Efficient Computation of View Subsets

Frank Dehne
Carleton University
Ottawa, Canada
frank@dehne.net

Todd Eavis
Concordia University
Montreal, Canada
eavis@cs.concordia.ca

Andrew Rau-Chaplin
Dalhousie University
Halifax, Canada
arc@cs.dal.ca

ABSTRACT

Over the past ten to fifteen years, data warehouse platforms have grown enormously, both in terms of their importance and their sheer size. Traditionally, such systems have been based upon a dimensional model known as the Star Schema that consists of a central fact table and a series of related dimension tables. Given the enormous size of the fact table, virtually all current systems augment the primary fact table with a small number of focused summary tables. Previous research has addressed the issue of the selection or identification of the most cost-effective summaries. However, the problem of efficiently computing a given view subset has received far less attention. In this paper, we present a suite of greedy algorithms for the construction of these view subsets. Experimental results demonstrate cost savings of between 20% and 70% relative to the naive alternatives, depending upon the degree of materialization required.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Query Processing, Relational Databases*

General Terms

Algorithms, Design, Performance

Keywords

OLAP, data cube, partial materialization

1. INTRODUCTION

In recent years, the size and significance of data warehousing (DW) platforms has grown enormously. Central to the data warehousing architecture is a denormalized logical model known as the Star Schema (the normalized version is known as the Snowflake Schema). A Star Schema consists of a single, very large fact table housing the measurement records associated with a given organizational process. During query processing, this fact table is joined to one or more dimension tables, each consisting of a relatively small number of records that define specific business entities (e.g., customer, product, store). A full data warehouse may contain multiple such Star Schema designs.

While the Star Schema forms the basis of the relational data warehouse, it can be extremely expensive to query the fact table directly, given that it often consists of tens of millions of records or more. In practice, DW designers typically review previous usage patterns in order to identify the most extensively queried summarization targets. They then augment the basic Star Schema with compact pre-computed summary tables that can be queried much more efficiently at run-time. Because these views essentially consist of a subset of the available dimensions, they can be joined with the dimension tables in exactly the same manner as the larger fact table; no new logic or SQL extensions are required.

Unfortunately, *computing* these subsets remains very expensive. While we can certainly construct an arbitrary subset of k views using a series of k independent SQL *GROUP BY* queries, doing so is enormously wasteful given the fact that many summary views will share some of the same attributes. Instead, we would prefer to utilize a single algorithm that is able to exploit the natural concurrency or overlap in related summary views.

In this paper, we present techniques that allow for the efficient computation of view subsets. Experiments show that our methods are effective in environments with as many as 16 dimensions — near the upper limit for a single corporate Star Schema — with an essentially linear increase in cost as additional views are added. In addition, we justify our approach with a simple *density estimator*. In fact, the estimator allows us to show that full cube materialization, apart from being immensely wasteful, is likely to be of no value whatsoever in terms of producing tangible gains in query performance.

The remainder of the paper is organized as follows. In Section 2, we briefly review related work. Our density estimator is described in Section 3, while our previous work in this area is reviewed in Section 4. The new methods are then discussed in detail. Section 5 presents optimized algorithms for subset construction, with Section 6 describing our technique for working in higher dimensional spaces. Experimental results are provided in Section 7, followed by final conclusions in Section 8.

2. RELATED WORK

While it is possible to summarize a d dimensional fact table in many different ways, a particularly important group of summary or *aggregate* tables is the one that consists of the $O(2^d)$ combinations of distinct dimension values. We generally refer to this set as the *data cube* [6]. A number of cube construction algorithms have been presented [1, 11, 2, 16]. Pipesort [1], for example, employs a a *top down* technique to organize views or "cuboids" into computational *pipelines*, while the BUC algorithm [2] uses a recursive partitioning mechanism to compute parent views from the more coarsely aggregated children.

More recently, researchers have explored compact representations of the aggregates found in all $O(2^d)$ views. QC-trees identify *equivalence classes* that summarize related records at different granularity levels [9], while the Dwarf Cube [15] achieves a similar result by reducing attribute redundancies. However, in both cases the resulting tree structures are quite complex and are not currently supported, or easily integrated into, existing DBMS systems. This point is echoed in [10], where a more conventional table-based model called CURE is proposed. Even here, however, the focus is again on the full cube which, as we shall see in Section 3, may not be the most appropriate target.

The notion of materializing a subset of the cube's views was first proposed by Harinarayan et al. [7]. They define the *cube lattice* and manipulate it using a greedy algorithm named BPUS that iteratively adds nodes to a set deemed to provide greatest *benefit*. In [13], Shukla et al. point out that BPUS effectively runs in $O(n^3)$ time. They provide a new $O(n \lg n)$ method called PBS that dramatically reduces execution cost. Kalnis et al. also use randomized search and simulated annealing to achieve comparable results [8].

