# Bottom-Up Computation of Sparse and Iceberg CUBEs

Kevin Beyer
Computer Sciences Department
University of Wisconsin – Madison
beyer@cs.wisc.edu

Raghu Ramakrishnan
Computer Sciences Department
University of Wisconsin – Madison
raghu@cs.wisc.edu

## Abstract

We introduce the Iceberg-CUBE problem as a reformulation of the datacube (CUBE) problem. The Iceberg-CUBE problem is to compute only those group-by partitions with an aggregate value (e.g., count) above some minimum support threshold. The result of Iceberg-CUBE can be used (1) to answer group-by queries with a clause such as HAVING COUNT(*) >= X, where X is greater than the threshold, (2) for mining multidimensional association rules, and (3) to complement existing strategies for identifying interesting subsets of the CUBE for precomputation.

We present a new algorithm (BUC) for Iceberg-CUBE computation. BUC builds the CUBE bottom-up; i.e., it builds the CUBE by starting from a group-by on a single attribute, then a group-by on a pair of attributes, then a group-by on three attributes, and so on. This is the opposite of all techniques proposed earlier for computing the CUBE, and has an important practical advantage: BUC avoids computing the larger group-bys that do not meet minimum support. The pruning in BUC is similar to the pruning in the Apriori algorithm for association rules, except that BUC trades some pruning for locality of reference and reduced memory requirements. BUC uses the same pruning strategy when computing sparse, complete CUBEs.

We present a thorough performance evaluation over a broad range of workloads. Our evaluation demonstrates that (in contrast to earlier assumptions) minimizing the aggregations or the number of sorts is not the most important aspect of the sparse CUBE problem. The pruning in BUC, combined with an efficient sort method, enables BUC to outperform all previous algorithms for sparse CUBEs, even for computing entire CUBEs, and to dramatically improve Iceberg-CUBE computation.

## 1 Introduction

Decision support systems frequently precompute many aggregates to improve the response time of aggregation queries. The datacube (CUBE) operator [6] generalizes the standard GROUP-BY operator to compute aggregates for every combination of GROUP BY attributes. For example, consider a relation Transaction(Product, Store, Customer, Sales). The sum of Sales over the CUBE of Product, Store, and Customer produces the sum of Sales for the entire relation (i.e., no GROUP BY), for each Product (GROUP BY Product), for each Store, for each Customer, for each pair: (Product, Store), (Product, Customer), and (Store, Customer), and finally for each (Product, Store, Customer) combination. In OLAP parlance, the grouping attributes are called *dimensions*, the attributes that are aggregated are called *measures*, and one particular GROUP BY (e.g., Product, Store) in a CUBE computation is sometimes called a *cuboid* or simply a *group-by*.

The basic CUBE problem is to compute all of the aggregates as efficiently as possible. Simultaneously computing the aggregates offers the opportunity to share partitioning and aggregation costs between various group-bys. The chief difficulty is that the CUBE problem is exponential in the number of dimensions: for $d$ dimensions, $2^d$ group-bys are computed. In addition, the size of each group-by depends upon the cardinality of its dimensions. If every store sold every product, then the (Product, Store) group-by would have $|\text{Product}| \times |\text{Store}|$ result tuples. However, as the number of dimensions or the cardinalities increase, the product of the cardinalities grossly exceeds the (fixed) size of the input relation for many of the group-bys. Even in our small example, if the data comes from a large department store, it is highly unlikely that a given customer purchased even 5% of the products, or shopped in more than 1% of the stores.

When the product of the cardinalities for a group-by is large relative to the number of tuples that actually appear in the result, we say the group-by is *sparse*. When the number of sparse group-bys is large relative to the number of total number of group-bys, we say the CUBE is sparse. As is well-recognized, given the large result size for the entire CUBE, especially on sparse datasets, it is important to identify (and precompute) subsets of interest.

This paper addresses CUBE computation over sparse CUBEs and makes the following contributions:

1. We introduce a variant of the CUBE problem, called Iceberg-CUBE in the spirit of [5], that allows us to selectively compute only those *partitions* that satisfy a user-specified aggregate condition (similar to SQL's

HAVING clause). The Iceberg-CUBE formulation can be viewed as a new *dynamic* subset selection strategy, complementing earlier approaches that statically identify group-bys (rather than partitions) to be precomputed [10, 8, 7, 3, 18]. The Iceberg-CUBE formulation also identifies precisely the subset of the CUBE that is required to mine multidimensional association rules in the framework described in [11].

2. We present a simple and efficient algorithm, called BUC, for the Iceberg-CUBE problem. BUC proceeds bottom-up (it builds the CUBE by starting from a group-by on a single attribute, then a group-by on a pair of attributes, and so on) in contrast to all prior CUBE algorithms. This feature enables the Iceberg aggregate selections to be pushed into the CUBE computation easily. We also outline an extension to compute precisely the CUBE subset identified for precomputation by the PBS algorithm [18].

3. We present an extensive performance evaluation based upon a complete implementation of BUC. We focus our comparison on the MemoryCube algorithm presented in [14], which was shown to be superior to earlier algorithms for computing sparse CUBEs. We show that BUC outperforms MemoryCube on a wide range of synthetic and real datasets, even when computing the full CUBE. Further, since MemoryCube does not exploit aggregate selections, BUC outperforms it significantly for Iceberg-CUBE computation.

The rest of this paper is organized as follows: Section 2 discusses sparse CUBEs, and we argue that the basic CUBE problem is not appropriate for high dimensions. In Section 3, we define the Iceberg-CUBE problem. Section 4 describes the previous work on computing the CUBE. Most previous work does not scale to sparse, high-dimensional CUBEs [14], and none can directly take advantage of the minimum support threshold in Iceberg-CUBE. In Section 5, we present a new algorithm for CUBE and Iceberg-CUBE computation called Bottom-Up Cube (BUC). Our performance analysis is described in Section 6. Our evaluation demonstrates that (in contrast to earlier assumptions) minimizing the aggregations or the number of sorts is not the most important aspect of the sparse CUBE problem. The pruning in BUC, combined with an efficient sort method, turns out to be the key to the performance gains, even for full CUBEs. We present our conclusions in Section 7.

## 2 Motivation

Ross and Srivastava computed the full CUBE on a real nine-dimensional dataset containing weather conditions at various weather stations on land for September 1985 [14, 9]. The dataset had 1,015,367 tuples (∼39MB). The CUBE on this dataset produces 210,343,580 tuples (∼8GB)—more than 200 times the input size!

We break down the output size distribution for this dataset in Figure 1. In both the graphs, the x-axis is the number of output tuples in a group-by relative to the input size. Figure 1a shows the number of group-bys that are smaller than a given relative size, and Figure 1b shows the amount of space (relative to the input size) needed to build all the group-bys that are less than a given size. The graphs convey a number of interesting points:

- About 20% of the group-bys performed very little aggregation (all of the group-bys with relative size of nearly 1). These group-bys are simply a projection of the input. (If the data were uniform and uncorrelated, then nearly 60% of the group-bys would perform little to no aggregation.) About 60% of the group-bys aggregated an average of no more than 4 tuples.
- Computing all group-bys that aggregate at least two input tuples on average (group-bys with relative size of 0.5) requires about 50 times the input size versus the 200 times for the full CUBE. Requiring an average of 10 tuples to be aggregated (i.e., 0.001% of the input tuples) shrinks the space requirement to 5 times the input.
- As noted in [14], simply writing the entire output to disk can take an inordinate amount of time, and can easily dominate the cost of computing the CUBE. By selecting the group-bys (or group-by partitions) that perform at least a little aggregation, the output time can be significantly reduced.

