
# ESE 589: LEARNING SYSTEMS

# PROJECT 1:
# IMPLEMENTATION OF STAR-CUBING ALGORITHM

## GUIDED BY:
## Dr. ALEX DOBOLI

PREPARED BY:

VARUN JAIN (111685202)
AMOD GANDHE (111678938)
SHASHANK RAO (111471609)

# I. Abstract

There is a massive amount of data getting generated each day - this statement is supported by one study which suggests that the amount of data that is being generated today in five minutes is equivalent to the data generated during the whole calendar year of 2003. This growing information set starts imposing constraints on extracting useful information out of large datasets. Constraints may be of a type, time, computational overhead on the system or many more aspects. Repetitive calculations on a dataset are not the best way to handle them. Hence, to reduce repetitive traversal over the same data, the concept of data cubes was formulated, which in turn generalized the information provided in the raw data set. Though generating data cube is helpful, data cube computation in itself is a resource consuming task to perform. This project deals with generalizing the data and computing data cubes efficiently by taking help of 'Star-Cubing' algorithm. Outcomes of projects are evaluated by considering various datasets as a reference. Changing Iceberg condition, change in results due to normalized and non-normalized data are some of the factors that are considered while evaluating the accuracy of code developed. Remaining part of the report will give a brief description about various cube computation algorithms and then will discuss the experimental results based on the proposed solution to the problem.

# II. Introduction

Ever since development in the field of data warehousing and OLAP, various methods were proposed - Iceberg cube computation, computation of compressed data cube, selective materialization of the cube - for efficient computation of data cubes. Different methods have its own use cases, but star-cubing prefers computation of full or Iceberg cubes. Star-cubing - given an m-dimensional data cube - looks to compute full or n-dimensional cubes which satisfy iceberg condition. In this algorithm, Iceberg condition works as a threshold, any cell that falls below this condition need not be considered while performing next calculations. Discussion about Iceberg condition is made in depth after a couple of sections.

MultiWay Aggregation, BUC, etc. are some of the cube computation algorithms which follow either top-down or bottom-up approach, while star-cubing leverages on both top-down and bottom-up approach. In case of MultiWay Aggregation, aggregation is done simultaneously over multiple dimensions but a downside of this algorithm is that it cannot rely on Apriori pruning. In contrast, BUC is based on bottom-up approach but cannot aggregate over multiple dimensions. star-cubing improves its efficiency, as it implements both top-down and bottom-up approach. This means that it implements upsides of both MultiWay Aggregation and BUC. For its implementation, Star Tree data structure is defined which performs lossless data compression and this will be evaluated towards the end of the report. Next section of the report will talk about related work by referring to data cube computation algorithms such as MultiWay Aggregation and BUC and will also compare these algorithms with star-cubing.

## III. Related Work

This section of the report provides a brief survey of various algorithms for data cube computation. As star-cubing follows both top-down as well as bottom-up algorithms, we believe, the intuition behind both approaches must be clear before going into details of the algorithm and its implementation. Hence, we will provide brief discussion about following algorithms:

- **MultiWay Aggregation**
- **BUC**

***MultiWay Aggregation***: It is an array-based top-down cubing algorithm. It uses multi-dimensional array structure to represent base cuboid and other sparse cubes. Chunks are a major part of this algorithm, as it reduces the memory overload of the system. It divides the array into chunks, which is nothing but a sub-cube that can fit into available memory for cube computation. The top-down approach for MultiWay aggregation can be realized from the following example:

Consider data set with 4 attributes/dimensions → A, B, C, D.

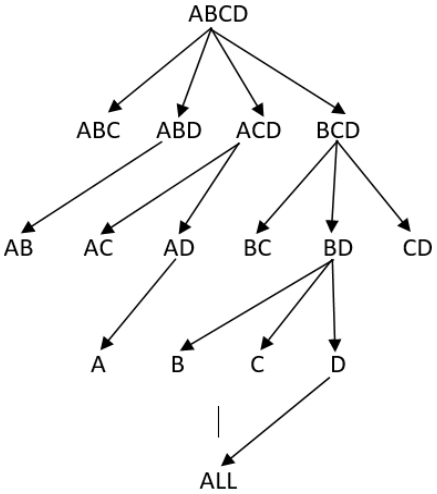



Fig.1: Top-Down Approach

It can be inferred from the diagram that MultiWay Aggregation starts with base cuboid by considering all dimensions and then travels towards apex cuboid. Also, it can benefit from shared dimension computation by storing intermediate results. But it fails in the case when the dimensionality of the dataset is so large that it cannot fit into adjacent memory elements. In addition to that, it cannot take advantage of Apriori pruning.

