ESE 589 Star-Cubing Algorithm

Fedrik Durao 103302962, Nehul Oswal 112820228

Introduction:

The Star Cubing Algorithm in this project's context is to produce cuboids that pass the iceberg condition determined by the user. The algorithm can be split into 3 parts: the creation and reduction of the star table, the creation of the star tree, and traversal of the star tree. The critical measurements used to evaluate the algorithm are execution time, memory usage, dimensionality of data, and number of data entries. The graphs of these measurements, and the relationships inferred by the graphs will be discussed later in the report.

Implementation:

1) Reduced Star Table
   a) readData() function creates a dictionary of all the elements in the DB with key value of the dictionary as the row number.
   b) readData_Count() function creates a nested dictionary in order to maintain a count of every unique element in each column.
   c) checkIceberg() function abstracts the data elements in the nested dictionary which do not satisfy the Iceberg condition.
   d) UpdateData() function iterates through the keys of the nested dictionary and abstracts the data elements stored in the dictionary created in the first step.
   e) Reduced_table() function creates the reduced star table(i.e. dictionary of tuples). This function utilizes the data structure set to identify unique tuples as set data structure can contain only unique tuples. This dictionary of tuples is then lexicographically sorted to obtain the reduced star table.
2) Star Tree Creation
   a) After the reduced star table has been computed and sorted, the result is passed into the makeTree() function which creates the star tree representation of the reduced star table. The nodes of the star tree have properties such as .elem, .children[], .parent, and .childNum which are necessary for the creation of the star tree. The other node properties are used during the tree traversal process.
   b) makeTree() function iterates through the reduced star table, and for every entry (or tuple) the loop will add every element in the tuple as a child of the root or a child of what was last inserted during that specific tuple's iteration. It also updates the count of the root as more children are added to the root. The actual insertion of each of these elements into the tree is done by the insertChild() function, which is called by the makeTree() function.
   c) insertChild() function takes in a new element to be added, the parent node, and the count of the new element. It will assign the parent with a new child with the appropriate properties (count, level, element, and parent) and update the '.childNum' of the parent to reflect the amount of children it now has. If the most recent child of the parent is the same as the new element, a new node is not produced, but instead the count of the existing child is updated and the existing child is returned instead of a new child. This function only looks at the most recent added child because it assumes the table is sorted,

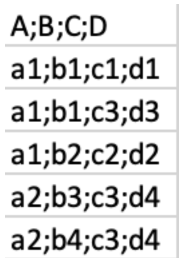
in which only the most recent element added (to that parent) will have a similar value to the next, as opposed to the earlier elements. When makeTree() finishes its iterations through the entries, it returns the root of the tree.

3) Star Tree Traversal
   a) traversal() function is given the root of the tree made by makeTree(), the previous reduced star table, and an iceberg condition. The root is used to start the traversal, the star table is used to calculate the dimensionality of the sub trees to be calculated, and the iceberg to be applied for pruning. This function loops with creatSubTrees() and goUp() until there are no longer any sub trees (which is the end of the traversal).
   b) createSubTrees() function takes in a tree node, a dimension length, an iceberg, a list of nodes traversed, and a list of sub trees. While the tree node has a child, this function will prune through the siblings of the child recursively, until a passing sibling is found (failed siblings get deleted). With the returned child that passes the iceberg, the child is passed into the insertIntoSubTrees() function, which for every existing sub tree, inserts the current node as a child of the tree's '.lastInsertion' child. Only the roots of the sub trees have this property, which is updated when a new node is inserted. Afterwards, a new sub tree is created by the 'getSubRoot' function which looks at the nodes that have been traversed, and the dimensionality of the database, to produce the corresponding new sub tree root. Once returned, the new sub tree is appended into the sub tree list and the current node (that has been inserted to all previous sub trees) is appended to the traversed nodes list. Then the creatSubTrees() function loops this whole process until a leaf is hit or no children at a node pass pruning.
   c) goUp() function executes explicitly after the createSubTrees() function is called in the 'traversal' function. This takes in the current node, list of sub trees, list of traversed nodes, and the iceberg. While the current node does not have a sibling it will iterate through each parent node until a node has a sibling. For every iteration the nodes traversed gets popped, the most recent sub tree gets printed and popped from the sub tree list. When a the sibling is found, it is inserted into the sub trees and the function is done.

