Optimizing Selections over Datacubes

Kenneth A. Ross[*]
Columbia University
kar@cs.columbia.edu

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

Abstract

Datacube queries compute aggregates over database relations at a variety of granularities. Often one wants only datacube output tuples whose aggregate value satisfies a certain condition, such as exceeding a given threshold. We develop algorithms for processing a datacube query using the selection condition internally during the computation. Thus, 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. 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. 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.

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 $(B_1, B_2 \dots B_k)$ 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 $\vec{B}_i$.

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

Figure 1. Selection over a projected data cube

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

Figure 2. Selection over a data cube

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 Figure 1 which we call a "projected datacube" because the aggregate is projected out of the result. In some cases we may need to know the exact value of sales too (Figure 2).

We can naively execute these queries as follows. Compute the datacube using any of the existing datacube algorithms [5, 1, 14] 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" [4]) then it seems that we may be wasting a lot of time computing aggregates that do not qualify.

Depending upon the aggregate function and the relational operator in the predicate, there are certain optimiza-

*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.

tions 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 datacube 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 that we are computing the projected datacube 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. □

In this paper we examine how we would make use of generalization and specialization to carry out selections over data cubes efficiently. We extend the `Memory-Cube` and `Partitioned-Cube` algorithms which can deal efficiently with sparse data [14]. `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 projected datacubes and datacubes while generalization applies only for projected datacubes. 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.

We further demonstrate that projected datacubes are, in general, easier to compute than datacubes that include the aggregate result. In particular, it is possible to compute projected datacubes with a `HAVING` clause on the median in a *distributive* fashion. In contrast, there is no known distributive algorithm for computing the median in the datacube output. Hence users can get efficient answers to queries such as "Find combinations of model, year and color for which the median sale is greater than \$10,000."

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 [7]. 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 $Q(\vec{B_i})$ 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.

Definition 1.1: We define a cuboid consisting of tuples with exactly n non-ALL dimension attribute values to be a n-cuboid. □

Using the categorizations of aggregate functions introduced in [5], we focus on the case of *distributive functions*.

Definition 1.2: Let $\cup$ denote multiset union, and let S, S' be multisets. An aggregate function g is *distributive* if there exists a binary function h such that for every nonempty S and S', $g(S \cup S') = h(g(S), g(S'))$. □

Examples of such aggregate functions include SUM, COUNT, MIN and MAX. A property that is important when considering potential optimizations of aggregate computation is the idempotent property.

Definition 1.3: Let S range over multisets over a domain D, and let the aggregate function g be a mapping from

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

Figure 3. Distributive Aggregate Functions and their properties

multisets over D to D. We say g is *idempotent* if for every nonempty multiset S, $g(\{g(S)\}) = g(S)$. □

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 $h = \bigcap$ and $h = \bigcup$ 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), $G(t_1)$ *op* $G(t_2)$ holds. 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.

As we shall see each of generalization and specialization behaves in complementary ways for concordant and discordant operators respectively.

2 Data Cube Algorithms

In this section we give a brief survey of existing datacube algorithms.

2.1 Array Based Algorithms

The array-based algorithm proposed by Gray et al. [5] 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 the base relation 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. [19] 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 Memory-Cube and Partitioned-Cube

`Partitioned-Cube` and `Memory-Cube` [14] are efficient algorithms for computing datacubes which work particularly well for sparse data.

`Partitioned-Cube` (Figure 4) 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 5) 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 $\begin{pmatrix} k \\ \lceil k/2 \rceil \end{pmatrix}$ 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.

2.3 Bottom-Up Cube

Bottom-Up Cube(BUC) [3] is a recent algorithm developed independently by researchers at the University of Wisconsin. It is similar to a version of `Partitioned-Cube` that never calls `Memory-Cube`. Instead the BUC algorithm builds the cuboid with a single attribute, followed by a cuboid with two attributes and so on. The intuition is that even though work is duplicated while computing these cuboids, this additional work is more than offset by the computation which can be pruned when a tuple does not meet minimum support. The `HAVING` clause considered is of the form `HAVING COUNT(*) >= X` which is equivalent to computing tuples having minimum support in an association rules context [2].