Note that the focus of the previous algorithms is to identify *which* set of views to materialize. Few papers address the problem of *how* this possibly large set can be most efficiently constructed. In the original PipeSort paper [12], a Steiner tree representation is suggested. However, this is unlikely to scale as the tree requires billions of edges for cubes of even eight or nine dimensions. It has also been suggested that the BUC algorithm might be used to compute the views beneath a specific lattice level. However, the specified adaptation is unable to support random subsets, nor does BUC efficiently compute views in the lower portion of the lattice, exactly the views that would comprise the bulk of a typical subset.

In [3], Dehne et al. propose an algorithm specifically targeted at arbitrary subset computation. This is a greedy method similar in style to BPUS in that it iteratively selects the most beneficial view, where the benefit is measured in terms of the view's ability to reduce the global cost of computation. However, while the algorithm does generate arbitrary subsets, it does so with a time complexity of $O(n^3), n = 2^d$. As such, it is unlikely to be effective in ranges beyond 5 or 6 dimensions, too limited to be viable in most practical DW environments.

3. THE DENSITY THRESHOLD

As noted, a number of recent algorithms such as DWARF and QC-trees have proposed compact representations of full data cubes. While the associated data structures are indeed smaller than the fully materialized cube, particularly in very high dimensional spaces, the question remains: Is the full cube really the appropriate target? There are two key observations in this respect. First, while a full data cube materializes all 2^d cuboids, there is an overwhelming

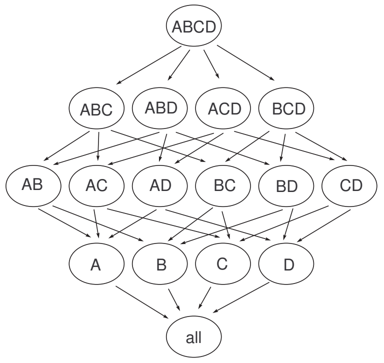


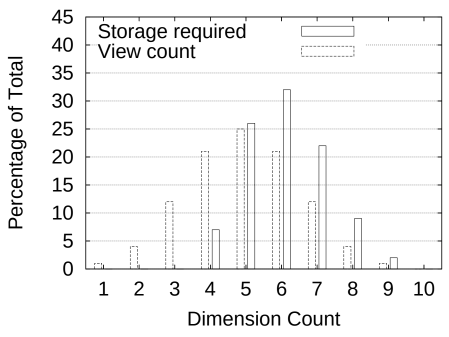
Figure 1: A 4-d cube lattice

preference among end users to query low dimension views. Such cuboids are more amenable to visualization and their simpler content can be intuitively converted into decision supporting knowledge.

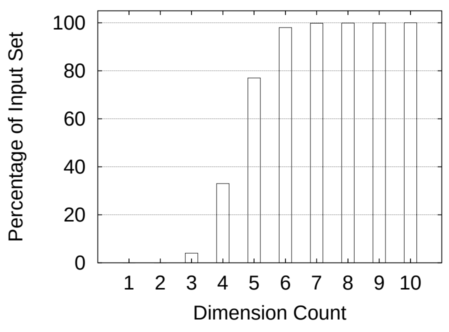
The second point regards the usefulness of the high dimensional views themselves. Specifically, do they actually contain information that is not easily obtained otherwise? In this regard we note that data cube processing costs are skewed heavily towards the upper portion of the cube lattice, where the largest views are found. Figure 1 illustrates the symmetrical structure of a simple four dimensional lattice. Figure 2(a), on the other hand, depicts the relationship between the number of views at a given level k of the graph — represented as $\binom{d}{k}$ — and the size of those views for a fairly typical 10-dimensional data set with mixed cardinality and one million records. Note that virtually all of the weight (i.e., table size) is associated with views of five to ten dimensions. At seven dimensions, for example, 12% of the views represent 22% of the weight, while the three-dimensional level has the same number of group-bys but less than one percent of the total weight.

In Figure 2(b), we more closely examine the record distribution (not just the file sizes) for the same problem instance. The total number of records in the aforementioned data cube is 642,197,905 — almost 650 times greater than the one million records in the input set. More striking is the effect of sparsity on the average record counts of views at each dimension. We can see that by six dimensions, the average view contains almost 97% of all records in the original input set, while at seven dimensions the ratio approaches 99.9%. In other words, in the upper portion of the lattice almost no aggregation takes place and the views are virtually identical to one another.

Of course, an increase in data set size also has an effect on record sparsity. One might imagine, then, that very large data sets — the kind we might expect to see in production environments — might produce dense views at much higher levels in the lattice. To assess this issue, we have developed a simple model that, for arbitrary fact tables, allows us to approximate the *density threshold*, the point at which the lattice becomes dramatically more sparse. We note that our method is intended to provide a *threshold approximation*, as the mixed cardinalities and inherent skew of specific data sets produce density thresholds that may, in practice, strad-



(a)



(b)

Figure 2: (a) View count and storage requirements. (b) Record sparsity.

dle several levels in the lattice. Nevertheless, it provides a very useful and informative picture of the degree of aggregation in the views of the complete lattice.

