Tech-Report Number CUCS-018-98

Optimizing Selections over Data Cubes

Kenneth A. Ross*
Columbia University
kar@cs.columbia.edu
Phone: (212) 939 7058
Fax: (212) 666 0140

Kazi A. Zaman
Columbia University
zkazi@cs.columbia.edu

Abstract

Datacube queries compute aggregates over database relations at a variety of granularities, and they constitute an important class of decision support queries. Often one wants only datacube output tuples whose aggregate value satisfies a certain condition, such as exceeding a given threshold. For example, one might ask for all combinations of model, color, and year of cars (including the special value "ALL" for each of the dimensions) for which the total sales exceeded a given amount of money.

Computing a selection over a datacube can naively be done by computing the entire datacube and checking if the selection condition holds for each tuple in the result. However, it is often the case that selections are relatively restrictive, meaning that a lot of work computing datacube tuples is "wasted" since those tuples don't satisfy the selection condition.

Our approach is to develop algorithms for processing a datacube query using the selection condition internally during the computation. By making use of the selection condition within the datacube computation, we can safely prune parts of the computation and end up with a more efficient computation of the answer. Our first technique, called "specialization", uses the fact that a tuple in the datacube does not meet the given threshold to infer that all finer level aggregates cannot meet the threshold. We propose a scheme of specialization transformations on the underlying data sets, using properties of the aggregates and threshold functions.

Our second technique is called "generalization", and applies in the case where the actual value of the aggregate is not needed in the output, but used just to compare with the threshold. Generalization uses the fact that a tuple meets the given threshold to infer that all coarser level aggregates also meet the threshold. We also propose a scheme of generalization transformations.

We demonstrate the efficiency of these techniques by implementing them within the sparse datacube algorithm of Ross and Srivastava. We present a performance study using synthetic and real-world data sets. Our results indicate substantial performance improvements for queries with selective conditions.

*This research was supported by a David and Lucile Packard Foundation Fellowship in Science and Engineering, by an NSF Young Investigator Award, by NSF grant number IIS-98-12014, and by NSF CISE award CDA-9625374.

1 Introduction

Datacube queries compute aggregates over database relations at a variety of granularities, and they constitute an important class of decision support queries. The databases may represent business data (such as sales data), medical data (such as patient treatments) or scientific data (such as large sets of experimental measurements).

The computation of a data cube query with k CUBE BY attributes $(B_1, B_2 \ldots B_k)$ involves computing the aggregates over a relation at 2^k granularities where each granularity is one of the the possible 2^k subsets of our k CUBE BY attributes. Attributes that are not present in such a subset are replaced by a special value ALL in the datacube result. We refer to each of these granularities as a *cuboid* and we use the notation $Q(\vec{B_i})$ to denote the cuboid at granularity $\vec{B_i}$.

Often one wants only datacube output tuples whose aggregate value satisfies a certain condition, such as exceeding a given threshold. For example, one might ask for all combinations of model, color, and year of cars (including the special value "ALL" for each of the dimensions) for which the total sales exceeded a given amount of money. This query takes the form of the left query of Figure 1. In some cases we may need to know the exact value of sales too (the right query of Figure 1).

```
SELECT   a,b,c,d
FROM     relation
CUBE BY  a,b,c,d
HAVING   aggregate(G) relop threshold
```

```
SELECT  a,b,c,d,aggregate(G)
FROM    relation
CUBE BY a,b,c,d
HAVING  aggregate(G) relop threshold
```

Figure 1: A selection over a data cube

We can naively execute these queries as follows. Compute the datacube using any of the existing datacube algorithms [4, 1, 10] and check if the predicate in the HAVING clause holds for each tuple in the datacube. This strategy is reasonable if a large proportion of the datacube result tuples satisfy the condition. However, if only a small fraction satisfy the condition (i.e, the query is an example of an "iceberg query" [3]) then it seems that we may be wasting a lot of time computing aggregates that do not qualify.

However depending upon the aggregate function and the relational operator in the predicate, there are certain optimizations we can make use of. In this paper we propose two kinds of optimization that we call *generalization* and *specialization*. We defer the formal details until Section 1.1. For now, we motivate these techniques with examples.

Example 1.1: (Specialization) Suppose that we are computing our example query

> Find all combinations of model, color, and year of cars (including the special value "ALL" for each of the dimensions) for which the total sales exceeded $100,000. Output the total sales also.

Suppose that during an intermediate step of the computation, we determine that the total sales for all green cars is below $100,000. Then we can immediately infer that datacube output tuples grouped by (model,color), (year,color), and (model,year,color) will never meet the threshold when the color is green. If the computation corresponding to those aggregates has not yet been performed, then perhaps we can avoid that computation altogether. □

Example 1.2: (Generalization) Suppose again that we are computing our example query "Find all combinations of model, color, and year of cars (including the special value "ALL" for each of the dimensions) for which the total sales exceeded $100,000." Note that in this case we do not need the total sales in the output. Suppose that during an intermediate step of the computation, we determine that the total sales for white 1998 Taurus cars is above $100,000. Then we can immediately infer that the class of white cars satisfies the condition, the class of 1998 Taurus cars satisfies the condition, etc. We can immediately output all of the additional seven "generalizations" of (white,1998,Taurus). If we have not yet performed the aggregation needed for some of these additional tuples, then we can potentially avoid such aggregation altogether. □

A relation is *sparse* with respect to a set of attributes if its cardinality is a small fraction of the size of the cross-product of the attribute domains. Sparseness exists for two distinct reasons: large domain sizes of some `CUBE BY` attributes and a large number of `CUBE BY` attributes in the datacube query. Real-World data in application domains is often very large and sparse. Hence, efficiently computing datacubes over large sparse relations is important.

In this paper we examine how we would make use of generalization and specialization to carry out selections over data cubes efficiently. Since sparse data sets are important we extend the `Memory-Cube` and `Partitioned-Cube` algorithms [10] which can deal efficiently with sparse data. `Memory-Cube` computes a set of paths which cover the search lattice and then computes the cuboids on each path in turn. We exploit specialization by altering the set of base tuples with which each path is computed without affecting the correctness of the result. Depending on the selectivity of our condition, we can reduce the number of tuples which are processed in each path leading to substantial improvements in performance. Generalization is incorporated into the algorithms by introducing marker tuples which allow us to skip computing aggregates which are known to satisfy our selection criteria. Specialization applies for both of the queries in Figure 1 while generalization applies only for the query on the left. We demonstrate the efficiency of these modifications by experiments carried out on synthetic and real-world data sets for a variety of selection conditions. These experiments support our overall conclusion that substantial work can be saved.

1.1 Notation and Terminology

The computation of the various cuboids are not independent of each other, but are closely related in that some of them can be computed using others. The relationship between cuboids can be captured in terms of the *search lattice* of the data cube [6]. Each granularity $\vec{B_i} \subseteq \{B_1, \ldots, B_k\}$ is a node in the search lattice, and there is an edge from node $\vec{B_i}$ to $\vec{B_j}$ if $\vec{B_j}$ is a subset of and has one fewer element than $\vec{B_i}$; $\vec{B_i}$ is said to be a *parent* of $\vec{B_j}$ in the search lattice. If there is a path from $\vec{B_i}$ to $\vec{B_j}$ in the search lattice, $\vec{B_i}$ is said to be of a finer granularity than $\vec{B_j}$, and $\vec{B_j}$ is said to be of a coarser granularity than $\vec{B_i}$. Paths in the search lattice precisely determine which of the cuboids can be computed from which others. In particular, cuboid $Q(\vec{B_j})$ can be computed using $Q(\vec{B_j})$ if and only if $\vec{B_j}$ is of coarser granularity than $\vec{B_i}$. A datacube tuple t_1 is more *general* than tuple t_2 if it can be produced from t_2 by replacing one or more of t_2's non-ALL attributes with ALL values. We can restate this by saying that t_2 is more *specialized* than t_1. t_1 and t_2 come from cuboids at different levels of the search lattice with a path from the cuboid of the more specialized tuple (t_2) to the cuboid of the more general (t_1) one.

Using the categorizations of aggregate functions introduced in [4], we focus on the case of *distributive functions.* An aggregate function $g()$ is said to be distributive if there is a function $h()$ such that $g(\{X_{i,j}\}) = h(\{g(\{X_{i,j}|i = 1, \ldots I\})|j = 1, \ldots, J\})$. Examples of such aggregate functions include SUM,COUNT,MIN and MAX. Within the class of distributive functions there are several properties we can check for. As we shall see, these properties are important when considering potential optimizations of aggregate computation.

