Computing Query Previews in the Flamenco System

Kevin Chen
UC Berkeley
Computer Science Division

January 26, 2004

Abstract

The Flamenco system [7, 6] is a web search interface that allows users to browse through large data sets using predefined hierarchical faceted metadata. It is built on top of a conventional relational database and currently scales to collections of several tens of thousands of items. In the current implementation, the system translates each user query into multiple SQL group-by commands in order to obtain query preview information for possible future queries. These group-by's take up a significant fraction of the query processing time. In this note, we describe an optimization that allows us to speed up the group-by computations dramatically. Our ideas have some similarity to the work of Beyer and Ramakrishnan on computing iceberg data cubes [4].

1 Introduction

The Flamenco system [7, 6] seeks to provide an innovative search interface for large collections of data on the web by closely integrating text search and browsing via *hierarchical faceted* metadata. By *faceted* we mean that metadata consists of several orthogonal attributes and by *hierarchical* we mean that each facet's values can be represented as a rooted tree via a partial ordering relation, or more generally as a directed acyclic graph or DAG.

As a motivating example of such a data set, we consider Spiro, a collection of roughly 40,000 architectural images from the architecture department at UC Berkeley. Each image is annotated with facets like its location, material or architect. These facets can be hierarchical, as in the case of location (e.g. $NorthAmerica \rightarrow USA \rightarrow California \rightarrow Berkeley$) or multi-valued, as in the case of material (e.g. a building can be made of both wood and brick). The multi-valued case is an important one and we shall be return to it again in subsequent sections.

Prior research on web search interfaces has established the importance of query previews [16] for possible next choices. In other words, a user should know ahead of time how many results he or she will obtain if a link is chosen. This eliminates the frustrating experience of obtaining zero or very few results one finds in many current search systems. Less research has been done on providing some form of approximate query previews (which could be as simple as just providing a binary indicator - "no results" versus "at least one result"), and it is not clear if such an approach would be either helpful to real users or significantly more efficient from a computational point of view. For the time being, we will consider only exact query previews, though we will revisit this issue later.

In the Flamenco system, providing these query previews involves computing a separate **SELECT** and **GROUP BY** SQL command for each facet in the metadata schema, each time the user makes a query to the system. In other words, if we had k facets in the metadata schema, each user query would be translated into k separate database *GROUP BY* queries. In practice, our collections usually have around ten facets, so this represents a slow down of a roughly a factor of ten over a system that does not provide query previews. In the current implementation, this computation is a major bottleneck in the running time for processing a query, severely limiting the scalability of the system.

In this note, we describe an optimization for this operation similar to the work of Beyer and Ramakrishnan on *iceberg data cubes* [4]. In section 2, we give some basic background on data cubes and algorithms for computing them. In section 3, we describe our techniques, and in section 4 we give experimental results for how well the techniques do in practice. For all our experiments, we use the Spiro collection as a test bed. Finally, we conclude in section 5.

2 Background on Data Cubes

The data cube operation was introduced by Gray et al. in 1996 [9] as a generalization of the standard SQL **GROUP BY** command. Given a collection of data and a set of k dimensions, the **CUBE BY** operator aggregates the items by *every* subset of the k dimensions [1]. For example, given a prototypical transaction database with dimensions like customers, stores, and time, a business may want to view its sales data aggregated by customer and store, time and store, just store by itself and so on. The **CUBE BY** operator allows the user to fetch all these views (in fact, all possible views) with a single simple (and hopefully, well-optimized) SQL command. The literature on data cubes has grown quite large over the last few years, and a number of good summaries of existing data cube work are available (see for example [4]) so here we summarize only the

[1]In the database and datacube literature, the term "dimension" is usually used synonymously with the term "attribute". In the context of the Flamenco system, we shall reserve the term "attribute" for the parts of the metadata which are not used by the system for navigation, but which are nevertheless associated with data items. In the Spiro example, an example of an attribute is the photo identification number of the image. Furthermore, we will handle the tree structure of the facets by treating each level of each facet as an independent dimension, so from now on, we shall eschew use of the terms "attributes" and "facets" and simply refer to "dimensions"

main trends and ideas most relevant to our work.