The model is based upon the notion of the "average" cardinality C_{avg} of the data cube. We use C_{avg} to then compute the *potential space* of a "typical" view at level k (i.e., a view with k attributes) of the lattice. By potential space, we are referring to the enclosing space of the cube, equivalent in value to the cardinality product $\prod_{i=i}^{d} C_i$, for $C_1, C_2, ...C_d$. Once this approximate potential space is defined, it can be passed to a probabilistic size estimator to compute the expected row count of a specific view. Feller, for example, shows that given a set of n elements, the expected number x of distinct elements obtained when randomly making r selections is $n - n(1 - 1/n)^r$ [4]. Alternatively, we can estimate the size of a given group-by as follows:

Proposition 1. *For an input set of size n, and a view v with a potential space of size S_v, we may estimate the number of records r in v by performing the summation*

$$x = \sum_{i=0}^{S_v} \frac{1}{(S_v - i)/S_v} = \sum_{i=0}^{S_v} \frac{S_v}{(S_v - i)},$$

terminating the summation when either $i \geq S_v$ or $x \geq n$.

Though the proof of equivalence with Feller's theorem is straightforward, we reserve the details to the longer version of this paper. With respect to the approximate potential space, it is defined as follows.

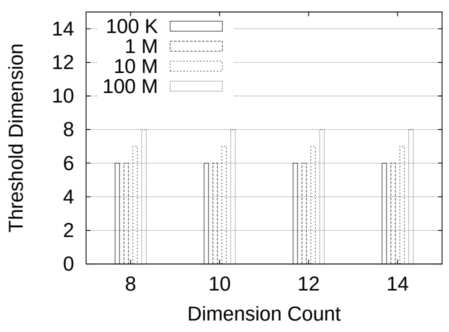


Figure 3: The density threshold for varying dimension counts and data set sizes.

Definition 1. *For a d-dimensional data cube with attribute cardinalities of C_1, $C_2 \ldots C_d$, we define the* **approximate potential space** *a_k of a view v with k attributes as*

$$\left(\sqrt[d]{\prod_{i=1}^{i=d} C_i}\right)^k$$

Here, we approximate the "average value" within the cardinality set for the full space by taking the d-th root of S_d. The k-th power of this approximated cardinality can then be used to estimate the size of the potential space for any view with k attributes. Now, for an approximated space a_k on a view v with k attributes, we may use our simple estimation model to approximate the number of records r in V as $\sum_{i=0}^{a_k} \frac{1}{(a_k - i)/a_k}$. Note that the output of this simple analytical model is a function of all of the relevant data cube parameters: d, k, C, and n. When applied to practical data cube problems, it can be used to predict the density threshold with impressive accuracy. For example, given the parameters of the problem represented by Figure 2(b), the model predicts a proportional size of 34% at 4 dimensions, 72% at 5 dimensions, and 95% at 6 dimensions, results that are within 3% of the observed values.

As previously noted, however, the method's real significance is that it permits us to assess the impact of data set size and dimension count on the position of the density threshold. Figure 3 provides an illustration of these interrelationships. Specifically, it depicts the density threshold on data sets of 8, 10, 12, and 14 dimensions, ranging in size from 100,000 to 100 million records. We note that for the purposes of this evaluation, we have defined the density threshold very conservatively so that a view is considered to be sparse only when it contains at least 99% of the records of the original fact table. Given this definition, the graph illustrates two very important points. First, it confirms the assertion that the density threshold does not increase with an increase in dimension count. In other words, for a fact table of a given size and dimensionality between 8–14, the threshold remains constant. Second, for a given dimension count, it takes a massive increase in input size to significantly increase the threshold. In fact, an order of magnitude size increase is required to move the threshold by a single level.

These results suggest that in DW/OLAP systems, the most appropriate target is likely not a fully materialized data

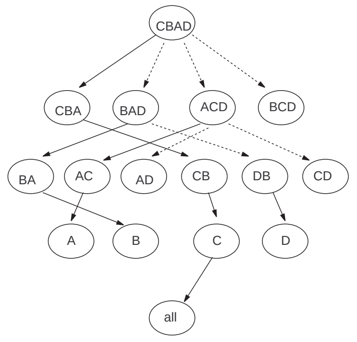


Figure 4: A PipeSort *scheduling tree*. Dashed lines are sort edges, solid lines are scans.

cube at all. Rather, a far more appropriate target is a partially materialized cube consisting of the *base cuboid* — the d-dimensional group-by that serves as a parent for all views — and some or all of the small, heavily summarized views below the fifth or sixth level of the lattice. With respect to the recently proposed tree-based structures, this observation is quite significant. The Dwarf, for example, is particularly effective at representing high dimensional spaces. However, the vast majority of high dimension summary views, apart from being rarely accessed in practice, are rendered largely irrelevant by the base cuboid. In terms of the lower dimension views, the tree-based models require the query engine to extract a small number of aggregate records from a massive and complex data structure. In this case, direct access to a focused set of small relational tables is an attractive alternative. More importantly, construction of these summary tables requires no significant extensions to current DBMS platforms. For this reason, we suggest that efficient partial cube materialization should be viewed as a fundamental OLAP problem.

4. OUR PREVIOUS MODEL

As previously noted, the methods in [3] provide the only functional partial cube implementation of which we are aware. Since our new methods also build upon a greedy approach, we briefly review the details of our previous research below.

