Data Cube Computing Method

The Star Cubing Approach

Introduction

OLAP is the most important decision support analysis tool in data warehouse. The traditional OLAP design is based on the historical data, it emphasizes on the offline batch computation which restricts real-time analysis. In today's rapidly changing business society, policymakers need to seize the fleeting business opportunities, so real-time analysis becomes more and more important on helping them make decisions in time.

Real-time OLAP asks for fast response ability while data warehouse is updating. In order to realize the real-time OLAP, data cube is an effective solution, precomputes all the possible query conditions user may raise. When user submits queries, data cube will return the expected result directly without the need for complex online aggregate computing. While the data volume of data warehouse is very huge, and keeps a highly growth trend, and the corresponding data cube size will be dozen times bigger than the original data volume.

Originally, the data cube concept was use for OLAP, it is also useful for data mining. The Multidimensional data mining is an approach to data mining that integrates data that based on OLAP analysis with knowledge discovery techniques.

Data cube 1computation is the most essential operations in data warehouse implementation. Efficient computation of data cube can greatly reduce the response time and enhance the performance of online analytical processing. But this kind of computation is a challenge for memory space and computation time.

The DATA MINING Concepts and Techniques Third Edition introduced the following general strategies for data cube computation:

Optimization technique 1: Sorting, hashing, and grouping.

Optimization techniques 2: Simultaneous aggregation and caching of intermediate results.

Optimization techniques 3: Aggregation from the smallest child when there exist multiple child cuboids.

Optimization techniques 4: The Apriori pruning method can be explored to compute iceberg cubs efficiently.

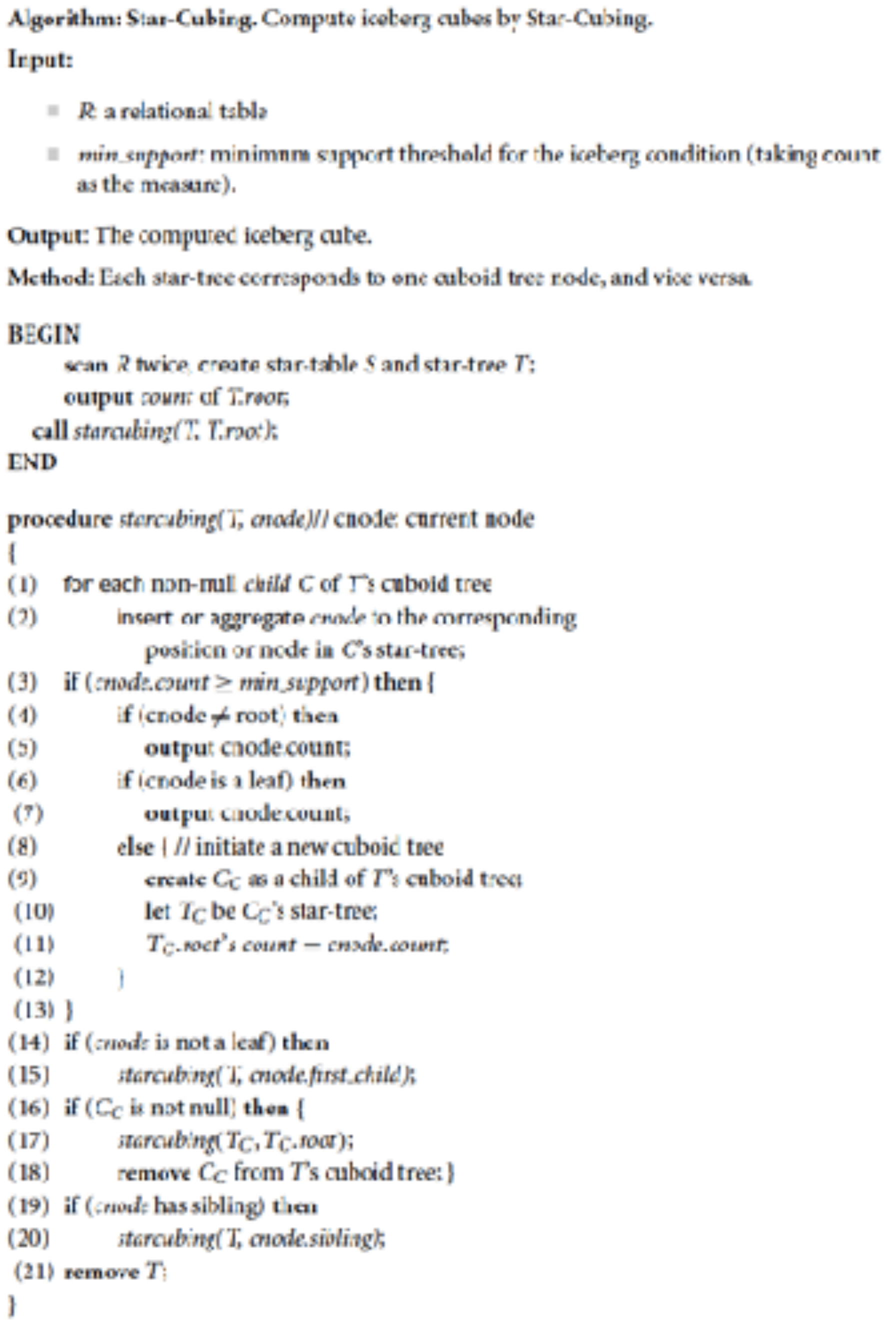
There are the following efficient approaches for data cube computation: Multiway array aggregation for full cube computation, BUC: computing iceberg cubes from the apex cuboid downward and Star-Cubing: computing iceberg cubes using a dynamic star-tree structure. Below is a table (Table 1) that summarized the major computational properties of the above three approaches.

