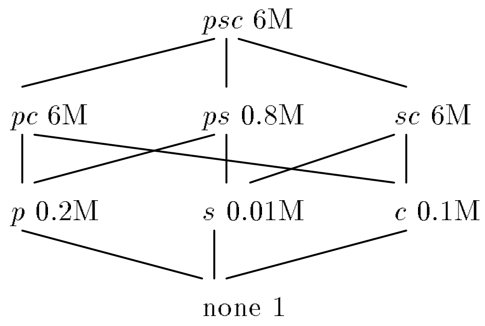


Figure 1: The eight views constructible by grouping on some of `part`, `supplier`, and `customer`

One possible user query is a request for an entire view. For example, the user may ask for the sales grouped by `part`. If we have materialized the view that groups only by `part` (view 5), we only need scan the view and output the answer. We can also answer this query using the view that groups by `part` and `customer` (view 2). In this case, since we have the total sales for each customer, for each part we need to sum the sales across all customers to get the result.

In this paper we assume the cost of answering a query is proportional to the number of rows examined. Thus, the cost of finding the total sales grouped by `part`, if (view 5) is materialized, is the cost of processing 0.2 million rows (the size of this view). To answer the same query using the `part`, `customer` view we would need to process 6 million rows.

Another kind of user query would ask only for the sales for a single part, say "widgets." If the views have no indexes, then we still have to scan the entire view (or half on the average) to answer this question. Thus, the same comparison, 0.2M rows for view 5 versus 6M rows for view 2, would apply to this query. If, however, the appropriate indexes are available in both views, finding sales of widgets requires only one row access from view 5, while in view 2 we would have to access an average of 6M/0.2M = 30 rows.[1] However, regardless of whether or not the materialized views are indexed, we expect that the cost of answering each of these queries — whole view or a single cell — would be proportional to the size of the view from which we answered the query. We shall discuss the cost model in more detail in Section 3.

There are some interesting questions we can now ask:

1. How many views must we materialize to get reasonable performance?
2. Given that we have space S, what views do we materialize so that we minimize average query cost?
3. If we're willing to tolerate an $X\%$ degradation in average query cost from a fully materialized data cube, how much space can we save over the fully materialized data cube?

In this paper, we provide algorithms that help us answer the above questions and provide near optimal results.

In the above example, a fully materialized data cube would have all the views materialized and thus have slightly more than 19 million rows.

Now let us see if we can do better. To avoid going to the raw data, we need to materialize the view grouping by `part`, `supplier`, and `customer` (view 1), since that view cannot be constructed from any of the other views. Now consider the view grouping by `part` and `customer` (view 2). Answering any query using this view will require us to process 6 million rows. The same query can always be answered using the view grouping by `part`, `supplier`, and `customer`, which again requires processing of 6 million rows. Thus there is no advantage to materializing the view grouping by `part` and `customer`. By similar reasoning, there is no advantage materializing the view grouping by `supplier` and `customer` (view 4). Thus we can get almost the same average query cost using only 7 million rows, an improvement of more than 60% in terms of space consumed and thus in the cost of creating the data cube.

Thus by cleverly choosing what parts of the data cube to materialize, we can reap dramatic benefits.

1.3 Related Work

Multi-dimensional data processing (also known as OLAP) has enjoyed spectacular growth of late. There are two basic implementation approaches that facilitate OLAP. The first approach is to eschew SQL and relational databases and to use proprietary multi-dimensional database (MDDB) systems and APIs for OLAP. So while the raw data is in relational data warehouses, the data cube is materialized in an MDDB. Users query the data cube, and the MDDB efficiently retrieves the value of a cell given its address. To allocate only space for those cells present in the raw data and not *every possible cell* of the data cube, a cell-address hashing scheme is used. Arbor's Essbase [ESS] and many other MDDBs are implemented this way. Note, this approach still materializes all the cells of the data cube present in raw data, which can be very large.

The other approach is to use relational database systems and let users directly query the raw data. The issue of query performance is attacked using smart indexes and other conventional relational query optimization strategies. There are many products like BusinessObjects and Microstrategy's DSS Agent that take this tack. However, MDDBs retain a significant performance advantage. Performance in relational database systems though can be improved dramatically by materializing the data cube into summary tables.

[1] Here we disregard the number of index nodes accessed.

The relational approach is very scalable and can handle very large data warehouses. MDDBs on the other hand have much better query performance, but are not very scalable. By materializing only selected parts of the data cube, we can improve performance in the relational database, and improve scalability in MDDBs. There are products in both the relational world [STG], and the MDDB world (Sinper's Spreadsheet Connector) that materialize only parts of the data cube. We believe however that this paper is the first to investigate this fundamental problem in such detail.

[GBLP95] discusses generalizing the SQL `GROUP-BY` operator to a data cube operator. They introduce the notion of "ALL" that we mention. However, they also claim the size of the entire data cube is not much larger than the size of the corresponding `GROUP-BY`. We believe differently.[2] As we saw in the TPC-D database, the data cube is usually much larger: more than three times larger than the corresponding `GROUP-BY` (`part`, `supplier`, `customer`).

1.4 Paper Organization

The paper is organized as follows. In Section 2 we introduce the lattice framework to model dependency among views. We also show how the lattice framework models more complex groupings that involve arbitrary hierarchies of attributes. Then in Section 3, we present the query-cost model that we use in this paper. Section 4 presents a general technique for producing near-optimal selections of materialized views for problems based on arbitrary lattices. In Section 5, we consider the important special case of a "hypercube" lattice, where the views are each associated with a set of attributes on which grouping occurs. The running example of Section 1.2 is such a hypercube.

2 The Lattice Framework

In this section we develop the notation for describing when one data-cube query can be answered using the results of another. As an example, in Section 1.2 we saw that for the data-cube, queries that we might wish to materialize are completely specified by giving the attributes in their `GROUP-BY` clause. We may thus denote a view or a query (which are the same thing) by giving its grouping attributes inside parenthesis. For example the query with grouping attributes `part` and `customer` is denoted by (`part`, `customer`). We also saw that views defined by supersets can be used to answer queries involving subsets.

2.1 The Dependence Relation on Queries

We may generalize the observations of Section 1.2 as follows. Consider two queries Q_1 and Q_2. We say $Q1 \preceq Q2$ if and only if $Q1$ can be answered using only the results of query Q_2. We then say that Q_1 is *dependent* on Q_2. For example, in Section 1.2, the query (`part`), can be answered using only the results of the query (`part`, `customer`). Thus (`part`) $\preceq$ (`part`, `customer`). There are certain queries that are not comparable with each other using the $\preceq$ operator. For example: (`part`) $\not\preceq$ (`customer`) and (`customer`) $\not\preceq$ (`part`).