- Idempotent: An aggregate function g is *idempotent* if $g(\{g(\{X_i\})\}) = g(\{X_i\})$.
- One Tuple Dependence (OTD):An aggregate function has one tuple dependence if $g(\{X_i\}) \in \{X_i\}$, i.e. the aggregate of a set is an element of the set.

Apart from the aggregate functions in SQL (MIN,MAX,SUM,COUNT), we also consider $\bigcup$ and $\bigcap$ as aggregate functions. If we use a bitmap representation for a set, then $\bigcap$ corresponds to the AND operation on bits and $\bigcup$ to the OR operation. Both of these operations are clearly distributive since we can pick $g = \bigcap$ and $g = \bigcup$ respectively.

One property of all the aggregate function discussed is that they have the property of monotonicity. For operators such as MAX, SUM (for non negative numbers), COUNT and $\bigcup$, the aggregate value is monotonically increasing as we move from a more specialized tuple to a more general one. The converse holds for operators like MIN and $\bigcap$. Let $G(t_i)$ denote the aggregate computed using aggregate function G which is associated with tuple t_i. A pair (G, op), where G is an aggregate function and op a relational operator, is said to be *concordant* if for all datacube tuples t_1 and t_2 (where t_1 is more general than t_2),

Function	Idempotent	OTD	Concordant Operators	Discordant Operators	Inverse
Min	Yes	Yes	$\leq$	$\geq$	
Max	Yes	Yes	$\geq$	$\leq$	
Sum (on R^+)	No	No	$\geq$	$\leq$	-value
Count	No	No	$\geq$	$\leq$	-value
$\bigcup$	Yes	No	$\supseteq$	$\subseteq$	
$\bigcap$	Yes	No	$\subseteq$	$\supseteq$	

Figure 2: Distributive Aggregate Functions and their properties

$G(t_1)$ *op* $G(t_2)$ holds ($G(t_i)$ denotes the aggregate computed using aggregate function G which is associated with tuple t_i). Similarly (G, op) is *discordant* if for all datacube tuples t_1 and t_2 (where t_1 is more general than t_2), $G(t_2)$ *op* $G(t_1)$ holds. Hence for MAX, $\geq$ is a *concordant* operator while $\leq$ would be a *discordant* operator.

In this paper we propose two optimizations which enable us to do better than than the naive evaluation strategy suggested above.

For concordant aggregates we have:

- **Generalization**: Suppose we have evaluated a datacube cuboid, $B_1B_2 \dots B_j$ from the base tuples, where $B_1, B_2 \dots B_n$ are the `CUBE BY` attributes. Each tuple at this cuboid has the value ALL for $n - j$ of its attributes. The Generalization principle states that if for tuple t of this cuboid, the condition `"aggregate(G) relationalOperator threshold"` is true, this condition will be true for any generalization of this tuple. In this case we would have $2^j - 1$ generalizations.

- **Specialization** : Similarly, if the condition `aggregate(G) relationalOperator threshold` is false for t, this condition will be false for any specialization of this tuple. The number of possible specializations for a tuple t, is given by $\prod_{i=j+1}^{n} Card_i$ where $Card_i$ denotes the cardinality of the ith attribute B_i.

Complementary versions of generalization and specialization apply for discordant aggregates.

2 Data Cube Algorithms

2.1 Array Based Algorithms

The array-based algorithm proposed by Gray et al. [4] is essentially a main memory algorithm, where all the tuples of the finest level of the datacube are kept in memory as a k dimensional array, where k is the number of `CUBE BY` attributes. The data structure needed for this algorithm will often not fit into memory for sparse relations even when R does. In this case, the algorithm does not apply. When the algorithm does apply, it requires just a single pass over the data. If we are computing a selection over the data cube, we will not be able to make use of specialization since we gain by this only if we examine a tuple multiple times. We can make use of generalization by outputting a tuple as soon as we know that it will be meet the threshold. This saves us the cost of further aggregations made to that counter in memory, which might be significant if the aggregate function itself is expensive to compute.

Zhou et al. [13] have proposed an array based algorithm that computes the datacube using array-chunking techniques. By managing the order in which chunks are processed, substantially less of the result array needs to be kept in memory at any one time than with the algorithm of Gray et al. Nevertheless, when the data is very sparse, this algorithm too may fail since it cannot allocate enough main memory to hold the needed parts of the result array.

2.2 PIPESORT

The PIPESORT algorithm [1] tries to optimize the overall computation of a datacube by providing a set of paths which cover the search lattice and then executing each path in turn. This algorithm makes use of cost estimates of the various ways to compute each cuboid $Q(\vec{B_j})$ to determine which cuboid will actually be used to compute the tuples of $Q(\vec{B_j})$.

Since this algorithm computes various paths in turn and each pass consists of examining all the data in the underlying relation, the techniques we describe with regard to the `Memory-Cube` algorithm of [10] are applicable to PIPESORT too.

2.3 OVERLAP

The OVERLAP algorithm [1] tries to minimize the number of disk accesses by overlapping the computation of the cuboids, by making use of the partially matching sort orders to reduce the number of sorting steps performed.

OVERLAP computes a cuboid tree which has the finest granularity cuboid as the root and covers the entire search lattice. Each cuboid is computed from the tuples of its parent cuboid in this tree. Next a set of cuboids is chosen that can be computed concurrently within the available memory constraints. At evaluation time, the cuboids in the "Partition" state are immediately available for pipelining purposes while those in the "SortRun" state are written to disk; the various runs are subsequently merged, further aggregating as necessary, and the result tuples are pipelined for further computation.

This algorithm sorts the tuples once. Our scheme for exploiting specialization relies on rearranging and altering the base tuples, hence it cannot directly be applied here. Our technique for generalization improves performance because it allows us to skip aggregations we do not have to carry out. If we are computing aggregations for multiple cuboids concurrently, we may not be able to skip tuples. Hence, OVERLAP is not a good candidate algorithm for our optimizations.

2.4 Memory-Cube and Partitioned-Cube

`Partitioned-Cube` and `Memory-Cube` [10] are efficient algorithms for computing datacubes which work particularly well for sparse data.

`Partitioned-Cube` (Figure 3) is an algorithm which uses a divide-and-conquer strategy to divide the problem of computing the datacube over a relation with T tuples and k `CUBE BY` attributes into $n+1$ sub-datacubes for a large number n. The first n of these sub-datacubes each has approximately T/n tuples and k `CUBE BY` attributes. The final sub-datacube has no more than T tuples and has $k-1$ `CUBE BY` attributes. Algorithm `Partitioned-Cube` assumes the existence of a subroutine `Memory-Cube` (Figure 4) which efficiently computes datacubes for relations that fit in memory.

The structure of `Partitioned-Cube` follows the recursive structure of datacubes themselves. A datacube is obtained by fixing each possible value of a `CUBE BY` attribute B_j in turn and computing the tuples in the corresponding sub-datacube, followed by computing the datacube tuples with the value ALL for B_j. Rather than rereading the input relation R for the ALL datacube we read the finest granularity cuboid F, which may be significantly smaller than R if there are many tuples in each group, and is never larger than R.

`Memory Cube` computes the various cuboids of the datacube using the idea of pipelined paths where each path requires the relation to be sorted in a particular attribute ordering. The paths are generated by an algorithm which generates $\binom{k}{\lceil k/2 \rceil}$ paths for a datacube with k `CUBE BY` attributes. These paths are processed in an order which allows us to share as much computation as possible across paths. `Memory-Cube` does not incur any I/O beyond the input of the relation and the output of the datacube itself.

We now examine how we would incorporate specialization and generalization into these algorithms. We first look at `Memory-Cube` and then at `Partitioned-Cube`. We then perform experiments with the modified version of these algorithms which show considerable speedup for queries with restrictive selections.

Algorithm `Partitioned-Cube`$(R, \{B_1, \ldots, B_m\}, A, G)$
INPUTS: A set of tuples R, possibly stored in horizontal fragments; `CUBE BY` attributes $\{B_1, \ldots, B_m\}$; attribute A to be aggregated, aggregate function $G()$.
OUTPUTS: The datacube result for R over $\{B_1, \ldots, B_m\}$ in two horizontal fragments F and D on disk. F contains the finest granularity tuples and D contains the remaining tuples. (F and D may themselves be further horizontally partitioned.)

