An Empirical Comparison of Methods for Iceberg-CUBE Construction

Leah Findlater and Howard J. Hamilton
Department of Computer Science
University of Regina, Regina, SK S4S 0A2, Canada
{findl1,hamilton}@cs.uregina.ca

Abstract

The Iceberg-Cube problem is to identify the combinations of values for a set of attributes for which a specified aggregation function yields values over a specified aggregate threshold. We implemented bottom-up and top-down methods for this problem. The bottom-up method included pruning. Results show that the top-down method, with or without pruning, was slower than the bottom-up method because of less effective pruning.

1. Introduction

Users of decision support systems often see data in the form of ***data cubes***, which are views of the data in two or more dimensions along some measure of interest [5,6]. Each cell in a cube represents the measure of interest for one combination of values. If the measure is *support* (tuple count), each cell contains the number of tuples in which that combination occurs. Given a database where each transaction represents the sale of a part from a supplier to a customer, each cell (p,s,c) in the part-supplier-customer cube represents the total number of sales of part p from supplier s to customer c [5]. From k attributes, 2^k different combinations can be defined. For the part-supplier-customer example, the eight possible combinations of attributes are: part; customer; supplier; part-customer; part-supplier; customer-supplier; part-customer-supplier; and none (no attributes). In the context of database retrieval, these combinations are called ***group-bys***.

Queries are performed on cubes to retrieve decision support information. The goal is to retrieve data in the most efficient way possible. Three means of achieving this goal are to pre-compute all cells in the cube, to pre-compute no cells, and to pre-compute some cells. For attributes $A_1,...,A_n$ with cardinalities $|A_1|,...,|A_n|$, the size of the cube is $\prod|A_i|$. Thus, the size increases exponentially with the number of attributes and linearly with the cardinalities of those attributes. To avoid the memory requirements of pre-computing the whole cube and the long query times of pre-computing none of the cube, most decision support systems pre-compute some cells.

The ***Iceberg-Cube problem*** is to pre-compute only those group-by partitions that satisfy an aggregate condition, such as a minimum support, average, min, max, or sum [1]. A ***group-by partition*** is an instance of a group-by where each attribute is restricted to a specific value. For the part-supplier group-by, one partition is <part-14, supplier-29>; with the support measure, the aggregated total is the number of tuples with these values. Queries on an "iceberg cube" retrieve a small fraction of the data in the cube, i.e., the "tip of the iceberg" [2].

Two straightforward approaches to this problem are top-down and bottom-up. The ***bottom-up approach*** starts with the smallest, most aggregated group-bys and works up to the largest, least aggregated group-bys. The ***top-down approach*** starts with the largest, least aggregated group-bys and works down to the smallest, most aggregated group-bys. For example, the bottom-up approach begins with the three separate part, supplier, and customer group-bys, while the top-down approach begins with the single part-supplier-customer group-by. Both approaches find the same set of group-bys.

In this paper, a comparison of the efficiency of bottom-up and top-down methods is provided. Section 2 presents the Iceberg-Cube problem and an example of it. Section 3 describes three approaches: bottom-up computation, top-down computation, and top-down computation with pruning. Section 4 presents results of testing the three algorithms on a student records database. Section 5 concludes the paper.

2. The Iceberg-Cube Problem

For a three-dimensional data cube and the support measure, the iceberg-cube problem may be represented as [1]:

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

where *minsup* represents the ***minimum support***, i.e., the minimum number of tuples a group-by must contain. The result of such a computation can be used to answer any queries on a combination of the attributes A,B,C that require COUNT(*) to be greater than minimum support.

Table 1 shows a relation with four attributes ABCD of varying cardinalities. The cardinality of A is 5 because A has 5 distinct values. The condition is that a combination of attribute values must appear in at least three tuples (*minsup* is 3). The output from the algorithm is shown in Table 2.