The $\preceq$ operator imposes a partial ordering on the queries. We shall talk about the views of a data-cube problem as forming a lattice [TM75]. In order to be a lattice, any two elements (views or queries) must have a least upper bound and a greatest lower bound according to the $\preceq$ ordering. However, in practice, we only need the assumptions that

[2] The analysis in [GBLP95], assumes that every possible cell of the data cube exists. However, in most cases, data cubes are sparse: only a small fraction of all possible cells are present. In such cases, the size of the data cube can be much larger than the corresponding `GROUP-BY`. In fact, the sparser the data cube, the larger is the ratio of the size of the data cube to the size of the corresponding `GROUP-BY`.

1. $\preceq$ is a partial order, and

2. There is a *top* element, a view upon which every view is dependent.

2.2 Lattice Notation

We denote a lattice with set of elements (queries or views in this paper) L and dependence relation $\preceq$ by $\langle L, \preceq \rangle$. For elements a and b of a lattice $\langle L, \preceq \rangle$, $a \prec b$ means that $a \preceq b$ and $a \neq b$.

The ancestors and descendants of an element of a lattice $\langle L, \preceq \rangle$, are defined as follows:

$$ancestor(a) = \{b \mid a \preceq b\}$$

$$descendant(a) = \{b \mid b \preceq a\}$$

Note that every element of the lattice is its own descendant and its own ancestor. The immediate proper ancestors of a given element a in the lattice belong to a set we shall call $next(a)$. Formally,

$$next(a) = \{b \mid a \prec b, \nexists c, a \prec c, c \prec b\}$$

2.3 Lattice Diagrams

It is common to represent a lattice by a *lattice diagram*, a graph in which the lattice elements are nodes and there is an edge from a below to b above if and only if b is in $next(a)$. Thus, for any two lattice elements x and y, the lattice diagram has a path downward from y to x if and only if $x \preceq y$.

EXAMPLE 2.1 The hypercube of Fig. 1 is the lattice diagram of the set of views discussed in Section 1.2. $\square$

2.4 Hierarchies

In most real-life applications, dimensions of a data cube consist of more than one attribute, and the dimensions are organized as hierarchies of these attributes. A simple example is organizing the time dimension into the hierarchy: `day`, `month`, and `year`. Hierarchies are very important, as they form a basis of two very commonly used querying operations: "drill-down" and "roll-up." Drill-down is the process of viewing data at progressively more detailed levels. For example, a user drills down by first looking at the total sales per year and then total sales per month and finally, sales on a given day. Roll-up is just the opposite: it is the process of viewing data in progressively less detail. In roll-up, a user starts with total sales on a given day, then looks at the total sales in that month and finally the total sales in that year.

In the presence of hierarchies, the dependency lattice $\langle L, \preceq \rangle$ is more complex than a hypercube lattice. For example, consider a query that groups on the time dimension and no other. When we use the time hierarchy given earlier, we have the following three queries possible: (`day`), (`month`), (`year`), each of which groups at a different granularity of the time dimension. Further,

$$(\texttt{year}) \preceq (\texttt{month}) \preceq (\texttt{day}).$$

In other words, if we have total sales grouped by `month`, for example, we can use the results to compute the total sales grouped by `year`. Hierarchies introduce query dependencies that we must account for when determining what queries to materialize.

To make things more complex, hierarchies often are not total orders but partial orders on the attributes that make up a dimension. Consider the time dimension with the hierarchy `day`, `week`, `month`, and `year`. Since months and years cannot be divided evenly into weeks, if we do the grouping by `week` we cannot determine the grouping by `month` or `year`. In other words: (`month`) $\not\preceq$ (`week`), (`week`) $\not\preceq$ (`month`), and similarly for `week` and `year`. When we include the `none` view corresponding to no time grouping at all, we get the lattice for the time dimension shown in the diagram of Fig. 2.

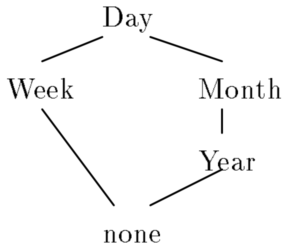


Figure 2: Hierarchy of time attributes

2.5 Composite Lattices for Multiple, Hierarchical Dimensions

We are faced with query dependencies of two types:

1. Query dependencies caused by the interaction of the different dimensions with one another. The example in Section 1.2 and the corresponding lattice in Fig. 1 is an example of this sort of dependency.

2. Query dependencies within a dimension caused by attribute hierarchies.

If we are allowed to create views that independently group by any or no member of the hierarchy for each of n dimensions, then we can represent each view by an n-tuple $(a_1, a_2, \ldots, a_n)$, where each a_i is a point in the hierarchy for the ith dimension. This lattice is called the *direct product* of the dimensional lattices [TM75]. We directly get a $\preceq$ operator for these views by the rule

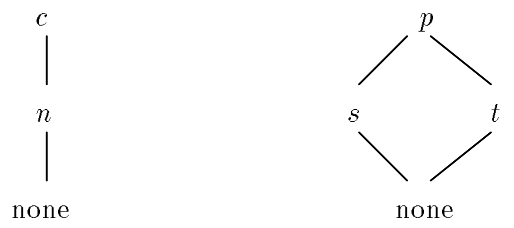
$$(a_1, a_2, \ldots, a_n) \preceq (b_1, b_2, \ldots, b_n) \text{ if and only if } a_i \preceq b_i \text{ for all } i$$

We illustrate the building of this direct-product lattice in the presence of hierarchies using an example based on the TPC-D benchmark.

EXAMPLE 2.2 In Example 1.1, we mentioned the TPC-D benchmark database. In this example we focus further on two dimensions: `part` and `customer`. Each of these dimensions is organized into hierarchies. The dimensional lattices for the dimension queries are given in Fig. 3. These dimension lattices have already been modified to include the attribute (`none`) as the lowest element.

The `customer` dimension is organized into the following hierarchy: individual customers, denoted by attribute c, are grouped together based on their country of residence, denoted by attribute n. The coarsest level of grouping is none at all, and this grouping is denoted by the attribute `none`.

For the `part` dimension, the individual parts are denoted by attribute p. These individual parts are grouped together based on their size denoted by attribute s. They are also grouped together based on their types, denoted by attribute t. Note neither of s and t is $\preceq$ the other. Finally, we have the attribute `none` as the smallest element in this lattice. The direct-product lattice is shown in Fig. 4. Note, when a dimension's value is `none` in a query, we do not specify the dimension in the query. Thus for example, (s,`none`) is written as (s). □