Clearly, for any reasonable data set, it is infeasible either to materialize (precompute) the entire data cube or to compute the entire cube on the fly in real time. Instead, the approach taken by virtually all researchers has been to preselect a portion of the cube to materialize, and using this, to compute the rest of the cube if necessary.

While the general **CUBE BY** operator is defined to handle all the standard SQL aggregation functions (i.e., sum, mean, median, max, etc.), for our application we shall only require the sum aggregation function. This simplifies things considerably since the sum is an example of an *algebraic aggregation function*, meaning that more restrictive aggregates can be used to compute less restrictive ones. In other words, given an aggregate A on some set of facets, one can derive the correct aggregate for any subset of those facets directly from A without having to access the raw data. This property also inspires the definition of the *parent-child aggregate relationship*: aggregate A is a child of aggregate B if and only if it can be derived directly from B without reaccessing the full database. Using this relationship, we can then define a partial ordering on the 2^k possible aggregates in the data cube, which is called the *aggregate lattice*.

The first data cube algorithms all pre-computed the aggregate lattice in a top-down fashion [19, 20, 13, 11, 10, 5, 2]. Clearly, if one precomputes the root of the lattice, then all subsequent aggregates can be derived from this one directly. The root aggregate will also be the largest table since it has the most aggregate dimensions and therefore the most rows. So, assuming that the cost of deriving a child aggregate from its parent is proportional to (or at least monotonic with) the number of rows in the parent, it may be advantageous to also precompute lower level aggregates of the aggregate lattice which have fewer rows than the root. Therefore, the basic strategy of these algorithms is to start from the root (top) of the lattice and search down the lattice until a level is found where the gain in running time ceases to be significant relative to the required space needed to store the additional aggregates. Various papers also explored other ideas such as using cost functions other than the number of rows in the table [3], how to choose indices along with the materialized views [11], how to solve the problem for non-algebraic aggregation functions [12], and how to implement the **CUBE BY** operation efficiently on a computer cluster [15].

Later, Ross and Srivastava [17, 18] pointed out that although in theory the size of the data cube grows exponentially in the number of dimensions, for real world datasets, the datacube usually becomes extremely sparse as the number of dimensions increase. For exmaple, suppose we have k dimensions and the number of distinct values for each dimension (the *cardinality* of the dimension) are $n_1, ..., n_k$ respectively, then in theory the root aggregate could have $\Pi_{i=1}^{k} n_i$ rows, but in practice, many of these rows are empty: there is no item that satisfies all the criteria exactly. In fact, the larger the number of dimensions, the more likely the datacube is to be sparse.

In an important paper, Beyer and Ramakrishnan [4] then observed that by doing data cube computations top-down in the aggregate lattice, the algorithm computes the most restrictive (i.e. the sparsest) aggregates in the lattice, whereas in practice, usually what one is most interested in are the densest aggregates. In many applications, sets with only a handful of items are less interesting than sets with many items. They introduced a new variant of the original **CUBE BY** operation called the **ICEBERG CUBE** [2] for which the user specifies some parameter X as the *minimum support threshold* necessary for a set to be considered interesting, and only sets containing enough items to meet this threshold are computed. They then proposed the bottom-up computation of the aggregate lattice for the computation of the iceberg cube. The rule is to stop the computation whenever the support of the aggregates becomes lower than the threshold parameter, X.

Although our work is most similar to that of the data cube community, there are several other lines of work which are related. A number of researchers have considered the problem of computing approximate group-bys and approximate data cubes. In particular, Archarya et al. [1] introduced the technique of congressional samples for computing approximate group-bys and Margaritis et al. [14] used Bayesian networks to approximate the data cube algorithms. However, these methods suffer from fairly high error rates (roughly 10 to 15%) and are extremely poor for very small set sizes. Since our application requires accurate measurements of sets with support as low as one element, approximate methods are not suitable for our application and we will not consider them further.