We refer to the original method as GreedyCube and note that it is based upon the pipeline model of the PipeSort [1]. There, the idea is to create a series of prefix ordered pipelines, in which each pipeline shares a common sort order. As such, it is possible to compute each view in the pipeline with a single linear pass through the pipeline's sorted head node. The pipelines are organized in what is referred to as a *schedule tree*, a graph that is ultimately composed of the 2^d views and a set of $2^d - 1$ *scan* edges and *sort* edges. Figure 4 illustrates a PipeSort *scheduling tree* and its constituent prefix pipelines.

Very simply, the basic GreedyCube technique works as follows. First, a partial tree is created from the user-defined view subset. This is an incremental, greedy approach that uses a simple costing strategy to add views in a bottom-up fashion to an initially empty tree. Specifically, the cost model is used to determine the cost of constructing a given view versus the savings it brings via its ability to compute existing views more efficiently. We refer to this initial graph as the *essential tree*. This tree is then augmented with *non-essential* views that have the potential to reduce the global construction cost by virtue of their ability to efficiently compute previously selected views (see Figure 5).This process continues in a greedy fashion until it is no longer possible to add any views that reduce the computational cost of the cube. Structurally, the algorithm is formulated as a series of three nested loops with time complexity $O(2^d)^3)$. As noted previously, the performance of this algorithm is only likely to be acceptable for very low dimensional spaces.

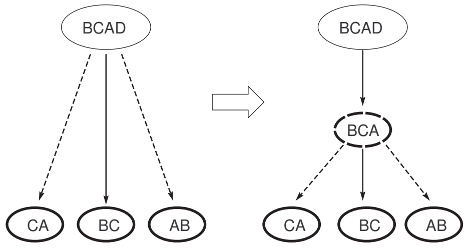


Figure 5: A partial cube in which the *non-essential* node BCA has been integrated into the user selected set.

5. A NEW QUADRATIC TIME VIEW CONSTRUCTION METHOD

Though our objective is the computation of view subsets, we will begin the discussion of our new work with a description of an algorithm that can be used for the computation of the full cube. Recall that the PipeSort uses a combination of sorts and scans to reduce global construction costs. Clearly, using a linear scan to compute a child from a parent is preferred to the use of a much more expensive sort. Still, some minimum number of sorts is actual required — since each pipeline begins with a sort — so the question becomes: How can we minimize this cost?

Let's us begin with a lattice of d dimensions. At a given level k (i.e., group-bys with k attributes), we have exactly $\begin{pmatrix} d \\ k \end{pmatrix}$ views. For each of these j views, there are exactly $d - k$ possible parents at level $k+1$. Assume that we divide the set j into two subsets, Q and M, such that Q contains the largest of the j views, and M the remainder. Now, if we use the cheaper scan edges to connect parents at level $k+1$ to children in Q, we will eventually be left with "required sorts" for the views of M, the set of smallest children. For each such view v in M, we are free to choose its smallest available parent at level $k+1$. Since, by definition, $|v| \leq |v'|$ for $v \in M, v' \in Q$, then $|w| \leq |w'|$ where w and w' represent the parents of *minimum required size* for v and v' respectively.

Algorithm 1 presents a new greedy method based upon this observation. We begin by identifying the largest "free" (i.e., not yet selected) view v in the lattice. Our objective is to build an entire pipeline beneath this point. We select the parent w of v by identifying the *smallest* view w at the preceding level (initially the fact table) that contains a superset of the attributes of v. The view w becomes the "sort" parent SP of v. Next, we recursively descend down through the lattice (using a simple supporting function called *ExtendPipeline*), selecting the *largest* child view amongst the available candidates and adding it to the pipeline with a "scan". This process continues until no "free" children can be found for the current *tail* node. We then execute the next iteration of the main loop, selecting the next largest view in the lattice and building another pipeline. The algorithm terminates once all the nodes of the lattice L have been added.

Algorithm 1 Recursive Tree Construction

```
Input: The full $d$-dimensional lattice $L$.
Output: An essential tree $E$.
1: Sort the views of L by estimated size.
2: repeat
3:     Select the next largest "free" view $v$.
4:     for all "free" views $w$ at previous level that contain a superset of the attributes of $v$ do
5:         $SP = w$, if $w <$ current $SP$
6:     Connect $SP$ to $v$ with a "sort" edge.
7:     ExtendPipeline($v$)
8: until all nodes have been added to $E$
```

With respect to the time complexity, we note that we begin with a $O(n \lg n)$ sort of the n views of the lattice, followed by a REPEAT loop in which we add a new pipeline. For each of the $O(n)$ views included during this progress, we select its parent and children from $O(d)$ total choices. Total cost is therefore bounded as $O(n \lg n) + O(dn)$. We note that since $\lg n = \lg 2^d = d$, the bound can be re-written as $O((n * d) + O(dn) = O(dn)$.