Code Verification:

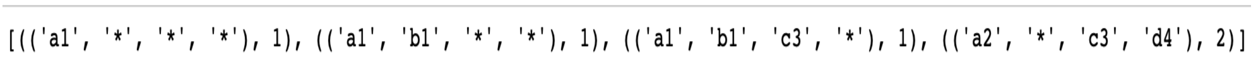
Before taking measurements of the algorithm for different databases, the algorithm was tested on smaller more readable databases made by hand. Similar to the one used in the textbook and class examples. As changes were made to the simple database, star cubing was done by hand and then compared to the results for the algorithm for each three parts. Many potential errors were eliminated while changing the small database and debugging the code. But there were indeed a few more errors afterwards when working with the large legitimate databases. These errors were eliminated after using the larger databases. Most of those late errors were related to traversing up the star tree early due to pruning. Without a large database, it was difficult to visualize problems with high pruning, as it wasn't verified exhaustively when testing the algorithm on the small database.

Below is our results of Star Cubing based on a simple database, modeled after class examples.



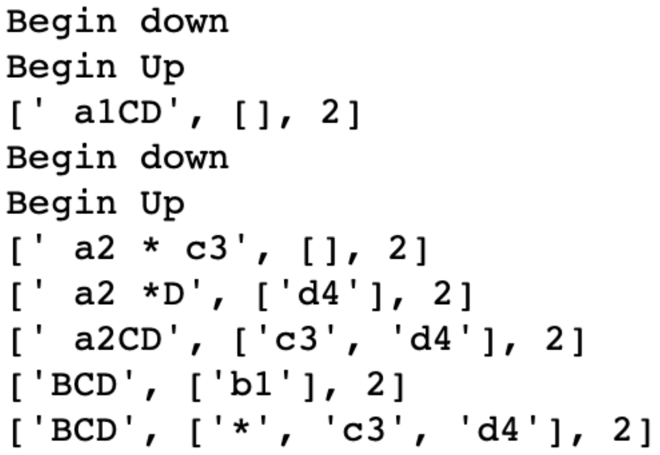
A;B;C;D
a1;b1;c1;d1
a1;b1;c3;d3
a1;b2;c2;d2
a2;b3;c3;d4
a2;b4;c3;d4

Figure 1: Original Database



```
[(('a1', '*', '*', '*'), 1), (('a1', 'b1', '*', '*'), 1), (('a1', 'b1', 'c3', '*'), 1), (('a2', '*', 'c3', 'd4'), 2)]
```

Figure 2: Reduced Star Table



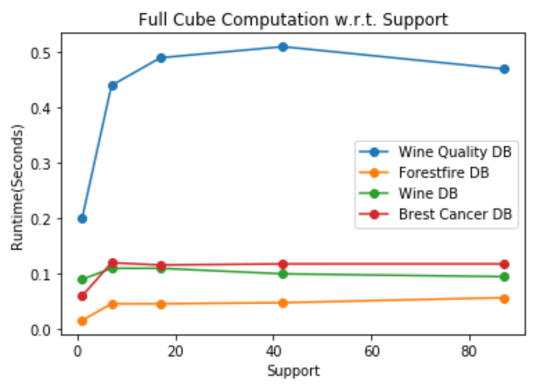
```
Begin down
Begin Up
[' a1CD', [], 2]
Begin down
Begin Up
[' a2 * c3', [], 2]
[' a2 *D', ['d4'], 2]
[' a2CD', ['c3', 'd4'], 2]
['BCD', ['b1'], 2]
['BCD', ['*', 'c3', 'd4'], 2]
```

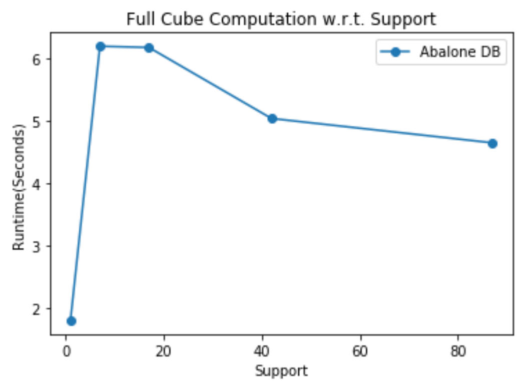
Figure 3: Star Tree Traversal Cuboids w/ pruning

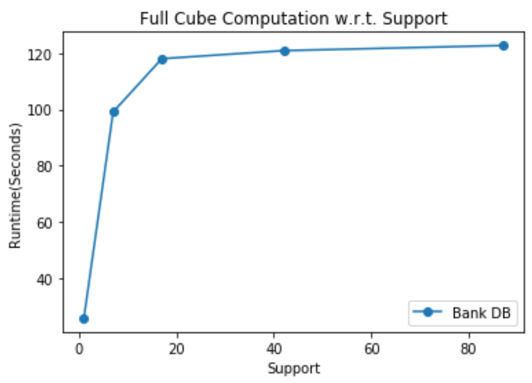
<u>Graphs and Relationships</u>:

Five relationships were depicted from graphing the results of Star Cubing the six databases: Wine_Quality, Wine, Forestfires, Breast_Cancer, Abalone, and Bank. The relationships observed are execution time v. entries, time v. dimensions, time v. support, memory v. entries, and cuboids v. support. These results were produced on a laptop with 12GB of RAM and an i7 processor.