Method:

If (R fits in memory) return `Memory-Cube`$(R, \{B_1, \ldots, B_m\}, A, G)$;
else { Choose an attribute B_j among $\{B_1, \ldots, B_m\}$;
  Scan R and partition on B_j into sets of tuples $R_1, \ldots, R_m$;
  `/*` $n \leq$ card(B_j) and $n \leq$ number of buffers in memory `*/`
  for$i = 1 \ldots n$
    let $(F_i, D_i) =$ `Partitioned-Cube`$(R_i, \{B_1, \ldots, B_m\}, A, G)$
  let $F =$ the union of the F_i's
  let $(F', D') =$ `Partitioned-Cube`$(F, \{B_1, \ldots, B_{j-1}, B_{j+1} \ldots, B_m\}, A, G)$
  let $D =$ the union of F', D' and the D_i's
  return (F, D) ;}

Figure 3: Algorithm `Partitioned-Cube`

INPUTS: A set of tuples R, that fits in memory; `CUBE BY` attributes $\{B_1, \ldots, B_m\}$; attribute A to be aggregated, aggregate function $G()$.
OUTPUTS: The datacube result for R over $\{B_1, \ldots, B_m\}$ in two horizontal fragments F and D on disk. F contains the finest granularity tuples and D contains the remaining tuples.

Method:
Sort R and combine all tuples that share all the values of $\{B_1, \ldots, B_m\}$;
`/*` Assume that tuples are sorted according to first sort order `*/`
for each sort order {
  initialize accumulators for computing aggregates at each granularity
  combine first tuple into finest granularity accumulator
  for each subsequent tuple t {
    Compare t with previous tuple, to find position j of the first sort order attribute at which they differ
    if (j exceeds number of common attributes with next sort order) then {
      resort segment from t'(last tuple where condition is satisfied) to tuple prior to t according to next sort order.
    }
    if (grouping attributes of t differ from finest granularity accumulator) {
      output and combine each accumulator into coarser granularity accumulator until grouping attributes match those of t;
      `/*` Number of combinings depends on sort order length and j `*/`
    }
    Combine current tuple with finest granularity accumulator;
  }
}

Figure 4: Algorithm `Memory-Cube`

3 Specialization

Let us define a datacube tuple to be *small* if its aggregate is below threshold and *large* if it is above. For concordant operators, we output *large* tuples; for discordant *small*. We can make use of Specialization by taking certain actions whenever we compute a *small* data cube tuple.

If the aggregate function has the OTD property (both MIN and MAX have this property) and the relational operator is concordant we can speed up Memory-Cube by simply dropping all base tuples for which the value of the attribute being aggregated is below the threshold. The OTD property ensures that this does not affect the correctness of our result. Note that if the relational operator is discordant we cannot make use of this optimization.

We can gain from this optimization in a number of ways:

1. The number of tuples for subsequent sort-orders is smaller. The time spent sorting will hence be less.
2. Since the number of tuples has decreased the time spent processing each sort-order is reduced.
3. We still maintain the exact attribute value for large tuples, so we can handle both variations of our sample query.

We can incorporate this optimization into Memory-Cube by incorporating a preprocessing check for each tuple t being processed.

If our distributive aggregate function does not have the OTD property, we cannot drop the tuples since this would affect the correctness of the result.

3.1 1-Specialization

Let us investigate how we would make use of a **1-Specialization** for an aggregate function g, a concordant relational operator op and a threshold which is a constant T.

Definition 3.1: We define a cuboid consisting of tuples with only one non-ALL attribute value to be a 1-cuboid. $\square$

1-Specialization applies when we have computed a datacube tuple in a 1-cuboid. If we have 4 cube by attributes; $< a_1, ALL, ALL, ALL : v_1 >$ would be one such example tuple (v_1 indicates the aggregate computed using function G for this tuple). If we know that this tuple is small (i.e $v \; op \; T$ is false), we can infer that there will be no datacube tuple with the value a_1 for attribute A in the result. Hence, we can replace all instances of a_1 in the base tuples with the a special blank value $\sqcup$. The benefit of this optimization is that if $< a_2, ALL, ALL, ALL : v_2 >$) is also small, we may have altered base tuples of the form $< \sqcup, b_1, c_1, d_1 : v >$ produced by both specializations. These tuples can be combined into a single tuple where v_1 and v_2 are merged using h, a function which allows us to merge tuples. As mentioned earlier for any distributive aggregate function g there exists a function h such that $g(\{X_{i,j}\}) = h(\{g(\{X_{i,j} | i = 1, \ldots I\}) | j = 1, \ldots, J\})$. For the functions MAX, MIN and SUM $h = g$. For the function COUNT, h = SUM. In a best case scenario, we reduce the number of tuples in the next sort order by a factor of the cardinality of attribute a. We do not increase the number of base tuples in this optimization.

Definition 3.2: Let us define 1-Specialization for a set of base tuples (R) and a datacube query with k `CUBE BY` attributes, an aggregate function g, a concordant relational operator op and a threshold (T). Consider a small 1-cuboid datacube tuple t with value x in attribute j. 1-Specialization consists of replacing x with a $\sqcup$ in the jth `CUBE BY` attribute for all tuples of R. $\square$

Example 3.1: We have a datacube with 4 `CUBE BY` attributes and the first sort order we are processing is $ABCD$. Hence the first lattice path which we compute is $ABCD \rightarrow ABC \rightarrow AB \rightarrow A \rightarrow \phi$. The next sort order is DAB and the corresponding lattice path is $DAB \rightarrow DA \rightarrow D$. Suppose both a_1 and a_2 are small.

Consider the following run of tuples:

$$\begin{array}{lcl} < a_1, b_1, c_1, d_1 : v_1 > & & \\ < a_1, b_1, c_1, d_4 : v_2 > & & \\ < a_1, b_1, c_2, d_3 : v_3 > & & < \sqcup, b_1, c_1, d_1 : h(\{v_1, v_6\}) > \\ < a_1, b_1, c_2, d_4 : v_4 > & & < \sqcup, b_2, c_1, d_1 : v_5 > \\ < a_1, b_2, c_1, d_1 : v_5 > & & < \sqcup, b_2, c_1, d_2 : v_{10} > \\ < a_2, b_1, c_1, d_1 : v_6 > & \Rightarrow & < \sqcup, b_1, c_2, d_3 : h(\{v_3, v_8\}) > \\ < a_2, b_1, c_1, d_4 : v_7 > & & < \sqcup, b_1, c_1, d_4 : h(\{v_2, v_7\}) > \\ < a_2, b_1, c_2, d_3 : v_8 > & & < \sqcup, b_1, c_2, d_4 : h(\{v_4, v_9\}) > \\ < a_2, b_1, c_2, d_4 : v_9 > & & \\ < a_2, b_2, c_1, d_2 : v_{10} > & & \end{array}$$

After using the 1-Specialization optimization, compacting tuples and sorting we obtain the run of tuples on the right from the run of tuples on the left. □

At this point we have to explain how we treat a tuple t which has one or more $\sqcup$'s as attribute values in the context of the `Memory-Cube` algorithm. If the $\sqcup$ occurs in an attribute which is not a grouping attribute in any of the nodes on the lattice path, we can treat these tuples in an identical fashion to other tuples.

In general, if a $\sqcup$ occurs as the ith attribute in the first node of a lattice path with j attributes and the number of nodes of the lattice path is k, we have a formula for determining which accumulators will need to be affected. Due to space constraints, we omit the derivation of this formula. We use this formula in `Memory-Cube` to combine a tuple with a $\sqcup$ with the appropriate accumulator.

When sorting tuples we can consider a $\sqcup$ to be less than any other value. The other aspect which needs to be addressed is the timing of when tuples are marked with a $\sqcup$. In `Memory-Cube` we sort tuples according to the sort order of the next path immediately after processing a tuple. Now we sort tuples only when we have finished computing a tuple of a 1-cuboid. We then scan all tuples which have just been processed and contribute to this tuple, and replace all values of a_i with $\sqcup$. After all the cuboids on a path have been computed, we can carry out the sorting and the subsequent compression step. If no 1-cuboids are being computed on a path, we can use the default of sorting tuples immediately.

3.2 2-Specialization

2-Specialization is similar to 1-Specialization. Here, we make use of the fact that if a tuple from a 2-cuboid is small, all tuples which are specializations of this tuple will be small too.

We can exploit this in a similar way to 1-Specialization by introducing special values $\sqcup$ into the tuples which contribute towards small tuple from the 2-cuboid. However, we cannot directly replace the non-ALL attribute values with $\sqcup$'s in the base tuples contributing to the small tuple of the 2-cuboid. The reason for this is illustrated in the following example.