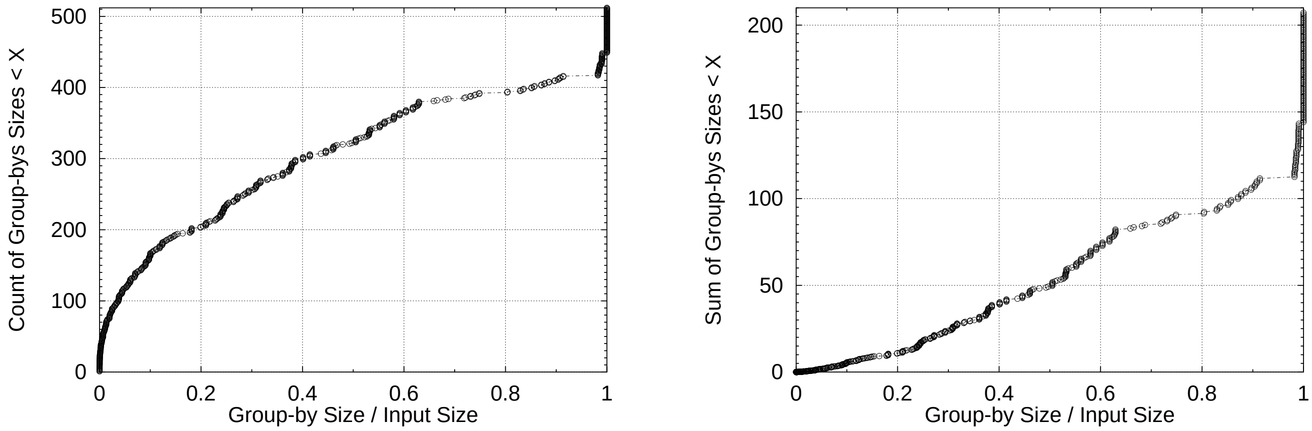
Because the space requirements for large CUBEs are so high, often we cannot realistically compute the full CUBE. We need a way to choose what portion of the CUBE to compute. A number of researchers have proposed computing a subset of the group-bys instead of the entire datacube [10, 8, 7, 3, 18]. The algorithms choose the group-bys to compute based on the available disk space, the expected size of the group-by, and the expected benefit of precomputing the aggregate. Under certain common conditions, [18] presents an algorithm called PBS that chooses to materialize the smallest group-bys (i.e., the fewest result tuples, which is the same as the group-bys that perform the most aggregation).

Choosing a subset of the group-bys to compute is certainly a reasonable way to reduce the output of full CUBE computation, and we show in Section 6 that BUC works particularly well with PBS. However, statically choosing a subset of the group-bys is not the only way to reduce the output; the next section describes a new alternative.

## 3 The Iceberg-CUBE Problem

The **Iceberg-CUBE** problem is to compute all group-by *partitions* for every combination of the grouping attributes that satisfy an aggregate selection condition, as in the HAVING clause of an SQL query. For concreteness, we will discuss the condition that a partition contain at least $N$ tuples; other aggregate selections can be handled as well, as described in Section 5.2. The parameter $N$ is called the *minimum support* of a partition, or *minsup* for short. Iceberg-CUBE with minsup of 1 is exactly the same as the original CUBE problem. Iceberg-CUBE with a minsup of $N$ is easily expressed in SQL with the CUBE BY clause:

```
SELECT  A,B,C,COUNT(*),SUM(X)
FROM    R
CUBE BY A,B,C
HAVING  COUNT(*) >= N
```




(a) Group-by Size

(b) Total Space Blow-up

Figure 1: Space requirements for the weather dataset

The precomputed result of an Iceberg-CUBE can be used to answer GROUP BY queries on any combination of the dimensions (A, B, and C, in this case) that also contains a HAVING COUNT(*) >= M, where M >= N. The count does not need to be stored if all queries (explicitly or implicitly) contain HAVING COUNT(*) >= N. That is, the user is simply not interested in small group-by partitions.

Iceberg-CUBE is not an entirely new problem, although this is the first time that it has been proposed for CUBE queries. The mining of *multi-dimensional association rules* (MDAR) uses a notion of minimum support that is equivalent to Iceberg-CUBE [11]. MDARs have the form $X \Rightarrow Y$. $X$ is called the *body* of the rule, and $Y$ is called the *head*. $X$ and $Y$ are sets of conjunctive predicates. For the purposes of this discussion, each predicate essentially has the form *attribute = value*, although they are a bit more general in [11]. The *support* for a rule $X \Rightarrow Y$ in a relation $R$ is the probability that a tuple of $R$ contains both $X$ and $Y$. In other words, the support is the count of tuples that are true for both $X$ and $Y$ divided by the number of tuples in $R$. The *confidence* of a rule is the probability that a tuple of $R$ contains $Y$ given that the tuple contains $X$ (i.e., $\text{Prob}(Y|X) = \text{count}(X \text{ and } Y)\ /\ \text{count}(X)$). The goal of mining MDARs is to find rules with a support of at least the *minsup* and a confidence of at least *minconf*.

The first step in mining MDARs, as with traditional association rules[1] [2], is to find the rules that meet minimum support. This is precisely the Iceberg-CUBE problem where $N = |R| * \text{minsup}_{\text{MDAR}}$. The results from Iceberg-CUBE can then be combined to find the rules that meet minimum confidence.

A precomputed Iceberg-CUBE is also useful for computing **iceberg queries** [5]. An iceberg query computes a single group-by and eliminates all tuples with an aggregate value below some threshold. For example:

```
SELECT   A,B,C,COUNT(*)
FROM     R
GROUP BY A,B,C
HAVING   COUNT(*) >= N
```

So the Iceberg-CUBE is essentially an iceberg query over a CUBE. Although we do not discuss this point further, the techniques of [5] can be used to refine BUC in some cases with large partitions.

## 4 Previous CUBE Algorithms

The CUBE was introduced in [6], and they outlined some useful properties for CUBE computation: (1) minimize data movement, (2) map string dimension attributes [and other types] to integers between zero and the cardinality of the attribute, and (3) use parallelism. Mapping dimensions to integers reduces space requirements (i.e., no long strings), eliminates expensive type interpretation in the CUBE code, and packing the domain between zero and the cardinality allows the dimensions to be used as array subscripts.

Three types of aggregate functions were identified. Consider aggregating a set of tuples $T$. Let $\{S_i | i = 1 \ldots n\}$ be a any complete set of disjoint subsets of $T$ such that $\bigcup_i S_i = T$, and $\bigcap_i S_i = \{\}$.

- An aggregate function $F$ is **distributive** if there is a function $G$ such that $F(T) = G(\{F(S_i)|i = 1 \ldots n\})$. SUM, MIN, and MAX are distributive with $G = F$. COUNT is distributive with $G = \text{SUM}$.

- An aggregate function $F$ is **algebraic** if there is a $M$-tuple valued function $G$ and a function $H$ such that $F(T) = H(\{G(S_i)|i = 1 \ldots n\})$, and $M$ is constant regardless of $|T|$ and $n$. All distributive functions are algebraic, as are Average, standard deviation, MaxN, and MinN. For Average, $G$ produces the sum and count, and $H$ divides the result.