BUC prunes computations which are specializations of a tuple but does not exploit generalizations. Since BUC seeks to trade partitioning costs against that of aggregation, it may not be a suitable algorithm if the aggregation operation is expensive.

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.

3.1 MIN and MAX

For MIN and MAX if 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. Note that if the relational operator is discordant we cannot make use of this optimization.

`Memory-Cube` 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 datacubes and projected datacubes.

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

3.2 Other Aggregates

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

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 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 (See Definition 1.2). 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 3.1: We 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. □

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:

$< a_1, b_1, c_1, d_1 : v_1 >$
$< a_1, b_1, c_1, d_4 : v_2 >$
$< a_1, b_1, c_2, d_3 : v_3 >$ $< \sqcup, b_1, c_1, d_1 : h(\{v_1, v_6\}) >$
$< a_1, b_1, c_2, d_4 : v_4 >$ $< \sqcup, b_2, c_1, d_1 : v_5 >$
$< a_1, b_2, c_1, d_1 : v_5 >$ $< \sqcup, b_2, c_1, d_2 : v_{10} >$

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 $R_1, \ldots, R_n$;
  /* $n \leq$ card(B_j) and $n \leq$ number of buffers in memory */
  for i = 1 . . . 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 4. Algorithm `Partitioned-Cube`

Algorithm `Memory-Cube`$(R, \{B_1, \ldots, B_m\}, A, G)$
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 5. Algorithm `Memory-Cube`

$< a_2, b_1, c_1, d_1 : v_6 > \Rightarrow < \sqcup, b_1, c_2, d_3 : h(\{v_3, v_8\}) >$
$< a_2, b_1, c_1, d_4 : v_7 > \quad < \sqcup, b_1, c_1, d_4 : h(\{v_2, v_7\}) >$
$< a_2, b_1, c_2, d_3 : v_8 > \quad < \sqcup, b_1, c_2, d_4 : h(\{v_4, v_9\}) >$
$< a_2, b_1, c_2, d_4 : v_9 >$
$< a_2, b_2, c_1, d_2 : v_{10} >$

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.

In a similar fashion to 1-Specialization we can define n-Specialization which is described in detail in [16]. The basic idea is to specialize on n-cuboids for larger values of n. We believe that this optimization is not as useful as 1-Specialization which is why we do not discuss it here.

3.3 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 projected datacubes 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. 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 first 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 $c_1^* < c_1$.

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 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 set valued aggregates such as $\bigcap$ over sets of large cardinalities, this would be the case whether we used a bit mapped representation or a list representation.

4.1 Discordant Operators

Our treatment of generalization has so far 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 a projected datacube with the aggregate function and the relational operator pair being discordant (e.g. MAX and $\leq$). We now want to output small datacube tuples rather than large ones. This enables us to handle datacubes in addition to projected datacubes.

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.

5 Partitioned Cube

We have dealt so far only with the `Memory-Cube` algorithm in [14], 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 8 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 is not present in any tuple in a 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 8) 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.

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

For concordant operators over MIN and MAX we only need to pass on those base tuples which are themselves large. For other cases we illustrate the algorithm used by means of the following example.

Example 5.1: We are computing a datacube with 5 `CUBE BY` attributes A, B, C, D, E. As indicated in Figure 6 we choose a partitioning attribute and split the data into sets depending upon the value of this attribute. We invoke `Memory-Cube` on each set of data. In this example

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;
/* $n \leq$ 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`

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 $\{B_1, \ldots, B_m\}$;
/* 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 {
if ((G is MIN or MAX)&&(tuple is small)) { Drop t from set of tuples }
Determine position l of first $\sqcup$ 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. Algorithm `Modified Memory-Cube` for Specialization and Generalization

