

Cross Table Cubing: Mining Iceberg Cubes from Data Warehouses*

Moonjung Cho

State University of New York at Buffalo, U.S.A.
mcho@cse.buffalo.edu

Jian Pei

Simon Fraser University, Canada
jpei@cs.sfu.ca

David W. Cheung

The University of Hong Kong, China
dcheung@csis.hku.hk

Abstract

All of the existing (iceberg) cube computation algorithms assume that the data is stored in a single base table, however, in practice, a data warehouse is often organized in a schema of multiple tables, such as star schema and snowflake schema. In terms of both computation time and space, materializing a universal base table by joining multiple tables is often very expensive or even unaffordable in real data warehouses. In this paper, we investigate the problem of computing iceberg cubes from data warehouses. Surprisingly, our study shows that computing iceberg cube from multiple tables directly can be even more efficient in both space and runtime than computing from a materialized universal base table. We develop an efficient algorithm, CTC (for Cross Table Cubing) to tackle the problem. An extensive performance study on synthetic data sets demonstrates that our new approach is efficient and scalable for large data warehouses.

1 Introduction

Given a base table $B(D_1, \dots, D_n, M)$ and an aggregate function, where D_1, \dots, D_n are n dimensions and M is a measure, a data cube consists of the complete set of group-bys on any subsets of dimensions and their aggregates using the aggregate function. A data cube in practice is often huge due to the very large number of possible dimension value combinations. Since many detailed aggregate cells whose aggregate values are too small may be trivial in data analysis, instead of computing a complete cube, an iceberg cube can be computed, which consists of only the set of group-bys whose aggregates are no less than a user-specified aggregate threshold.

In the previous studies, several efficient algorithms have been proposed to compute (iceberg) cubes efficiently from a *single* base table, with simple or complex measures, such as BUC [1] and H-Cubing [4]. All of them assume that the data is stored in a single base table. However, a data warehouse in practice is often organized in a schema of multiple tables, such as star schema or snowflake schema. Although mining iceberg cube from single table becomes more and more efficient, such algorithms cannot be applied directly to real data warehouses in many applications.

*This research is supported in part by NSF Grant IIS-0308001, a President's Research Grant, an Endowed Research Fellowship Award and a startup grant in Simon Fraser University. All opinions, findings, conclusions and recommendations in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

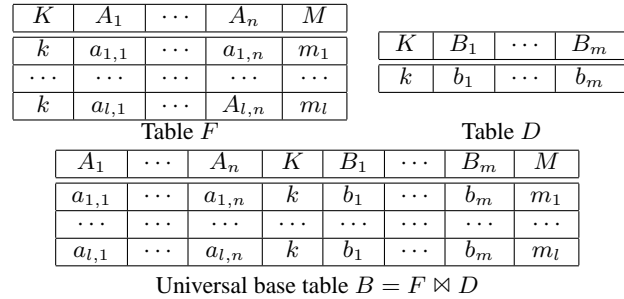


Figure 1: A simple case of computing iceberg cube from two tables.

EXAMPLE 1. (INTUITION) Consider computing the iceberg cube from tables F and D in Figure 1. Suppose attribute M is the measure. A rudimentary method may first compute a universal base table $B = F \bowtie D$, as also shown in the figure, and then compute the iceberg cube from B . However, such a rudimentary method may suffer from two non-trivial costs.

Space cost. As shown in the figure, the tuple in table D is replicated l times in the universal base table B , where l is the number of tuples in the fact table. Moreover, every attribute in the tables appears in the universal base table. Thus, the universal base table is wider than any table in the original database. In real applications, there can be a large number of tuples in the fact table, and hundreds of attributes in the database. Then, the dimension information may be replicated many times, and the universal base table may be very wide – containing hundreds of attributes.

Time cost. The large base table may have to be scanned many times and many combinations of attributes may have to be checked. As the universal base table can be much wider and larger than the original tables, the computation time can be dramatic.

Can we compute iceberg cubes directly from F and D without materializing the universal base table B ? The following two observations help.