(a) Customer (b) Part

Figure 3: Hierarchies for the `customer` and `part` dimensions

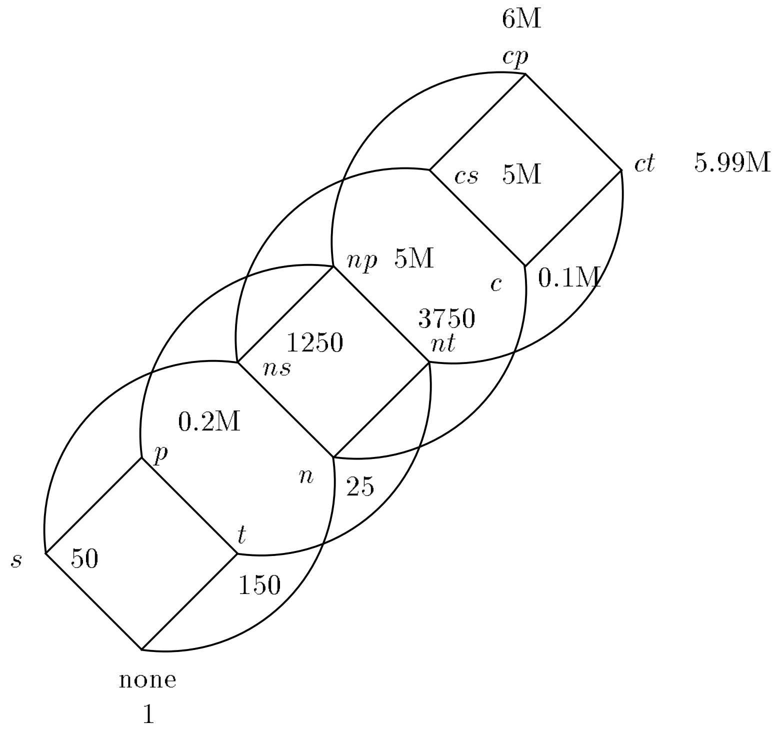


Figure 4: Combining two hierarchical dimensions

The lattice framework, we present and advocate in this paper, is advantageous for several reasons:

1. It provides a clean framework to reason with dimensional hierarchies, since hierarchies are themselves lattices. As can be seen in Fig. 4, the direct-product lattice is not always a hypercube when hierarchies are not simple. Current data cube approaches are unable to integrate dependencies caused by dimensional hierarchies cleanly with the dependencies caused by inter dimensional interaction, as we do. As we shall see shortly in Section 4, doing so is very important in deciding which views need to be materialized for best query performance.

2. We can model the common queries asked by users better using a lattice framework. Users usually don't jump between unconnected elements in this lattice, they move along the edges

of the lattice. In fact, drill-down is going up (going from a lower to higher level) a path in this lattice, while roll-up is going down a path.

3. The lattice approach also tells us in what order to materialize the views. Thus for example, let us decide to materialize views $S = \{Q_1, \ldots, Q_N\}$. Since some queries in set S may be dependent on others, we need not go to the raw data to materialize every view. By using views that are already materialized to materialize other views in S, we can reduce the total materialization time dramatically. Doing so translates to reducing the time required to create the data cube. Consider the lattice, $\langle S, \preceq \rangle$, in which all the views in S are elements. Perform a topological sort of S based on the $\preceq$ operator. Arrange the elements of S in descending order with respect to operator $\preceq$. Let the order be $Q_1^s, \ldots, Q_N^s$. We materialize the views in this order. The first few views have no proper-ancestor elements in S. To materialize these views we must access raw data. Thereafter, all the views can be materialized from views materialized earlier. In materializing a view Q_i^s, we use its proper-ancestor in S, Q_j^s, $j < i$, with the smallest number of rows.

3 The Cost Model

In this section, we review and justify our assumptions about the "linear cost model," in which the time to answer a query is taken to be equal to the space occupied by the view from which the query is answered. We then consider some points about estimating sizes of views without materializing them and give some experimental validation of the linear cost model.

3.1 The Linear Cost Model

Let $\langle L, \preceq \rangle$ be a lattice of queries (views). To answer a query Q we choose an ancestor of Q, say Q_A, that has been materialized. We thus need to process the table corresponding to Q_A to answer Q. The cost of answering Q is a function of the size of the table for Q_A. In this paper, we choose the simplest cost-model:

- The cost of answering Q is the number of rows present in the table for that query Q_A used to construct Q.

As we discussed in Section 1.2, not all queries ask for an entire view, such as a request for the sales of all parts. It is at least as likely that the user would like to see sales for a particular part or a few parts. If we have the appropriate index structure, and the view (`part`) is materialized, then we can get our answer in $O(1)$ time. If there is not an appropriate index structure, then we would have to search the entire (`part`) view, and the query for a single part takes almost as long as producing the entire view.

If, for example, we need to answer a query about a single part from some ancestor view such as (`part`, `supplier`) we need to examine the entire view. It can be seen that a single scan of the view is sufficient to get the `total sales` of a particular part. Now on the other hand if we wish to find the `total sales` for each part from the ancestor view (`part`, `supplier`), we need to do an aggregation over this view. We can use either hashing or sorting (with early aggregation) [G93] to do this aggregation. The cost of doing the aggregation is a function of the amount of memory available and the ratio of the number of rows in the input to that in the output. In the best case, a single pass of the input is sufficient (for example, when the hash table fits in main memory). In practice, it has been observed that most aggregations take between one and two passes of the input data.

While the actual cost of queries that ask for single cells, or small numbers of cells, rather than a complete view, is thus complex, we feel it is appropriate to make an assumption of uniformity. That is, either the data structure used to store views supports efficient access to the desired cell, in which case the time required is proportional to the number of rows that must be aggregated to compute the value in the cell, or the data structure does not, in which case the behavior of single-cell and full-view queries are essentially the same.

In the first case, where a suitable data structure exists, we shall make the further assumption that, over time, many queries asking for (different) cells of the same view Q will occur. Some number of these queries, as a group, will have performance equivalent to that of a single query for the full view Q. That is, either their total time will equal that of reading Q, if it is materialized, or the time will be that of reading view Q_A, the preferred, materialized ancestor of Q. Thus, we may avoid the question of whether full-view or single-cell queries predominate, and treat all queries as full-view queries. Thus:

- We assume that all queries are identical to some element (view) in the given lattice.

Clearly there are other factors, not considered here, that influence query cost. Among them are the clustering of the materialized views on some attribute, and the indexes that may be present. More complicated cost models are certainly possible, but we believe the cost model we pick, being both simple and realistic, enables us to design and analyze powerful algorithms. We believe that our analysis of the algorithms we develop in Sections 4 and 5 reflects their performance under other cost models as well as under the model we use here.

3.2 Experimental Examination of the Linear Cost Model

A substantial validation of our cost model is shown in Fig. 5. On the TPC-D data, we asked for the total sales for a single supplier, under four conditions, using views of different granularities. We find an almost linear relationship between size and running time of the query. This linear relationship can be expressed by the formula:

$$T = m * S + c$$

Here T is the running time of the query on a view of size S. c gives the fixed cost: the overhead of running this query on a view of negligible size. In this case, the first row of the table in Fig. 5, gives the fixed cost of 2.07 seconds. m is the ratio of the query time to the size of the view, after accounting for the fixed cost. As can be seen in Fig. 5 this ratio is almost the same for the different views.