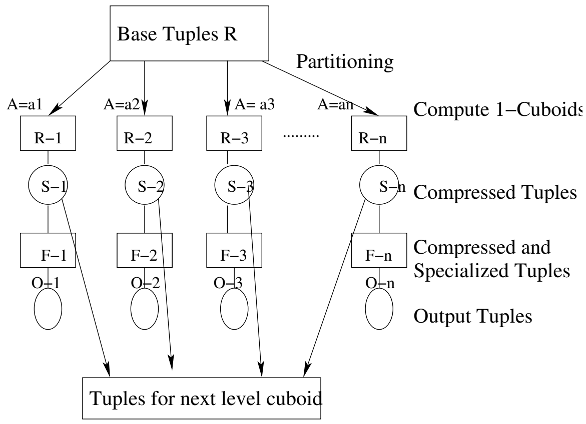

Figure 8. Modified Partitioned-Cube

we partition by attribute A (Figure 8) to get sets of tuples $R_1, R_2 \ldots R_n$. We compress each R_i by collapsing tuples with the same CUBE BY attributes to obtain S_i's. We modify `Partitioned-Cube` here to compute all the large tuples of the 1-cuboids which can be done while we are partitioning the data itself. Since we have this information we can now carry out specialization on each set of S_i tuples *before* invoking `Memory-Cube`. Corresponding to each S_i we now have a set of tuples, F_i. We carry out our `Memory-Cube` computation on each F_i to obtain the sub datacube results O_i. In `Partitioned-Cube` the tuples output in each computation of `Memory-Cube` are used for computing cuboids not containing the partitioning attribute. This would be equivalent to using the O_i tuples which we can't do since this contains just those datacube tuples which meet the selection condition. The F_i tuples cannot be used since they contain the $\sqcup$'s produced by specialization. Using the S_i tuples is superior to using the uncompressed R_i's only if the gain due to collapsing tuples offsets the I/O cost of writing and then rereading the S_i tuples. □

Generalization is unaffected by whether the relational operator is concordant or discordant. For discordant operators we can carry out specialization only for MAX and MIN.

6 Holistic Aggregates

For holistic aggregates like the median or other quantiles, data cube computation algorithms cannot compute a datacube tuple from its parents, so each tuple has to be computed directly from the base data. In `Memory-Cube` this means that we have to omit the preprocessing step where we replace the base data with the finest level aggregates.

Surprisingly, for holistic projected-datacube queries of the kind shown in Figure 9, we can evaluate the query in

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

Figure 9. Example Holistic Query

a fashion similar to that of distributive aggregates. Instead of maintaining the value of the median in each data cube tuple we maintain two counts: c_1 which corresponds to the `count` of the tuple and c_2 which corresponds to the number of contributing tuples for which the measure attribute satisfies the clause `v relop threshold`. We can use these two aggregates to check if any data cube tuple satisfies the `HAVING` clause by computing c_2/c_1. If we are computing the median and the relational operator is $>$, we have to check if this number is less than 0.5 . In this fashion we can compute any quantile both for $>$ and $<$. We can similarly compute the median for $=$ by maintaining three counters instead of two.

Lemma 6.1: We can check if a quantile is greater than less than a threshold by maintaining two counts and computing these counts distributively. □

This also works for `Partitioned-Cube`: we change to the count representation during the first partitioning step. This is a big win over the computing the exact median since we cannot use `Partitioned-Cube` for holistic aggregates when the data exceeds memory capacity. As demonstrated in Section 8, distributive computation is much more efficient than a nondistributive one. However, we can't apply specialization and generalization in the holistic case.

7 Multiple Selection Conditions

In our previous examples there has always been a single selection condition in the `HAVING` clause. In the most general case we can have any combination of aggregation functions connected by any combination of boolean operators. The aggregation functions may be distributive, algebraic or holistic and the boolean operator may be `OR` or `AND`.

Let us consider the case where our selection conditions are of the form `distributiveAggregate relop threshold` (where relop is a concordant operator) and are connected only by `AND`'s (the expression is in CNF). In this case, we can carry out 1-Specialization whenever any of the aggregation functions is not satisfied. In the compressed tuple we will have to maintain the all the aggregates being checked. Thus, we cannot exploit this optimization if we have a mixture of distributive and holistic aggregates in the `HAVING` clause.