First, for any combination of attributes in table D , the aggregate value is $m = \text{aggr}(\{m_1, \dots, m_l\})$. Therefore, if m satisfies the iceberg condition, then every combination of attributes in D is an iceberg cell. Here, we compute these iceberg cells using table D only, which contains only 1 tuple. In the rudimentary method, we have to use many tuples in table B to compute these iceberg cells.

Second, for any iceberg cell involving attributes in table F , the aggregate value can be computed from table F only. In other words, if we find an iceberg cell in F , we can

enumerate a whole bunch of iceberg cells by inserting more attributes in D and the aggregate value retains. Please note that we only use F , which has only $(n + 1)$ attributes, to compute these iceberg cells. In the rudimentary method, we have to compute these iceberg cells using a much wider universal base table B .

Although the observations here are based on an over-simplified case, as shown in the rest of the paper, the observations can be generalized. ■

In this paper, we make the following contributions. First, we address the problem of mining iceberg cubes from data warehouses of multiple tables. We use star schema as an example. Our approach can be easily extended to handle other schemas in data warehouses, such as snowflake schema.

Second, we develop an efficient algorithm, CTC (for *Cross Table Cubing*), to compute iceberg cubes. Our method does not need to materialize the universal base table. Instead, CTC works in three steps. First, CTC propagates the information of keys and measure to each dimension table. Second, the local iceberg cube in each table is computed. Last, the global iceberg cube is derived from the local ones. We show that CTC is more efficient in both space and runtime than computing iceberg cube from a materialized universal base table.

Last, we conduct an extensive performance study on synthetic data sets to examine the efficiency and the scalability of our approach. The experimental results show that CTC is efficient and scalable for large data warehouses.

The rest of the paper is organized as follows. In Section 2, we formulate the problem and review related work. Algorithm CTC is developed in Section 3 by examples. A performance study is briefly reported in Section 4.

2 Problem Definition and Related Work

Without loss of generality, we assume that the domains of the attributes in the tables are exclusive, except for the foreign keys K_i in the fact table F referencing to the primary keys K_i in dimension table D_i .

EXAMPLE 2. (STAR SCHEMA) Consider the data warehouse DW in Figure 2. We will use this data warehouse as the running example in the rest of the paper.

The star schema is shown in Figure 2(a). In data warehouse DW , the fact table $Fact$ has 3 dimensions, namely A , B and E . The measure is M . Dimensions B and E reference to dimension tables D_1 and D_2 , respectively. In data warehouse DW , the universal base table $T_{base} = Fact \bowtie D_1 \bowtie D_2$ is shown in Figure 3. ■

Let $B = (A_1, \dots, A_m, M)$ be a universal base table, where A_1, \dots, A_m are either dimensions or attributes in dimension tables. A cell $c = (a_1, \dots, a_m)$ is called an *aggregate cell*, where $a_i \in A_i$ or $a_i = *$ ($1 \leq i \leq m$). The *cover* of c is the set of tuples in B that match all non- $*$ a_i 's, i.e., $cov(c) = \{t \in B \mid \forall a_i \neq *, t.A_i = a_i\}$.

For an aggregate function $aggr()$ on the domain of M , $aggr(c) = aggr(cov(c))$.

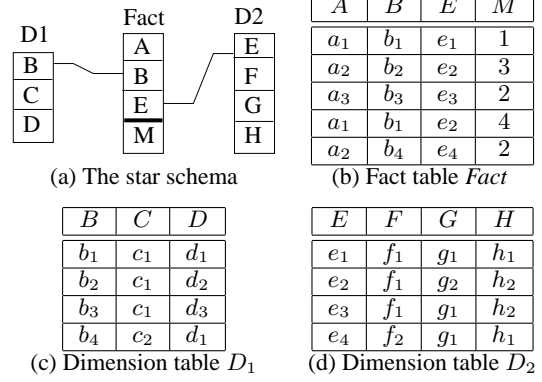


Figure 2: Data warehouse DW as the running example.