3 Our approach

For the following discussion, we shall assume that we have topologically sorted each facet hierarchy and split it into its constituent levels. We will call these levels the *dimensions* of the database and treat each one as independent. In the case where the facet is flat, the dimension corresponds to the facet itself. Otherwise in general, a facet has more than one dimension. Usually, the facet hierarchy is a tree and this simply corresponds to taking each level of the tree (defined as all the nodes at some distance from the root) to be a dimension.

In it's most basic form, the Flamenco system allows users to navigate through a collection of items by choosing, at each step, one facet (e.g. architect) and one value from that facet (e.g. Gehry) thus narrowing the result set to only those items whose metadata includes the assignment $architect = Gehry$. In the next step, the user repeats this process, thus narrowing the result set further until it becomes small enough to exhaustively scan through the entire result set.

Now recall that the "cardinality" of a dimension is defined to be the number of distinct values that dimension can take. Our approach is based on the following intuition: suppose that all dimensions are completely independent from one another and suppose we can partition the dimensions into those that have high cardinality (e.g., building name) and those with low cardinality (e.g., period). When the user begins by selecting (i.e., aggregating by) a value from a

[2] The name "iceberg" is a reference to the work on iceberg queries by Fang et al. [8]. An iceberg is a general database query which returns all items with value above a certain user-specified threshold.

high cardinality dimension like "building name" to begin with, the working set size is immediately cut down to a very small number. Therefore, all subsequent group-by information on this small set can be quickly computed by the trivial brute-force approach (i.e. computing all the group-by's individually from the raw data). On the other hand, when the user picks a value from a low cardinality dimension, the set size is large and so the brute-force approach is very inefficient. In this case we would like to precompute the answers and store them in a database table. In general, such precomputation takes up a large amount of storage space, but since our dimension has low cardinality, the resulting table has relatively few rows. This simple insight is the crux of our method. To summarize the main idea once again: query preview information can be trivially computed when the set size is small, otherwise, if the set size is big, it can be precomputed and stored in a table because the low cardinality implies the table will be small in size. In the rest of this note, we shall abuse the definition of the data cube somewhat by referring to this table as a data cube, even though technically it is not the full cube.

Any time the user picks a high cardinality facet, all subsequent group-bys are efficient because the initial select command reduces the number of elements that need to be considered by a huge factor. Also, every combination of low cardinality facets is precomputed and therefore efficient. The only case which is still slow is if the user repeatedly selects a chain of low cardinality facets (e.g. material = brick followed by architect = Frank Lloyd Wright followed by Location = N. America etc.) such that the set size remains big after each select. In this case, only the query previews corresponding to other low cardinality facets are available from the precomputed cube table, while the other query previews corresponding to high cardinality facets still need to be manually computed. Since the precomputed previews can be retrieved virtually instantaneously, and we can typically precompute about half of all the facets, we can speed up this query by approximately a factor of two.

Now our problem has now been reduced to choosing the appropriate dimensions to be included in the cube table. Obviously, in general, it will not always be the case that we can partition the facets into two obvious clusters - those with high cardinality and those with low cardinality - so it will be necessary to have an algorithm that in general chooses which dimensions to preaggregate. We consider this problem in the following sections.

4 Handling Multi-Valued Facets

One feature of our data sets which, to our knowledge, other researchers have not yet considered and which we have thus far ignored, is the problem of multiple values. In this case, the sum function is no longer an algebraic aggregate function. (Recall that the definition of an algebraic aggregate function is one for which a less restrictive aggregate can be derived directly from a more restrictive one.) For example, consider the case where a single item x is assigned two materials, wood and brick. In this case the cube table would have a count

Material	People	Location
Brick	Architect	N. America
Metal	Historical Figure	S. America
Glass	Agency	W. Europe

Table 1: Example of three facets with their facet values. Note that although in this example each facet has three values, in general each facet can have an arbitrary number of values.

of one for each of wood and brick. However, if we then tried to aggregate the counts, this single item, x, would then contribute two to the total count instead of one.