Source	Size	Time (sec.)	Ratio
From cell itself	1	2.07	not applicable
From view (`supplier`)	10,000	2.38	.000031
From view (`part, supplier`)	800,000	20.77	.000023
From view (`part, supplier, customer`)	6,000,000	226.23	.000037

Figure 5: Growth of query response time with size of view

3.3 Determining View Sizes

Our algorithms require knowledge of the number of rows present in each view. There are many ways of estimating the sizes of the views short of materializing all the views. One commonly used approach is to run our algorithms on a statistically significant but small subset of the raw data. In such a case, we can get the sizes of the views by actually materializing the views. We use this subset of raw data to determine which views we want to materialize. And we only materialize a few views from the entire raw data.

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 the largest attribute in each dimension). 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` and `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 Optimization of Data-Cube Lattices

Our most important objective is to develop techniques for optimizing the space-time tradeoff when implementing a lattice of views. The problem can be approached from many angles, since we may in one situation favor time, in another space, and in a third be willing to trade time for space as long as we get good "value" for what we trade away. In this section, we shall begin with a simple optimization problem, in which

1. We wish to minimize the average time taken to evaluate a view.

2. We are constrained to materialize a fixed number of views, regardless of the space they use.

Evidently item (2) does not minimize space, but in Section 4.6 we shall show how to adapt our techniques to a model that does optimize space utilization.

Even in this simple setting, the optimization problem is NP-complete; there is a straightforward reduction from Set-Cover. Thus, we are motivated to look at heuristics to produce approximate solutions. The obvious choice of heuristic is a "greedy" algorithm, where we select a sequence of views, each of which is the best choice given what has gone before. We shall see that this approach is always fairly close to optimal and in some cases can be shown to produce the best possible selection of views to materialize.

4.1 The Greedy Algorithm

Suppose we are given a data-cube lattice with space costs associated with each view. In this paper, the space cost is the number of rows in the view. Let $C(v)$ be the cost of view v. Suppose also that there is a limit k on the number of views, in addition to the top view, that we may select. After selecting some set S of views (which surely includes the top view), the *benefit* of view v relative to S, which we denote $B(v, S)$, is defined as follows.

1. For each $w \preceq v$, define the quantity B_w by:

(a) Let u be the view of least cost in S such that $w \preceq u$. Note that since the top view is in S, there must be at least one such view in S.

(b) If $C(v) < C(u)$, then $B_w = C(v) - C(u)$. Otherwise, $B_w = 0$.

2. Define $B(v, S) = \sum_{w \preceq v} B_w$.

In perhaps simpler terms, we compute the benefit of v by considering how it can improve the cost of evaluating views, including itself. For each view w that v covers, we compare the cost of evaluating w using v and using whatever view from S offered the cheapest way of evaluating w. If v helps, *i.e.*, the cost of v is less than the cost of its competitor, then the difference represents part of the benefit of selecting v as a materialized view. The total benefit $B(v, s)$ is the sum over all views w of the benefit of using v to evaluate w, providing that benefit is positive.

Now, we can define the *Greedy Algorithm* for selecting a set of k views to materialize. The algorithm is shown in Fig. 6.

```
S = {top view};
for i=1 to k do begin
    select that view v not in S such that B(v,S) is maximized;
    S = S union {v};
end;
resulting S is the greedy selection;
```

Figure 6: The Greedy Algorithm

EXAMPLE 4.1 Consider the lattice of Fig. 7. Eight views, named a through h have space costs as indicated on the figure. The top view a, with cost 100, must be chosen. Suppose we wish to choose three more views.

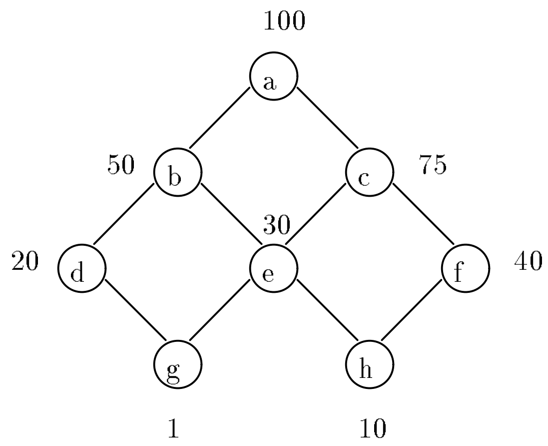


Figure 7: Example lattice with space costs

To execute the greedy algorithm on this lattice, we must make three successive choices of view to materialize. The column headed "First Choice" in Fig. 8 gives us the benefit of each of the views besides a. When calculating the benefit, we begin with the assumption that each view is evaluated using a, and will therefore have a cost of 100.

If we pick view b to materialize first, then we reduce by 50 its cost and that of each of the views d, e, g, and h below it. The benefit is thus 50 times 5, or 250, as indicated in the row b and first column of Fig. 8. As another example, if we pick e first then it and the views below it — g and h — each have their costs reduced by 70, from 100 to 30. Thus, the benefit of e is 210.

	First Choice	Second Choice	Third Choice
b	$50 \times 5 = 250$		
c	$25 \times 5 = 125$	$25 \times 2 = 50$	$25 \times 1 = 25$
d	$80 \times 2 = 160$	$30 \times 2 = 60$	$30 \times 2 = 60$
e	$70 \times 3 = 210$	$20 \times 3 = 60$	$20 + 20 + 10 = 50$
f	$60 \times 2 = 120$	$60 + 10 = 70$	
g	$99 \times 1 = 99$	$49 \times 1 = 49$	$49 \times 1 = 49$
h	$90 \times 1 = 90$	$40 \times 1 = 40$	$30 \times 1 = 30$

Figure 8: Benefits of possible choices at each round

Evidently, the winner in the first round is b, so we pick that view as one of the materialized views. Now, we must recalculate the benefit of each view V, given that the view will be created either from b, at a cost of 50, if b is above V, or from a at a cost of 100, if not. The benefits are shown in the second column of Fig. 8.

For example, the benefit of c is now 50, 25 each for itself and f. Choosing c no longer improves the cost of e, g, or h, so we do not count an improvement of 25 for those views. As another example, choosing f yields a benefit of 60 for itself, from 100 to 40. For h, it yields a benefit of 10, from 50 to 40, since the choice of b already improved the cost associated with h to 50. The winner of the second round is thus f, with a benefit of 70. Notice that f wasn't even close to the best choice at the first round.

Our third choice is summarized in the last column of Fig. 8. As an example of the most complex calculation, the benefit of e is 50. That number consists of 20 for each of e and g, which would reduce their costs of 50 (using d) to 30 (using e), plus 10 for the reduction of the cost of h from 40 (using f) to 30 (using e). The winner of the third round is d, with a benefit of 60, gained from the improvement to its own cost and that of g.

The greedy selection is thus b, d, and f. These, together with a, reduces the total cost of evaluating all the views from 800, which would be the case if only a was materialized, to 420. That cost is actually optimal.

□

EXAMPLE 4.2 Let us now examine the lattice suggested by Fig. 9. This lattice is, as we shall see, essentially as bad as a lattice can be for the case $k = 2$. The greedy algorithm, starting with only the top view a, first picks c, whose benefit is 4141. That is, c and the 40 views below it are each improved from 200 to 99, when we use c in place of a.