| A | B | C | D | E | F | G | H | M |
|-------|-------|-------|-------|-------|-------|-------|-------|---|
| a_1 | b_1 | c_1 | d_1 | e_1 | f_1 | g_1 | h_1 | 1 |
| a_2 | b_2 | c_1 | d_2 | e_2 | f_1 | g_2 | h_2 | 3 |
| a_3 | b_3 | c_1 | d_3 | e_3 | f_1 | g_1 | h_2 | 2 |
| a_1 | b_1 | c_1 | d_1 | e_2 | f_1 | g_2 | h_2 | 4 |
| a_2 | b_4 | c_2 | d_1 | e_4 | f_2 | g_1 | h_1 | 2 |

Figure 3: The universal base table T_{base} .

For an *iceberg condition* C , where C is defined using some aggregate functions, a cell c is called an *iceberg cell* if c satisfies C . An *iceberg cube* is the complete set of iceberg cells.

EXAMPLE 3. (ICEBERG CUBE) In base table T_{base} (Figure 3), for aggregate cell $c = (*, b_1, *, d_1, *, f_1, *, *)$, $cov(c)$ contains 2 tuples, the first and the fourth tuples in T_{base} , since they match c in dimensions B , D and F . We have $COUNT(cov(c)) = 2$.

Consider iceberg condition $C \equiv (COUNT(c) \geq 2)$. Aggregate cell c satisfies the condition and thus is in the iceberg cube. ■

Problem definition. The *problem of computing iceberg cube from data warehouse* is that, given a data warehouse and an iceberg condition, compute the iceberg cube. Limited by space, we only discuss data warehouses in star schema in this paper. ■

For aggregate cells $c = (a_1, \dots, a_m)$ and $c' = (a'_1, \dots, a'_m)$, c is called an *ancestor* of c' and c' a *descendant* of c if for any $a_i \neq *$, $a'_i = a_i$ ($1 \leq i \leq m$), denoted by $c' \sqsubseteq c$. An iceberg condition C is called *monotonic* if for any aggregate cell c , if C holds for c , then C also holds for every ancestor of c . In this paper, we only consider monotonic iceberg conditions, such as $COUNT(c) \geq v$, $MAX(c) \geq v$, $MIN(c) \leq v$, $SUM(c) \geq v$ (if the domain of the measure consists of only non-negative numbers).

Many approaches have been proposed to compute data cubes efficiently from scratch (e.g., [6, 1]). In general, they speed up the cube computation by sharing partitions, sorts, or partial sorts for group-bys with common dimensions.

Fang et al. [2] proposed the concept of iceberg queries and developed some sampling algorithms to answer such

queries. Beyer and Ramakrishnan [1] introduced the problem of iceberg cube computation in the spirit of [2] and developed algorithm BUC. BUC conducts bottom-up computation and can use the monotonic iceberg conditions to prune.

H-cubing [4] uses a hyper-tree data structure called H-tree to compress the base table. The H-tree can be traversed bottom-up to compute iceberg cubes. It also can prune unpromising branches of search using monotonic iceberg conditions. Moreover, a strategy was developed in [4] to use weakened but monotonic conditions to approximate non-monotonic conditions to compute iceberg cubes. The strategies of pushing non-monotonic conditions into bottom-up iceberg cube computation were further improved by Wang et al. [5]. A new strategy, divide-and-approximate, was developed.

All of the previous studies on computing (iceberg) cubes assume that a *universal base table is materialized*. However, many real data warehouses are stored in tens or hundreds of tables. *It is often unaffordable to compute and materialize a universal base table for iceberg cube computation*. This observation motivates the study in this paper.

3 CTC: A Cross Table Cubing Algorithm

For an iceberg cell c with respect to a monotonic iceberg condition, its projections on the fact table and the dimension tables must also be local iceberg cells. Instead of directly computing the iceberg cube from a universal base table, we can compute local iceberg cubes from the fact table and the dimension tables, respectively. Then, we can try to derive the global iceberg cube from the local ones.

Based on the above observation, algorithm CTC works in three steps. First, the aggregate information is propagated from the fact table to each dimension tables. Then, the iceberg cubes in the propagated dimension tables as well as in the fact table (i.e., the *local iceberg cubes*) are mined independently using the same iceberg cube condition. Last, the iceberg cells involving attributes in multiple dimension tables are derived from the local iceberg cubes.

