

BUC Algorithm for Iceberg Cubes: Implementation and Sensitivity Analysis

George E. Nasr and Celine Badr

School of Engineering and Architecture
Lebanese American University, Byblos, Lebanon
e-mail: genasr@lau.edu.lb

Abstract

The Iceberg-Cube problem restricts the computation of the data cube to only those group-by partitions satisfying a minimum threshold condition defined on a specified measure. In this paper, we implement the Bottom-Up Computation (BUC) algorithm for computing Iceberg cubes and conduct a sensitivity analysis of BUC with respect to the probability density function of the data. The distributions under consideration are the Gaussian, Geometric, and Poisson distributions. The Uniform distribution is used as a basis for comparison. Results show that when the cube is sparse there is a correlation between the data distribution and the running time of the algorithm. In particular, BUC performs better on Uniform followed by Poisson, Gaussian and Geometric data.

Introduction

In presence of huge amounts of data, the need to transform data into useful knowledge by finding and extracting interesting patterns and existing associations is accentuated. It is crucial that efficient analysis tools are made available to produce reliable information to be used for decision-making, process control, information management, and query processing (Han and Kamber 2001; Fang et al. 1998). Thus knowledge discovery in databases (KDD), also known as data mining, takes a significant role. To combine raw data from heterogeneous sources and store it separately from operational databases, data warehouses have come into sight. They are maintained separately from operational databases and provide different functionalities, mainly OLAP-Online Analytical Processing. Since data warehouses are viewed as multidimensional repositories, they are modeled as n-dimensional data cubes, summarizing a specified aggregate function over the dimensions called the measure. These data cubes can be used for answering queries needed for decision support. When a threshold is introduced for the measure, the number of results satisfying this minimum support requirement often becomes small compared to the total possible results of group by aggregations. In this case, instead of materializing the entire cube and consuming

storage space, or even materializing none of the cuboids and suffering from on-the-fly computation time, it is more efficient to pre-compute only those group-by partitions that satisfy the specified minimum support condition. This is known as the Iceberg-Cube problem. In this paper, we implement the Bottom-Up Computation algorithm introduced in (Beyer and Ramakrishnan 1999). The BUC algorithm computes the cube proceeding bottom-up from the most aggregated cuboids to the least aggregated ones, outperforming other algorithms for cube computation. In addition, we present the results of a sensitivity analysis of the BUC algorithm with respect to the probability distribution of the data varying from Uniform to Gaussian, Geometric and Poisson. The running time of the algorithm is presented for each case. In the case of sparse results, BUC performance proves to degrade in an increasing order when the data distribution varies from Uniform to Poisson, Gaussian, and Geometric. This study is significant as non-uniform data distributions are commonly found in a wide range of application domains (Ross and Srivastava 1997).

Data Cubes Computation

Data warehouses architecture is structured as a multidimensional database, where each dimension corresponds to one or more attributes, and each cell contains the value of an aggregate measure. This multidimensional data model presents the data in the form of an n-dimensional data cube. The data cube measure is classified according to the used numerical aggregate function which can be distributive (count(), sum(), min(), max()), algebraic (avg(), std_dev()), or holistic (median(), rank()). The cube dimensions may contain hierarchies, and they are usually collected at the lowest level of detail. Higher levels are aggregations of the lower ones in the hierarchy. This model allows the user to view data from different angles by performing operations such as roll-up to higher levels of the concept hierarchies, drill-down to lower detailed data levels, slice and dice selections, pivot, and some other visualization operations. Each level of summarization in this hierarchy represents a group-by and may be referred to as cuboid. If n is the number of dimensions in the data cube, the total number of cuboids to

be computed is 2^n . This formula applies when there are no hierarchies associated with each dimension. On the other hand, when dimensions have hierarchies, that is, they are based on abstraction, the total number of cuboids becomes $N_c = \prod [i=1 \text{ to } n] (L_i + 1)$, where L_i is the number of levels of the i^{th} dimension. This means that the number of cuboids grows significantly as n increases, making it impractical to compute them all. Consequently, in order to provide fast response time to OLAP queries, and to avoid huge storage requirements, it is crucial to select a proper subset of the possible cuboids to be precomputed. This is known as partial materialization (Han and Kamber 2001).

Iceberg Cubes

Often in OLAP queries, a minimum support is introduced as a measure of the interestingness of the results. For example, when the aggregate measure is Count(), the minimum support is the minimum number of tuples in which a combination of attribute values must appear to be considered frequent.