For our second choice, we can pick either b or d. They both have a benefit of 2100. Specifically, consider b. It improves itself and the 20 nodes at the far left by 100 each. Thus, with $k = 2$, the greedy algorithm produces a solution with a benefit of 6241.

However, the optimal choice is to pick b and d. Together, these two views improve by 100 each, themselves and the 80 views of the four chains. Thus, the optimal solution has a benefit of 8200.

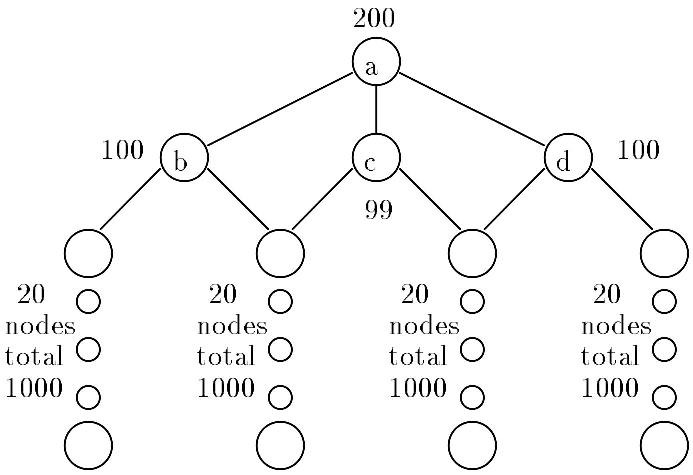


Figure 9: A lattice where the greedy algorithm does poorly

the ratio of greedy/optimal is 6241/8200, which is about 3/4. In fact, by making the cost of c closer to 100, and by making the four chains have arbitrarily large numbers of views, we can find examples for $k = 2$ with ratio arbitrarily close to 3/4, but no worse. □

4.2 An Experiment With the Greedy Algorithm

We ran the greedy algorithm on the lattice of Fig. 4, using the TPC-D database described earlier. Figure 10 shows the resulting order of views, from the first (top view, which is mandatory) to the twelfth and last view. The units of Benefit, Total Time and Total Space are number of rows. Note, the average query time is the total time divided by the number of views (12 in this case).

Number	Selection	Benefit	Total Time	Total Space
1.	*cp*	infinite	72M	6M
2.	*ns*	24M	48M	6M
3.	*nt*	12M	36M	6M
4.	*c*	5.9M	30.1M	6.1M
5.	*p*	5.8M	24.3M	6.3M
6.	*cs*	1M	23.3M	11.3M
7.	*np*	1M	22.3M	16.3M
8.	*ct*	0.01M	22.3M	22.3M
9.	*t*	small	22.3M	22.3M
10	*n*	small	22.3M	22.3M
11.	*s*	small	22.3M	22.3M
12.	none	small	22.3M	22.3M

Figure 10: Greedy order of view selection for TPC-D-based example

This example shows why it is important to materialize some views and also why materializing all views is not a good choice. The graph in Fig. 11 has the total time taken and the space consumed on the Y-axis, and the number of views picked on the X-axis. It is clear that for the first few views

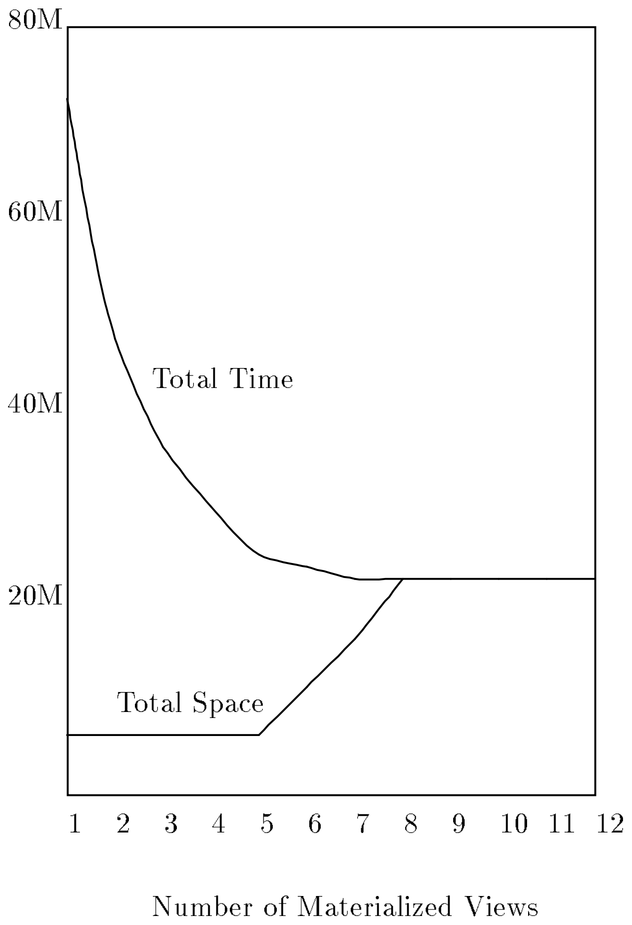


Figure 11: Time and Space for the greedy view selection for the TPC-D-based example.

we pick, with minimal addition of space, the query time is reduced substantially. After we have picked 5 views however, we cannot improve total query time substantially even by using up large amounts of space. For this example, there is a clear choice of when to stop picking views. If we pick the first five views — *cp*, *ns*, *nt*, *c*, and *p* — (*i.e.*, $k = 4$, since the top view is included in the table), then we get almost the minimum possible total time, while the total space used is hardly more than the mandatory space used for just the top view.

4.3 A Performance Guarantee for the Greedy Algorithm

We can show that no matter what lattice we are given, the greedy algorithm never performs too badly. Specifically, the benefit of the greedy algorithm is at least 63% of the benefit of the optimal algorithm. The precise fraction is $(e-1)/e$, where e is the base of the natural logarithms.

To begin our explanation, we need to develop some notation. Let $v_1, v_2, \ldots, v_k$ be the k views selected in order by the greedy algorithm. Let a_i be the benefit achieved by the selection of v_i, for $i = 1, 2, \ldots, k$. That is, a_i is the benefit of v_i, with respect to the set consisting of the top view and $v_1, v_2, \ldots, v_{i-1}$.

Let $w_1, w_2, \ldots, w_k$ be the optimal set of k views, *i.e.*, those that give the maximum total benefit. The order in which these views appear is arbitrary, but we need to pick an order. Given the w's in order $w_1, w_2, \ldots, w_k$, define b_i to be the benefit of w_i with respect to the set consisting of the top view plus $w_1, w_2, \ldots, w_{i-1}$. Define $A = \sum_{i=1}^{k} a_i$ and $B = \sum_{i=1}^{k} b_i$.

Now we must put an upper bound on the b's in terms of the a's. In the greedy solution, we can look at any view and see how much that view's cost has improved. We can also attribute its improvement to various of the v_i's.