Table 1: Summary of three Approaches (Algorithms)

Apporach(Algorithm)	Simultaneous Aggregation	Partition & Prune
MultiWay	Yes	No
BUC	No	Yes
Star-Cubing	Yes	Yes

Base on previous studies, Star-Cubing (FIG 1) combines the strengths of the other methods we have studied up to this point. It integrates top-down and bottom-up cube computation and explores both multidimensional aggregation and Apriori-like pruning. It performs lossless data compression on a data structure called a star-tree to achieve a goal of reducing the computation time and memory requirements.

FIG 1 : Star-Cubing Algorithm



Algorithm: Star-Cubing. Compute iceberg cubes by Star-Cubing.

Input:

- *R*: a relational table
- *min_support*: minimum support threshold for the iceberg condition (taking count as the measure).

Output: The computed iceberg cube.

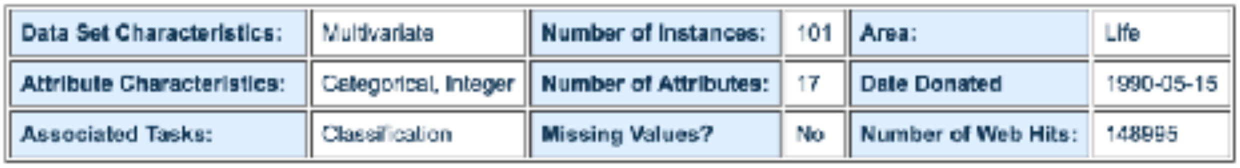
Method: Each star-tree corresponds to one cuboid tree node, and vice versa.

```
BEGIN
      scan R twice, create star-table S and star-tree T;
      output count of T.root;
   call starcubing(T, T.root);
END

procedure starcubing(T, cnode)// cnode: current node
{
(1)   for each non-null child C of T's cuboid tree
(2)         insert or aggregate cnode to the corresponding
               position or node in C's star-tree;
(3)   if (cnode.count ≥ min_support) then {
(4)         if (cnode ≠ root) then
(5)            output cnode.count;
(6)         if (cnode is a leaf) then
(7)            output cnode.count;
(8)         else { // initiate a new cuboid tree
(9)            create C_C as a child of T's cuboid tree
(10)           let T_C be C_C's star-tree;
(11)           T_C.root's count = cnode.count;
(12)        }
(13) }
(14) if (cnode is not a leaf) then
(15)        starcubing(T, cnode.first_child);
(16) if (C_C is not null) then {
(17)        starcubing(T_C, T_C.root);
(18)        remove C_C from T's cuboid tree; }
(19) if (cnode has sibling) then
(20)        starcubing(T, cnode.sibling);
(21) remove T;
}
```