Our solution to this problem is to store the counts for *all* possible subsets of the cube dimensions rather than just the largest one. Then, rather than performing group-by operations on the largest aggregate to compute the smaller ones, we just precompute everything and store them all. Note that this does not increase the space complexity in an assymptotic sense. Previously the worst-case space complexity was $\Pi_{i=1}^{k} n_k$ while now this has increased to $\Pi_{i=1}^{k}(n_k + 1)$. In practice, we see that it involves an increase by only a modest constant factor (a factor of three in our experiments).

Now we can see in hindsight that the data structure we have achieved by this line of thinking is precisely the structure produced by the bottom-up data cube algorithm of Beyer and Ramakrishnan. In both problems, the crucial observation is that the small sets need not be precomputed. In their applications, this was because the small sets were deemed to not be of interest to the user. In our case, while we are most definitely interested in these small sets, it turns out that these sets are easy to compute via a brute-force approach.

Although the previous discussion may seem slightly complicated, the final idea is extremely simple, as can be easily seen from an example. Suppose we select two facets to be in our datacube table with facet values as shown in Table 1. The resulting data cube table would look something like the example in Table 2.

Then to get the query preview information for any query that consists of a combination of the facets in the data cube table, one can retrieve the counts by simply indexing into the table and reading off the count values.

4.1 Choosing the cube dimensions

Beyer and Ramakrishnan [4] noted that the dimension ordering is vital to the performance of their iceberg cube algorithm, and they gave three criteria for ranking dimensions:

- Cardinality: The cardinality of a dimension is the number of distinct values of the dimension. A lower cardinality implies fewer rows in the cube table, so we prefer to choose low cardinality facets for the cube.

Material	People	Location	Count
Brick	Architect	N. America	2780
Brick	Architect	S. America	8
Brick	Architect	W. Europe	3420
Brick	Historical Figure	W. Europe	6
Brick	Agency	N. America	4
Metal	Architect	N. America	1152
Metal	Architect	S. America	6
Metal	Architect	W. Europe	1644
Metal	Agency	N. America	22
Glass	Architect	N. America	1247
Glass	Architect	S. America	86
Glass	Architect	W. Europe	1934
Glass	Agency	N. America	22

Table 2: Example of a data cube table using the facet values in table 1 and actual count values from the Spiro dataset

- Skew: In the database literature the skew is defined as the square of the ℓ_2 norm of the facet interpreted as a vector. Intuitively, the higher the skew, the farther away the facet is from the uniform distribution. The *effective cardinality* of the facet is lower if the skew is high because it is likely that subsequent facets will not appear at all in some of the small sets.
- Correlation: Two facets are highly correlated if their values are highly correlated (for e.g. in table 1, if Agency is highly correlated with Brick and Architect with Glass, we would say People and Material are highly correlated.) Again, the effective cardinality is lower if a facet is highly correlated with a previous facet.

There are a few rules of thumb we can assume about our data sets. In practice, we can generally assume that the dimensions of different facets are essentially independent. For example, a building in North America is equally likely to be made of brick or steel. It should not be the case that all the buildings in the database that are in North American are also made of steel, because then having separate facets at all is meaningless for searching. Therefore, we can assume that the metadata is well constructed and that dimensions from different facets look independent.

Similarly, since some dimensions come from the same facet (i.e. one is an ancestor of the other in a single facet tree or DAG), these facets are very well correlated. For example, it is always the case that a building with second level location "USA" also has first level location "N. America". Similarly, there are no items in the collection assigned with the metadata "W. Europe" and "USA". This means that this cell in the datacube table will be empty and therefore not stored at all. Thus, in practice, it is probably best to concentrate only on dimensions corresponding to the top levels in the facet trees, and not to worry at all about levels lower in the tree. Not only does including these lower levels not affect the performance much (since they are well-correlated with their ancestor levels) but also, because of the way the Flamenco browsing system works, a user usually selects all the ancestor levels on some path down the facet tree to get to a particular level [3], and therefore we are almost certain to see more queries to the system consisting of high level queries as opposed to lower ones.