EXAMPLE 4.3 In Example 4.1, the cost associated with g improved from 100 to 20. Of that improvement, 50 is attributed to the selection of b for the greedy solution and 30 to the subsequent selection of d. □

Next, we want to compare the improvement to an arbitrary view u effected by the v's and by the w's. Figure 12 suggests how the improvement in the cost of some u might be partitioned among the v's (top bar) and w's (lower bar). We have suggested that the improvement in u is due to v_1, v_3, and v_7, in the amounts suggested. Likewise, the larger improvement in u due to the w's is divided among w_2, w_3, w_5, and w_6, as shown.

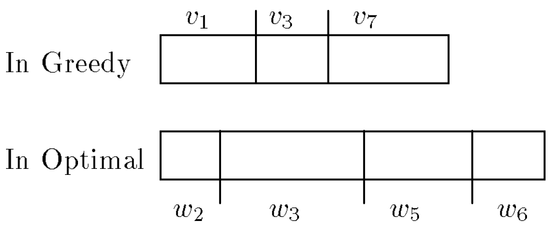


Figure 12: The improvement to any view u can be attributed to v's or w's

To show the upper bound on B/A we need to attribute each piece of the benefits b_i either to none of the v's or to one particular v_j. To do so, we examine the improvements to each view u as suggested in Fig. 12.

EXAMPLE 4.4 We can attribute the contribution of w_2 wholly to v_1, because the region for w_2 is contained within the region for v_1. The contribution of w_3 is divided among v_1, v_5, and v_7, in the proportions shown. The contribution of w_6 is not attributed to any of the v's, while part of w_5's contribution is attributed to v_7 and part is not attributed. □

In general, order the improvements in the same sequence that the views are chosen. Define x_{ij} to be the sum over all views u in the lattice of the amount of the benefit b_i (from w_i) that is attributed to v_j. An important inequality that holds for each j is

- $\sum_{i=1}^{k} x_{ij} \leq a_j$

That is, the sum of the pieces of the benefit of v_j that is attributed to the various w's cannot exceed the benefit of v_j.

We can observe several inequalities from the fact that none of the w's was picked in place of one of the v's (except in the case where $w_i = v_i$). Our first observation:

- For all i: $b_i \leq a_1$.

For if not, then w_i would have been picked first by the greedy algorithm instead of v_1. Similarly, considering the second choice by the greedy algorithm tells us:

- For all i: $b_i - x_{i1} \leq a_2$.

The reason is that the benefit w_i brings to the competition for second choice is b_i minus the amount of benefit of w_i that was covered by v_1; the latter is x_{i1}. The benefit a_2 must be at least as great as any competing choice. Generalizing the above, for each j we can write:

- For all i: $b_i - x_{i1} - x_{i2} - \cdots - x_{i,j-1} \leq a_j$.

If we sum each of the above equations over i and remember that

1. $\sum_{i=1}^{k} b_i = B$
2. $\sum_{i=1}^{k} a_i = A$
3. $\sum_{i=1}^{k} x_{ij} \leq a_j$

we get the family of inequalities in Fig. 13.

$$\begin{array}{ll}
(1) & B \leq ka_1 \\
(2) & B \leq ka_2 + a_1 \\
(3) & B \leq ka_3 + a_1 + a_2 \\
 & \quad \cdots \\
(k) & B \leq ka_k + a_1 + a_2 + \cdots + a_{k-1}
\end{array}$$

Figure 13: Inequalities bounding the benefit of the optimal solution

We thus conclude that B is no greater than the minimum of the right sides of the inequalities of Fig. 13. It is easy to show that if for a given $a_1, a_2, \ldots, a_k$ the right sides are unequal, then we can transfer some quantity from some a_j to a_{j-1} or a_{j+1}, resulting in no change to A but a looser bound on B. Our conclusion is that

- For a fixed A, the tightest bound on B occurs when all the right sides in Fig. 13 are equal.

Notice, however, that the difference between the ith right side and the $(i+1)$st right side is $ka_{i+1} - (k-1)a_i$. Since this difference must be 0, we conclude that $a_i = \frac{k}{k-1}a_{i+1}$. For these values of the a's, we observe:

- $A = \sum_{i=0}^{k-1} (\frac{k}{k-1})^i a_k$, because the terms of the sum are $a_1, a_2, \ldots, a_k$.
- $B \leq k(\frac{k}{k-1})^{k-1} a_k$. This bound comes most easily from the first inequality of Fig. 13, but it could come from any of the inequalities, since they have equal right sides.

Our conclusion, taken from the ratio of the above formulas, is:

$$\begin{aligned}
A/B &\geq \textstyle\sum_{i=0}^{k-1} (\frac{k}{k-1})^{i-k+1} \\
&\geq \tfrac{1}{k}(1 + \tfrac{k-1}{k} + (\tfrac{k-1}{k})^2 + \cdots + (\tfrac{k-1}{k})^{k-1}) \\
&\geq 1 - (\tfrac{k-1}{k})^k
\end{aligned}$$

For example, for $k = 2$ we get $A/B \geq 3/4$; *i.e.*, the greedy algorithm is at least 3/4 of optimal.

4.4 Matching the Worst Possible Ratio for the Greedy Algorithm

We saw in Example 4.2 that for $k = 2$ there were specific lattices that approached 3/4 as the ratio of the benefits of the greedy and optimal algorithms. In fact,

- For any k, the ratio $A/B = 1 - (\frac{k-1}{k})^k$ can be reached.

We omit the detailed construction of the sequence of bad cases, one for each k, from this paper. However, the intuitive idea is as follows. Construct a lattice with k identical subtrees, each of whose roots gives the same, large benefit B. These k nodes will be the optimal choice.

Now add to the lattice one node v_1 that includes among its descendants $1/k$th of each of these subtrees, plus something extra so v_1's benefit is just slightly higher than B. Thus, v_1 will be the first greedy choice. Then, add to the lattice a node v_2 that covers $1/k$th of that portion of each subtree that was not covered by v_1, plus something extra, so v_2's benefit is just higher than $B(k-1)/k$. Note that after v_1 was picked, the benefit of each of the subtree roots shrinks to $B(k-1)/k$, so v_2 becomes the second greedy choice. Continue in this manner, with each of the greedy-choice nodes covering $1/k$th of that portion of the subtrees that was not covered by any of the previous greedy choices. It is then possible to show that, while the optimal choice has benefit kB, the benefit of the greedy choices is just $(1 - (\frac{k-1}{k})^k)B$. We conclude that

- For all k, the lower bound on the ratio of the greedy and optimal benefits is exact. That is, the ratio $1 - (\frac{k-1}{k})^k$, shown in Section 4.3 actually occurs for at least one lattice for each k.