Example 3.2: Consider a base tuple of the form $< a_1, b_1, c_1, d_1 : v_1 >$ which contributes to the small tuple $< a_1, b_1, ALL, ALL : V >$. If we replace the base tuple with $< \sqcup, \sqcup, c_1, d_1 : v_1 >$ we would no longer have the correct answer when computing tuples of the form $< a_i, ALL, c_i, ALL : V_2 >$. Since we ignore tuples which have a $\sqcup$ in the grouping attributes for a particular sort order, the tuple $< \sqcup, \sqcup, c_1, d_1 : v_1 >$ would be ignored while computing $< a_1, ALL, c_1, ALL >$. However, it is clear that V_2 is dependent on v_1. □

We can remedy the situation by replacing each base tuple with 3 new tuples. Let us say the small tuple computed has non-ALL attributes in attributes p and q and they take the value of x and y respectively. Of the three new tuples, one of these tuples will have x replaced with a $\sqcup$, one would have y replaced with a $\sqcup$ and the third would have both replaced with $\sqcup$'s. The introduction of $\sqcup$'s in the first two tuples give us potential for compression while the third tuple is introduced for correctness. If the original tuple has the value v for the attribute being aggregated, the first two would have v too while the third would have the *inverse* of v. The inverse of an attribute exists for non idempotent distributive aggregate functions like SUM and COUNT. The inverse function here consists of negating the value v (Normally for COUNT we are negating 1, however if we are 2-specializing a tuple which has been formed from multiple tuples while computing a previous path we need to negate its current count). This third tuple ensures correctness while computing cuboids to which both the earlier tuples would contribute. If the aggregate function is idempotent (MAX,MIN,$\bigcup$,$\bigcap$), we are not affected by double counting since the value of the aggregate will remain unchanged. In this case we only need to introduce the first two tuples.

Example 3.3: Continuing our earlier example we would replace $< a_1, b_1, c_1, d_1 : v_1 >$ with 3 tuples : $< \sqcup, b_1, c_1, d_1 : v_1 >$, $< a_1, \sqcup, c_1, d_1 : v_1 >$, $< \sqcup, \sqcup, c_1, d_1 : inv(v_1) >$. The third tuple is necessary so we do not have an error while computing cuboid CD; both $< \sqcup, b_1, c_1, d_1 : v_1 >$ and $< a_1, \sqcup, c_1, d_1 : v_1 >$ contribute to $< ALL, ALL, c_1, d_1 : h(\{v_1, v_1\}) >$ and so we have counted the original tuple twice. The third tuple compensates for this double counting. □

While 1-Specialization does not increase the number of tuples, 2-Specialization may lead to a net *increase* in the number of tuples if we do not have substantial savings in the compression step. The compression factor depends upon the sparseness of the data as well as the number of tuples in the 2-cuboid for which the aggregate is below the threshold.

3.3 n-Specialization

In a similar fashion to 2-Specialization we can define n-Specialization. In this case we introduce $2^n - 1$ tuples for each original tuple if the aggregate function is invertible but not idempotent. Each of these tuples is formed by replacing one or more of the n attribute values with $\sqcup$'s. We can assign values of the attribute being aggregated to the new tuples in the following fashion. We assign the inverse of the original aggregated value to all tuples in which there are an even number of $\sqcup$'s and the original value to those tuples with an odd number of $\sqcup$'s. Due to space constraints, we omit the proof of correctness.

Example 3.4: Consider the 3 specialization case, with tuple $< a_1, b_1, c_1, ALL, ALL : V >$ having just been computed and shown be to small with respect to a concordant aggregate. We can replace $< a_1, b_1, c_1, d_1, e_1 : v >$ with $2^3 - 1 = 7$ tuples namely:

$$\begin{gathered} < \sqcup, \sqcup, \sqcup, d_1, e_1 : v > \\ < a_1, \sqcup, \sqcup, d_1, e_1 : inv(v) > \\ < \sqcup, b_1, \sqcup, d_1, e_1 : inv(v) > \\ < \sqcup, \sqcup, c_1, d_1, e_1 : inv(v) > \\ < a_1, b_1, \sqcup, d_1, e_1 : v > \\ < a_1, \sqcup, c_1, d_1, e_1 : v > \\ < \sqcup, b_1, c_1, d_1, e_1 : v > \end{gathered}$$

□

We only need to introduce all $2^n - 1$ elements if the aggregate function is non idempotent like COUNT and SUM. If the aggregate function is idempotent, we can ensure correctness by introducing a minimum of n tuples where these n tuples would be the ones with only one instance of $\sqcup$.

The major gains of n-Specialization come from the reduction in the number of tuples as we move from sort order to the next. However, as n increases so does the likelihood of a net increase in the number of tuples.

We also need to work out how to deal with n-Specialization in the context of memory cube as in a single pipeline we may be having multiple specializations originating due to different levels. We notice that for $i < j$, i-Specialization dominates j-Specialization. We only need to carry out the adjustment for i-Specialization. We can handle this by delaying carrying out specializations until we determine whether we will be carrying out any 1-Specializations.

3.4 Generating Paths for `Memory-Cube`

In `Memory-Cube` we generate a set of paths which cover the search lattice and execute them in turn. To make better use of specialization, it is better to execute paths containing a 1-cuboid earlier on, since 1-Specialization gives us the maximum benefit by decreasing the number of tuples. After executing these paths, we can use the previous heuristic of ordering paths lexicographically to maximize work shared across sort orders.

4 Generalization

The Generalization principle states that if a datacube tuple is large then all generalizations of this tuple are also large. If the data cube has n `CUBE BY` attributes and the tuple computed has m non-ALL attributes, the tuple has $2^m - 1$ generalizations. The Generalization optimization holds only for the left query of Figure 1 where we do not need to know the exact value of the aggregate. Let us see how this optimization would be incorporated into the `Memory-Cube` algorithm.

Whenever a large tuple is computed we know that all generalizations of this tuple are large. Some of these generalizations will lie in cuboids on the current path being executed. We can automatically flush the accumulators for these cuboids which are maintained for the pipelined execution of the path.

We can adjust for the remaining tuples in the following manner by adding a special extra tuple to the set of base tuples. This extra tuple would contain marked values of the non-ALL attributes. This marked tuple would be given special treatment on subsequent lattice paths. When sorting these marked tuples, we use the following rule: A marked value a_i^* is treated such that $a_{i-1} < a_i^* < a_i$. On a subsequent lattice path, we treat marked tuples as follows. If all the grouping attributes for the first node on the lattice path correspond to a `*`'d attribute, we can immediately reach the conclusion that tuple currently being computed is large and so are all of its generalizations along the path currently being computed. In a similar fashion we can flush the accumulators for these cuboids.

Example 4.1: If the current lattice path being examined is $ABCD \rightarrow ABC \rightarrow AB \rightarrow A \rightarrow \phi$, and we have just computed a large tuple $< a_1, b_1, c_1, ALL : v_1 >$, the tuples about which we can make generalizations are the following:
1. $< a_1, b_1, ALL, ALL : v_2 >$
2. $< a_1, ALL, c_1, ALL : v_3 >$
3. $< ALL, b_1, c_1, ALL : v_4 >$
4. $< a_1, ALL, ALL, ALL : v_5 >$
5. $< ALL, b_1, ALL, ALL : v_6 >$
6. $< ALL, ALL, c_1, ALL : v_7 >$
7. $< ALL, ALL, ALL, ALL : v_8 >$

Here, tuples 1,2,4 and 7 lie on the same lattice path and can be handled by flushing the accumulators.

We introduce a tuple $< a_1^*, b_1^*, c_1^*, ALL >$. Let the subsequent lattice path be $CAD \rightarrow CA \rightarrow C$. When we encounter tuple $< a_1^*, b_1^*, c_1^*, ALL >$ we have yet to encounter any tuples of the form $< a_1, b_1, c_1, d_1 >$ since we treat ALL as being less than any other attribute value.

At this point, we know that we do not have to compute any aggregates for the CA and C cuboids for a_1 and c_1. We next have to alter values for the CA cuboid when we encounter a new value of a. At this point we can write out the values for $< a_1, ALL, c_1, ALL >$ and $< ALL, ALL, c_1, ALL >$ without further computation. We cannot skip on computing CAD tuples. □

Since we know what the lattice paths are going to be beforehand, we can encode the last lattice path for which the special tuple would be required and discard the marked tuple after we have computed that lattice path.

It is always a win to generalize for tuples which lie along the current lattice path since we are reducing the number of aggregations. It is not clear that our scheme of introducing marked tuples will always cause an improvement in performance. Since we are introducing extra tuples there will be an increase in the costs of sorting. Furthermore, the benefits of introducing this tuple depend upon how many computations it allows us to skip. This would be affected by which cuboids have already been computed as well as the number of non-ALL attributes in the tuple being generalized.