In our experiments we used the heuristic of considering only top level dimensions, ordering them by their cardinality, and greedily selecting facets until the datacube table become too large. Other heuristics are possible, but as we have already discussed above, for real world data sets, it seems unlikely that anything will significantly outperform the trivial heuristic we have chosen.

5 Experiments

We implemented our ideas in the Flamenco system for the Spiro collection of architectural photographs. The resulting datacube table contains a total of 21373 rows. Note that the items table that stores the items in the collection plus their attributes by itself has over 36000 rows, so our table does not represent a huge overhead. In addition, we require a full index on the table. The time for building the table is not trivial but since this is a preprocessing step, we feel this is quite reasonable. In all, six facets were chosen for the cube - role, location, object type, material, style and concept.

5.1 Queries from the Logs

The Flamenco system has been in use by UI researchers and art or architecture professionals for some time now and we have now accumulated a large log of their searches. Even when we exclude non-standard Flamenco searches such as search box queries and queries from within UC Berkeley and from the Google web crawler, we are left with over 80000 queries, of which more some 26000 are unique. Since our system caches most result pages, we assume that a query is slow only the first time it is sent to the server and so we restricted our attention only to the unique queries [4]. In other words, we assumed that all subsequent trials of the same query would hit the cache and thus take negligible time. (Note that this is not always true since not all queries are cached - however, this only gives the old brute-force implementation an advantage.)

We sampled around 1000 unique queries and issued them to our database 5 times and took the median time. We used the median because we saw a fairly high variance in processing time (most likely due to the presence of other

[3] In some cases, the user may jump directly to some level of a facet hierarchy without first visiting its ancestors via a free text search or by expanding the search from the endgame.

[4] In fact, our method can be seen as complentary to the existing system cache. The existing cache stores the small item sets because these are efficient to store. Our method takes care of the large item sets.

Implementation	Previous	New
Average time	2.42 seconds	0.883 seconds
Max time	49.2 seconds	24.79 seconds
num queries over 10 sec	48	9
num queries over 5 sec	148	28
num queries over 1 sec	442	223

Table 3: Experimental Results

high priority processes running on the same server) and the median removes the effect of such outliers. In some cases, a server or network error stops the query from running, in which case we throw out the query. After this, we were left with 1017 queries for which we have reliable timing information.

Two primary measures were used to measure the effectiveness of our implementation versus the existing implementation: the average time taken per query and the maximum time taken per query. Recall that our methods are expected to perform roughly the same as the old implementation for low cardinality result sets and beat the old implementation by a factor of roughly 2 or more for the high cardinality result sets. Furthermore, only the database group-by processing was timed. In the real system, the system also has to fetch the images (in our case over the network) and send them again over the network to the browser. Due to the high variance in network congestion, we exclude this portion of the query from our timing experiments.

In addition, we analyzed how many of the queries in each implementation were "slow" - defined as taking more than 1, 5 or 10 seconds. We summarize the results of the experiments in table 3

Finally we sorted the queries in order of slowest to fastest and plotted the time they took both on a regular scale and with the x-axis on a log scale in figures 5.1 and 5.1. From these figures it is quite apparent that our implementation works much better than the old implementation. However, from these two figures it looks as if the plots might be Zipf distributed which would indicate that most of the weight in the distribution occurred in the tail, thus implying that almost all the queries were short queries to begin with anyway.

6 Discussion

The Flamenco interface is only one possible web search interface, suitable for particular types of data collections and users. For certain applications, it is possible that a checkbox-style interface may be more desirable. For example, in such an interface it would be simpler to select a large number of facet terms and to provide functionality for boolean ANDs and ORs in a way the user can easily understand. Examples of such interfaces can be easily found on the World Wide Web.