As $k \to \infty$, $(\frac{k-1}{k})^k$ approaches $1/e$, so $A/B \geq 1 - \frac{1}{e} = (e-1)/e = 0.63$. That is, for no lattice whatsoever does the greedy algorithm give a benefit less than 63% of the optimal benefit. Conversely, the sequence of bad examples suggested by Section 4.4 shows that this ratio cannot be improved upon.

4.5 Cases Where Greedy is Optimal

The analysis of Section 4.3 also lets us discover certain cases when the greedy approach is optimal, or very close to optimal. Here are two situations where we never have to look further than the greedy solution.

1. If a_1 is much larger than the other a's, then greedy is close to optimal. To see why, consider the last inequality in Fig. 13. It essentially says that $B \leq a_1$. Since A is approximately a_1 in this case, we have $B \leq A$ (approximately).

2. If all the a's are equal then greedy is optimal. In proof, consider the first inequality of Fig. 13. It says that $B \leq ka_1$, but since all the a's are equal, $ka_1 = A$. Similarly, if the a's are approximately equal, then B is approximately A, and the greedy approach is near-optimal.

4.6 Extensions to the Basic Model

There are at least two ways in which our model fails to reflect reality.

1. The views in a lattice are unlikely to have the same probability of being requested in a query. Rather, we might be able to associate some probability with each view, representing the frequency with which it is queried.

2. Instead of asking for some fixed number of views to materialize, we might instead allocate a fixed amount of space to views (other than the top view, which must always be materialized).

Point (1) requires little extra thought. When computing benefits, we weight each view by its probability. The greedy algorithm will then have exactly the same bounds on its performance: at least 63% of optimal.

Point (2) presents an additional problem. If we do not restrict the number of views selected, but fix their total space, then we need to consider the benefit of each view *per unit space* used by a materialization of that view. The greedy algorithm again seems appropriate, but there is the additional complication that we might have a very small view with a very high benefit per unit space, and a very large view with almost the same benefit per unit space. Choosing the small view excludes the large view, because there is not enough space available for the large view after we choose the small.

On the other hand, if no view's space is more than some fraction f of the total space allowed, then the same analysis as given above says that the ratio of the benefits of the greedy and optimal algorithms is no less than $0.63 - f$.

5 The Hypercube Lattice

Arguably, the most important class of lattices are the *hypercubes*, in which the views are vertices of an n-dimensional cube for some n. The intuition is that there are n attributes $A_1, A_2, \ldots, A_n$ on which grouping may occur and an $(n+1)$st attribute B whose value is aggregated in each view. That is, each view is of the form

```
SELECT C1, C2,...,Cp, SUM(B)
FROM R
GROUP BY C1, C2,...,Cp;
```

where the C's are some subset of the A's. Figure 1 was an example of a hypercube lattice with $n = 3$, taken from the TPC-D benchmark database.

The top view groups on all n attributes. We can visualize the views organized by ranks, where the ith rank from the bottom is all those views in which we group on i attributes. There are $\binom{n}{i}$ views in rank i.

5.1 The Equal-Domain-Size Case

We can, of course, apply the greedy algorithm to hypercube lattices, either looking for a fixed number of views to materialize, or looking for a fixed amount of space to allocate to views. However, because of the regularity of this lattice, we would like to examine in more detail some of the options for selecting a set of views to materialize.

In our investigations, we shall first make an assumption that is unlikely to be true in practice: all attributes $A_1, A_2, \ldots, A_n$ have the same domain size, which we shall denote r. The consequence of this assumption is that we can easily estimate the size of any view. In Section 5.4, we shall consider what happens when the domain sizes vary. It will be seen that the actual views selected to materialize will vary, but the basic techniques do not change to accommodate this more general situation.

When each domain size is r, and data in the data cube is distributed randomly, then there is a simple way to estimate the sizes of views. The combinatorics involved is complex, but the intuition should be convincing. Suppose only m cells in the top element of our lattice appear in the raw data. If we group on i attributes, then the number of cells in the resulting cube is r^i. To a first approximation, if $r^i \geq m$, then each cell will contain at most one data point, and m of the cells will be nonempty. We can thus use m as the size of any view for which $r^i \geq m$. On the other hand, if $r^i < m$, then almost all r^i cells will have at least one data point. Since we may collapse all the data points in a cell into one aggregate value, the space cost of a view with $r^i < m$ will be approximately r^i. The view size as a function of the number of grouped attributes is shown in Fig. 14.

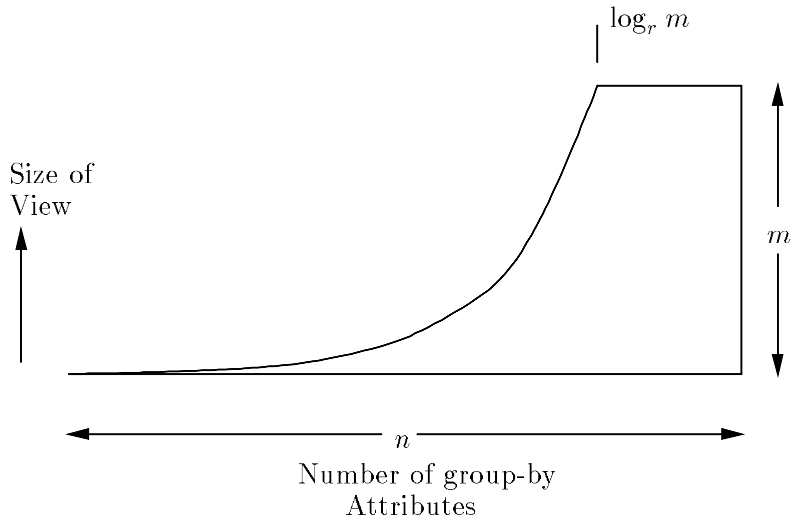


Figure 14: How the size of views grows with number of grouped attributes

The size of views grows exponentially, until it reaches the size of the raw data, and then ceases to grow. Notice that the actual data taken from Fig. 1 almost matches this pattern. The top view and the views with two grouping attributes have the same, maximum size, except that the view *ps* (`part`, `supplier`) has somewhat fewer rows, due to the fact that the benchmark explicitly sets it to have fewer rows.

5.2 The Space-Optimal Solution

One natural question to ask when investigating the time/space tradeoff for the hypercube is what is the average time for a query when the space is minimal. Space is minimized when we materialize only the top view. Then every query takes time m, and the total time cost for all 2^n queries is $m2^n$.

5.3 The Time-Optimal Solution

At the other extreme, we could minimize time by materializing every query. Then, the time to execute each query would be equal to its own size, and the total space needed to store the data cube would equal the time taken to execute each query once. However, under our model, the views whose size is m, which will be all those queries that group by at least $k = \log_r m$ attributes, will have the maximum size m, and these may as well be executed through the top view.

Define the *rank* of a query or view to be the number of attributes on which it is grouped. Queries organize themselves into ranks as suggested in Fig. 15. There are two parameters of the problem whose relationship determines what the time and space are for this case.