Of course, our objective in this paper is to produce scheduling trees for partial cube problems. We note that the original PipeSort algorithm cannot be directly utilized as it relies on a bipartite graph matching phase that requires all views at every level of the lattice to be generated. By contrast, a unique feature of our new approach is that we can actually modify the base algorithm to permit pipelines to "skip" levels in the lattice. Specifically, we extend the processing model to allow it to include *all* possible parents and children of v in the view selection process. An arbitrarily-defined partial set S can then be materialized. In terms of cost complexity, we note that instead of $O(d)$ potential parents and children for each node v, we have $O(n)$ such candidates. Consequently, the run-time for the modified recursive partial cube algorithm is $O(n \log n) + O(n^2) = O(n^2)$.

5.1 Adding Non Essential Views

Having generated the essential tree, we must now provide an efficient mechanism for adding *non essential* or non-selected views so that (i) the global cost is further reduced and (ii) we maintain the $O(n^2)$ upper bound of Algorithm 1. To do this, we utilize a technique by which views can be added directly to the tree, without the need to review all candidate nodes before selecting a single "best view". Specifically, the new algorithm greedily adds candidate views as soon as it determines that a new node has the potential to produce *any* reduction in the global tree cost. In this respect, the order of consideration is critical. Specifically, it is possible to commit to an alteration to a pipeline — say with the insertion of a scan edge — only to later find a much smaller non-essential view that could have more cheaply supported the same child set. Figure 6 provides a simple example.

In this respect, we note that the potential for error when greedily adding non-essential views is actually most pro-

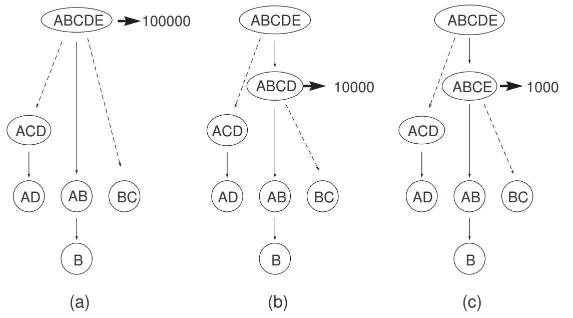


Figure 6: (a) The original tree (b) ABCD is inserted with a scan edge (c) A more cost effective solution; i.e., the "inclusion" cost of ABCE is just 1000.

nounced when selecting views from the guiding graph in a top down fashion. This is the case since (i) the unnecessary addition of larger views in the upper levels represents a larger penalty relative to the global cost, and (ii) if views are sorted and then evaluated by size, a bottom up traversal ensures that all possible children of v have already been added to the tree by the time v is considered for inclusion. This is the case since a view u can only be a child of v if $size(u) \leq size(v)$.

Algorithm 2 presents a new method for adding non essential views. We take as input an essential tree E and a *potential set* $P = L - S$. To begin, the views of P are sorted by size, this time in *ascending* order. Once the sorted list is created, we proceed by processing each candidate to determine possible parents and children in E. If a positive benefit for the inclusion of v is calculated, we add v to E and move on to the next view in the list. The algorithm terminates when we have examined all views from P.

Algorithm 2 Addition of Non Essential Views

```
Input: An essential tree $E$ and a potential set $P$.
Output: A complete scheduling tree.
1: Sort the views of $P$ in ascending order by size.
2: for each view $i$ in the sorted set $P$ do
3:     Select most cost effective parent in $E$
4:     Select the child set that would provide greatest cost savings if constructed from $i$
5:     if net benefit $> 0$ then
6:         add $i$ to $E$ as per the previous plan
```

With respect to cost, note that the algorithm consists of a sorting phase, followed by a view inclusion phase. The sort requires time $O(n \log n)$, where n is equivalent to the number of views in P. In the inclusion phase, we simply examine the $O(n)$ parents and children for each of the $O(n)$ views in P. The result is an $O(n \log n) + O(n^2) = O(n^2)$ bound.

Note that for a partial cube problem whose input is an n-node graph, the run-time of an $O(n^2)$ solution will exceed that of an $O(n^3)$ solution if the dimension count for the quadratic solution is increased by 50%. We can see that this is the case by solving the following simple inequality: $(2^{d\beta})^2 \geq (2^d)^3 = \beta \geq 3/2$, where β represents the multiplicative factor for the dimension count. So, for example, if the $O(n^3)$ solution is *practically feasible* for 8 dimensions on a given computing platform, this would imply that the new $O(n^2)$ algorithm is viable in the range of 12 dimensions on the same machine.

6. IMPROVING SCALABILITY

While the $O(n^2)$ complexity of the new model allows us to comfortably work in 10–12 dimensional spaces, it is sometimes necessary to support even larger cube problems. To do so, we have developed a heuristic pruning technique to reduce the search space without compromising the quality of the schedule trees. Recall that a candidate node is added to the current spanning tree *if and only if* the cost of its physical creation is less than that of the savings it bring to the generation of its potential children. In fact, we can actually make a stricter statement, namely that a candidate node v will not be added to the spanning tree R if it cannot improve the construction cost of at least two child nodes already in R (only one of which can share the parent's sort order). While the proof of this statement is relatively straightforward, and consists of a case-by-case enumeration of the alternatives, we will leave the details to the longer version of this paper.