3.1 Propagation Across Tables

EXAMPLE 4. (PROPAGATING AGGREGATE INFORMATION) Consider our running example data warehouse (Figure 2) again. To propagate the aggregate information from the fact table *Fact* to the dimension tables D_1 and D_2 , we create a new attribute *Count* in every dimension table. By scanning the fact table once, the number of occurrences of each foreign key value in the fact table can be counted. Such information is registered in the column of *Count* in the dimension tables, as shown in Figure 4. Hereafter, the propagated dimension tables are denoted as PD_1 and PD_2 , respectively, to distinguish from the original dimension tables.

In the rest of the computation, we only use the fact table and the propagated dimension tables PD_1 and PD_2 . We will show that the iceberg cube computed from these three tables is the same as the one computed from the universal base table. ■

| B | C | D | $Count$ |
|-------|-------|-------|---------|
| b_1 | c_1 | d_1 | 2 |
| b_2 | c_1 | d_2 | 1 |
| b_3 | c_1 | d_3 | 1 |
| b_4 | c_2 | d_1 | 1 |

(a) Propagated PD_1

| E | F | G | H | $Count$ |
|-------|-------|-------|-------|---------|
| e_1 | f_1 | g_1 | h_1 | 1 |
| e_2 | f_1 | g_2 | h_2 | 2 |
| e_3 | f_1 | g_1 | h_2 | 1 |
| e_4 | f_2 | g_1 | h_1 | 1 |

(b) Propagated PD_2

Figure 4: The propagated dimension tables.

This computation of the aggregates on the keys is implemented as group-by aggregate queries on the key attributes in the fact table. Only the fact table is needed to conduct such queries. The aggregate information is appended to the records in the dimension tables after the aggregates are computed. In general, we extend every dimension table to include a measure column.

CTC never really joins multiple tables. Instead, it only conducts group-by queries on each key attribute and propagates the aggregates to the corresponding dimension table. When there are multiple dimension tables, propagating the aggregates is much more cheaper than joining multiple tables and materializing a universal base table. We notice that there are efficient indexing techniques to join tables in star schema fast. Most of those techniques can also be used to propagate the aggregates to dimension tables efficiently.

3.2 Computation of Local Iceberg Cubes Local iceberg cubes on propagated dimension tables can be computed using an adaption of any algorithms for iceberg cube computation. For each iceberg cell c , we maintain the histogram of primary key values that the tuples in $cov(c)$ carry.

EXAMPLE 5. (COMPUTING LOCAL ICEBERG CUBE)

We can compute the iceberg cube on propagated dimension table PD_2 (Figure 4(b)) with respect to condition $C \equiv COUNT(c) \geq 2$ using algorithm BUC [1]. One advantage of computing iceberg cubes on propagated dimension tables is that *one tuple in the propagated dimension table may summarize multiple tuples in the corresponding projection of the universal base table*. Thus, we reduce the number of tuples in the computation.

For each iceberg cell, we record the histogram of primary key values that the tuples in the cell carry. For example, for iceberg cell $(*, f_1, *, *)$ with count 4, we record the set of primary key values $\{e_1, e_2, e_3\}$ that the tuples having f_1 carry. This is called the *signature* of the iceberg cell. It will be used in the future to derive global iceberg cells. To maintain the signature, we can use a vector of m bits for every iceberg cell, where m is the number of distinct values appearing in attribute E (the primary key attribute) in table PD_2 . ■

Let D be a dimension table and K be a primary key attribute such that K is used in the fact table as the foreign key referencing to D . For an iceberg cell c in D , the *signature* of c , denoted as $c.sig$, is the set of primary key values (i.e., values in K) that appear in the tuples in $cov(c)$ in D . Clearly, to maintain the signatures in D , we only need m bits, where m is the number of distinct values in K that

appear in the fact table. m is at most the number of tuples in D , and no more than the cardinality of K .