If we consider the case where we have multiple selections with aggregate functions in CNF, we still have to decide the order in which to evaluate different conditions. This is particularly important if the aggregation operation is expensive. This problem has been studied extensively [8, 9] in the database literature. A major issue tackled by recent papers is when expensive predicates should be *pushed* or *pulled* before a join. Since we do not have joins in our scenario our approach will be different. The approach first proposed in [18] orders predicates based on their *rank*, where the rank of a predicate is based on its cost and its selectivity:
$rank = selectivity * (cost - per - application)$ and $selectivity = \frac{cardinality-of-output}{cardinality-of-input}$. Predicates are then evaluated in order of ascending rank. We cannot use this scheme without modification for data cubes since the selectivity of a predicate differs for each cuboid and we are processing more than a single cuboid in a run.

Example 7.1: We are computing the aggregation function $\bigcap$ over a set which we represent as a bitmap in each individual tuple. Let the predicate be $\bigcap(bitmap) \supset \emptyset$. If the set cardinality is n and in each bitmap an average of $n/2$ bits are set, the selectivity of the predicate and hence its rank will be high. If we aggregate k tuples for each tuple in the coarser cuboid, the expected number of bits set in a tuple will be $n*(1/2)^k$. This number is considerably smaller than the $n/2$ we started with leading to a very different rank. □

To handle this we propose the following modification to the predicate ordering algorithm if the selection consists of distributive range predicates connected conjunctively. If cuboids $a_1, a_2, \ldots, a_n$ are being computed in a particular run, we compute the global selectivity as $\frac{\sum_{i=1}^{n}(selectivity_i * size_i)}{\sum_{i=1}^{n}(size_i)}$ where $size_i$ denotes the size in tuples of cuboid a_i. We can estimate the size of a cuboid using techniques described in [17]. Estimating the selectivity of a predicate is normally carried out over the base data by using histograms. In Example 7.1 we use a bitmap representation for a set and the cost of each $\bigcap$ operation is a constant. If we used a linked list representation instead of a bitmap representation the cost of each operation would not be value independent.

We outline how to estimate the selectivity of a predicate for different aggregate functions.

1. COUNT: If we assume that the data is uniformly distributed we can estimate the average count of a tuple by computing the size of the cuboid, taking its reciprocal and multiplying by the size of the base data (n). We use this value to obtain an estimate of 0 or 1 for predicates of the form $COUNT(x) > t$ or $COUNT(x) < t$.

2. MAX/MIN: Calculate the fraction ($frac = base/n$) of the base dataset for which the measure attribute is greater (less) than the threshold. We can use the estimate of the COUNT to determine the selectivity of the predicate by computing $1 - (1 - base/n)^{count}$ for $MAX(x) > t$.

3. SUM: If we assume that the measure attribute takes values drawn from a random distribution with a mean of μ and a standard deviation of σ, the distribution of the sum of $count$ such random variables will be given by $(count * \mu, count * \sigma)$. We can now use Chebyshev's theorem which states that the probability of any random variable X will assume a value within k standard deviations of the mean is at least $1-1/k^2$. We can compute how many standard deviations away from the threshold the mean is, and then use this theorem to estimate the selectivity.

4. $\bigcap$: Assume the base dataset has an average of a bits set out of a possible total of b for a bitmap based set representation. The expected number of bits set is given by $b * (a/b)^{count}$. We can compute a similar estimate for $\bigcup$.

We use the selectivities in conjunction with the execution costs to order predicate evaluations. We can summarize the applicability of our optimizations for multiple selection conditions in the following table.

Connectives	Generalization	Specialization
Disjunctive	Yes	No
Conjunctive Concordant	No	Yes
Conjunctive Discordant	No	No

8 Experimental Results

We have implemented the modified version of the `Memory-Cube` algorithm presented in Figure 7. We implemented 1-Specialization. We implemented generalization only for tuples in the same path so that performance is not worse than the unmodified version. 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 potentially 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 $\binom{n}{\lceil n/2 \rceil}$ 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 [15]. 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 10 (a) and (b) we investigate the effect 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 COUNT. Figure 10 (a) has uniformly distributed data while Figure 10 (b) 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.