Experimental Design

This project using Star-Cubing algorithm to classify a set of zoo data (Table 2) on https://archive.ics.uci.edu/ml/datasets/Zoo.

Table 2： Properties of Zoo Data



Data Set Characteristics:	Multivariate	Number of Instances:	101	Area:	Life
Attribute Characteristics:	Categorical, Integer	Number of Attributes:	17	Date Donated	1990-05-15
Associated Tasks:	Classification	Missing Values?	No	Number of Web Hits:	148995

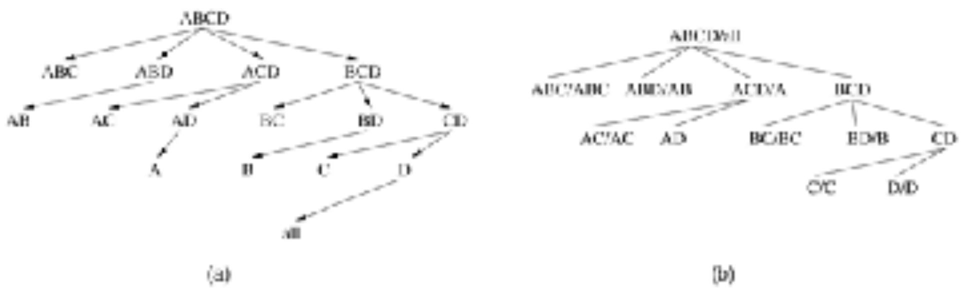
1. Pruning Shared Dimensions

Share Dimension is an important concept of the Star-Cubing algorithm. A set of shared dimensions is a set of the prefix dimensions of the parent node and the set of maximum prefix dimensions of all nodes in the subtree in a top-down cuboids spanning tree.

Taking ABCD as the base cuboid, the FIG 2 below shows Top-down computation with bottom-up growing shared dimensions. As the Fig display, ABD/AB means cuboid ABD has shared dimension AB.

FIG 2 Top-down computation with bottom-up growing shared dimensions

(a) Top-down and (b) integration of top-down and bottom-up.



Since the shared dimensions are identified in the tree expansion, so it makes no recomputing them later. Shared dimensions facilitates shared computation. If the value in the shared dimension A is a1 and it is not satisfy the iceberg condition, the whole subtree should be pruned since they are all more specialized versions of a1.

2. Star-tree Construction

For the input R （Table 3）which is a table of A, B,C and D dimensions and five tuples. Let the iceberg condition is min_support = 2, which makes only a1, a2, b1, c3, d4 meet condition, others are below the threshold and become star-nodes (Table 4).

Table 3：Input R - The Base Table

(Before Star Reduction)

A	B	C	D	count
a_1	b_1	c_1	d_1	1
a_1	b_1	c_4	d_3	1
a_1	b_2	c_2	d_2	1
a_2	b_3	c_3	d_4	1
a_2	b_4	c_3	d_4	1

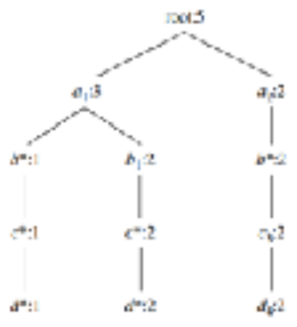
Table 4: Compressed Base Table

(After Star Reduction)

A	B	C	D	count
a_1	b_1	*	*	2
a_1	*	*	*	1
a_2	*	c_3	d_4	2

Base on the Compressed base table which after star reduction, create the Star-Tree (FIG 3).