3.3 Computation of Global Iceberg Cubes The set of global iceberg cells can be divided into two exclusive subsets: the ones having some non- $*$ values on the dimension attributes in the fact table, and the ones whose projections on the fact table are $(*, \dots, *)$. We handle them separately.

EXAMPLE 6. (ICEBERG CELL INVOLVING FACT TABLE) Now, we consider the iceberg cells that contain some non- $*$ values in the dimension attributes in fact table *Fact*. To find such iceberg cells, we first apply an iceberg cube computing algorithm, such as BUC [1], to the fact table.

For example, we find $(a_1, *, *) : 2$ is an iceberg cell in the fact table. In the cover of $(a_1, *, *)$ (i.e., the first and the fourth tuples in Figure 2(b)), b_1 appears in attribute B , which references to dimension table D_1 . Thus, for any local iceberg cell c in PD_1 whose signature contains b_1 , such as $(b_1, *, *)$, $(*, c_1, *)$, and $(*, c_1, d_1)$, the join¹ of $(a_1, *, *)$ and c , such as $(a_1, b_1, *, *, *, *, *)$, $(a_1, *, c_1, *, *, *, *, *)$ and $(a_1, *, c_1, d_1, *, *, *, *)$, must be a global iceberg cell of count 2 (yielding to the measure of the iceberg cell in the fact table). As another example, iceberg cell $(a_1, *, c_1, *, *, *, *, *)$, e_1 and e_2 appear in attribute E , which reference to dimension table D_2 . Thus, for any local iceberg cell c in PD_2 whose signature contains e_1 or e_2 , such as $(*, f_1, *, *)$, can be a global iceberg cell, if the overlap of the signatures can lead to an aggregate value satisfying the iceberg condition. Then, we can further join them to get iceberg cell $(a_1, *, c_1, *, *, f_1, *, *)$.

In such a recursive way, we can find all the global iceberg cells that contain some values in the attributes in fact table *Fact*. Limited by space, we omit the details here. ■

We never need to join the fact table with any dimension tables to generate a global iceberg cell. Instead, we join the local iceberg cells based on the signatures. Recall that we maintain the signatures using bitmap vectors, the matching of signatures is efficient. To facilitate matching, we also index the iceberg cells in the dimension tables by their signatures. Another advantage of the algorithm is that, a local iceberg cell is found only once but is used many times to join with other local iceberg cells to form global ones. If we compute the global iceberg cells from the universal base table, we may have to search the same portion of the universal base table for the (local) iceberg cell many times for different global iceberg cells. The cross table algorithm eliminates the redundancy in the computation.

EXAMPLE 7. (JOINING LOCAL ICEBERG CELLS) We consider how to compute the global iceberg cells in data warehouse *DW* (Figure 2) that do not contain any non- $*$ value in

¹Let c_1 and c_2 be aggregate cells on tables T_1 and T_2 , respectively, such that if T_1 and T_2 have any common attribute then c_1 and c_2 have the same value in every such common attribute. The *join* of c_1 and c_2 , denoted as $c_1 \bowtie c_2$, is the tuple c such that (1) for any attribute A that c_1 has a non- $*$ value, c has the same value as c_1 on A ; (2) for any attribute B that c_2 has a non- $*$ value, c has the same value as c_2 on B ; (3) c has value $*$ in all other attributes.

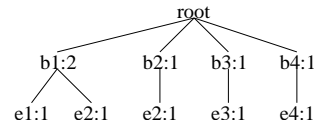


Figure 5: The H-tree for foreign key attribute values.