In Figure 10 (c) and (d), 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. Figure 10 (c) shows how the modified algorithm compares to the original one. In Figure 10 (d), 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 (e) and (f) 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 8.1: We also experiment upon real-world data on cloud coverage [6]. The data used corresponds to measurements of the amount of cloud coverage throughout the globe over a period of one month, September 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 (a) and (b) 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. Figure 11 (b) indicates that for restrictive selections it is better to have many compression steps, but there is little to be gained in doing so when that is not the case. □

We carry out experiments to show the effect of generalization when we have a relatively expensive aggregation operation. An example of such a query follows.

Example 8.2: 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." □

As we can see in Figure 11(c), generalization leads to great improvements in performance. 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. 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

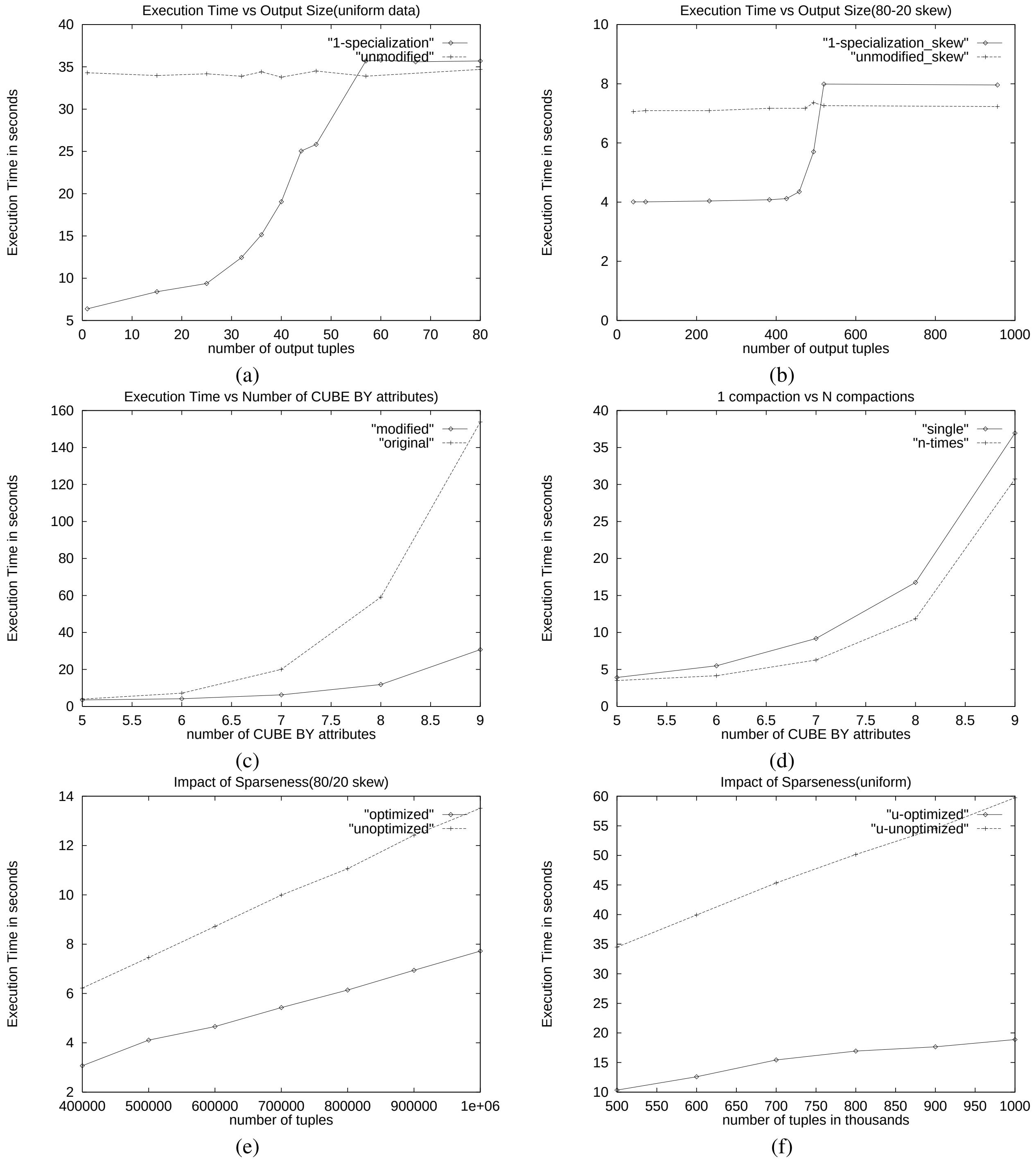

Figure 10. Experimental Results 1

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

We carry out experimentation with the same parameters but this time using a discordant relational operator (we output those datacube tuples with fewer than a certain number of elements in a set) in conjunction with our aggregate function. In this case, we can handle both datacubes and projected datacubes. This is because we do not need to