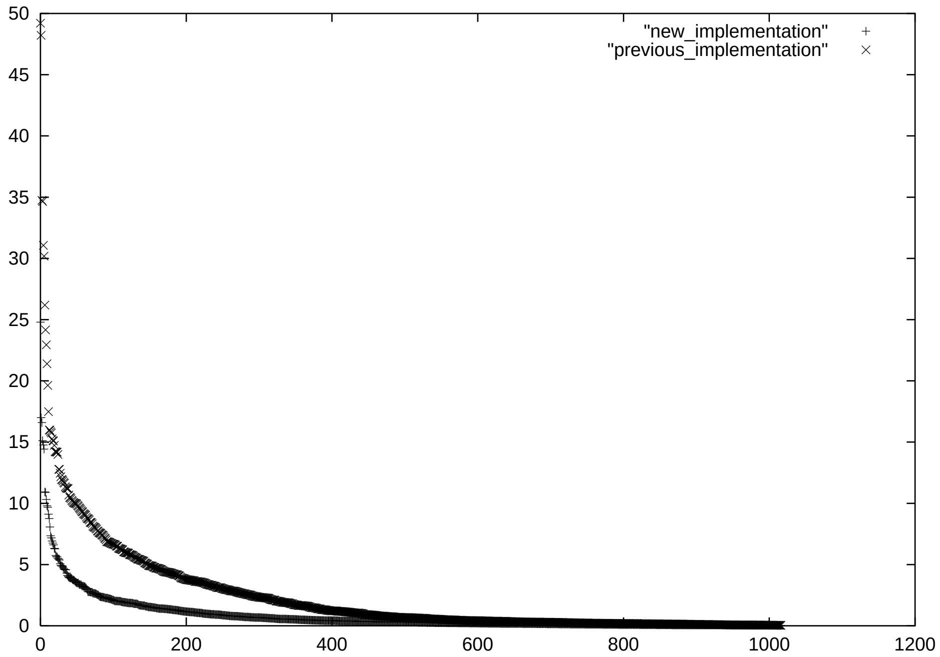

Figure 1: time vs. rank of queries

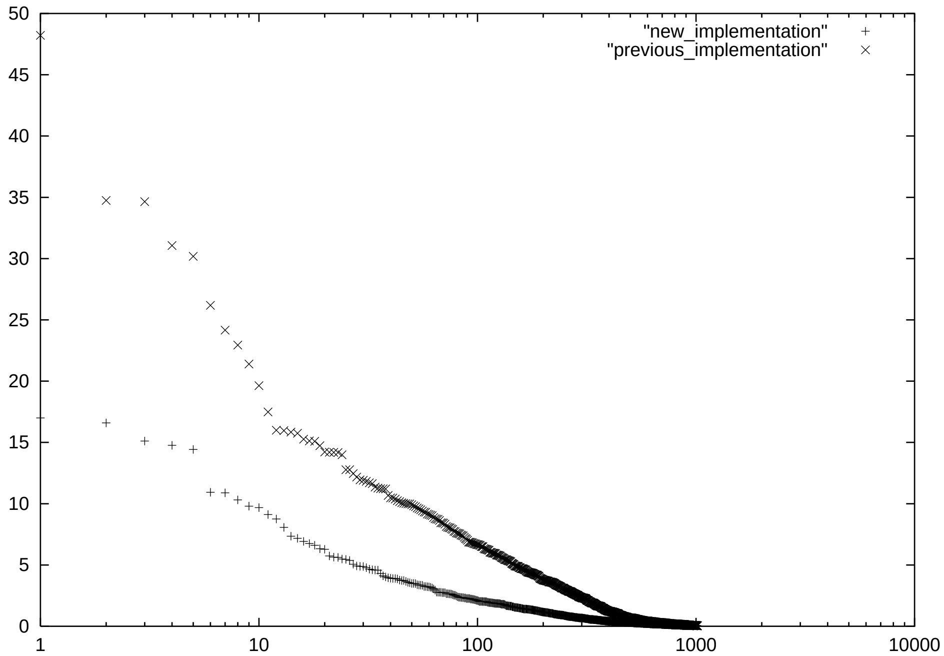

Figure 2: time vs. logrank of queries

The Achilles' heel of checkbox interfaces is that the user usually checks too many boxes and is then left with the frustrating experience of very few results or worse, no results at all. Therefore an interesting question is whether query previews can be generated on the fly for these interfaces. In such an implementation, as each box is checked, the count of the implied result set would be updated.