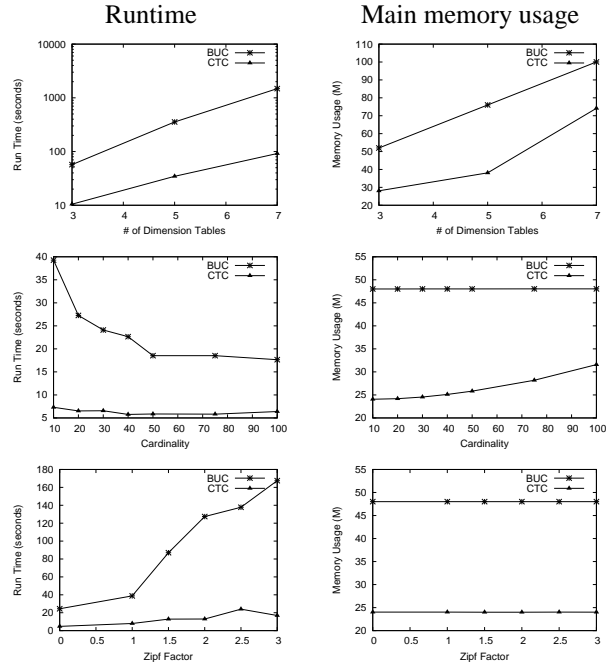


Figure 6: Experimental results – Part 1.

attributes in the fact table. Those global iceberg cells can be divided into two subsets: (1) the descendants of some local iceberg cells in PD_1 , and (2) the descendants of some local iceberg cells in PD_2 but not descendant of any local iceberg cells in PD_1 . In both cases, we only consider the cells that do not contain any non- $*$ value in the key attributes.

To find the first subset, we consider the local iceberg cells in PD_1 one by one. For example, $(*, c_1, *)$ is a local iceberg cell in PD_1 with signature $\{b_1, b_2, b_3\}$. To find the local iceberg cells in PD_2 that can be joined with $(*, c_1, *)$ to form a global iceberg cell, we should collect all the tuples in the fact table that contain either b_1, b_2 or b_3 , and find their signature on attribute E .

Clearly, to derive the signature on attribute E for a local iceberg cell in table PD_1 by collecting the tuples in the fact table is inefficient, since we have to scan the fact table once for each local iceberg cell. To tackle the problem, we build an H-tree [4] using only the foreign key attributes in the fact table, as shown in Figure 5.

With the H-tree, for a given signature on attribute B , it is efficient to retrieve the corresponding signature on attribute E . For example, for $(*, c_1, *)$, its signature (on B) is $\{b_1, b_2, b_3\}$. From the H-tree, we can retrieve its signature on E is $\{e_1, e_2, e_3\}$, i.e., the union of the nodes at level E that are descendants of b_1, b_2 or b_3 .

Then, we can search the iceberg cells in dimension table PD_2 . For example, iceberg cell $(*, *, g_1*)$ in dimension

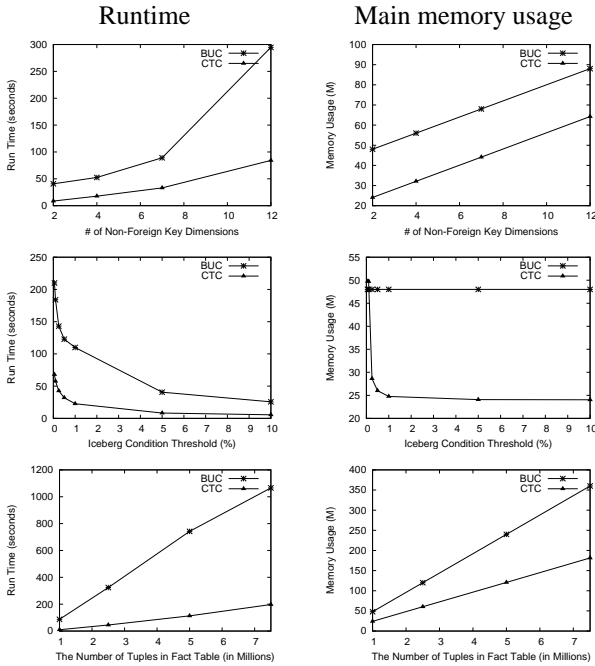


Figure 7: Experimental results – Part 2.

table PD_2 has signature $\{e_1, e_3, e_4\}$. The intersection of the two signatures is $\{e_1, e_3\}$. From the H-tree, we know that the total aggregate of tuples having e_1 or e_3 and b_1, b_2 or b_3 is 2 (the sum of the first and the fourth leaf nodes in the H-tree). Thus, the two iceberg cells can be joined and $(*, *, c_1, *, *, *, g_1, *)$ is a global iceberg cell.