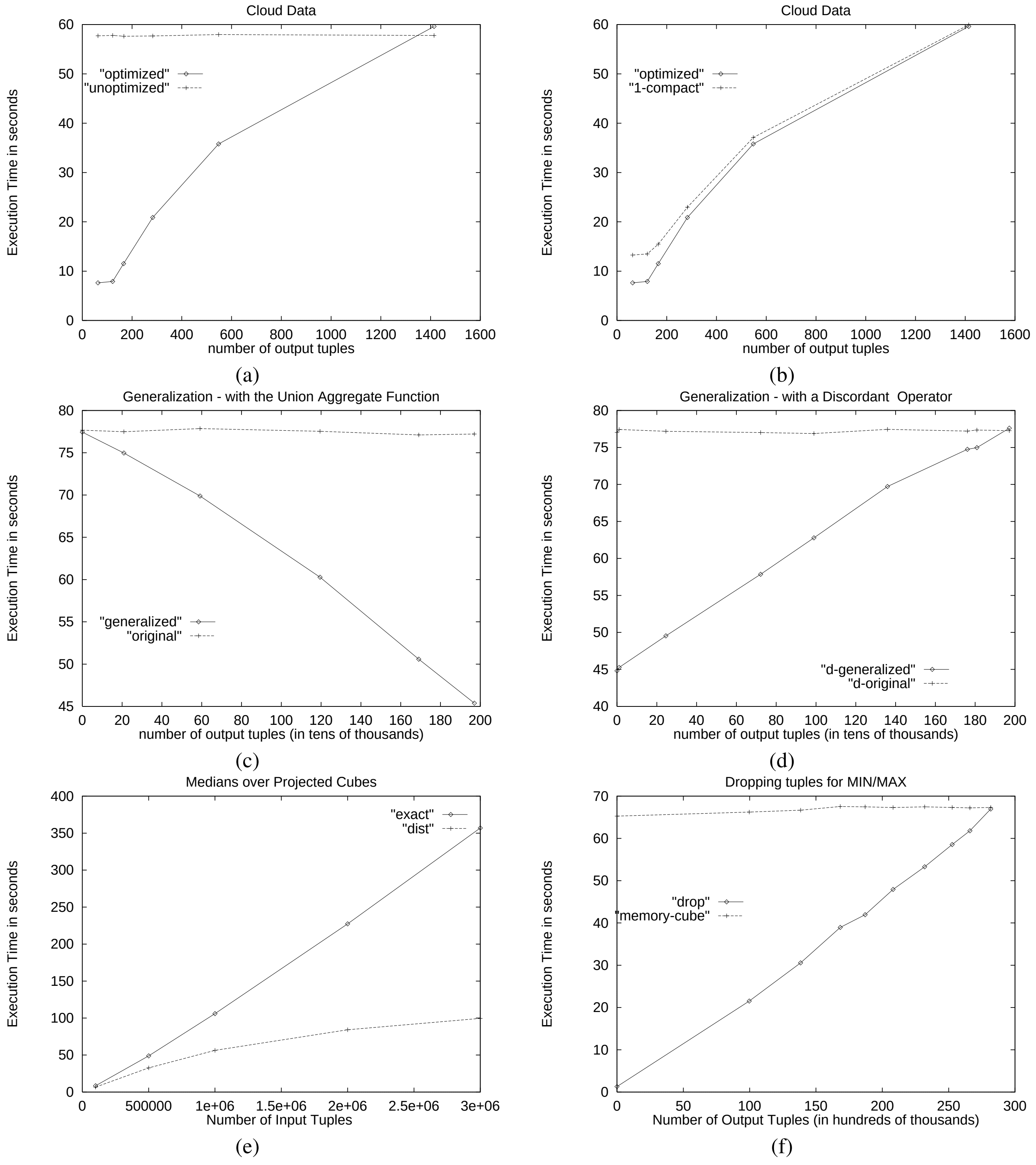

Figure 11. Experimental Results 2

output the tuples we generalize. The results in Figure 11(d) show that there is a substantial improvement in performance when the number of output tuples is small.

We compare the difference between the distributive and holistic versions of `Memory-Cube` for projected datacube computation. The left hand graph of Figure 11 (e) indicates that the improvement in performance for the distributive case increases with the size of the input relation. In the holistic case the median is computed in a quicksort like algorithm that recursively processes one of the two partitions.

This algorithm is linear in the average case.

In our final experiment, we show the improvement in performance when the aggregation operation is MIN or MAX and the relational operator is concordant. This is the case where we can drop all small tuples. In Figure 11(f), 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.

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 for projected datacubes when the relational operator is concordant. When the relational operator is discordant it holds for datacubes too. 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. Additionally, we show that for projected datacubes the median is easier to compute. The class of projected datacube queries is an important class of queries which hasn't been considered separately before.

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.

Extensions of our optimizations for range queries and hierarchies are discussed in [16]. Other algorithms for computation of the datacube are described in [1]. Work on reasoning with aggregation constraints is described in [10, 12]. The idea of moving predicates for query optimization has been investigated in [11]. The monotonic properties of aggregations has been studied in [13].

References

[1] S. Agarwal et al. On the computation of multidimensional aggregates. In *Proceedings of the 22nd International VLDB Conference*, Bombay, 1996.