The gains of generalization are more pronounced when the the cost of computing an aggregate is expensive. If we were dealing with sets of large cardinalities this would be the case whether we used a bit mapped representation or a list representation.

5 Concordances and Discordances

Our treatment of specialization and generalization has focussed on concordant operators. Let us see how our modifications to `Memory-Cube` are affected if the relational operator is discordant rather than concordant.

Consider the left query from Figure 1 with the aggregate function and the relational operator pair being discordant (e.g. MAX and $\leq$). The situation is reversed as we now want to output small datacube tuples rather than large ones.

From the perspective of generalization, we have to make just one change. For generalizations along the same path, we can flush the accumulators but in this case we do not output the corresponding tuples. We can still use our technique of introducing marked tuples. Marked tuples can be used to skip computing aggregates, once again the aggregations skipped are not output.

Let us see what happens in the case of 1-Specialization (the situation will be similar for n-Specialization) when we have just computed a small tuple from a 1-cuboid. Clearly all specializations of this tuple will also be small. We cannot use our earlier technique of replacing all attribute values with $\sqcup$'s since now we would need to output these specialized tuples on subsequent paths.

A possible line of attack could be to try using our technique of introducing $\sqcup$'s for large tuples from a 1-cuboid. Unfortunately, this is where the difference between concordant and discordant operators comes into play. In this case, all specializations of the the tuple are not necessarily large and hence may need to be output.

6 Partitioned Cube

We have dealt so far only with the `Memory-Cube` algorithm in [10], however we need to address how the optimizations , specialization and generalization could be used in conjunction with `Partitioned-Cube`. The key idea behind this algorithm is divide and conquer. The problem of computing a relation R with T tuples and k `CUBE BY` attributes into $n+1$ sub-datacubes, for a large number n. The first n of these sub data cubes each have approximately T/n tuples and k `CUBE BY` attributes. the final sub-datacube has no more than T tuples and $k-1$ `CUBE BY` attributes. We modify the Partitioned Cube algorithm to optimize for selections. The outline of the process illustrated in Figure 5 follows.

We first split the relation on the basis of a partitioning attribute a, where each sub-datacube corresponds to a distinct value of a. While carrying out the partitioning scan we can simultaneously generate all 1-cuboids. This is done by maintaining in memory counters for each of the distinct attribute values. Since we are performing this step for 1-cuboids the memory requirements of this is bounded by the sum of the cardinalities of each attribute. We use this information while executing each of the individual sub-datacubes.

1. If a particular partitioning attribute a_i in not present in any tuple in 1-cuboid for a, it means that we do not have to compute this sub-datacube at all.
2. Before we start processing each sub-datacube we can carry out a 1-Specialization step over the tuples. In this step, each value of each attribute in a tuple is replaced by a $\sqcup$ if that value is not present in the tuples of the corresponding 1-cuboid. We can compact these tuples before we process the sub data cube. This allows us to gain by reducing the number of tuples before processing any of the sort orders. We carry out this step even if we do not have to compute the sub data cube since we will require these tuples later for computing the sub-datacube with $k-1$ attributes.

The tuples passed to each of the n subcubes (marked with R's in Figure 5) each have a distinct value of the partitioning attribute. This immediately means that any generalization of a tuple in datacube i will either be in the same datacube or in the datacube with $k-1$ attributes. Within each datacube we can carry out generalization and specialization.

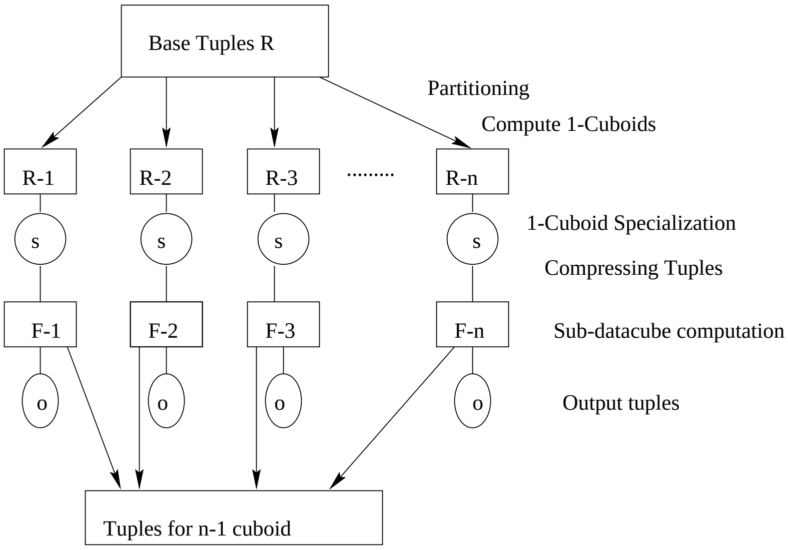

Figure 5: Modified Partition-Cube

The next issue we have to address is which tuples should be used for the sub datacube with $k-1$ attributes.

For concordant operators with the OTD property we only need to pass on those base tuples which are themselves large. If we do not have the OTD property we cannot adopt this scheme nor can we use the post processed F tuples. The reason for this is illustrated by the following example:

Example 6.1: We are computing a datacube with 5 `CUBE BY` attributes A, B, C, D, E. In the partitioning step, we partition by attribute A. Consider the sub datacube with value a_i in all its constituent tuples. We use information from the 1-cuboids computed during the partitioning phase to compact the T tuples to obtain the F tuples. Assume the first lattice path we compute is $ABCDE \rightarrow ABCD \rightarrow ABC \rightarrow AB \rightarrow A$, We are operating upon the tuples marked with F's in Figure 5. If we compute tuple $< a_i, b_j, ALL, ALL, ALL >$ and it turns out to be small, we can replace b_j with a $\sqcup$ in all the base tuples for the rest of this subcube. If we had a tuple $< a_i, b_j, c_x, d_y, e_z >$, we would replace it with $< a_i, \sqcup, c_x, d_y, e_z >$. As in earlier cases we gain if we have similar tuples of this form since we can compact them into a single tuple. However, when computing the sub datacube with 4 `CUBE BY` attributes we would have to compute a tuple of the form $< ALL, b_j, c_x, ALL, ALL >$. The tuple $< a_i, b_j, c_x, d_y, e_z >$, which we earlier $\sqcup$'ed, contributes to this tuple. Hence, it is necessary for us to use a copy of the tuples which have not been modified with $\sqcup$'s (the tuples marked with F's in Figure 5 prior to the sub-cube computation). This is superior to tuples marked with T's in Figure 5 since we are making use of the 1-cuboid information to compress tuples. $\square$

It is likely that the subcube with $k-1$ attributes is smaller than it normally would be since we are using tuples marked F and not those marked T. This might enable us to carry out the computation of the $k-1$ sub datacube in memory without having to resort to another partitioning step.

Generalization is unaffected by whether the relational operator is concordant or discordant. For discordant operators we can carry out specialization only in the case where OTD property holds. We have the same caveat about using the original base tuples for the $k-1$ subcube.

Algorithm `Modified Partitioned-Cube`$(R, \{B_1, \ldots, B_m\}, A, G)$
INPUTS: A set of tuples R, possibly stored in horizontal fragments; `CUBE BY` attributes $\{B_1, \ldots, B_m\}$,a concordant selection condition S; attribute A to be aggregated, aggregate function $G()$.
OUTPUTS: The datacube result for R over $\{B_1, \ldots, B_m\}$ as D on disk.

Method:

If (R fits in memory) return `Memory-Cube`$(R, \{B_1, \ldots, B_m\}, A, G, S)$;
else { Choose an attribute B_j among $\{B_1, \ldots, B_m\}$;
  Scan R and partition on B_j into sets of tuples $R_1, \ldots, R_m$;
  Determine large tuples of all 1-cuboids;
  `/*` $n \leq \text{card}(B_j)$ and $n \leq$ number of buffers in memory `*/`
  for$i = 1 \ldots n$ {
    Using 1-cuboid information transform R_i to F_i
    let $(F_i, D_i) =$ `Partitioned-Cube`$(F_i, \{B_1, \ldots, B_m\}, A, G, S)$
  }
  let $F =$ the union of the F_i's
  let $(F', D') =$ `Partitioned-Cube`$(F, \{B_1, \ldots, B_{j-1}, B_{j+1} \ldots, B_m\}, A, G, S)$
  let $D =$ the union of D' and the D_i's
  return (D) ;}

Figure 6: Algorithm `Modified Partitioned-Cube`

7 Experimental Results

We have implemented the modified version of the `Memory-Cube` algorithm presented in Figure 7. This version of `Memory-Cube` is modified so that performance is not worse than with the unmodified version. Hence, we implement only 1-Specialization and generalization for tuples in the same path. Specialization will have the maximum effect when we have lots of small tuples in the 1-cuboid while generalization will help us benefit when we have many large tuples in the finer cuboids of the datacube. Thus, our optimizations would provide gains in both of these complementary cases.