FIG 3 Compressed Base Table Star-Tree



3. Star-Cubing

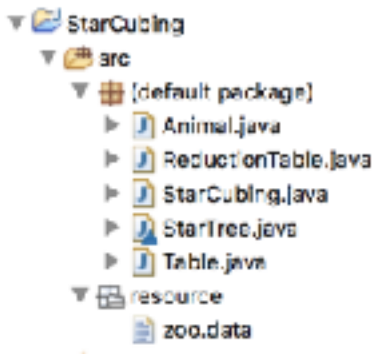
This is the final step which performs multi-way aggregation. It works by conducting a depth first search at the “root” of the start-tree above.

Implementation and Results

1. Implementation

The Star-Cubing algorithm implementation (FIG 5) contains 4 classes which deal with convert the original data to Objects, compress base table, create Star-tree and perform the final multi-way aggregation and thus finish calculating the iceberg cube.

FIG 5 Star-Cubing algorithm implementation



```
▼ StarCubing
  ▼ src
    ▼ (default package)
      ▶ Animal.java
      ▶ ReductionTable.java
      ▶ StarCubing.java
      ▶ StarTree.java
      ▶ Table.java
    ▼ resource
        zoo.data
```

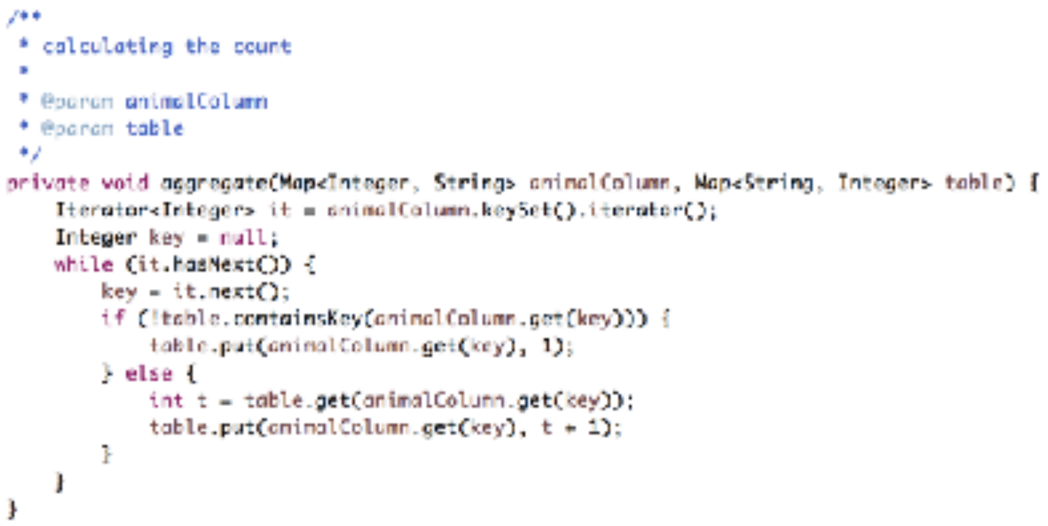
Step 1: Convert Original Data to Objects

In Animal. java, using class Animal to make the 18 properties of the zoo data set as an Objects to enable the subsequent processing.