[1] Note that the traditional association rule problem can be mapped to the MDAR problem be making each item an attribute (i.e., a dimension) with a value of zero or one (and using predicates with a value of 1). Unfortunately, BUC is ill-suited for this problem because (1) the dimensionality is extremely high (50,000 items is not uncommon), (2) the cardinality of every dimension is extremely low (only two values), and (3) the dimensions tend to be highly skewed.

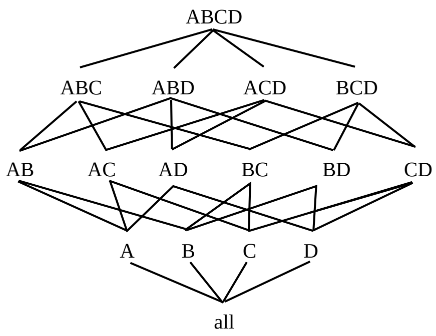



Figure 2: 4-Dimensional Lattice

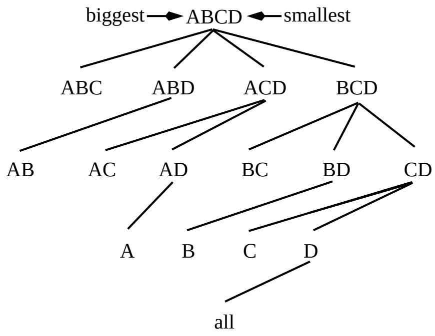



Figure 3: Sample Processing Tree

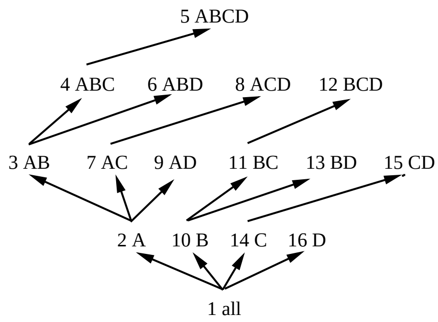



Figure 4: BUC Processing Tree

- An aggregate function $F$ is **holistic** if it not algebraic. For example, Median and Rank are holistic.

Algebraic functions have the key property that more detailed aggregates (i.e., more dimensions) can be used to compute less detailed aggregates. This property induces a partial ordering (i.e., a lattice) on all of the group-bys of the CUBE. A group-by is called a **child** of some **parent** group-by if the parent can be used to compute the child (and no other group-by is between the parent and the child). Figure 2 depicts a sample lattice where A, B, C, and D are dimensions, nodes represent group-bys, and the arcs show the parent-child relationship.

All of the algorithms use this lattice view of the problem. The goal is to take advantage of as much commonality as possible between a parent and child. In general, the algorithms recognize that group-bys with common attributes can share partitions, sorts, or partial sorts. The algorithms differ on exactly how they exploit this commonality. The following subsections provide details on each of the previous proposed algorithms.

### 4.1 PipeSort, PipeHash, and Overlap

- PipeSort, proposed in [16, 1], searches the space of possible sort orders for the best set of sorts that convert the CUBE lattice into a processing tree (e.g., Figure 3). The search attempts to minimize the number of sorts, while at the same time it seeks to compute a group-by from its smallest parent. The authors recognized that many of the sorts will have a common prefix, so they optimized the sorting procedures to take advantage of partial sorts. Except when sorting, PipeSort uses at most $d+1$ memory cells for each of the simultaneous aggregates, where $d$ is the number of dimensions.

  [14] points out that PipeSort performs *at least* $\binom{d}{\lceil d/2 \rceil}$ sorts, where $d$ is the number of dimensions. When computing sparse CUBEs, many of the intermediate results that are sorted cannot fit in memory, so external sort will be used.

- PipeHash, also proposed in [16, 1], computes a group-by from its smallest parent in the lattice. For example, if the attributes in Figure 2 are ordered such that $A \leq B \leq C \leq D$, and group-by size estimates are proportional to the product of the cardinalities (as is the case with attribute independence assumptions), then the processing tree produced is shown in Figure 3. PipeHash uses a hash table for every simultaneously computed group-by. If all of the hash tables cannot fit in memory, PipeHash partitions the data on some attribute and processes each partition independently. PipeHash suffers from two problems. First, it does not overlap as much computation as PipeSort because PipeSort computes multiple group-bys with one sort where as PipeHash must re-hash the data for every group-by. Second, PipeHash requires a significant amount of memory to store the hash tables for the group-bys even after partitioning.

- Overlap, proposed in [4, 1], aims to overlap as much sorting as possible by computing a group-by from a parent with the maximum sort-order overlap. The algorithm recognizes that if a group-by shares a prefix with its parent, then the parent consists of a number of partitions, one for each value of the prefix. For example, the $ABC$ group-by has $|A|$ partitions that can be sorted independently on $C$ to produce the $AC$ sort order.

  Overlap chooses a sort order for the root of the processing tree, and then all subsequent sorts are some suffix of this order. Once the processing tree is formed, Overlap tries to fit as many partitions in memory as possible to avoid writing intermediate results. If enough memory is available, Overlap can make one pass over a sorted input file.

  With the same assumptions on the lattice as we used for PipeHash, Overlap also produces the processing tree in Figure 3. Actually, under these assumptions, choosing the smallest parent minimizes the partition sizes and the number of partitions, which is ideal for Overlap. For example, $BD$ could be computed from $ABD$, or $BCD$, but $BCD$ produces partitions that are proportional to $|B|$ while $ABD$ produces partitions proportional to $|BD|$. $CD$ could be computed from either $ACD$ or $BCD$, either would produce the same partition sizes, but the number of partitions in $BCD$ is proportional to $|B|$ while $ACD$ is proportional to $|A|$, so $BCD$ produces fewer partitions. Thus, picking the smallest parent is quite similar to picking for the most overlap.

  [14] argues that the Overlap on sparse CUBEs also produces a large amount of I/O by sorting intermediate

results (at least quadratic in the number of dimensions).

Because these algorithms can generate significant I/O for intermediate results or require large amounts of main memory, we do not consider them further.

### 4.2 ArrayCube

An array-based algorithm for computing the CUBE was described in [19]; we call this ArrayCube. The algorithm is very similar to Overlap, except that it uses in-memory arrays to store the partitions and to avoid sorting. The algorithm expects its input to be in an array-based file-structure, but it can be extend to use relational input and output with little or no performance penalty. The input order for ArrayCube is slightly different from the input order for Overlap because the array file-structure is "chunked". A chunk of a n-dimensional array is a n-dimensional subarray that corresponds loosely to a page. The array is stored in units of chunks to provide multidimensional clustering; chunking is not useful for computing the cube.

This algorithm is unique in two regards. First, it requires no tuple comparisons, only array indexing. Second, the array file-structure offers compression as well as indexing. The other methods could benefit from compression as well, and with the amount of redundancy in the CUBE output, we expect high compression ratios. The algorithm is most effective when the product of the cardinalities of the dimensions is moderate. Unfortunately, if the data is too sparse, this method becomes infeasible because the in-memory arrays become too large to fit in main-memory.