When a threshold is introduced for the aggregate measure, the number of results having a value above the minimum support is often very small relative the large amount of input data, as compared to the tip of an iceberg. In this case, the group by is described as sparse because the number of possible partitions in it is large relatively to the number of partitions that actually satisfy the above-threshold condition. When the number of sparse group by's becomes a high percentage of the total number of group by's, the cube is categorized as sparse. In this case, it is preferable to compute the Iceberg cube that holds only the partitions where the cells measure evaluates to an above-threshold value.

The SQL statement of a typical Iceberg-Cube for a three-dimensional data cube is expressed in (Beyer and Ramakrishnan 1999) as:

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

where *minsup* is the value of required minimum support. In distributive cubes, it is possible to take advantage of the fact that higher-level granularities are computed from lower-level ones. That is, if at the more aggregated group by the Count does not evaluate to a value greater than minimum support, then it is subsequent that the less aggregated group by's will not either making it possible to skip the computation for those higher levels. This process constitutes the basis for the algorithms tackling the Iceberg-Cube problem. Since real-time data is frequently sparse (Ross and Srivastava 1997), the importance of those algorithms increases with their efficiency in computing sparse datacubes.

Bottom-Up Computation Algorithm

Kevin Beyer and Raghu Ramakrishnan have proposed an algorithm, BottomUpCube (BUC), that computes sparse and Iceberg cubes (Beyer and Ramakrishnan 1999). Their algorithm is inspired by previous ones presented in (Ross and Srivastava 1997), and combines partitioning to pruning for more efficiency. BUC performance analysis proved to be notably faster than its closest competitors. Also, BUC, which has been limited to simple measures, such as Count or Sum, was extended in (Han et al. 2001) to compute Iceberg cubes with complex measures like Average.

Figure 1 shows the processing tree of BUC for three dimensions. As illustrated, BUC begins the cube computation from the most aggregated group-bys up to the less aggregated one, as opposed to other cube algorithms. It is a recursive algorithm that takes advantage of minimum support pruning since it avoids recursion to lower levels of aggregation (i.e., higher levels in the processing tree) if the *minsup* condition is not satisfied at the current level.

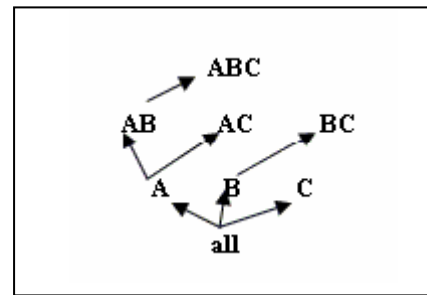


Figure 1: BUC Processing Tree

As a first step, BUC takes the entire input and aggregates it to evaluate the specified measure, Count for instance. If the value meets the threshold condition, BUC considers the first attribute dimension A. The input is then ordered by A, and partitioned to obtain the count of each distinct value of A. In the case where the count of a specific attribute value x is above *minsup*, x is considered to be frequent and outputted to the results. It follows that the tuples containing x in the first attribute of the input relation are further examined by BUC. Those tuples constitute the new input to the recursive call of BUC; ordering and partitioning are next done on the following dimension, B. Similarly, only the tuples containing frequent attribute values of B in the current input are processed in a new recursive call on dimension C. Along the recursive process, the frequent combinations found are sent to output. When all attribute values are considered in the last dimension, the algorithm recurses back to the previous level, and considers the next attribute value of B. Also, when all attribute values of B are considered, BUC recurses back to dimension A to examine the next attribute

value of A. Finally, when no more frequent counts can be found, the algorithm returns and all frequent combinations are in the output relation.

BUC Implementation

With its dynamic selection BUC has proven to outperform all previous algorithms elaborated for Iceberg Cube computation. However, since its performance is not optimal when the data is non-uniform, and since real-world applications data is often non-uniform, this work aims at investigating BUC performance in the case of dimensions having different non-uniform distributions. In particular, data having Gaussian, Geometric or Poisson distribution are examined as compared to the Uniform distribution.

To test the performance of the BUC algorithm, many factors are taken into consideration. This study considers computing Iceberg Cubes of three dimensions with respect to one measure, the monotonically decreasing distributive aggregate function Count. The variable parameters of interest are the input data size, the cardinality of the dimensions, the minimum support value chosen and the distribution of the input data.