In addition, we observe that as we move from the bottom to the top of P, it becomes progressively more unlikely that a candidate v will be able to reduce the global tree cost. This is so because as we move upwards through P, we incrementally increase the number of dimensions in v and therefore its sparsity. Eventually, v will be almost identical in size to its potential parent views. This is significant since parent views consist of a superset of the attributes in v and can thus also serve as parents for any of the potential children of v. In other words, the value of v as a non-essential addition declines as we move towards the top of G since it is increasingly unlikely that it will be more cost effective than any number of other candidates.

Algorithm 3 incorporates the previous observations into an effective heuristic solution. Beginning at the base cuboid, the method moves downwards through the lattice, selecting potentially "useful" views and adding them to a *minimal set* M that will be used to provide candidate nodes to our greedy algorithm. The logic for inclusion is as follows. For a given candidate node v in L, we will assume that its largest parent is already in G. We do this since (i) we know that any parent of v can also serve as a parent of the children of v, and (ii) the largest parent would produce the maximum possible benefit for v. Furthermore, we have also suggested that the inclusion of v requires that v have at least two child nodes. Therefore, for each candidate node v, we conservatively estimate the cost of the inclusion of v versus the existing alternative (i.e., of computing its 2 or more children from the current parent w). With scan cost = s and sort cost = S, we must have $newCost < oldCost$ or $s(w) + s(v) + S(v) < s(w) + S(w) = s(v) + S(v) < S(w)$. Figure 7 illustrates how the evaluation is performed.

For the sake of flexibility, the algorithm is augmented with a user-defined *confidence factor* β that determines how aggressively to prune the lattice. A β value of one implies the assumption that v will only have two children. With increasing values of β, the algorithm becomes more conservative in that it allows for the possibility that v may have many children.

In terms of cost complexity, an examination of the logic demonstrates that we make a linear pass through the lattice, looking for views to prune. At each step, we check the $O(d)$ possible parents of v. Because we are pruning the complete lattice, there are only $O(d)$ possibilities (at the level above), not $O(n)$. Thus the upper bound is $O(d * n)$. Finally, we note that the pruning algorithm is the only method to iterate over the full lattice.

Algorithm 3 Pruning for High Dimensions

```
Input: A lattice $L$, and a confidence factor $\beta$.
Output: A minimized set $M$.
1: for all candidate nodes $v$ in $L$ do
2:   From the $O(d)$ potential parents in $L$, find the smallest parent $w$ of $v$
3:   if $scanCost(v) + (\beta * sortCost(v)) \leq \beta * sortCost(w)$ then
4:     add $v$ to $M$
```

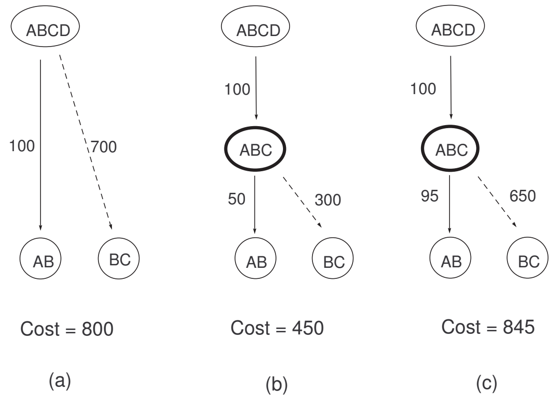


Figure 7: (a) The original tree. (b) A dense ABC node offers great benefit ($450 < 800$). (c) In contrast, a sparse ABC node increases the cost ($845 > 800$).

7. EXPERIMENTAL RESULTS

In this section, we present experimental results for the partial cube algorithms discussed in the paper. All tests were conducted on a 3.3 GHz CPU, with 2 GB of memory and a standard 120 GB SATA drive. Unless otherwise stated, the default input set consists of one million records, with 10 dimensions of varying cardinalities of 10–100000 (recall that data set size actually has little effect upon the density threshold). Point distribution is uniform since, in the context of subset generation, the use of large cardinality ranges provides far greater costing complexity than data skew. Further, synthetic test sets allow us to establish and explore a broader range of core parameters. We note that we have also utilized real data sets, and observed similar results, but have not included them here due to space constraints.

7.1 Quality of Full Cube Scheduling Trees

While the selection of an optimal set (full or partial) is NP-hard, and as such a provably optimal subset cannot be easily defined, we do of course have the original PipeSort algorithm that has been shown to produce excellent trees for the full cube. Figure 8(a) illustrates the quality/weight of the "Recursive Pipeline" trees produced by our new methods relative to PipeSort trees, as well as to the cubic time Greedy Cube algorithm described in [3]. Note that *both* of the greedy partial cube algorithms produce trees that are less than six percent larger than those of the PipeSort for dimension counts between six and twelve (the cubic time method is only charted up to 10 dimensions). In fact, the new $O(n^2)$ recursive pipeline algorithm produced trees in six to 12 dimensions that were less than 0.1% larger than those generated via the PipeSort.