### 4.3 PartitionedCube and MemoryCube

PartitionedCube and MemoryCube, described in [14], are designed to work together. PartitionedCube partitions the data on some attribute into memory-sized units (similar to PipeHash but for a different reason), and MemoryCube computes the CUBE on each in-memory partition. The key observation they made is that buffering intermediate group-bys in memory — something all the previous algorithms do (except PipeSort, which does not buffer anything) — requires too much memory for large sparse CUBEs. Instead, they chose to buffer the partitioned *input* data for repeated in-memory sorts, similar to PipeSort, although they present a new algorithm that picks the minimum number of sorts (which is exactly the largest tier in the lattice = $\binom{d}{\lceil d/2 \rceil}$). Once each partition for the first partitioning attribute is processed (which is half of the CUBE), the input is repartitioned on the next attribute.

Since this algorithm is designed for sparse CUBEs, we consider this to be the best existing algorithm for the CUBE problems discussed in this paper (sparse CUBEs and Iceberg-CUBEs). Therefore, we only compare BUC to this algorithm.

In recent independent work, researchers at Columbia University also found that pushing the HAVING clause into CUBE computation is beneficial [15]. They describe how to take advantage of HAVING predicates in PartitionedCube/ MemoryCube.

```
Procedure BottomUpCube(input, dim)
Inputs:
  input: The relation to aggregate.
  dim: The starting dimension for this iteration.
Globals:
  constant numDims: The total number of dimensions.
  constant cardinality[numDims]: The cardinality of
      each dimension.
  constant minsup: The minimum number of tuples in a
      partition for it to be output.
  outputRec: The current output record.
  dataCount[numDims]: Stores the size of each partition.
      dataCount[i] is a list of integers of size
      cardinality[i].
Outputs:
  One record that is the aggregation of input.
  Recursively, outputs CUBE(dim, ..., numDims) on
      input (with minimum support).
Method:
 1: Aggregate(input); // Places result in outputRec
 2: if input.count() == 1 then // Optimization
       WriteAncestors(input[0], dim); return;
 3: write outputRec;
 4: for d = dim ; d < numDims ; d++ do
 5:   let C = cardinality[d];
 6:   Partition(input, d, C, dataCount[d]);
 7:   let k = 0;
 8:   for i = 0 ; i < C ; i++ do // For each partition
 9:     let c = dataCount[d][i]
10:     if c >= minsup then // The BUC stops here
11:       outputRec.dim[d] = input[k].dim[d];
12:       BottomUpCube(input[k ... k+c], d+1);
13:     end if
14:     k += c;
15:   end for
16:   outputRec.dim[d] = ALL;
17: end for
```

Figure 5: Algorithm BottomUpCube (BUC)

## 5 Algorithm Bottom-Up Cube

We propose a new algorithm called BottomUpCube (BUC) for sparse CUBE and Iceberg-CUBE computation. The idea in BUC is to combine the I/O efficiency of PartitionedCube/ MemoryCube, but to take advantage of minimum support pruning like Apriori [2]. BUC was inspired by the algorithms in [14], particularly PartitionedCube. BUC is similar to a version of algorithm PartitionedCube that never calls MemoryCube.

To achieve pruning, BUC proceeds from the bottom of the lattice (i.e., the smallest / most aggregated group-bys), and works its way up towards the larger, less aggregated group-bys. All of the previous algorithms compute in the opposite direction. Since parent group-bys are used to compute child group-bys, the algorithms cannot avoid computing the parents.

The details of BUC are in Figure 5. The first step is to aggregate the entire input (line 1) and write the result (line 3). (Line 2 is an optimization that we discuss

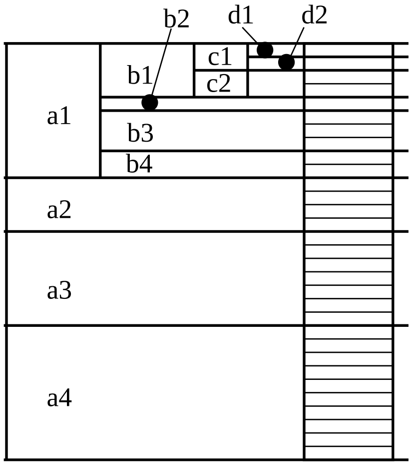



Figure 6: BUC Partitioning

below. For now we ignore that line.) For each dimension $d$ between dim and numDims, the input is partitioned on dimension $d$ (line 6). On return from Partition(), dataCount contains the number of records for each distinct value of the $d$-th dimension. Line 8 iterates through the partitions (i.e., each distinct value). If the partition meets minimum support (which is always true for full CUBEs because minsup is one), the partition becomes the input relation in the next recursive call to BottomUpCube, which computes the (Iceberg) CUBE on the partition for dimensions $d+1$ to numDims. Upon return from the recursive call, we continue with the next partition of dimension $d$. Once all the partitions are processed, we repeat the whole process for the next dimension.

Figure 4 shows the BUC processing tree (i.e., how it covers the lattice). The numbers indicate the order in which BUC visits the group-bys. Figure 6 illustrates how the input is partitioned during the first four calls to BottomUpCube (assuming minsup is one). First BUC produces the empty group-by. Next, it partitions on dimension $A$, producing partitions $a1$ to $a4$, and then it recurses on partition $a1$. The $a1$ partition is aggregated and produces a single tuple for the $A$ group-by. Next, it partitions the $a1$ partition on dimension $B$. It recurses on the $\langle a1, b1\rangle$ partition and writes a $\langle a1, b1\rangle$ tuple for the $AB$ group-by. Similarly for $\langle a1, b1, c1\rangle$, and then $\langle a1, b1, c1, d1\rangle$, but this time it does not enter the loop at line 4. Instead it simply returns only to recurse again on the $\langle a1, b1, c1, d2\rangle$ partition. BUC then returns twice and then recurses on the $\langle a1, b1, c2\rangle$ partition. When this is complete, it partitions the $\langle a1, b1\rangle$ partition on $D$ to produce the $\langle a1, b1, D\rangle$ aggregates.

Once the $\langle a1, b1\rangle$ partition is completely processed, BUC proceeds to $\langle a1, b2\rangle$. This partition consists of a single tuple. If we ignore line 2 for the moment, then BUC will recurse for $\langle a1, b2, c\rangle$, $\langle a1, b2, c, d\rangle$, and $\langle a1, b2, d\rangle$ aggregating and partitioning a single tuple. While the result is correct, it's a fruitless exercise. We add line 2 to the algorithm so that the aggregates on this tuple are computed just once, and then the result tuple is written to each of its ancestor group-bys in WriteAncestors() by simply setting the dimension values appropriately (in this case: $\langle a1, b2\rangle$, $\langle a1, b2, c\rangle$, $\langle a1, b2, c, d\rangle$, and $\langle a1, b2, d\rangle$).

The elimination of the aggregation and partitioning of single tuple partitions is a key factor in the success of BUC on sparse CUBEs because many partitions have a single tuple. Figure 1 shows that 20% of the *group-bys* consisted almost entirely of single tuple partitions! On one generated dataset, this optimization improved the computation by more than 40%.

### 5.1 Iceberg-CUBE with BUC

To this point, we considered only full CUBE computation (i.e., minsup = 1). The optimization for a single tuple partition is similar to how BUC processes an Iceberg-CUBE. When a small partition is found, instead of writing for all of the group-bys, BUC simply skips the partition (line 10) and does not consider any of the partition's ancestors. The pruning is correct because the partition sizes are always decreasing when BUC recurses, and therefore none of the ancestors can have minimum support.