This study is conducted on a large number of numerical tuples, obtained from pseudo-random generating classes. To generate the test data, we used the Colt 1.0.2 package provided by the Open Source Libraries for High Performance Scientific and Technical Computing in Java of CERN institute. The classes needed for generating Uniform, Gaussian, Geometric, and Poisson distributions were selected and customized in order to obtain the needed data in the appropriate format. The input consists of flat files of integer numbers with the respective statistical distributions. The size of the data varied from 5000 numbers, to 10000, 25000, and 50000 numbers. For each input size and distribution, we generated files with cardinalities of 25, 100, and 500. Moreover, an additional java class was implemented to read from these files and populate the database relations. Each test case was formed by reading integers from 3 uncorrelated files of equal size, equal cardinality, and similar distribution, and inserting them simultaneously into the 3 dimensions of the input relation to BUC.

The original BUC algorithm is given in (Beyer and Ramakrishnan 1999). For our implementation, we adopted a detailed variant of the algorithm inspired by (Findlater and Hamilton 2001) as listed in Figure 2. BUC algorithm was implemented using Java Language, compiled under JBuilder 7.0. Java Database Connection was used to access the database relations stored on an Oracle 9i local database server. In order to have in-memory processing of relations, we used the CachedRowSet class since it does not maintain an open connection to the database and stores data completely in memory. External partitioning was not needed since test input could be entirely placed in main memory.

In this investigation, experiments were conducted on a dedicated Pentium 4 computer, with CPU speed of 1500 MHz, and 512 MB RAM. The reported values represent the average of values collected over multiple test runs on identical data.

```

Class BUC(inputRelation, dim, startingAttr)
  level = level + 1
  newTable = "temptable" + dim
  for d = dim to numDims
    curTable = Run ("select * from " + inputRelation +
      " order by " + fields[d])
    if curTable.TupleCount <= 1 then return end if
    C = cardinality[d]
    dataCount = partition(inputRelation, d, C)
    k = 0
    for i = 0 to C - 1
      count = dataCount[i + 1]
      if count >= minsup then
        if level = 1 then startingAttr = d end if
        strTemp = curTable[k, startingAttr]
        for j = d - level + 2 to d
          strTemp = strTemp + "-" + curTable[k, j]
        end for
        Run ("insert into Results (Combination, Count)
          values (" + strTemp + "," + count + ")")
        Run ("select into " + newTable + " from " +
          inputRelation + " where " + fields[d] + " = " +
          curTable[k,d])
        ret_val = BUC(newTable, d + 1, startingAttr)
        end if
        k = k + count
      end for
    end for
  Run ("drop table " + newTable)
  level = level - 1
return

```

```

Function partition(inputRelation, d, C)
  int count[C]
  strGroup = fields[d]
  newTable = "temptable" + d
  Run ("select " + strGroup + ", count(*) as Count into
    " + newTable + " from " + inputRelation + " group by
    " + strGroup)
  for i = 0 to newTable.RecordCount - 1
    count[i + 1] = newTable [i, newTable.FieldCount - 1]
  end for
  count[0] = d
return count

```

Figure 2: BUC Algorithm

Performance Results

The running time of the algorithm was recorded in each test. The average values computed are presented in graphical format. Figures 3, 4, and 5 depict time measurements for a minimum support value of 50. Figures 6, 7, and 8 give the results for a minimum support of 100 whereas Figures 9, 10, and 11 show the results for a minimum support of 500. Each set of figures illustrates the execution times of three test cases where the dimensions cardinality is 25, 100, and 500, respectively. The values on the x-axis represent the size of the sample data, in thousands of tuples. The y-axis values represent the time. Actual measurements were taken in milliseconds. For clarity of the plots, the values are converted to seconds. The values for the four different data distributions are plotted simultaneously on each graph. As seen in the plots, the only constant general conclusion that can be made is that the performance of BUC is similar for the Gaussian and Poisson distributions. On the other hand, in the case where *minsup* is 500 the results are sparse for all considered distributions. Moreover, these results clearly show that BUC performs best for Uniform data distribution while exhibiting increasing performance degradation for the Poisson, Gaussian, and Geometric distributions respectively.