1) Time v. Support

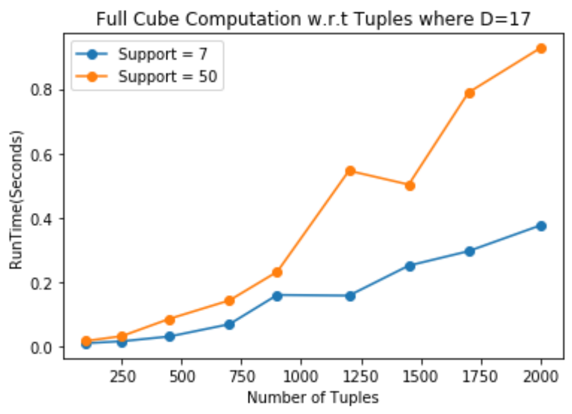


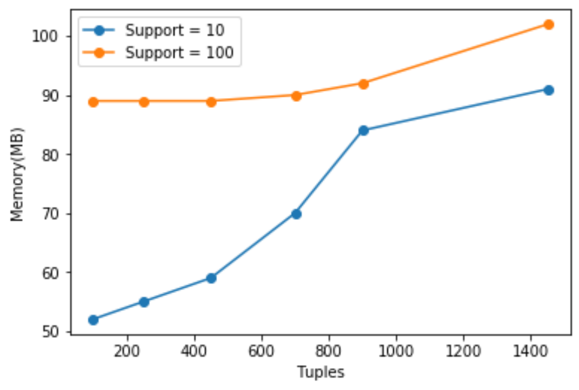




The results were expected to have runtime decrease as the support increases. This holds true for some instances in these graphs. As support increases, the graphs flatten out as expected. But there seems to be issues with this implementation for low support values, where instead of runtime being high, and begins to decrease with support, it starts out initially low. When analyzing the Cuboid v Support graph, this suggest that the algorithm has high complexity when dealing with pruning. Also the runtime for the Bank database was much higher then the rest of the databases, suggesting that Star cubing is not the optimal algorithm for this database.

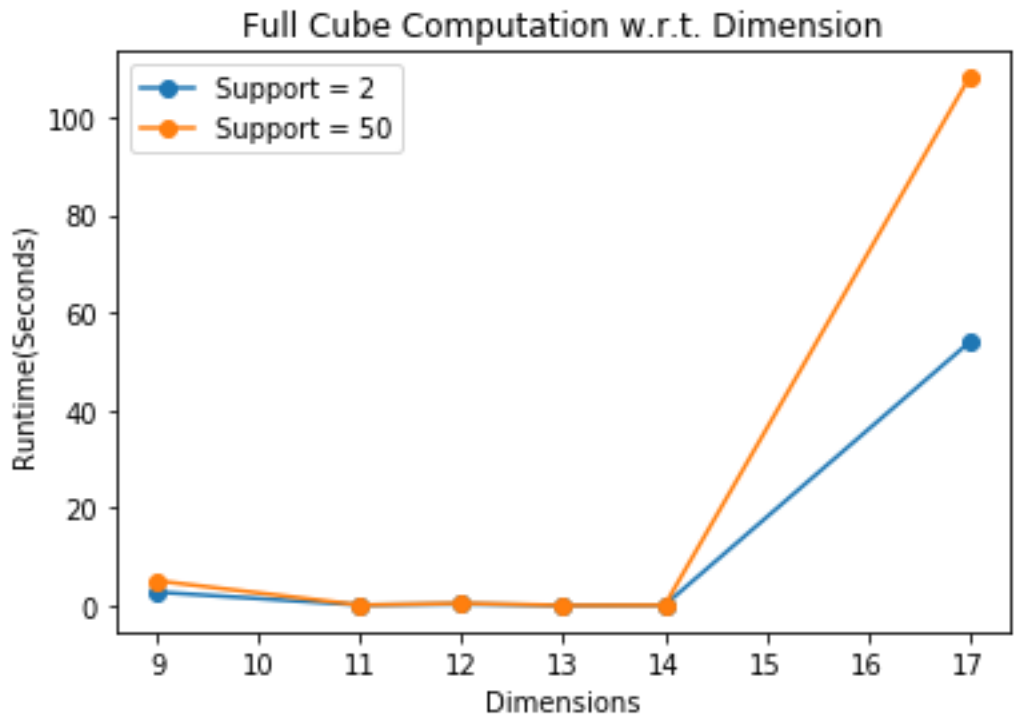
2) Time v. Entries (or Tuples), and Memory v. Entries