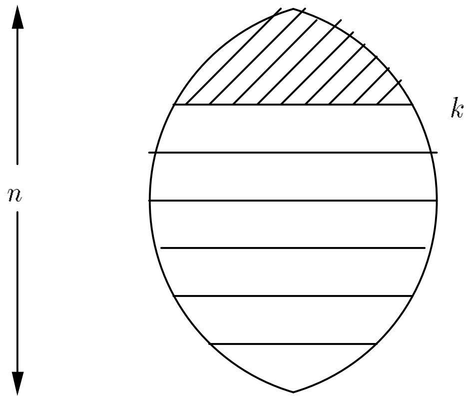


Figure 15: The ranks of the hypercube lattice

1. The rank k at which the "cliff" in Fig. 14 occurs. That is, $k = \log_r m$, or $r^k = m$.

2. The rank j such that $r^j\binom{n}{j}$ is maximized. This quantity is the sum of the sizes of the queries of rank j, provided that j is still in the growing portion of the curve in Fig. 14. It is not hard to show that at the maximum, $r = j/(n-j)$, or $j = nr/(r+1)$.

If $k > j$, then the bulk of the space and time are used for queries that have rank approximately j. That is, to a first approximation, all 2^n queries use space $r^j = r^{nr/(r+1)}$, and the total space and time are both $(2r^{r/(r+1)})^n$.

On the other hand, if $k < j$, then almost all the time and space is consumed executing queries in the flat part of the curve of Fig. 14. Thus, except for some materialized queries in ranks less than k, whose total space is negligible, we can execute all the queries using the top view. Thus, the space used is little more than m (recall that we do not have to materialize views in ranks between k and $n-1$).

Now, let us consider the time requirements. It is again possible to neglect the "small" queries of rank below k. Each of the queries of rank k or more requires time m, so we have only to estimate the total number of these queries. there are two subcases.

1. $k < j$ and $k \leq n/2$. Here, at least half the views take m space and time, so the the time is approximately $m2^n$. Thus, in this case the time and space are both approximately that of the space-optimal solution.

2. $k < j$ and $k > n/2$. Now, most of the space (and therefore time) is concentrated around rank j, but there are relatively few of these views. The total time is approximated by $\sum_{i=j}^{n}\binom{n}{i}r^i$. Since the total time per rank decreases as i grows above j, we can approximate this sum by its first term: $\binom{n}{j}r^j$.

Figure 16 summarizes the time and space used for the three tradeoff points studied.

5.4 Extension to Varying Domain Sizes

Suppose now that the domains of each attribute do not each have r equally-likely values. The next simplest model is to assume that for each dimension, values are equally likely, but the number of values varies, with r_i values in the ith dimension for $i = 1, 2, \ldots, n$.

Strategy	k, j, and n	Space	Time
Space-optimal		m	$m2^n$
Time-optimal	$k > j$	$(2r^{r/(r+1)})^n$	$(2r^{r/(r+1)})^n$
	$k < j$ and $k \leq n/2$	m	$m2^n$
	$k < j$ and $k > n/2$	m	$\binom{n}{j} r^j$

Figure 16: Summary of time- and space- optimal strategies for hypercube

Now, the "cliff" suggested in Fig. 14 does not occur at a particular rank, but rather the cliff is distributed among ranks. If a view groups by a set of attributes whose domain sizes multiply to something less than m, the total number of rows in the raw data, then that view behaves as if it is "below the cliff," *i.e.*, of rank less than k. If the product of the domain sizes for the grouping attributes of a view exceeds m, then this view is "on top of the cliff." However, the fundamental behavior suggested by Fig. 14 is unchanged. As we "drill down" by grouping on progressively more attributes, our views stay fixed at the maximum size, until at some point the size drops, whereupon the size of views decreases exponentially.

The analysis of the space-optimal case does not change. For the time-optimal case, we need to replace the condition $k < j$ by "most of the space used by the various queries is among those views whose product of domain sizes for grouped attributes is less than m." Likewise, $k \leq n/2$ becomes "most views have a product of domain sizes for their grouped attributes that exceeds m."

6 Conclusions

In this paper we have investigated the problem of deciding which set of cells (views) in the data cube to materialize in order to minimize query response times. Materialization of views is an essential query optimization strategy for decision-support applications. In this paper, we make the case that the right selection of the views to materialize is critical to the success of this strategy. We use the TPC-D benchmark database as an example database in showing why it is important to materialize some part of the data cube but not all of the cube.

Our second contribution is a lattice framework that models multidimensional analysis very well. Our greedy algorithms work on this lattice and pick the right views to materialize, subject to various constraints. The greedy algorithm we give performs within a small constant factor of the optimal solution for many of the constraints considered. Finally, we looked at the most common case of the hypercube lattice and investigated the time-space trade-off in detail.

6.1 Future Work

We are investigating the following topics.

- Progressively more realistic cost models. Models that capture indexing and clustering are the first step in this direction.

- Data cubes that are stored in MDDB systems. The poor scalability of many MDDB systems is due to the fact that they materialize the entire data cube. Thus the selection of the right cells to materialize is very important here too.

- Dynamic materialization. The views, in some sense, form a memory hierarchy with differing access times. In conventional memory hierarchies, data is usually assigned to different memory stores (like cache, or main memory) dynamically based on the run time accesses. It would be interesting to compare dynamic materialization with the static materialization scheme we investigate in this paper.

Acknowledgements

We thank Bala Iyer and Piyush Goel at IBM Santa Teresa Labs for help with the experiments.

References

[CS94] S. Chaudhuri and Kyuseok Shim. Including Group-By in Query Optimization. In *Proceedings of the Twentieth International Conference on Very Large Databases (VLDB)*, pages 354–366, Santiago, Chile, 1994.

[ESS] Arbor Software Inc. Multidimensional Analysis: Converting Corporate Data into Strategic Information. White Paper. At http://www.arborsoft.com/papers/multiTOC.html

[G93] Goetz Graefe. Query Evaluation Techniques for Large Databases. In *ACM Computing Surveys*, Vol. 25, No. 2, June 1993.

[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.

[GHQ95] A. Gupta, V. Harinarayan, and D. Quass. Aggregate-Query Processing in Data Warehousing Environments . In *Proceedings of the 21st International VLDB Conference*, pages 358-369, 1995.

[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.

[OG95] P. O'Neill and G. Graefe. Multi-Table Joins Through Bitmapped Join Indexes. In *SIGMOD Record*, pages 8-11, September 1995.

[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.

[Rad95] Alan Radding. Support Decision Makers With a Data Warehouse. In *Datamation*, March 15, 1995.

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

[TM75] J. P. Tremblay and R. Manohar. *Discrete Mathematical Structures with Applications to Computer Science.* . McGraw Hill Book Company, New York, 1975.

[X94] John Xenakis, editor. Multidimensional Databases. In *Application Development Strategies*, April 1994.

[YL95] W. P. Yan and P. A. Larson. Eager Aggregation and Lazy Aggregation. In *Proceedings of the 21st International VLDB Conference*, pages 345-357, 1995.