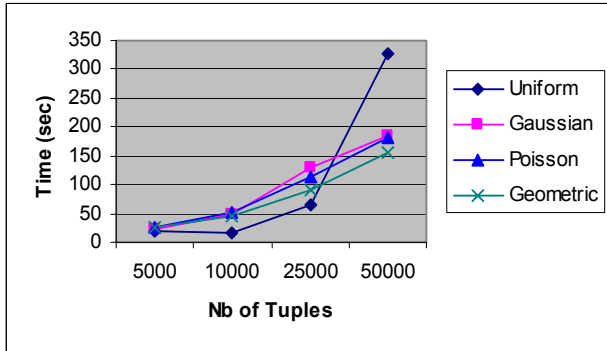


Figure 3: Running Time for Card = 25 and Minsup = 50

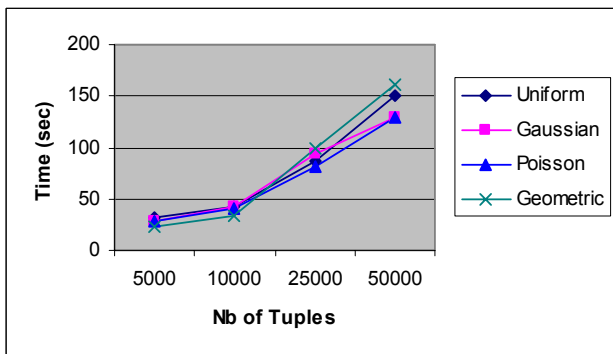


Figure 4: Running Time for Card = 100 and Minsup = 50

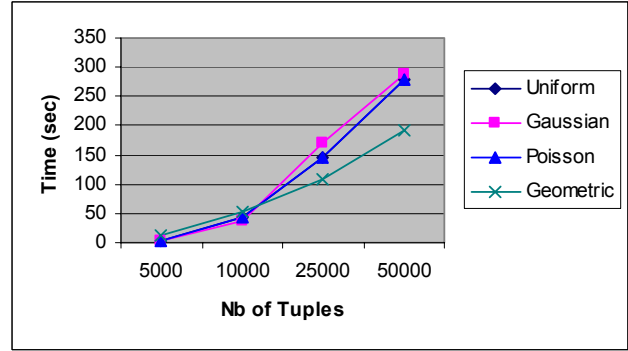


Figure 5: Running Time for Card = 500 and Minsup = 50

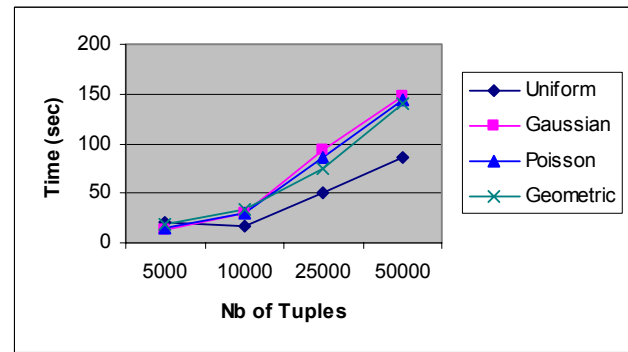


Figure 6: Running Time for Card = 25 and Minsup = 100

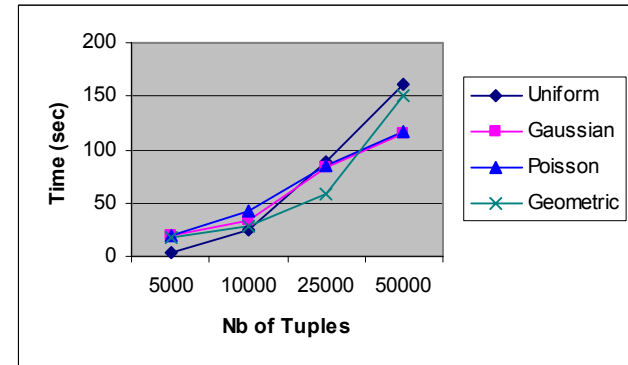


Figure 7: Running Time for Card = 100 and Minsup = 100

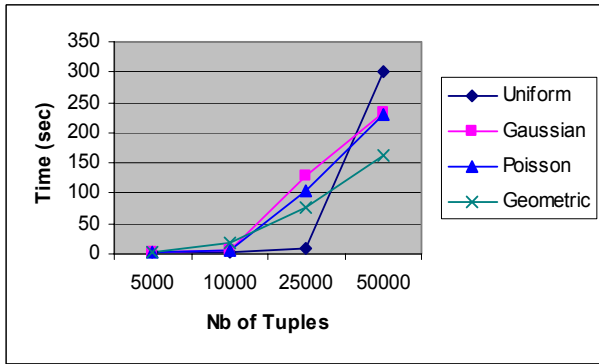


Figure 8: : Running Time for Card = 500 and Minsup = 100

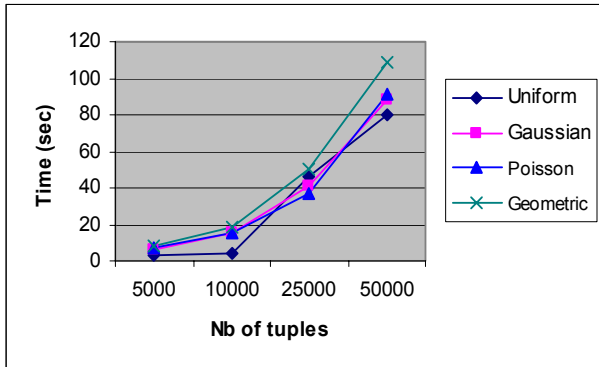


Figure 9: Running Time for Card = 25 and Minsup = 500

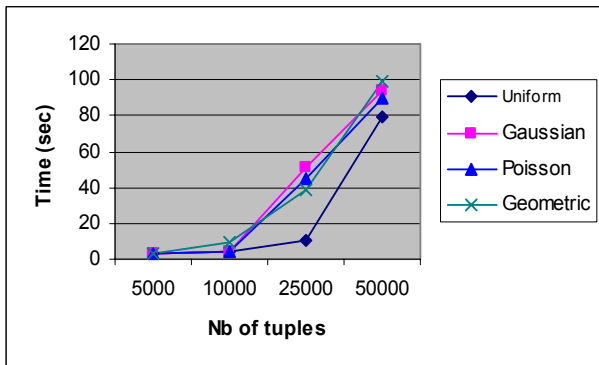


Figure 10: Running Time for Card = 100 and Minsup = 500

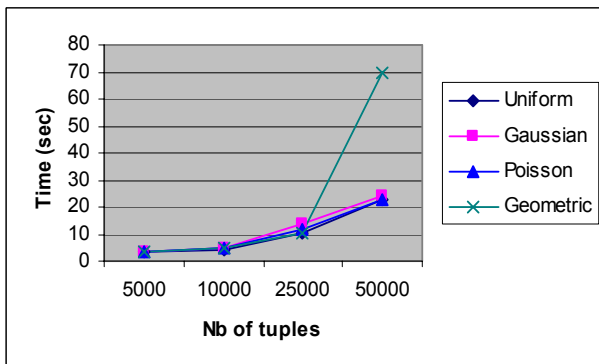


Figure 11: Running Time for Card = 500 and Minsup = 500

Conclusion

In this paper, the Bottom-Up Computation algorithm is implemented on Iceberg cubes of three dimensions using the monotonically decreasing distributive aggregate function Count. The performance of the BUC algorithm on non-uniform data is investigated. In particular, algorithm behavior on data having Gaussian, Geometric or Poisson distribution is examined as compared to its performance on data with Uniform distribution. Results show that there is a correlation between