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

[3] K. Beyer and R. Ramakrishnan. Bottom-up computation of sparse and iceberg cubes. In *Proceedings of the 1999 ACM SIGMOD Conference*.

[4] M. Fang et al. Computing iceberg queries efficiently. In *Proceedings of the 24th International VLDB Conference*, New York, August 1998.

[5] J. Gray et al. Data Cube: A relational aggregation operator generalizing group-by, cross-tab and sub-total. *Data Mining and Knowledge Discovery*, 1(1), 1998.

[6] C. J. Hahn et al. 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.

[7] V. Harinarayan et al. Implementing data cubes efficiently. In *Proceedings of the 1996 ACM SIGMOD Conference*.

[8] J. Hellerstein and M. Stonebraker. Predicate migration: Optimizing queries with expensive predicates. In *Proceedings of the 1993 ACM SIGMOD COnference*.

[9] A. Kemper et al. Optimizing disjunctive queries with expensive predicates. In *Proceedings of the 1994 ACM SIGMOD Conference*.

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

[11] A. Levy, I. Mumick, and Y. Sagiv. Query optimization by predicate movearound. In *Proceedings of the 20th International VLDB Conference*, Santiago, 1994.

[12] K. Ross et al. Foundation of aggregation constraints. *Theoretical Computer Science*, 193(1):149–179, 1998.

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

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

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

[16] K. Ross and K. Zaman. Optimizing selections over data cubes. Technical Report CUCS-011-98, Department of Computer Science, Columbia University, 1998.

[17] A. Shukla et al. Storage estimation for multidimensional aggregates in the presence of hierarchies. In *Proceedings of the 22nd International VLDB Conference*, Bombay, 1996.

[18] W. Smith. Various optimizers for single-stage production. *Naval Research Logistics Quarterly*, 3:59–66, 1956.

[19] Y. Zhou et al. An array-based algorithm for simultaneous multidimensional aggregates. In *Proceedings of the 1997 ACM SIGMOD Conference*.