Moreover, if we have more than 2 foreign key attributes, once all the global iceberg cells that are descendants of local iceberg cells in dimension table PD_1 are computed, the level of attribute B in the H-tree can be removed and the remaining sub-trees can be collapsed according to the next attribute, E . That will further reduce the tree size and search cost.

The second subset of global iceberg cells, i.e., the ones that are descendants of some local iceberg cells in PD_2 , but not of PD_1 , are exactly $(*, *, *, *) \bowtie c$, where c is a local iceberg cell in PD_2 . ■

The space complexity of the H-tree in CTC is $O(kn)$, where k is the number of dimension tables and n is the number of tuples in the fact table. In many cases, the H-tree is smaller than the fact table and much smaller than the universal base table. The signatures of local iceberg cells can be stored on disk and do not have to be maintained in main memory.

4 Experimental Results

In this section, we briefly report an extensive performance study on computing iceberg cubes from data warehouses in star schema, using synthetic data sets. All the experiments are conducted on a Dell Latitude C640 laptop computer with a 2.0 GHz Pentium 4 processor, 20 G hard drive, and 512 MB main memory, running Microsoft Windows XP operating

system. We compare two algorithms: BUC [1] and CTC. Both algorithms are implemented in C++.

We generate synthetic data sets following the Zipf distribution. By default, the fact table has 5 dimensions, 1 million tuples and the cardinality of each dimension is set to 10; we set 3 dimension tables, and each dimension table has 3 attributes; the Zipf factor is set to 1.0.

In a data warehouse generated by the above data generator, if there are n dimensions in the fact table and k dimension tables ($n \geq k$), and there are l attributes in each dimension table, then the universal base table has $(l \cdot k + (n - k))$ dimensions. Thus, by default, a data warehouse has 11 dimensions. In all our experiments, we use aggregate function `count()`. Therefore, the domain, cardinality and distribution on the measure attribute have no effect on the experimental results. By default, we set the iceberg condition to “`COUNT(*) ≥ number of tuples in fact table × 5%`”.

In all our experiments, the runtime of CTC is the elapsing time that CTC computes iceberg cube from multiple tables, including the CPU time and I/O time. However, the runtime of BUC is only the time that BUC computes iceberg cube from the universal base table, including the CPU time and I/O time. That is, the time of deriving the universal table is not counted in the BUC runtime. We believe that such a setting does not bias towards CTC.

To simplify the comparison, we assume that the universal base table can be held into main memory in our experiments. When the universal base table cannot be held into main memory, the performance of BUC will be degraded substantially. CTC does not need to store all the tables in main memory. Instead, it loads tables one by one. The local iceberg cells can be indexed and stored on disk. One major consumption of main memory in CTC is to store the H-tree for the fact table. As shown before, the H-tree is often smaller than the fact table and much smaller than the universal base table. When the H-tree is too large to fit into main memory, the disk management techniques as discussed in [4] and also the techniques for disk-based BUC can be applied.

The experimental results are shown in Figures 6 and 7, while the curves are self-explained. By the extensive performance study using synthetic data sets, we show that CTC is consistently more efficient and more scalable than BUC. The performance of BUC in our experiments is consistent in trend with the results reported in [1].

References

- [1] K. Beyer and R. Ramakrishnan. Bottom-up computation of sparse and iceberg cubes. In *SIGMOD'99*.
- [2] M. Fang et al. Computing iceberg queries efficiently. In *VLDB'98*.
- [3] J. Gray et al. Data cube: A relational operator generalizing group-by, cross-tab and sub-totals. In *ICDE'96*.
- [4] J. Han et al. Efficient computation of iceberg cubes with complex measures. In *SIGMOD'01*.
- [5] K. Wang et al. Pushing aggregate constraints by divide-and-approximate. In *ICDE'03*.
- [6] Y. Zhao et al. An array-based algorithm for simultaneous multidimensional aggregates. In *SIGMOD'97*.