The pruning is similar to the pruning in Apriori [2]. The major difference between Apriori (appropriately adapted for Iceberg-CUBE computation) and BUC is that Apriori processes the lattice breadth first instead of depth first. In our example, Apriori would compute the $A$, $B$, $C$, and $D$ group-bys in one pass of the input. A candidate set would be created from all the partitions that meet minimum support. For example, if $\langle a3\rangle$ and $\langle b2\rangle$ made minimum support, then $\langle a3, b2\rangle$ would be added to the candidate set. The input would be read a second time and the candidate pairs would be pruned for minimum support. Now the remaining pairs are combined to form the candidate triples. If $\langle a3, b2\rangle$, $\langle a3, c5\rangle$, and $\langle b2, c5\rangle$ all made minimum support, then $\langle a3, b2, c5\rangle$ is a candidate in the third pass.

Apriori can prune group-bys one step earlier than BUC. For example, if the partition $\langle a3\rangle$ met minimum support but $\langle b2\rangle$ did not, BUC will consider the $\langle a3, b2\rangle$ partition but Apriori will not. The problem with using Apriori is the candidate set usually cannot fit in memory because little pruning is expected during first few passes of the input. BUC trades pruning for locality of reference and reduced memory requirements.

We implemented the modified Apriori algorithm and compared it to BUC. As expected, when the output size is large, Apriori needs too much memory and performs terribly. We then compared the algorithms with extremely skewed input: all duplicates. Apriori needed very little memory, but it still performed significantly worse than BUC.

### 5.2 Additional Pruning Functions

As described in [13], functions other than count can be used for pruning. The pruning function must be monotonic.[2] For any two sets (of tuples, in our case) $S$ and $T$ such that $S \subseteq T$, a function $f$ is monotonicly decreasing if $f(S) <= f(T)$. (If $f$ is monotonically increasing, the inequality in the prune expression must be reversed. We are actually interested in monotonically decreasing boolean functions, true < false: if $f(T)$ is false then $f(S)$ is false for any subset $S$ of $T$). Count is monotonicly decreasing, since the count of a subset is

[2]The functions are called anti-monotonic in [13].

certainly larger than the count of the superset. Min and max are monotonic, as is the sum of positive numbers. Also, if $f$ and $g$ are two monotonically decreasing boolean functions, then $f \wedge g$ and $f \vee g$ are both monotonically decreasing boolean functions.

Average, however, is not monotonic. [13] describes how some non-monotonic functions can still be used to prune by replacing it with a conservative monotonic function. For example, if the average of positive numbers must be above $X$, we can prune with the sum above $X$ and the minimum below $X$. If the sum of both positive and negative numbers must be above $X$, then we can prune with the sum of only the positive numbers above $X$.

These additional pruning functions can be quite useful in practice. For example, we can prune aggregates with little sales (HAVING SUM(sales) >= S), or those aggregates that do not include any young or old people (HAVING MIN(age) <= young OR MAX(age) >= old). These functions can not only be used for aggregate precomputation, but also for mining multi-dimensional association rules.

To use additional pruning predicates in BUC, add:

**if** CanPrune() **then return**;

after the call to Aggregate() and before line 2 (where CanPrune() is the pruning predicate).

The class of predicates that BUC can use for pruning is by no means the only useful HAVING predicates. For example, consider HAVING COUNT(*) < X. In this case, the user wants the groups with little aggregation, which occur towards the top of the lattice. Now, the previous CUBE algorithms can prune their computation, but BUC cannot. When computing large CUBEs however, this predicate will not reduce the output size much, so the output time will dominate the cost of computation. Note that even if a predicate cannot be used for pruning in BUC, it can still be used for reducing the CUBE output.

### 5.3 Partitioning

The majority of the time in BUC is spent partitioning the data, so optimizing Partition() is important. When 'input' does not fit in main memory, the data must be partitioned to disk. This can be done with hash-partitioning, followed by a partitioning within each bucket (which hopefully now fits in main memory), or external-sort can be used. When performing an external partitioning, the aggregation step (at line 1 of BUC) can be combined with the partitioning.

Once 'input' fits in main memory, which hopefully occurs after partitioning on the first dimension, we can use in-memory sorting or hashing to partition the data. Note that once 'input' fits into memory on some call to BUC, for all recursive calls from that point, 'input' will fit in memory. Therefore, an implementation of BUC should have BUC-External and BUC-Internal, where processing starts with BUC-External and switches to BUC-Internal when the input fits in memory. Our current implementation does not perform external partitioning.

Our implementation uses a linear sorting method called CountingSort [17]. CountingSort excels at sorting large lists that have a sort key of moderate cardinality (i.e., many duplicates). The algorithm requires that the sort key be an integer value between zero and its cardinality, and that the cardinality is known in advance. When an attribute of a relation meets this property, we say the attribute is **packed**.

Our implementation of BUC assumes that all of the dimensions are packed. This same assumption is used in [19]. This assumption is reasonable because strings and other types are usually mapped into integers to save space and eliminate type interpretation. Also, in a star-schema [12], the dimension values are often system generated integers that are used as keys to the dimension tables. If the dimensions are not packed in the input, they can be packed when the input is first read by creating a hashed symbol table for each dimension as described in [6], and the mapping can be reversed when tuples are output (or a simple pre- and post-processing pass can be used).

We found the use of CountingSort to be an important optimization to BUC. For example, when sorting one million records with widely varied key cardinality and skew, QuickSort ran between 3 and 10 times slower than CountingSort. CountingSort is faster not only because it sorts in O(N) time, but also because it does not perform any key comparisons. When using CountingSort, we do not even need comparisons to find the partition boundaries, because the counts computed in CountingSort can be saved for use in BUC.

CountingSort cannot be easily used in other CUBE algorithms because they perform sorts on several dimensions (composite keys). (However, CountingSort is a *stable* sort. This means that a sort on a composite key can be achieved by calling CountingSort for each key attribute in reverse order.)

Unfortunately, the advantage of CountingSort over QuickSort slowly degrades as the ratio of the number of tuples to the cardinality decreases. When the number of tuples is significantly less than the cardinality of the partitioning dimension, QuickSort is faster than CountingSort. BUC produces many sorts with small partitions, so our implementation switches to QuickSort when the number of tuples in the partition is less than 1/4 the cardinality. (Also, QuickSort switches to InsertionSort when the number of tuples is less than 12.)

#### 5.3.1 Dimension Ordering

The performance of BUC is sensitive to the ordering of the dimensions. The goal of BUC is to prune as early as possible; i.e., BUC wants to find partitions that do not meet minimum support (or the other pruning criteria, or that only have one tuple). For best performance, the most **discriminating** dimensions should be used first. Remember that the first dimension is used in half of the group-bys, so it has the most potential for savings.

How discriminating a dimensions is depends upon several factors:

- **Cardinality:** The cardinality of a dimension (the number of distinct values) determines the number of partitions that are created. The higher the cardinality, the smaller the partitions, and therefore the closer BUC is to pruning some computation.

- **Skew:** The skew in a dimension affects the size of each partition. Skewed dimensions also have a smaller *effective* cardinality when used as the second or later partitioning attribute because it is likely that infrequent values will not appear in some partition. The more uniform a dimension (i.e., the less skew), the better it is for pruning.

- **Correlation:** If a dimension is correlated with an earlier partitioning dimension, then its effective cardinality is reduced. Correlation decreases pruning.

We experimented with two heuristics for ordering the dimensions. The first heuristic is to order the dimensions based on decreasing cardinality. The second heuristic is to order the dimensions based on increasing maximum number of duplicates. When the data is not skewed, the two heuristics are equivalent. Section 6.5 gives a synthetic example where the second heuristic out-performs the first, but on we found little difference on real datasets.