the data distribution and the running time of the algorithm. In the case of sparse cube, BUC performance proves to degrade in an increasing order when the data distribution varies from Uniform to Poisson, Gaussian, and Geometric. Moreover, testing results using different values for the size of input data, the cardinality of dimensions and the required minimum support are included.

References

- Beyer, K. and Ramakrishnan, R. 1999. Bottom-Up Computation of Sparse and Iceberg CUBEs. *SIGMOD Record* 28(2): 359-370.
- Fang, M., Shivakumar, N., Garcia-Molina, H., Motwani, R. and Ullman, J.D. 1998. Computing Iceberg Queries Efficiently. In *Proceedings of the 24th VLDB Conference*, 299-310. New York.
- Findlater, L. and Hamilton, H. J. 2001. An Empirical Comparison of Methods for Iceberg-CUBE Construction. In *Proceedings of the 14th International FLAIRS Conference*, 244-248. Menlo Park, Calif: AAAI Press.
- Han, J., and Kamber, M. 2001. *Data Mining Concepts and Techniques*. 1 – 99: Academic Press. Morgan Kaufmann Publishers.
- Han, J., Pei, J., Dong, G. and Wang, K. 2001. Efficient Computation of Iceberg Cubes with Complex Measures. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, 1-12. Santa Barbara, Calif.
- Ross, K.A., and Srivastava D. 1997. Fast Computation of Sparse Datacubes. In *Proceedings of the 23rd VLDB Conference*, 116-125. Athens, Greece.