step 2: Star-Table Reduction

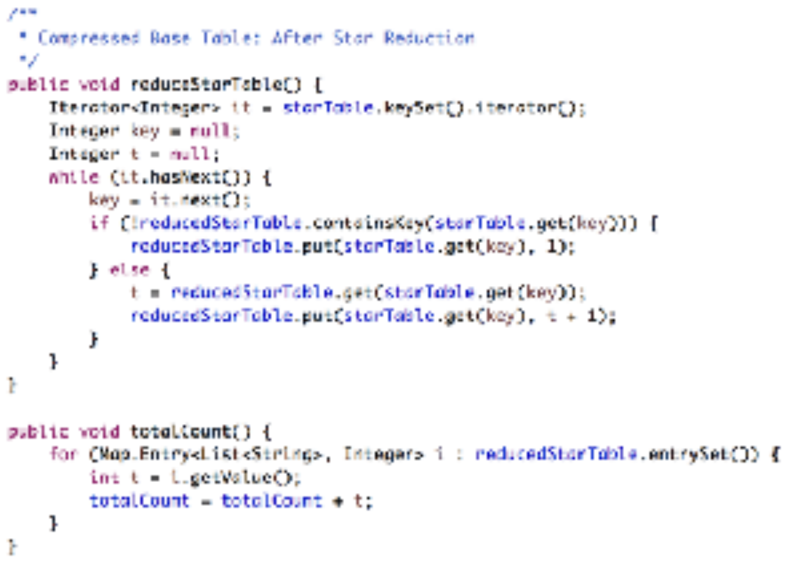
Creating and initialize the Star-table, and then calculating the count of original data enable to perform the computation of one-dimensional aggregates for all attributes(FIG 6). By applying the min_sup in the iceberg condition and collapsing star-nodes to generate the reduced base table(FIG 7).

FIG 6 One-Dimensional Aggregates



```
/**
 * calculating the count
 *
 * @param animalColumn
 * @param table
 */
private void aggregate(Map<Integer, String> animalColumn, Map<String, Integer> table) {
    Iterator<Integer> it = animalColumn.keySet().iterator();
    Integer key = null;
    while (it.hasNext()) {
        key = it.next();
        if (!table.containsKey(animalColumn.get(key))) {
            table.put(animalColumn.get(key), 1);
        } else {
            int t = table.get(animalColumn.get(key));
            table.put(animalColumn.get(key), t + 1);
        }
    }
}
```

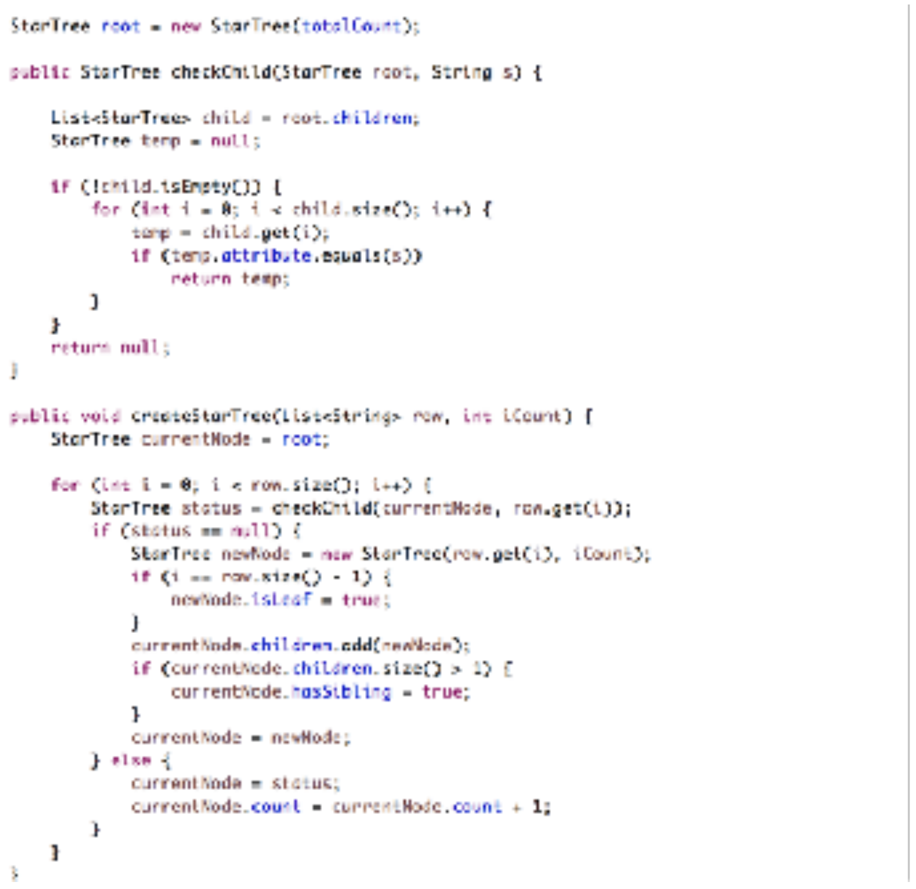
FIG 7 Compress Base Table by Star Reduction



```
/**
 * Compressed Base Table: After Star Reduction
 */
public void reduceStarTable() {
    Iterator<Integer> it = starTable.keySet().iterator();
    Integer key = null;
    Integer t = null;
    while (it.hasNext()) {
        key = it.next();
        if (!reducedStarTable.containsKey(starTable.get(key))) {
            reducedStarTable.put(starTable.get(key), 1);
        } else {
            t = reducedStarTable.get(starTable.get(key));
            reducedStarTable.put(starTable.get(key), t + 1);
        }
    }
}

public void totalCount() {
    for (Map.Entry<List<String>, Integer> i : reducedStarTable.entrySet()) {
        int t = i.getValue();
        totalCount = totalCount + t;
    }
}
```