We implemented this algorithm in `C++` and it computes the datacube of a partition that fits in memory. The data is read in or synthetically generated internally. Data is assumed to consist of 4-byte integer values on all grouping and aggregated attributes, and it is assumed that no extraneous attributes are present. We report results for the SUM and COUNT operators over a single attribute.

We ran the datacube algorithm on a 200 MHz UltraSparc single processor with 256MB of RAM. The algorithm was run when no other processes were active on the system. We measured both the CPU time and the elapsed time. In all our experiments these two measurements were within three percent; we use the CPU time in our results. The time was measured from the point after the input relation was read or generated until the end of the datacube computation.

A large fraction of the computing the datacube is spent sorting since as the number of paths required to cover the search lattice is is $\binom{n}{\lceil n/2 \rceil}$ for n `CUBE BY` attributes and each path requires a sorting step. To minimize the sorting cost we adopt a scheme described in detail in [11]. We modify the input tables in such a way as to reduce the sorting cost, and then reconstitute the original values at the end. This is done by applying Huffman coding independently to each attribute domain to get bit strings corresponding to each attribute value. We then carry out a counting sort using a composite surrogate key formed by concatenating the Huffman codes from each of the attributes in a sort order. The counting sort operation counts the frequency of each b-bit prefix of this surrogate key for some b. A cumulative histogram is created, and another pass through the data puts it in the order of its most significant b bits. Values that share the same b-bit prefix are then ordered using quicksort. This scheme leads to substantial speedup over quicksort.

In Figure 8 we investigate the effect of on performance time when we vary the threshold in the selectivity condition. We use a synthetic dataset with 500000 tuples and 6 `CUBE BY` attributes and aggregate function

INPUTS: A set of tuples R, that fits in memory; `CUBE BY` attributes $\{B_1, \ldots, B_m\}$,a concordant selection condition S; attribute A to be aggregated, aggregate function $G()$.
OUTPUTS: The datacube result for R over $\{B_1, \ldots, B_m\}$ in two horizontal fragments F and D on disk. F contains the finest granularity tuples and D contains the Output tuples.

```
Method:
Sort R and combine all tuples that share all the values of {B_1, ..., B_m};
/* Assume that tuples are sorted according to first sort order */
for each sort order {
        initialize accumulators for computing aggregates at each granularity
        combine first tuple into finest granularity accumulator
        for each subsequent tuple t {
            if ((S has OTD property)&&(tuple is small)) { Drop t from set of tuples }
            Determine position l of first ⊔ in grouping attributes of tuple
            Compare t with previous tuple, to find position j of the first sort
            order attribute at which they differ
            if (grouping attributes of t differ from finest granularity accumulator) {
                    combine each accumulator into coarser granularity
                    accumulator until grouping attributes match those of t;
                    Output those accumulator values which are large.
                      Flush all accumulators for coarser granularities .
                    /* Number of combinings depends on sort order length and j */
                    if (we have just computed a small tuple in a 1-cuboid)
                      introduce blanks into contributing base tuples;
            }
            Combine current tuple with appropriate accumulator;
        }
        Sort according to next sort order making use of shared work if possible
        Merge duplicate tuples produced by specialization.
}
```

Figure 7: Modified Memory-Cube for Specialization and Generalization

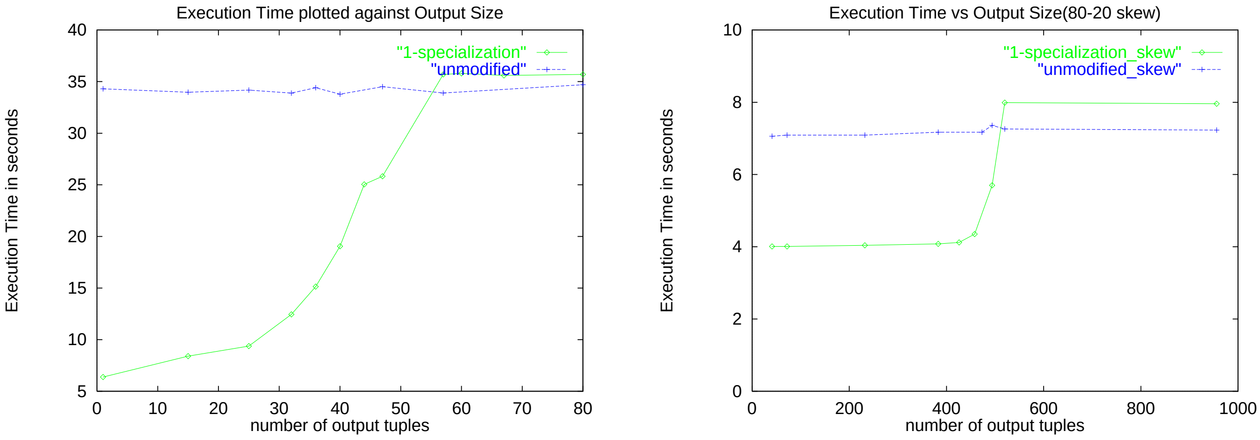

Figure 8: Varying the size of the output

COUNT. The left figure has uniformly distributed data while the right hand figure has 80/20 skew. We see a considerable improvement in performance when the selection condition is restrictive. The degradation of our algorithm when we do not have any opportunity for 1-Specialization is small, the difference in performance is caused by the code introduced for handling $\sqcup$'s. Note that the selectivity of the condition in terms of the number of output tuples cannot be made across data sets. With a particular set of data, we might have many fine output tuples and a few coarse tuples. With a different dataset we might have the same number of output tuples but with a different breakup between coarse and fine tuples.

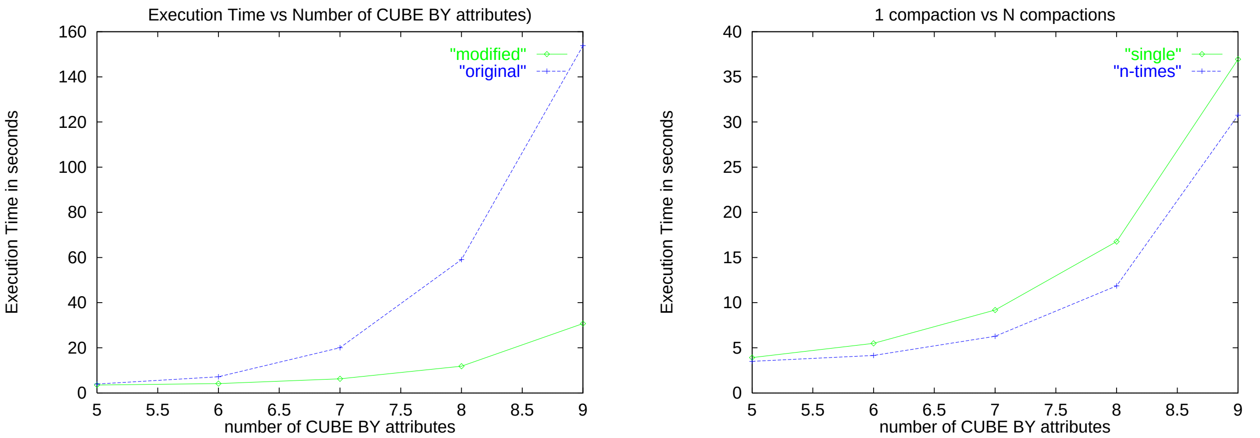

Figure 9: Varying number of CUBE BY attributes with constant $|R|$

In Figure 9, we vary the the number of `CUBE BY` attributes for a fixed number of base tuples. In each case we generate 500000 tuples with a 80/20 skew and a fixed output size. We can fix the output size by appropriately choosing the threshold for each case. The left hand graph shows how the modified algorithm compares to the original one. In the right hand graph, we study the impact of having a single compression step against multiple compression steps in the `Memory-Cube` algorithm. We can have a single step in which we compress tuples after carrying out all possible 1-Specializations. Alternatively, whenever we carry out 1-Specialization we can compress the tuples. The graph indicates that the second alternative is better.

In Figure 10 we study the impact of varying the size of the base relation with a fixed output size and a constant number of `CUBE BY` attributes. We see that the performance gap between the two versions of `Memory-Cube` widens as the number of tuples increases for both skewed and uniformly generated data.