### 5.4 Collapsing Duplicates

In the presence of high skew or correlation, a few group-by values can account for most of the tuples in the input. For example, when computing CUBE($A, B, C, D$), the partition $\langle a3, b2, c7, d4\rangle$ could contain 90% of the original input. When this occurs, it is worthwhile to collapse the duplicate partitioning values to a singe tuple (using aggregation).

If skewed data is expected to be common, we suggest changing the top-level call to BUC to collapse duplicates. This can be done by making a copy of the BUC procedure called BUC-Dedup and starting the computation at BUC-Dedup. Then, replace the Partition() function at line 6 of BUC-Dedup with a function that not only partitions the data on dimension $d$ but collapses all of the duplicates on dimensions $d \ldots numDims - 1$.

Collapsing duplicates has three disadvantages. First, if the data has few duplicates, there is a modest extra cost of trying to eliminate them. Second, the 'input' to BUC is now the result of aggregation, and if a large number of aggregates are computed on a small number of measure fields, less tuples will fit in memory. Third, and most importantly, holistic aggregate functions can no longer be computed because the Aggregate() function at line 1 receives partially aggregated data.

#### 5.4.1 Switching To ArrayCube

BUC can perform poorly when each recursive partitioning does not significantly reduce the input size. For example, consider computing the CUBE on a relation that is 64 times the size of main memory, with ten dimensions that each have two distinct values. In this case, BUC needs to partition on six attributes before the input fits in memory. The CUBE is actually dense, not sparse, so previous algorithms, in particular ArrayCube [19], will perform better than BUC.

However, this effect can occur even when computing a sparse CUBE. Consider the previous example again, but say one dimension (call it $A$) has a cardinality of 100,000. The result CUBE is much more sparse, but half the group-bys are just as dense as they were before (i.e., the ones that do not use $A$). If $A$ is used as the first partitioning attribute, then BUC efficiently computes all the group-bys on $A$, but performs poorly on the remaining group-bys.

We suggest switching from BUC to ArrayCube whenever the product of the remaining cardinalities is reasonably small but the number of input tuples is still large. The switch can occur at any point, but a logical choice would be to use ArrayCube on the last dimensions during to topmost call to BUC. For example, when computing CUBE($A, B, C, D$), if $A$ is large but $B$, $C$, and $D$ are small, use BUC to compute all the group-bys that use $A$, and use ArrayCube to compute the remaining group-bys starting at group-by $BCD$ (refer to Figure 4).

When switching to ArrayCube, we can no longer prune any of the computation, although the output can still be pruned. However, all of the group-bys must be relatively small (and therefore aggregate many records) to fit in memory, which implies that most of those group-bys are likely to be computed in any case. Another downside to using ArrayCube is that it does not support holistic aggregate functions.

This optimization is at odds with collapsing duplicates. If we reconsider the first example in this section (10 dimensions each with cardinality of 2), collapsing duplicates on the original input will reduce the relation to at most $2^{10} = 1024$ tuples. More work needs to be done to determine which strategy is best.

### 5.5 Using BUC with PBS

BUC works well with the PickBySize (PBS) algorithm for choosing group-bys to precompute [18]. PBS chooses the group-bys with the smallest expected (output) size (i.e., the group-bys that perform the most aggregation). PBS chooses entire group-bys, not partitions like Iceberg-CUBE.

Since PBS chooses by the size of the group-by, if some node is chosen, then all of its children must have been chosen. For example, if $ABC$ is selected, then $AB$, $AC$, $BC$, $A$, $B$, $C$, and the empty group-by must all be selected. Selecting group-bys this way produces a *frontier* in the lattice. Every group-by below the frontier is selected. Another interesting point from [18] is that the BPUS algorithm in [10] also tends to pick group-bys towards the bottom of the lattice.

BUC can easily be extended to compute only selected aggregates. When PBS is used, BUC can stop when it hits the frontier. This means that BUC can compute only the selected group-bys, an no others.

### 5.6 Minimizing Aggregations

Since BUC proceeds bottom-up, it does not take advantage of algebraic functions to reduce the aggregation costs. On the positive side, BUC can efficiently compute holistic functions, unlike most of the previous algorithms (MemoryCube can efficiently compute holistic functions as well.) In our experiments we found that aggregation costs were only small percent of the processing costs. Even when computing 16 aggregates on a full CUBE, less than 1/4 of the processing time was attributed to aggregation (see Section 6.3). However, BUC *can* take advantage of algebraic functions by aggregating the results of the recursive call from any one iteration of the loop at line 8. This complicates the algorithm a bit, and when we implemented it, BUC actually ran more slowly!

### 5.7 Memory Requirements

BUC relies on a significant, but reasonable, amount of working memory. As mentioned previously, BUC tries to fit a partition of tuples in memory as soon as possible. Say the partition has $N$ tuples, and each tuple requires $T$ bytes. Our implementation uses pointers to the tuples, and CountingSort requires a second set of pointers for temporary use. Let $C_1 \ldots C_d$ be the cardinality of each dimension, and $C_{max}$ be the maximum cardinality. CountingSort uses $C_{max}$ counters, and BUC uses $\sum C_i$ counters. If the counters and pointers are each four bytes, the total memory requirement in bytes for BUC is:

$$N(T+8) + 4\sum_{i=1}^{d} C_i + 4C_{max}$$

The memory requirements can be reduced by switching to QuickSort. When using QuickSort, the second set of pointers and all of the counters are not needed. Also, the counters in BUC are not necessary to use CountingSort. If the counters are not used, BUC will have to search for the partition boundaries.

## 6 Performance Analysis

We received an executable for MemoryCube from Prof. Ken Ross. Receiving his executable not only saved us time, but also allowed us to do a fair comparison with code that they optimized. Their implementation included a number of performance improvements that are not described in [14], but they are expected to appear in the forthcoming journal version of the paper. It is sufficient to say that they report this version is three times faster than the original version used in [14].

We implemented BUC for main memory only (no external partitioning). The implementation of MemoryCube had the same restriction because it did not come with PartitionedCube. This is not a problem because PartitionedCube / MemoryCube and BUC have equivalent I/O performance. We ran our tests on a 300MHz Sun Ultra 10 Workstation with 256MB of RAM. We measured the elapsed time, but since the implementation of MemoryCube reads text files, we did not count the time to read the file, and we did not output any results. The input to the programs was all integer data, and all the dimensions were packed between 0 and their cardinality. We estimated the I/O time based upon the number of tuples in the input and output, assuming no partitioning was required. Our system had a sequential I/O rate of 5MB/sec, so we used that figure in estimating I/O times.

### 6.1 Full CUBE Computation

The first experiment compares BUC with MemoryCube for full CUBE computation. We randomly generated one million tuples. We varied the number of dimensions (group-by attributes) from 2 to 11 (11 was a compiled limit for MemoryCube). We repeated the experiment with three different cardinalities: 10, 100 and 1000 (all dimensions had the same cardinality). The results are shown in Figure 7. The graphs show a number of interesting points:

- The CUBE gets more sparse as the number of dimensions increases and as the cardinality increases. As a result, the output gets extremely large, so the output time dominates the cost of computation. The same result was observed in [14].
- With a cardinality of 10, BUC and MemoryCube have comparable performance. BUC begins to improve on MemoryCube at 10 dimensions.
- As cardinality increases, the time for MemoryCube marginally increases or stays about the same. BUC, however, dramatically improves as the cardinality increases. With 11 dimensions and a cardinality of 1000, BUC is over 4 times faster than MemoryCube. The reason that BUC improves so much is that the CUBE is getting significantly more sparse as cardinality increases,[3] so BUC can stop partitioning and aggregating and simply write the answer to all ancestors. (Even though we do not output the result, we still make the correct number of calls the output function.)