A	B	C	D
a1	b2	c1	d1
a2	b1	c1	d2
a2	b2	c2	d2
a3	b2	c2	d1
a3	b3	c1	d1
a4	b3	c3	d2
a5	b2	c2	d1

Table 1: Sample Relation

Combination	Count
b2	4
b2-c2	3
c1	3
c2	3
d1	4
d2	3

Table 2: Group-bys with Adequate Support

3. Approach

We consider three algorithms for computing Iceberg Cubes. All methods yield identical output. The output is the combinations of values for attributes that meet the specified condition, as well as the aggregate value for each. Figure 1 shows a 4-dimensional lattice that represents all combinations of four attributes and the relationships between these combinations. We must determine for each combination whether or not it has a value or values that satisfy the minimum support requirement.

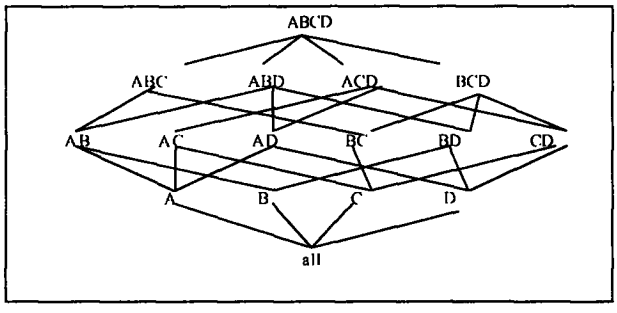

Figure 1: The 4-Dimensional Lattice for ABCD [1]

3.1 Bottom-Up Computation

The Bottom-Up Computation (BUC) algorithm is given in Figure 2 [1,3]. This algorithm computes the cube beginning with the smallest, most aggregated group-bys and recursively works up to the largest, least aggregated group-bys. The processing order can be seen in Figure 3, where BUC begins processing at the leaves of the tree and recursively works its way upward. If a group-by does not satisfy the minimum support condition, then the algorithm does not recurse to calculate the next largest group-by.

The BUC algorithm begins by taking the entire input and aggregating it. No specific aggregate function is described in [1]. For our experiments, we performed an

```
Algorithm BUC(D, nextAttr, lastAttr, comb)
for d := nextAttr to lastAttr  (* remaining attribs *)
  T := Run "Select * from " D " order by " name[d]
  (* attribute d of D is also attribute d of T *)
  if tuple count of T is one or less then return end if
  Count := partition(T, d)  (* array of counts *)
  k = 1
  for i := 1 to |Count|   (* unique values of attrib d *)
    if Count[i] >= minsup then
      Insert (comb "-" T[k,d], Count[i]) into Results
      T' := Run "Select from" T "where"
              name[d] "=" T[k,d]
      BUC(T', d + 1, comb "-" T[k,d])
    end if
    k := k + Count[i]
  end for
end for
```

Figure 2: BUC Algorithm

ORDER BY of the input on attribute *d* (line 2) and counted tuples by linearly partitioning on attribute *d* (line 4) [3].

Suppose BUC is called with *minsup* of 40% (three tuples), the relation in Table 1 as input, nextAttr of 1, lastAttr of 4, and combination name "". When *d* is 1, tuples are ordered by attribute A and then Partition is called on attribute A, producing a count for each unique value of A in the entire table. No value of A has a count of at least *minsup*, so the algorithm repeats with *d* as 2 (attribute B). After Partition is called, the count for b_1 is less than *minsup*, so the algorithm checks the count for b_2, which is 4. The tuple $<b_2,4>$ is inserted into the Results table. The algorithm then recurses on only those tuples that contain b_2. With these three tuples, Partition is called on attribute C. The count for c_1 is only 1, so the algorithm looks at c_2. The count for c_2 is 3 so $<b_2\text{-}c_2,3>$ is inserted into the Results table and the algorithm recurses on partition c_2. This time after Partition is called on D, no counts are greater than *minsup* so the algorithm returns. This process continues until the Results table contains all of the combinations and counts seen in Table 2.

To improve the running time of this algorithm, the attributes should be ordered from lowest to highest cardinality [1]. This ordering increases pruning on the first few recursions and reduces the number of recursive calls.