Example 7.1: We also experiment upon real-world data on cloud coverage [5]. The data used corresponds to measurements of the amount of cloud coverage throughout the globe over a period of one month, September

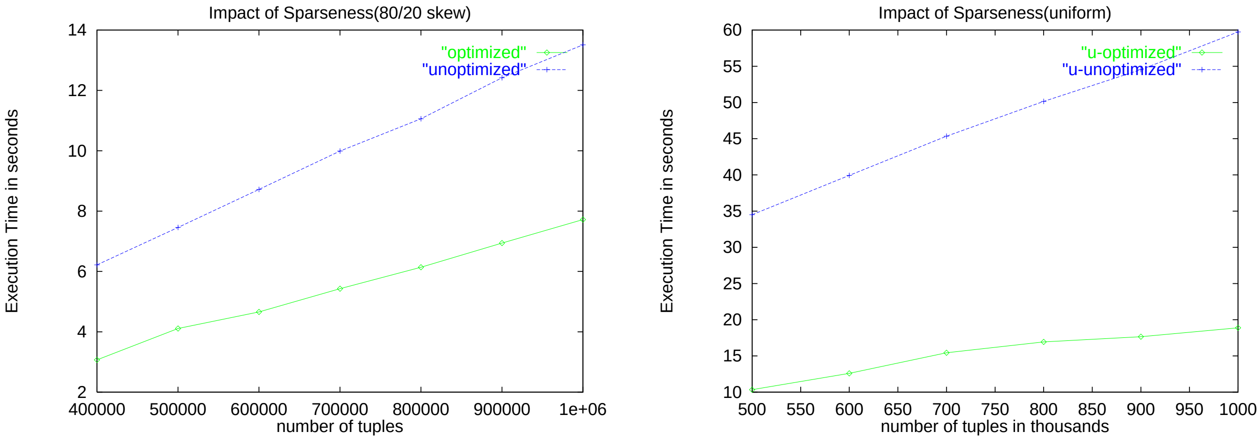

Figure 10: Varying $|R|$ with fixed output, `CUBE BY` attributes

1985. We use a data set containing 117,635 tuples for measurements made over the ocean. We have chosen 9 `CUBE BY` attributes out of a possible 20 fields. We have carried out experiments with SUM as the aggregate function. In the graphs of Figure 11 we show the difference in performance between the unmodified and modified versions of `Memory-Cube` as well as the difference in performance between having a single compression step and n compression steps. We see that the modified version of `Memory-Cube` does better than the unmodified version for restrictive selections. The right graph indicates that for restrictive selections it is better to have many compression steps, but there is nothing to be gained in doing so when that is not the case. □

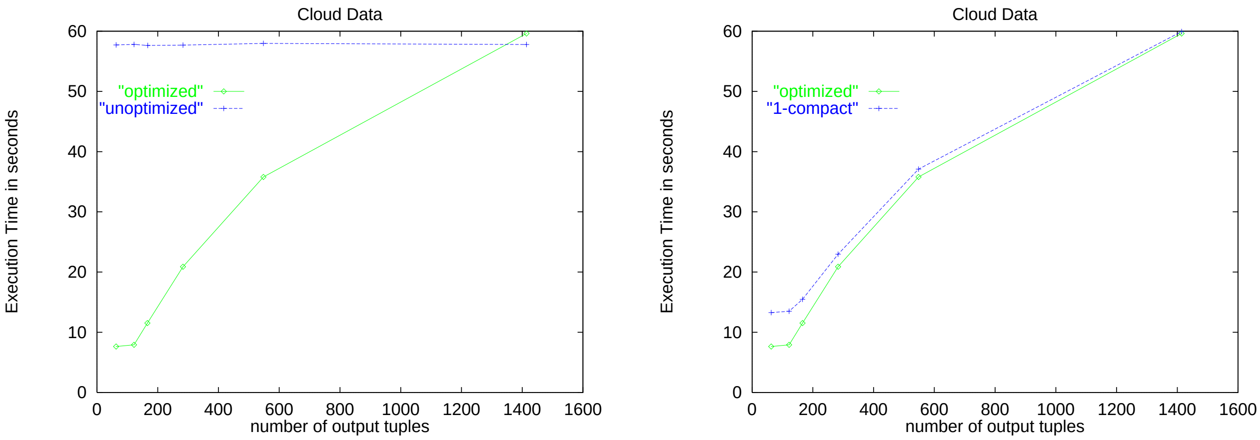

Figure 11: Cloud Data

We carry out experiments to show the effect of generalization when we have a relatively expensive aggregation operation. The left hand graph of Figure 12 shows the performance using a simulated[1] user-defined expensive aggregate on a single aggregation column for a datacube with a large number of output tuples. As we can see in Figure 12, generalization leads to great improvements in performance under these circumstances. In the right hand graph, we use an actual user defined expensive aggregation function. Our function is a variation of the union function where along with the elements of a set we also maintain the count of the number of elements in a set. The set is stored as a bitmap using many aggregation columns (0123456789 in this example). Note that the count is a *distinct* count, and could not be computed without the set being explicitly represented. We output only those datacube tuples which have a more than a certain

[1] We artificially made the aggregate more expensive by including some irrelevant computation.

number of elements in a set. This experiment is carried out on a synthetically generated uniform dataset with 100000 tuples. The query used had 6 `CUBE BY` attributes.

Example 7.2: Returning to our example of cars, suppose that instead of using sales as the aggregate column, we use "color". We use a bitmap set representation for the set of colors used at a particular granularity. As discussed above, we also keep a count of the number of (distinct) colors. We can apply the generalization optimization above for a query like "Find combinations of manufacturers and models for which cars of more than a certain number of colors were sold." □

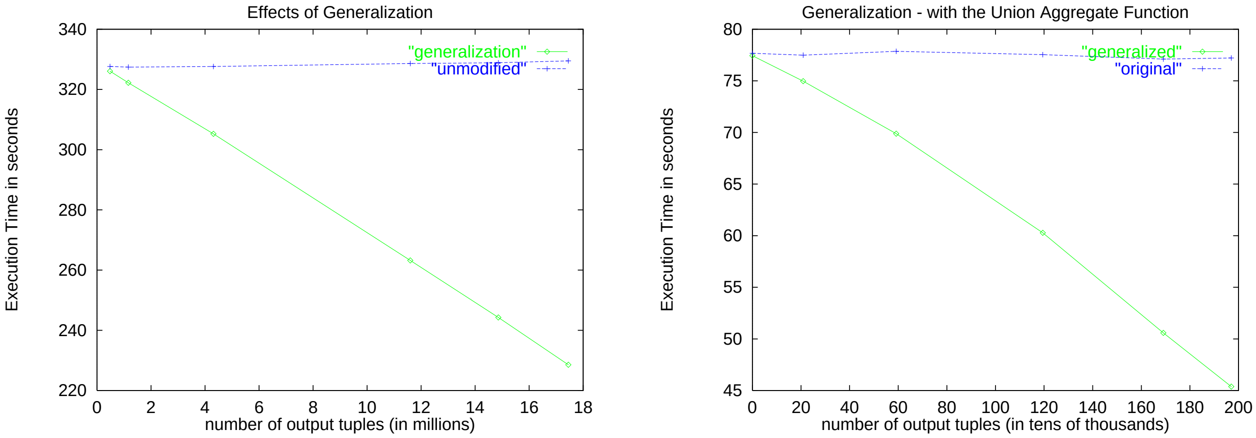

Figure 12: Effects of Generalization

We carry out experimentation with the same parameters but this time using a discordant relational operator (we output those datacube tuples with less than a certain number of elements in a set) in conjunction with our aggregate function. In this case, we can handle the right hand query of Figure 1 too. This is because we do not need to output the tuples we generalize. The results in Figure 13 (left graph) show that there is a substantial improvement in performance when the number of output tuples is small.

In our final experiment, we show the improvement in performance when the relational operator has the OTD property and we can drop small tuples. In the right hand graph of Figure 13, we use the real-world cloud data set, MAX as the aggregate function and $\geq$ as the relational operator. We can see that the improvement in performance is the greatest when the number of output tuples is small.