### 6.2 Iceberg-CUBE Computation

This experiment explores the effect of minimum support pruning in BUC. The input is the 11 dimensional data from the previous section. The cardinality is again 10, 100, or 1000. The minimum support was 1 (i.e., full CUBE), 2, 10, or 100. (Remember that the minimum support is the minimum number of records for a group-by partition to be output. A minimum support of 10 is 0.001% of the data.)

The results are shown in Figure 8 and Figure 9. A minimum support of 10 decreases the time for BUC significantly: 37%, 75%, and 85% for cardinalities 10, 100, and 1000 respectively. In addition, MemoryCube now takes twice as long as BUC for a cardinality of 10. The other major effect is the I/O time no longer dominates the computation, even with a minimum support of 2.

### 6.3 Additional Aggregates

BUC does not try to share the computation of aggregates between parent and child group-bys, only the partitioning costs. To verify that partitioning is the major expense, not aggregation, we computed a full CUBE on 10 dimensional data with a cardinality of 100 and one million tuples. The results are shown in Figure 10. Computing one aggregate accounts for less than 7% of the total cost. Computing 16 aggregates is still only 23% of the total cost. If any algorithm sacrifices partitioning to try and overlap the aggregate computations, these percentages will only decrease. This suggests that optimizing the partitioning is the right approach for sparse CUBEs.

### 6.4 PBS

We ran an experiment to determine how BUC and MemoryCube compare when used with PBS. We generated a 10 dimensional dataset with a cardinality of 100. Since all the dimensions are the same, every group-by with the same number of dimensions has the same estimated size

[3] Even though the CUBE gets more sparse as dimensionality increases, the problem is still exponentially harder, so BUC can never get faster with added dimensions.

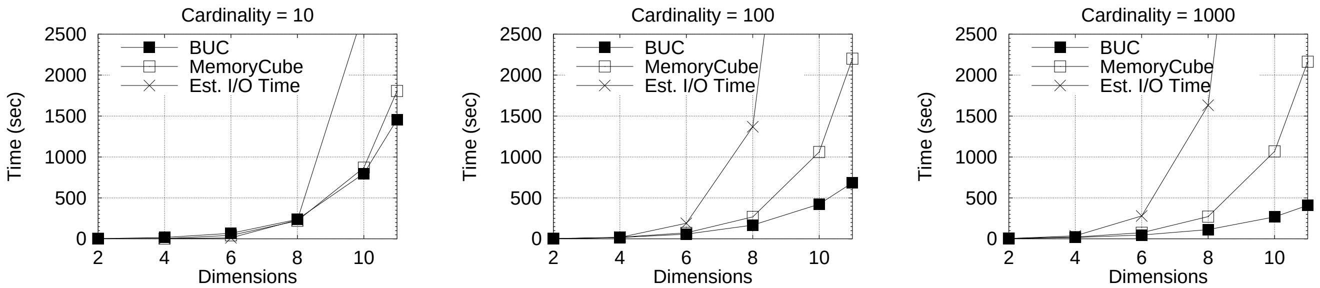



Figure 7: Full CUBE computation

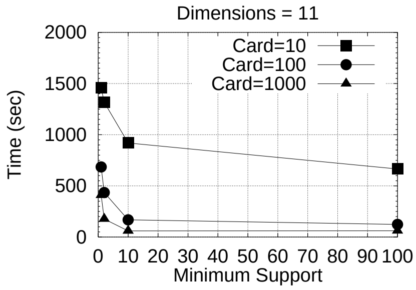



Figure 8: BUC with min. support

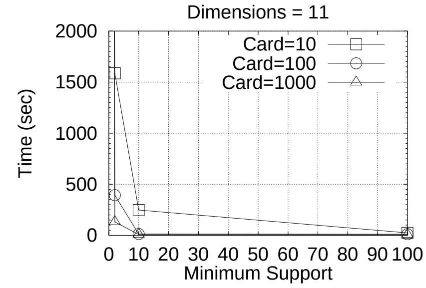



Figure 9: Est. I/O with min. support

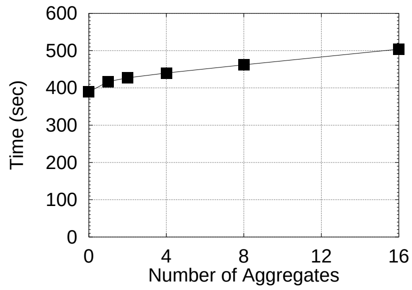



Figure 10: Additional aggregates

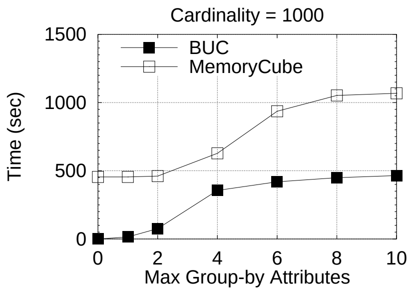



Figure 11: Limited dimensions

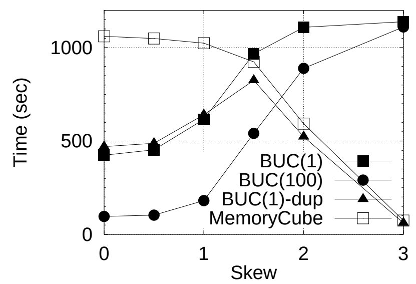



Figure 12: Increasing skew

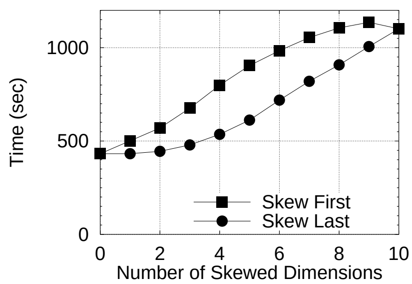



Figure 13: Skewed dimension order

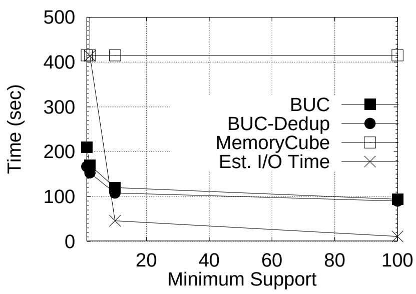



Figure 14: Weather data

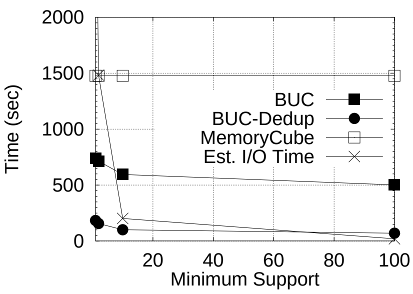



Figure 15: Mail-order sales data

(e.g., $|AB| = |AC| = |BC|$). The implementation of MemoryCube had an option to limit the maximum number of attributes used in a group-by (i.e., the maximum number of non-ALL values), and we implemented the same feature in BUC. We varied the maximum number of grouping dimensions from 0 to 10. Figure 11 shows that MemoryCube and BUC followed a similar trend, but that BUC was always significantly faster. The performance of MemoryCube did not change between 0 and 2 dimensions, probably because their implementation is not optimized for this case.