7.2 Run Time Performance on the Full Cube

Figure 8(b) depicts the run time for the PipeSort and the two partial cube methods for the tests in the previous section. Note that a logarithmic axis was necessary since the run-time for the Greedy Cube algorithm grew extremely quickly beyond eight dimensions. For example, at 10 dimensions the run-time was just over 2800 seconds, with both PipeSort and our new quadratic algorithm completing in under 20 seconds. By extrapolation, we estimate the time required at 14 dimensions to be roughly 14 months ($O(n^3)$) versus 5 minutes ($O(n^2)$).

7.3 Computing Partial Cubes

In this section, we compare the schedule trees produced by the two partial cube algorithms on selected sets of views. In the absence of a viable partial cube alternative, we establish a baseline by first (i) building the full data cube and keeping only the selected set, and (ii) calculating each of the selected views by a separate sort of the raw data set. The baseline is defined as the faster of the two.

We then randomly select subsets consisting of 10%, 25%, and 50% of the views in the full space. Because of the run-time of the cubic time method, we restrict ourselves to dimension counts in the range of six to nine. Figures 8 (c), (d), and (e) present the results. We note the gradual decrease of relative weight reduction as we move toward higher dimensions. This is to be expected since an increase in dimension count implies an increase in the sparsity of intermediate views which, in turn, implies an increase in size relative to the raw data set. As such, the relative cost savings will decline slightly in higher dimensions. Nevertheless, savings of between 20% and 60% are generated by the new algorithm, with no discernible advantage for the more expensive $O(n^3)$ method.

Note, however, that the use of randomly selected subsets such as these tends to underestimate the cost savings for an important class of partial cube problems. Specifically, in high dimension spaces, users and administrators typically select the majority of views from the lower portion of the lattice since such views are more intuitive to visualize and interpret. Figure 8(f) illustrates the relative cost reductions when the selected views are limited to those containing three attributes or less (the algorithm, of course, is free to add larger, non-essential views). For the sake of clarity, we restrict the experiment to the new quadratic time algorithm. Under these circumstances, the algorithm consistently reduces the weight of the schedule tree by 60% to 70% for dimension counts up to 14.

7.4 Addition of Non-Essential Views

As previously noted, one of the most important cases for partial cube execution is the selection of views in the lower regions of the lattice. Figure 8(g) illustrates the effect of adding non selected views when the selected set consists entirely of views with three or less attributes. Notice that for both methods the addition of non essential views reduces the cost of the essential spanning tree by an additional 30% to 50%. Moreover, these weight reductions are consistently large from 6 to 14 dimensions. The conclusion to be drawn here is that there is really no reason to ever choose the Greedy Cube algorithm since, not only is it not viable on problems of practical size, but it simply does not produce better schedule trees on either the full or partial cube.

7.5 Pruning the Guiding Graph

Our objective in this section is to (a) determine how significantly the potential set can be pruned and (b) understand the impact of adjusting the confidence factor. With respect to the first issue, Figure 8(h) demonstrates that as we increase the dimension count — assuming a confidence factor of one — the percentage of views pruned increases steadily from a low of 2% (one of 64 views) at six dimensions to a high of 74% (48,496 of 65,536) at 16 dimensions. Not surprisingly, the practical benefit is significant. At 16 dimensions, for example, the pruned input size is just 1/4 the size of the original guiding graph. With an $O(n^2)$ algorithm, this translates into a factor of 16 performance improvement.

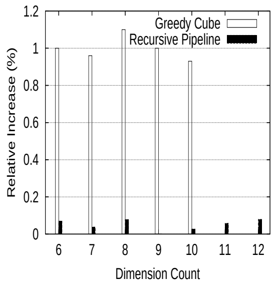
In terms of the second issue, Figure 8(i) presents results in 14 dimensions for confidence factors from one to three (and views with three attributes or less). We note that as the confidence factor increases, there is a huge drop off in the number of views pruned, from 56% to 34% to almost zero when the confidence factor is three. Clearly, a conservative approach to pruning will have a significant impact upon run time performance. More importantly, however, there is virtually no impact upon the quality/size of the tree as we become more conservative. As such, we can conclude that aggressive pruning is a low-risk option for improving the scalability of the partial cube algorithm.

8. CONCLUSION

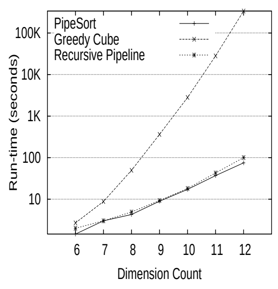
In this paper, we have presented a suite of greedy methods for the computation of partial cubes in practical DW/OLAP spaces. While full cube computation has received greater attention in the literature, it is the partial cube problem that is actually far more relevant in large production environments. While greedy methods have been used in the past — for both view identification and selection — their cubic time cost complexity often makes them impractical for more realistic settings. Using a series of observations regarding the nature of data cube spaces, we have developed methods that produce impressive scheduling trees in quadratic time. With the addition of a simple pruning method, the algorithms have been run effectively at up to 16 dimensions. In addition, the proposed methods can be integrated into existing DBMS platforms with very little effort. In fact, it is possible to run the current methods against a fact table and then store the computed partial set directly into a standard DBMS. As a result, we believe the new methods represent the most practical approach currently available for the important problem of data cube subset generation.