Step 3: Create the Star-Tree (FIG 8)

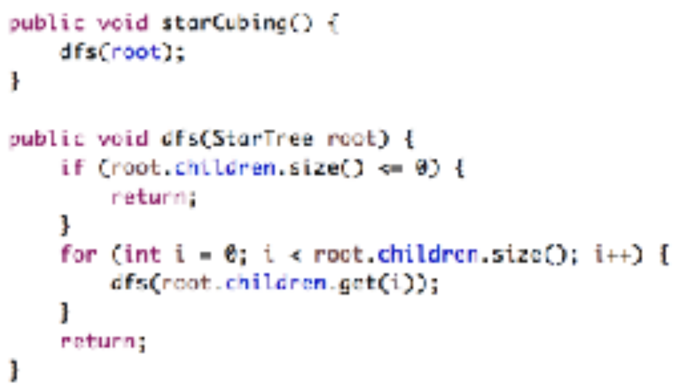
FIG 8 Star-Tree



```
StarTree root = new StarTree(totalCount);

public StarTree checkChild(StarTree root, String s) {

    List<StarTree> child = root.children;
    StarTree temp = null;

    if (!child.isEmpty()) {
        for (int i = 0; i < child.size(); i++) {
            temp = child.get(i);
            if (temp.attribute.equals(s))
                return temp;
        }
    }
    return null;
}

public void createStarTree(List<String> row, int iCount) {
    StarTree currentNode = root;

    for (int i = 0; i < row.size(); i++) {
        StarTree status = checkChild(currentNode, row.get(i));
        if (status == null) {
            StarTree newNode = new StarTree(row.get(i), iCount);
            if (i == row.size() - 1) {
                newNode.isLeaf = true;
            }
            currentNode.children.add(newNode);
            if (currentNode.children.size() > 1) {
                currentNode.hasSibling = true;
            }
            currentNode = newNode;
        } else {
            currentNode = status;
            currentNode.count = currentNode.count + 1;
        }
    }
}
```

Step 4: Star-Cubing (FIG 9), conducting a depth first search at the “root” of the start-tree to performs multi-way aggregation

FIG 9 Star-Cubing



```
public void starCubing() {
    dfs(root);
}

public void dfs(StarTree root) {
    if (root.children.size() <= 0) {
        return;
    }
    for (int i = 0; i < root.children.size(); i++) {
        dfs(root.children.get(i));
    }
    return;
}
```

2. Results

Enter 5 for the min_sup in the iceberg condition, the run time is 0.014s and the Star table and Compressed Base Table attached as an .txt attachment named Appendix A .

Conclusion

The Star-Cubing algorithm successfully apply to data sets. It achieve the goal of computing the Iceberg Cube with saving time and memory space. However, sometimes, when star tree gets very wide, it makes more time in construction, traversal and aggregation and pruning.