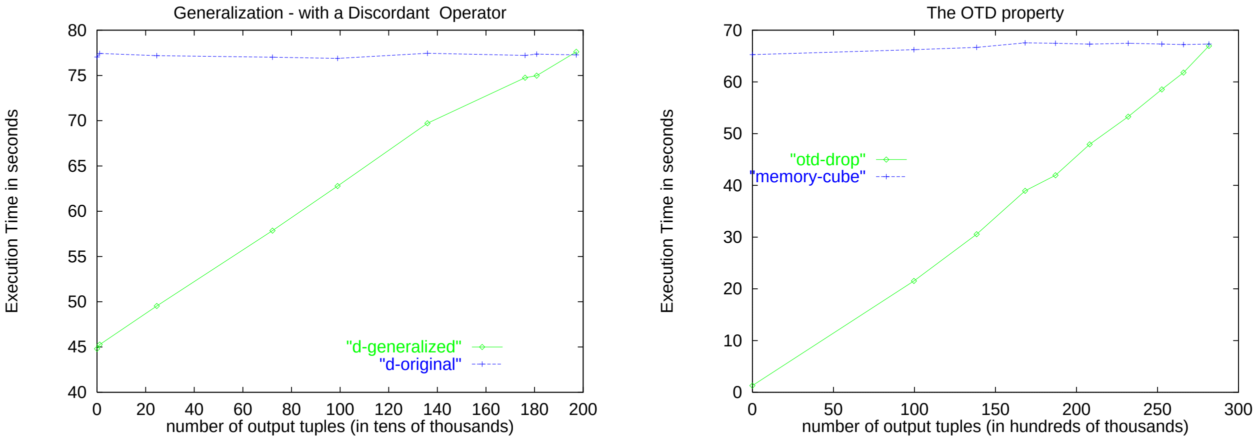

Figure 13: Effect of Discordant Operators and the OTD property

$L_1 = \{\text{Large 1-Cuboids}\}$
for $(k = 2; L_{k-1} \neq \phi; k++)$ do begin
$C_k = Generate(L_{k-1})$
forall tuples $t \in R$ do begin
$C_t = subset(C_k, t)$; (Generates all possible candidate cuboids from a tuple)
forall candidates $c \in C_t$
aggregate t into c
end
$L_k = \{c \in C_k | aggregate\ is\ large\}$
end
$Answer = \bigcup_k L_k$.

Figure 14: Modified Apriori Algorithm

8 Related Work

The gains of our optimizations depend upon a number of factors, one of which is the fraction of datacube tuples (f) which constitute the output. If we are dealing with a concordant relational operator and we know that f is very small, a variant of the Apriori algorithm [2] applies. This algorithm is outlined in Figure 14 and proceeds as follows.

In the first step of the algorithm, we compute all 1-cuboids, namely all tuples which have only one non-ALL element. The amount of memory required for this step is proportionate to the sum of the cardinalities of each of the attributes of the relation. We can tighten this bound by initializing an in-memory counter only for those attribute values which are actually present in the base tuples. Once we have finished a pass over the data, we can determine which of them are large by checking against the threshold. We output all large 1-cuboids (L_1) and discard all small 1-cuboids. In the next step we generate all large 2-cuboids, however we do not use the naive technique of initializing in-memory counters for all possible 2-cuboids. We initialize in-memory counters as follows . We know that if our selection against the threshold does not hold for a particular tuple in a 1-cuboid, it will not hold for for any specialization of the tuple either. Hence, we do not need not initialize in memory counters for any of the 1-cuboid tuples discarded in the previous pass. We generate the set of possible counters (C_2) by choosing all possible pairs from L_1. C_2 should be small since we have discarded all small 1-cuboids. We now scan the data to determine which of the in-memory counters (L_2) satisfy the condition on the threshold.

In a similar fashion, we can generate L_k from L_{k-1}. The only addition in the generic case is that we have an additional pruning step to remove all candidate counters in C_k which have generalizations not present in C_{k-1}. We do not need this step for $k = 2$.

This algorithm is useful only if we know that the number of output tuples is going to be small. If this is not the case, we have the same deficiency as Gray's algorithm; namely, that the number of in-memory counters used is large. If we have n cube by attributes, on the $\lceil n/2 \rceil$'th iteration, we would have $\binom{n}{\lceil n/2 \rceil}$ cuboids where each cuboid had $card^{\lceil n/2 \rceil}$, if each attribute had the same cardinality *card*. This memory requirement of $\binom{n}{\lceil n/2 \rceil} * card^{\lceil n/2 \rceil}$ could easily be in excess of main memory. In fact if we had sufficient memory, in this situation it would be more efficient to directly apply Gray's algorithm rather than split the work across multiple passes.

To restate the main point above, A-priori will work adequately when the number of aggregates to compute is small. (In the terminology of [2], the number of cuboids with high "support" is small.) However if that is not the case, A-priori will perform poorly because not all counters can fit in memory. In contrast, out techniques "degrade gracefully" in that they reduce to the original `partitioned-cube` and `memory-cube` if the selection condition is not restrictive.

1-Specialization and generalization along the same path give a large gain in performance if the inputs are suitable, if not they will not affect the performance of our algorithm noticeably. This is not the case for the modified version of the Apriori algorithm, while it may give excellent results if f is small, it is no longer a viable alternative when f is large. Additionally, this optimization can be used only when the relational operator is concordant.

Work on reasoning with aggregation constraints includes [7, 12], while the idea of moving predicates for query optimization has been investigated in [8]. The monotonic properties of aggregations has been studied in [9].

9 Conclusions

Datacube queries with selections are important because in decision support environments we are often interested in knowing for which tuples in a datacube a certain condition holds. We have proposed two different ways by which we can use the selection condition internally during the computation of such queries. By making use of the selection condition within the datacube computation, we can safely prune parts of the computation and end up with a more efficient computation of the answer. Our first technique, called "specialization", uses the fact that a tuple in the datacube does not meet the given threshold to infer that all finer level aggregates cannot meet the threshold. We propose a scheme of specialization transformations on the underlying data sets, using properties of the aggregates and threshold functions.

Our second technique is called "generalization", and applies in the case where the actual value of the aggregate is not needed in the output, but used just to compare with the threshold. Generalization uses the fact that a tuple meets the given threshold to infer that all coarser level aggregates also meet the threshold. We also propose a scheme of generalization transformations.

We demonstrate the efficiency of these techniques by implementing them within the sparse datacube algorithm of Ross and Srivastava. We present a performance study using synthetic and real-world data sets. Our results indicate substantial performance improvements for queries with selective conditions.

References

[1] S. Agarwal, R. Agrawal, P. Deshpande, A. Gupta, J. Naughton, R. Ramakrishnan, and S. Sarawagi. On the computation of multidimensional aggregates. In *Proceedings of the 22nd International Conference on Very Large Databases*, Mumbai,India, 1996.

[2] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proceedings of the 20th International Conference on Very Large Databases*, Santiago, September 1994.

[3] M. Fang, N. Shivakumar, H. Garcia-Molina, R. Motwani, and J. Ullman. Computing iceberg queries efficiently. In *Proceedings of the 23rd International Conference on Very Large Databases*, New York, August 1998.

[4] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Datacube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. In *Proceedings of the IEEE Conference on Data Engineering*, pages 152–159. IEEE Computer Society, 1996.

[5] C. J. Hahn, S. G. Warren, and J. London. Edited synoptic cloud reports from ships and land stations over the globe, 1982-1991. Available from http://cdiac.esd.ornl.gov/cdiac/ndps/ndp026b.html, 1994.

[6] V. Harinarayan, A. Rajaraman, and J. Ullman. Implementing data cubes efficiently. In *Proceedings of the 1996 ACM SIGMOD Conference on Management of Data*, Montreal,Canada, 1996. Association for Computing Machinery.

[7] A. Levy and I. Mumick. Reasoning with aggregation constraints. In *Proceedings of the Conference on Extending Database Technology, (EDBT-96)*, Avignon, France, 1996.

[8] A. Levy, I. Mumick, and Y. Sagiv. Query optimization by predicate movearound. In *Proceedings of the 20th International Conference on Very Large Databases*, Santiago, September 1994.

[9] K. Ross and Y. Sagiv. Monotonic aggregation in deductive databases. *Journal of Computer and System Sciences*, 54(1):79–97, 1997.

[10] K. Ross and D. Srivastava. Fast computation of sparse datacubes. In *Proceedings of the 23rd International Conference on Very Large Databases*, Athens,Greece, 1997.

[11] K. Ross and D. Srivastava. Fast computation of sparse datacubes. *Full Version*, 1998.

[12] K. Ross, D. Srivastava, P. Stuckey, and S. Sudarshan. Foundation of aggregation constraints. *Theoretical Computer Science*, 193(1):149–179, 1998.

[13] Y. Zhou, P. Deshpande, and J. Naughton. An array-based algorithm for simultaneous multidimensional aggregates. In *Proceedings of the 1997 ACM SIGMOD Conference on Management of Data*, pages 159–170, Tucson, Arizona, May 1997. Association for Computing Machinery.