### 6.5 Skew

As mentioned previously, BUC is sensitive to skew in the data. In all of the previous experiments, the data was generated uniformly (i.e., no skew). We ran an experiment on 10 dimensional data with cardinality of 100 that varied the skew simultaneously in all dimensions. We used a Zipf distribution to generate the data. Zipf uses a parameter $\alpha$ to determine the degree of skew. When $\alpha = 0$, the data is uniform, and as $\alpha$ increases, the skew increases rapidly: at $\alpha = 3$, the most frequent value occurred in about 83% of the tuples.

The results in Figure 12 show that the performance of BUC does degrade as skew increases. BUC with a minimum support of 100 even converges on BUC for full CUBE. The performance of MemoryCube, however, improved with skew because the implementation collapses duplicate group-by values. We added the duplicate collapsing code to BUC as described in Section 5.4. This version is called BUC-Dedup in the graph. With this modification, BUC degraded until the deduplication compensated for the loss of pruning. At which point, BUC and MemoryCube have similar performance.

We ran another experiment were we varied the number of skew dimensions, each with the same cardinality. Figure 13 shows that placing the skewed dimensions last in the dimension ordering is significantly better than placing the skewed dimensions first.

### 6.6 Weather Data

Figure 14 shows the time for MemoryCube and BUC on a real nine-dimensional dataset containing weather conditions at various weather stations on land for September 1985 [9]. The dataset contained 1,015,367 tuples. The attributes were ordered by cardinality: station-id (7037), longitude (352), solar-altitude (179), latitude (152), present-weather (101), day (30), weather-change-code (10), hour (8), and brightness (2). Many of the attributes were highly skewed, and some of the attributes were significantly correlated (e.g., only one station was at one (latitude, longitude)).

This experiment shows that BUC is effective on real data, even with high skew and correlation. BUC is 2 times faster than MemoryCube for full CUBE computation, and 3.5 times faster when minimum support is 10. The graph also shows that a minimum support of just 2 tuples significantly reduces the I/O cost (4.3 times faster). With a minimum support of 10, the I/O costs drop drastically (39 times faster than full CUBE). We also ran BUC with the code to collapse duplicates. For full CUBE, this version of BUC ran in 167 seconds, which is a 20% improvement.

### 6.7 Mail-order Data

We ran MemoryCube and BUC on a second real dataset. This data is sales data from a mail-order clothing company. We limited the dataset to two million tuples to keep the relation in memory. The dataset has ten dimensions: the first three digits of the customer's zip code (920), product number (793), add space in the catalog (361), order date (319), page in the catalog (212), category (40), colors (21), gender of the product (8), catalog id (2), and focus indicator (2). This dataset contains extreme correlation. The product number, page, category, colors, gender, and focus attributes are all strongly correlated. Collapsing duplicates on all of the group-bys (i.e., creating the $\langle D_1, D_2, \ldots, D_{10}\rangle$) produced less than 1.4 million distinct tuples.

The results of the experiment are depicted in Figure 15. Even with the correlation, BUC is still 2 times faster than MemoryCube for a full CUBE. With duplicate elimination, BUC becomes 8 times faster than MemoryCube, and with a minimum support of 10, BUC is 14.6 times faster!

## 7 Conclusions

We introduced the Iceberg-CUBE problem and demonstrated its viability as an alternative to static selection of group-bys. We discussed how Iceberg-CUBE relates to full CUBE computation, multi-dimensional association rules, and iceberg queries.

We presented a novel algorithm called BUC for Iceberg-CUBE and sparse CUBE computation. BUC builds the CUBE from the most aggregated group-bys to the least aggregated, which allows BUC to share partitioning costs and to prune the computation. We also described how BUC complements group-by selection algorithms like PBS. BUC can be extended to support dimension hierarchies, and it can be easily parallelized. Exactly the best way to implement these features is left for future research.

Our experiments demonstrated that BUC is significantly faster at computing full sparse CUBEs than its closest competitor, MemoryCube. For example, BUC was eight times faster than MemoryCube on one real dataset. For Iceberg-CUBE queries, our experiments also showed that BUC improves upon its own performance, with speedups of up to four times with a minimum support of ten tuples.

**Acknowledgements**

We thank Prof. Ed Robertson for his comments on a previous version of this paper. We are particularly grateful to Prof. Ken Ross for his helpful comments and for supplying his implementation of MemoryCube. We also thank the anonymous referees for their comments. This work was supported in part by ORD contract 144-ET33 and NSF research grant IIS-9802882

## References

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

[2] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. of the 20th VLDB Conf.*, pages 487–499, Santiago, Chile, Sept. 1994.

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

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

[5] M. Fang, N. Shivakumar, H. Garcia-Molina, R. Motwani, and J. D. Ullman. Computing iceberg queries efficiently. In *Proc. of the 24th VLDB Conf.*, pages 299–310, New York, New York, August 1998.

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

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

[8] H. Gupta, V. Harinarayan, A. Rajaraman, and J. D. Ullman. Index selection for OLAP. In *Proc. of the 13th ICDE*, pages 208–219, Manchester, UK, 1997.

[9] C. Hahn, S. Warren, and J. London. Edited synoptic cloud reports from ships and land stations over the globe, 1982-1991. http://cdiac.esd.ornl.gov/-cdiac/ndps/ndp026b.html, http://cdiac.esd.ornl.gov/-ftp/ndp026b/SEP85L.Z, 1994.

[10] V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing data cubes effeciently. In *Proc. of the ACM SIGMOD Conf.*, pages 205–216, 1996.

[11] M. Kamber, J. Han, and J. Y. Chiang. Metarule-guided mining of multi-dimensional association rules using data cubes. In *Proceeding of the 3rd Intl. Conf. on Knowledge Discovery and Data Mining*, Newport Beach, CA, Aug. 1997. Also Techical Report CS-TR 97-10, School of Computing Science, Simon Fraser University, May 1997.

[12] R. Kimball. *The Data Warehouse Toolkit.* John Wiley and Sons, Inc, 1996.

[13] R. T. Ng, L. V. Lakshmanan, J. Han, and A. Pang. Exploratory mining and pruning optimizations of constrained associations rules. In *Proc. of the ACM-SIGMOD Conf. on Management of Data*, pages 13–24, Seattle, WA, June 1998.

[14] K. A. Ross and D. Srivastava. Fast computation of sparse datacubes. In *Proc. of the 23rd VLDB Conf.*, pages 116–125, Athens, Greece, 1997.

[15] K. A. Ross and K. A. Zaman. Optimizing selections over data cubes. Technical Report CUCS-018-98, Columbia University, Nov 1998. http://www.cs.-columbia.edu/ library/1998.html.

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

[17] R. Sedgewick. *Algorithms in C*, chapter Chapter 8, page 112. Addison-Wesley Publishing Company, 1990.

[18] A. Shukla, P. M. Deshpande, and J. F. Naughton. Materialized view selection for multidimensional datasets. In *Proc. of the 24th VLDB Conf.*, pages 488–499, New York, New York, August 1998.

[19] Y. Zhao, P. M. Deshpande, and J. F. Naughton. An array-based algorithm for simultaneous multidimensional aggregates. In *Proc. of the ACM SIGMOD Conf.*, pages 159–170, 1997.