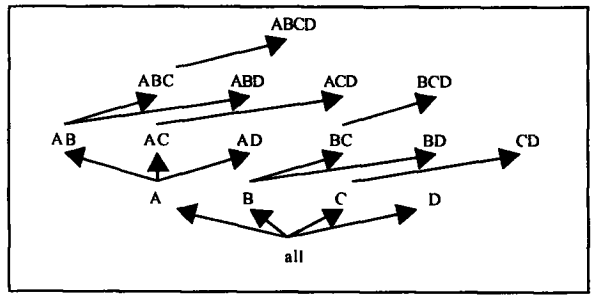

Figure 3: BUC Processing Tree

```
Algorithm TDC(D)
for ordering := Order.iterator(name)
    OnePass(D, ordering, |ordering|)
end for

Function OnePass(D, ordering, size)
T := Run "select " makeString(ordering) " from " D
    "order by " makeString(ordering)
for j := 1 to size do Hold[j] := T[1, j] end for
for i := 1 to |T|  (* for every tuple *)
  Next[1] := T[i , 1] (* first attribute of current tuple*)
  for j := 1 to size  (* for every attribute *)
    if j > 1 then Next[j] := Next[j - 1] "-" T[i , j] end if
    if Next[j] != Hold[j] then  (* finished sequence *)
      if Count[j] >= minsup then
        Insert (Hold[j], Count[j]) into Results
      end if
      Count[j] := 0
      Hold[j] := Next[j]
    end if
    Count[j] := Count[j] + 1
  end for
end for
```

Figure 4: TDC Algorithm

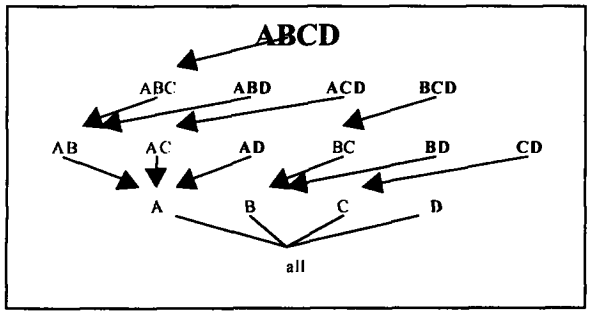

Figure 5: TDC Processing Tree

3.2 Top-Down Computation

The Top-Down Computation (TDC) algorithm is shown in Figure 4. This algorithm is based on an algorithm given in [4]. It uses orderings, produced by the Order subalgorithm to reduce the number of passes through the database. Order is a relatively straightforward iterator implemented as a recursive function [3]. With n attributes, Order produces 2^{n-1} orderings, and TDC calls OnePass on each. For example, for attributes A, B, C, and D, the 8 possible orderings are ABCD, ABD, ACD, AD, BCD, BD, CD and D. When called with a specific ordering, OnePass counts not only that combination but also any combination that is a prefix of that ordering. For ordering ABCD, the combinations A, AB, ABC and ABCD are also counted. These combinations appear on the same path in the processing tree in Figure 5. Separate calls are made to OnePass for the other seven orderings.

If Order is called on the attributes in Table 1, then the first ordering passed to TDC is ABCD. OnePass performs an ORDER BY on the database using the specified ordering. It then visits every tuple in the resulting table to determine the count for all prefixes of ABCD.

For Table 1, $<a_1,b_2,c_1,d_1>$ is copied into *Hold* and the counts are incremented to 1. For the second pass *Next*[1] = a_2, so *Count*[1] is checked. It is less than *minsup* (where *minsup* = 40% from the previous example), so the new tuple $<a_2,b_1,c_1,d_2>$ is copied into *Hold* and the counts are reset to 0. No value of A has count greater than or equal to *minsup* so OnePass returns with no output. The same occurs for the next three orderings: ABD, ACD, and AD.

The fifth ordering is BCD. When TDC is called, an ORDER BY is done on the attributes B, C, and D. The first tuple, $<b_1,c_1,d_2>$, is copied into *Hold* and all attribute counts are incremented to 1. Since the second tuple contains a different B value than the first, the current counts are checked. All are less than *minsup*, so they are reset to 0 and $<b_2,c_1,d_2>$ is copied into *Hold*. On the next iteration *Next* = $<b_2,c_2,d_1>$, so *Hold*[1] is set to *Next*[1] and *Count*[1] is incremented. *Hold*[2] is different than *Next*[2], so *Count*[2] is checked. It is less than *minsup*, so c_2 is copied into *Hold*[2] and *Count*[2] is reset to 0. This process continues with no output until tuple six.