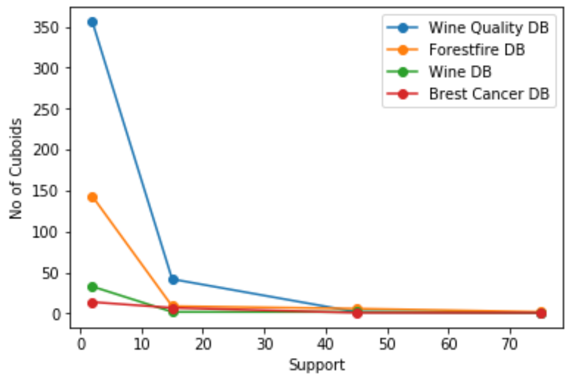
As expected, the runtime increases for increasing number of data entries. Although with larger supports, the runtime increases, just as it did in the previous graphs. Memory usage seems to be increasing with the amount of data entries, naturally. Although again the same issue occurs where for larger supports, memory usage increases instead of decreases. This suggests that the pruning is not effectively deleting non-useful data.

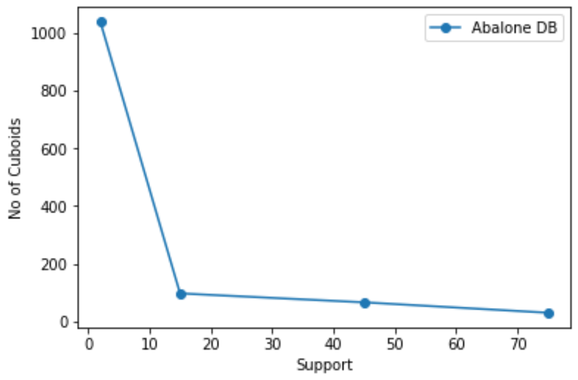
3) Time v. Dimensions



As the dimensionality of a database increases, so does the time to compute the star cuboids. The graph above agrees with this, although up to a certain point the runtime remains constant. Suggesting that runtime degradation was minimal for small dimensionalities. And again the data is inferring that larger supports increase runtime, which is the fault of this implementation.

4) Number of Cuboids v. Support





With increasing support, the number of resulting cuboids should decrease, since cuboids of small support contains entries of larger supports but not vice versa. This is accurately illustrated from these two graphs. Since the resulting cuboids do decrease with support, that suggests that the algorithm is pruning correctly. But the higher runtimes with increasing supports suggests that this implementation is pruning correctly but not effectively.

Challenges and Revelations:

Most challenges with the algorithm came from the traversal of the star tree part. The creation of the reduced star table and star tree were trivial in comparison. Creating correct sub tree roots and adding nodes at the correct positions on the sub trees proved to be very difficult. To create a correct sub tree root, all traversed nodes need to be recorded accurately while traversing down or up the tree. And to insert nodes at the correct location in a sub tree, the most recently inserted node and depth of the node to be accounted for, which can change while traversing up or down the star tree.

Before implementing the Star Cubing Algorithm, the value in creating a star tree and traversing it was not quite clear to the group. The reduced star table seemed reduced as it was. But after trying to implement the tree traversal step with pruning, it became clear how significant this step is to the algorithm. While the reduced star table seems very reduced, when the data is recorded as a star tree, the data is overlayed in a way that makes it easier to visualize where more pruning can occur. Thus offering more reduction.

Some things may have been improved with this implementation of the Star cubing Algorithm. The pruning done during tree traversal still needs some fine tuning, since it is the primary suspect for increasing runtime despite producing fewer cuboids for increasing supports. The memory management was ill maintained. Despite removing nodes from the star tree or sub trees, these objects were not truly being destroyed and removed in memory. In Java, unused objects are destroyed, but that luxury is not available in Python and was realized during the time of writing this report. Also, during the traversal up the star tree, when a sibling was found, it was not pruned to find the ideal sibling that passes the iceberg. This lead to slight inefficiencies that could have been solved by more timely pruning. Also, if the reduced star tables' dimensions were sorted with descending cardinality first and then sorted lexicographically (only lexicographic sorting was implemented), then the execution time of the tree traversal would be improved. This will lead to larger base trees that are closer to the star tree root, and smaller trees further from the root. This will lead to earlier pruning of nodes that would otherwise get pruned only later in the algorithm. Thus resulting in a quicker execution time. Also, there was no discretization done to improve the effectiveness of the algorithm. Most time was spent getting the algorithm to work and plotting the results. If data discretization was implemented, the execution time of the reduced star table production would have been improved. By splitting the database into sections with similar data, and producing quicker reduced star tables, then concatenating the tables, and then reducing it further would yield faster reduced star table execution.

Grid Implementation:

kyvkvlkuvu