9. REFERENCES

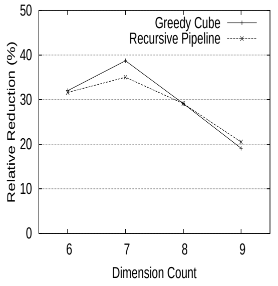
[1] S. Agarwal, R. Agrawal, P. Deshpande, A. Gupta, J. Naughton, R. Ramakrishnan, and S. Sarawagi. On the computation of multidimensional aggregates. pages 506–521, 1996.



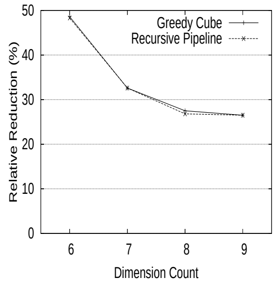
(a)



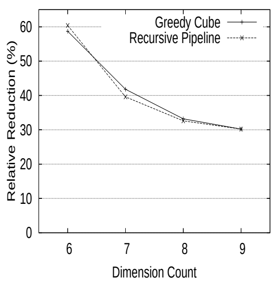
(b)



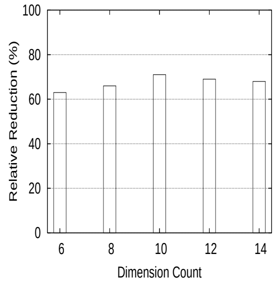
(c)



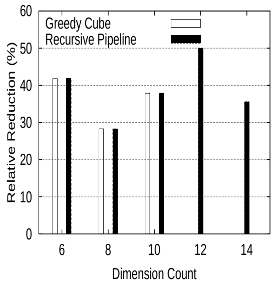
(d)



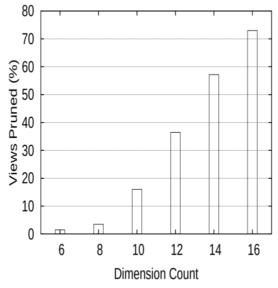
(e)



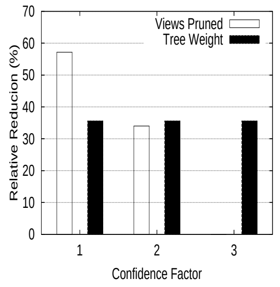
(f)



(g)



(h)



(i)

Figure 8: (a) Tree costs versus PipeSort (b) The run-time cost versus PipeSort. Tree reduction for subsets of (c) 10%, (d) 25%, (e) 50%. (f) Tree costs for 3 dimensions or less, (g) non-essential view addition for 3 dimensions or less (h) view pruning (i) confidence factor effect.

[2] K. Beyer and R. Ramakrishnan. Bottom-up computation of sparse and iceberg cubes. *ACM SIGMOD*, pages 359–370, 1999.

[3] F. Dehne, T. Eavis, and A. Rau-Chaplin. Top-down computation of partial ROLAP data cubes. *HICSS*, page 80223c, 2004.

[4] W. Feller. *An introduction to probability theory and its applications.* John Wiley and Sons, 1957.

[5] P. Flajolet and G. Martin. Probabilistic counting algorithms for database applications. *Journal of Computer and System Sciences*, 31(2):182–209, 1985.

[6] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *ICDE*, pages 152–159, 1996.

[7] V. Harinarayan, A. Rajaraman, and J. Ullman. Implementing data cubes. *ACM SIGMOD*, pages 205–216, 1996.

[8] P. Kalnis, N. Mamoulis, and D. Papadias. View selection using randomized search. *Data Knowledge Engineering*, 42(1):89–111, 2002.

[9] L. Lakshmanan, J. Pei, and Y. Zhao. QC-trees: An efficient summary structure for semantic OLAP. In *ACM SIGMOD*, pages 64–75, 2003.

[10] K. Morfonios and Y. Ioannidis. CURE for cubes: Cubing using a ROLAP engine. In *VLDB*, pages 379–390, 2006.

[11] K. Ross and D. Srivastava. Fast computation of sparse data cubes. *VLDB*, pages 116–125, 1997.

[12] S. Sarawagi, R. Agrawal, and A. Gupta. On computing the data cube. Technical Report RJ10026, IBM Almaden Research Center, San Jose, California, 1996.

[13] A. Shukla, P. Deshpande, and J. Naughton. Materialized view selection for multidimensional datasets. *VLDB*, pages 488–499, 1998.

[14] A. Shukla, P. Deshpande, J. Naughton, and K. Ramasamy. Storage estimation for multidimensional aggregates in the presence of hierarchies. *VLDB*, pages 522–531, 1996.

[15] Y. Sismanis, A. Deligiannakis, N. Roussopoulos, and Y. Kotidis. Dwarf: Shrinking the PetaCube. *ACM SIGMOD*, pages 464–475, 2002.

[16] Y. Zhao, P. Deshpande, and J. Naughton. An array-based algorithm for simultaneous multi-dimensional aggregates. *ACM SIGMOD*, pages 159–170, 1997.