When tuple six is reached, *Hold* = $<b_2,c_2,d_2>$ and *Next* = $<b_3,c_1,d_1>$. At this point *Count* = [4, 3, 1], so $<b_2,4>$ and $<[b_2,c_2],3>$ are both inserted into the Results table because they have counts greater than or equal to *minsup*. All counts are reset to 0 and *Next* is copied into *Hold* as before.

This process continues until the current ordering is completed and the orderings BD, CD and D are also completed. After this process, Results holds all attribute value combinations that satisfy minimum support.

3.3 Top-Down Computation with Pruning (TDC-P)

The TDC-P algorithm adds pruning to TDC. An integer array called *frequent* is used in OnePass-P in parallel with *count*. This array is initialized to 0. When an itemset is found to be above *minsup*, *frequent*[*i*] is set to 1, where *i* corresponds to the last attribute in the itemset. At the end of OnePass-P, *frequent* is analysed and the lowest index that still holds a 0 found. The number of attributes minus this index is returned to TDC-P. TDC-P passes this number to the Order-P iterator, which immediately returns from recursive calls this many times (levels) and then continues execution. The algorithm is given in [3].

For example, for Table 1 and a *minsup* of 3, on the first ordering (ABCD) no value of attribute A satisfies *minsup*. OnePass-P returns the number of attributes (4) minus the index for attribute A (1), which is 3. When Order-P is next called, it skips three levels (anything starting with ABC, AB, or A). This skips three orderings (ABD, ACD and AD). All combinations counted by these orderings include A, yet we already know that no value of A is greater than *minsup*, so we can safely eliminate these orderings. Then Order-P continues by creating the next itemset, BCD.

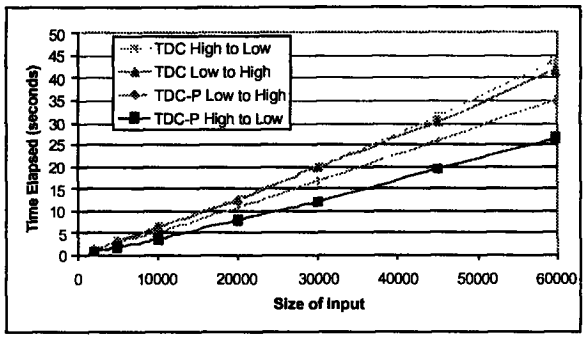

Figure 6: Pruning Effectiveness of TDC-P with Four Input Attributes

TDC-P calls OnePass-P on this itemset and execution continues. The final output is the same as for the BUC and TDC algorithms.

4. Results

The data we used to test these algorithms is grading data from the University of Regina's student record system. The original table has 59689 records and 21 attributes, most of which have cardinalities below 150. We ran tests on the original table and subsets of 45000, 30000, 20000, 10000, 5000, and 2000 records. Input for each test run included one of these tables and three or more attribute names to calculate the attribute value combinations from. We call these attribute names the ***input attributes*** for a test run.

The algorithms were implemented using Microsoft Access and Visual Basic. Testing was done on a Pentium III 600 MHz PC with 256MB of RAM. We measured elapsed time on a dedicated machine. Time for input and output was included. The values we report for elapsed time represent averages for 10 runs on identical data. In previous testing of BUC, the actualization of output was not implemented [1].

Test Series 1: Pruning Effectiveness of TDC-P

First, we measured the effectiveness of the pruning in TDC-P. For all data sets, TDC-P ran at least as fast as TDC, and it was faster for most data sets. A representative example, using 4 input attributes, is shown in Figure 6. TDC-P's pruning gives it faster execution times than TDC for both runs.

The execution time for the TDC-P algorithm depends on the ordering of the input attributes, as does the BUC algorithm [1]. This ordering is based on the cardinality of the attributes. As can be seen in the top two lines in Figure 6, the ordering does not significantly affect the execution time of the TDC algorithm because no pruning occurs. For four input attributes, n is 4, and the TDC algorithm makes $2^{n-1} = 8$ passes over the data for both orderings. TDC visits every tuple the same number of times regardless of the order of the attributes.