A trivial way to implement this would be to take our techniques and transfer them over to the other interface. However, our methods are still too slow for checkbox interfaces. The slow down from having to compute query previews for each click would defeat the purpose of having this interface in the first place. One possibility, however, is that an approximate solution might be given if it can be implemented significantly more efficiently than an exact method.

References

[1] Swarup Acharya, Philip Gibbons, and Viswanath Poosala. Congressional samples for approximate answering of group-by queries. In *SIGMOD*, 2000.

[2] S. Agarwal, R. Agrawal, P.M. Deshpande, A. Gupta, and J.F. Naughton. On the computation of multidimensional aggregates. In *Proc. of the 22nd VLDB Conference*, pages 506–521, 1996.

[3] E. Baralis, S. Paraboschi, and E. Teniente. Materialized view selection in a multidimensional database. In *Proceedings of the 23rd VLDB Conference*, pages 98–112, 1997.

[4] Kevin Beyer and Raghu Ramakrishnan. Bottom-up computation of sparse and iceberg cubes. In *SIGMOD*, 1999.

[5] P.M. Deshpande, S. Agarwal, J.F. Naughton, and R. Ramakrisnan. Computation of multidimensional aggregates. Technical Report 1314, University of Wisconsin, 1996.

[6] Jennifer English, Marti Hearst, Rashmi Sinha, Kirsten Swearinghan, and Ka-Ping Yee. Flexible search and navigation using faceted metadata.

[7] Jennifer English, Marti Hearst, Rashmi Sinha, Kirsten Swearinghan, and Ka-Ping Yee. Hierarchical faceted metadata in site search interfaces. In *CHI*, 2002.

[8] Min Fang, Narayanan Shivakumar, Hector Garcia-Molina, Rajeev Motwani, and Jeffrey Ullman. Computing iceberg queries efficiently. In *Proc. 22nd International Conference on Very Large Data Bases*, pages 307–317, 1996.

[9] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Datacube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. In *Proc. of ICEE ICDE*, pages 152–159, 1996.

[10] H. Gupta. Selection of views to materialize in a data warehouse. In *Proc. of the 6th ICDT*, pages 98–112, 1997.

[11] H. Gupta, Harinarayan V., A. Rajaraman, and J.D. Ullman. Index selection for olap. In *Proceedings of the 13th ICDE*, pages 208–219, 1997.

[12] Jiawei Han, Jian Pei, Guozhu Dong, and Ke Wang. Efficient computation of iceberg cubes with complex measures. In *SIGMOD*, May 2001.

[13] V. Harinarayan, A. Rajaraman, and J.D. Ullman. Implementing data cubes efficiently. In *Proceedings of the ACM SIGMOD Conference*, pages 205–216, 1996.

[14] Dimitris Margaritis, Christos Faloutsos, and Sebastian Thrun. Netcube: A scalable tool for fast data mining and compression. In *Proceedings of the 27th VLDB Conference*, 2001.

[15] Raymond T. Ng, Alan Wagner, and Yu Yin. Iceberg-cube computation with pc clusters. In *SIGMOD*, May 2001.

[16] Catherine Plaisant, Ben Shneirderman, Khoa Doan, and Tom Bruns. Interfaces and data architecture for query preview in networked information systems. *ACM Transactions on Information Systems*, 17(3):320–341, 1999.

[17] K.A. Ross and D. Srivastava. Fast computation of sparse datacubes. In *Proceedings od the 23rd VLDB Conference*, 1997.

[18] K.A. Ross and K.A. Zaman. Optimizing selections over data cubes. Technical Report CUCS-018-98, Columbia University, November 1998.

[19] Sunita Sarawagi, Rakesh Agrawal, and Ashish Gupta. On computing the data cube. Technical Report RJ10026, IBM Almaden Research Center, 1996.

[20] A. Shukla, P.M. Deshpande, and J.F. Naughton. Materialized view selection for multidimensional datasets. In *Proceedings of the 24th VLDB Conference*, 1998.