TDC-P executed faster with attributes ordered from high to low cardinality than with them ordered from low to high cardinality. High to low cardinality permits pruning at an earlier stage. In the tests used for Figure 6, TDC-P made 7 passes over the data when the ordering was from low to high, but only 5 when it was high to low.

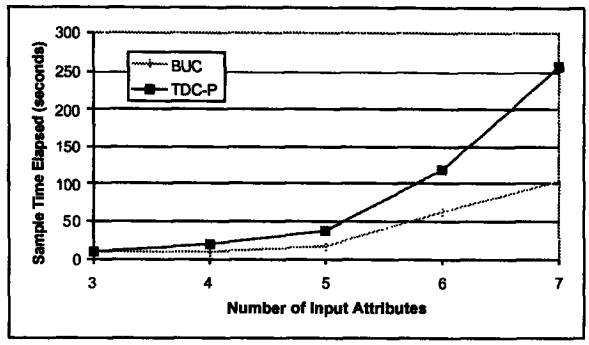

Figure 7: Varying Number of Attributes

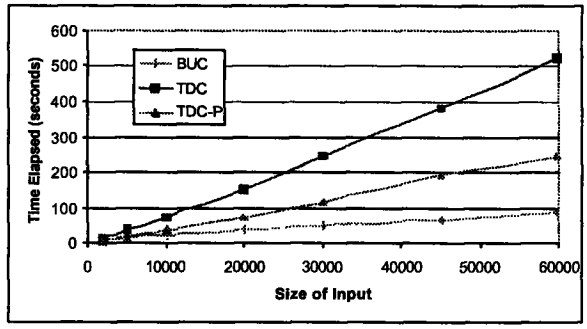

Figure 8: Seven Input Attributes

Test Series 2: Varying Number of Attributes

We tested the three algorithms with varying numbers of attributes. Figure 7 shows that BUC's computation time increases more slowly than TDC-P's as the number of input attributes increases from 3 to 7. All three algorithms have similar running times for three attributes. However, for 7 attributes, the execution time of TDC-P is significantly longer than that of BUC for seven attributes, as shown in Figure 8.

Test Series 3: Varying Cardinality of Attributes

We investigated the effect of varying the cardinality of the attributes on the performance of the three algorithms. For these experiments, we defined an attribute as having ***high cardinality*** if it had at least 150 distinct values and ***low cardinality*** otherwise. Since the person-identifier (PIDM) attribute had a cardinality of about 20000 for the full relation, it offered the best opportunity for early pruning. This experiment allowed us to compare the pruning capability of the algorithms. We tried all low cardinalities (Low), all high cardinalities (High), and a mix of low and high cardinalities (Mixed).

Figure 9 shows the execution times with three low-cardinality input attributes. The similar running times of TDC and TDC-P indicate that little pruning occurred in TDC-P. Since BUC also had similar running times, it must not have pruned much either. In Figure 10, with PIDM and

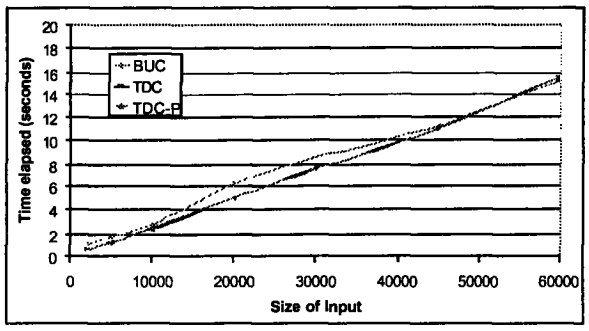

Figure 9: Three Attributes - Low Cardinalities

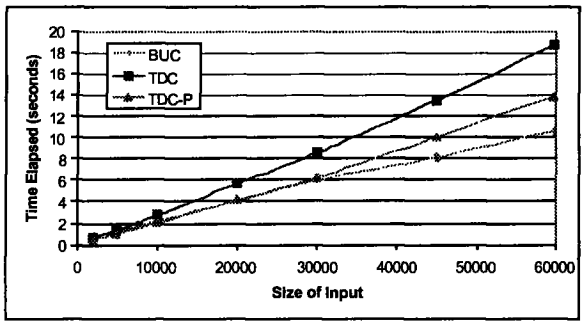

Figure 10: Three Attributes - High Cardinalities

two other higher cardinality attributes, BUC and TDC-P ran faster than TDC, with BUC pruning most effectively.

Figures 11 and 12 show the results of similar tests run with six attributes instead of three. BUC is faster than TDC-P and TDC, on both low and mixed cardinality attributes.

5. Conclusion

We described three algorithms for the Iceberg-Cube problem, which is to identify combinations of attribute values that meet an aggregate threshold requirement. The results of our testing show that the BUC algorithm is faster than the TDC and TDC-P algorithms and produces the same output. TDC-P was more effective than TDC, but could only compete with BUC when there were very few attributes as input.

The BUC algorithm ran faster than the TDC-P algorithm because it begins with the smallest group-bys possible. These small group-bys are significantly faster to compute than the large group-bys (with which TDC-P starts). Although BUC and TDC-P both employ pruning, TDC-P performs a considerable amount of computation on the large group-bys before pruning becomes effective.

This weakness of the TDC-P algorithm in comparison to the BUC algorithm becomes more apparent as the number of attributes increases, because it takes an increasingly long time for TDC-P to process the first few group-bys. The conclusion is that BUC prunes earlier and more effectively than TDC-P. BUC is the algorithm of choice for this problem.

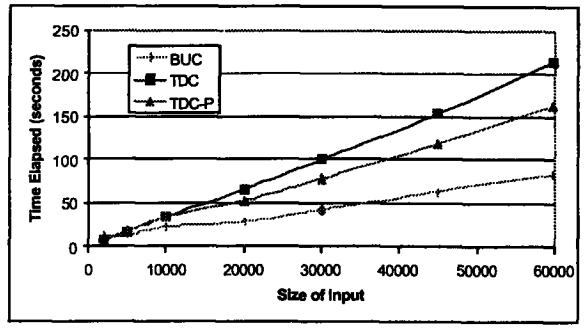

Figure 11: Six Attributes - Low Cardinalities

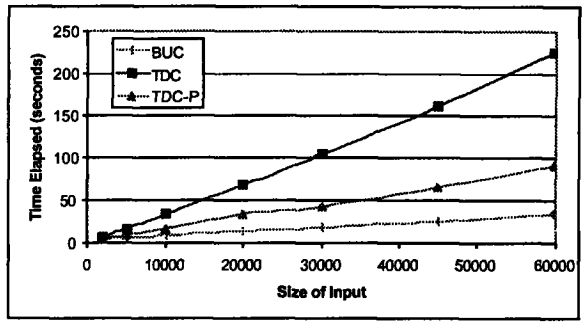

Figure 12: Six Attributes - High and Low Cardinalities

Acknowledgements: We thank Steve Greve for code, and the Natural Science and Engineering Research Council of Canada for an URSA award (LF), a Research grant (HH), and Networks of Centres of Excellence funding provided in cooperation with PRECARN via IRIS (HH).

References

[1] K. Beyer and R. Ramakrishnan. Bottom-Up Computation of Sparse and Iceberg CUBEs. *SIGMOD Record*, 28(2):359-370, 1999.

[2] M. Fang, N. Shivakumar, H. Garcia-Molina, R. Motwani, and J. D. Ullman. Computing Iceberg Queries Efficiently. In *Proc. of the 24th VLDB Conference*, pages 299-310, New York, August 1998.

[3] L. Findlater and H. J. Hamilton. An Empirical Comparison of Methods for Iceberg-CUBE Construction. Technical Report CS-2000-04, University of Regina. Regina, Canada.

[4] S. Greve. *Experiments with Bottom-up Computation of Sparse and Iceberg Cubes*. CS831 Course Project, Department of Computer Science, University of Regina. Regina, Canada, May, 2000.

[5] V. Harinarayan, A. Rajaraman and J. Ullman. Implementing Data Cubes Efficiently. *SIGMOD Record*, 25(2):205-216, 1996.

[6] S. Sarawagi, R. Agrawal, and A. Gupta. *On Computing the Data Cube*. Technical Report RJ10026, IBM Almaden Research Center, San Jose, CA, 1996.