***BUC:*** BUC employs bottom-up approach by starting computation from apex cuboid and moving forward towards base cuboid. In MultiWay, a base cuboid is the parent of all cuboids, in contrast, in BUC, cuboids with fewer dimensions become a parent to the cuboid having more dimension. Adopting bottom-up approach, BUC starts with a single dimension and then reaches to base cuboid following Apriori pruning. The diagram below illustrates logic behind the bottom-up approach.

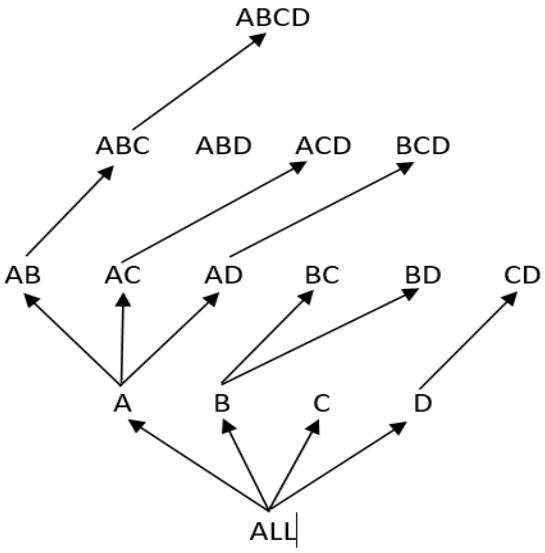



Fig.2: Bottom-Up Approach

BUC leverages on Apriori pruning which does not care about a cell which does not satisfy some threshold condition and hence, reduces the computation of cubes which probably do not contain useful information. Partitioning and aggregation are the major costs in BUC's cube computation since recursive partitioning does not reduce the input size. Moreover, BUC is sensitive to skew in the data - the performance of BUC degrades as skew increases.

Both the algorithms discussed below follow either of the approaches and hence have some restriction. However, star-cubing is based on both the approaches and so, it provides multidimensional simultaneous aggregation of data. Table No.1 provides a quick comparison between methodologies followed by algorithms discussed before.

| **Algorithm** | **Top-Down Approach** | **Bottom-Up Approach** |
|---|---|---|
| MultiWay Aggregation | Used | Not Used |
| BUC | Not Used | Used |
| Star-Cubing | Used | Used |

Table No.1: General comparison between Algorithms

# IV. Generic Algorithm

Star-cubing algorithm integrates bottom-up and top-down computational models in the following way – while computing globally, it uses bottom-up model; however, there is a sub layer underneath which is based on a top-down model, which explores the notion of shared dimensions. Implementation of the star-cubing algorithm involves the following generic steps:

***Obtain the compressed base table:***

Consider a given dataset with T tuples of A attributes. Suppose that the cardinality of each attribute is an element of the set $\{C_1, C_2, C_3, \ldots, C_A\}$, where $C_1$ is the cardinality of the first attribute, $C_2$ is the cardinality of the second attribute and so on, i.e., each attribute has a different cardinality. For each attribute, the number of occurrences of different values is calculated using Hash Map. Occurrences less than Iceberg condition are replaced with * values. For example, let Iceberg condition be *i*. This means that if attribute 1 has some value (out of the $C_1$ different types) with an occurrence lesser than *i*, then those occurrences are replaced with '*' since they do not meet the minimum threshold (Iceberg condition). Effectively, all the cardinal elements of an attribute which does not meet the minimum required threshold are marked as * in the table. By merging these cardinal values, we can build a highly compressed table. While replacing these cardinal values with *, they are saved in the star-table.

***Create the star-tree data structure:***

Using the compressed table, we form the star-tree data structure. The structure initially has a root with an aggregate count of the total number of tuples in the dataset. Its next level will contain all cardinal elements of the first attribute in lexical order, i.e., if $v_1$, v*, $v_3$, …., $v_A$ are the different cardinal elements of the first attribute, then they will be placed under the root with the leftmost branch as *, followed by $v_1$, followed by $v_3$, and so on. This followed by the all the cardinal elements of the second attribute. But now, in this case, the cardinal elements of the second attribute will be considered only after considering the values of the first attribute. This means that if w*, w*, $w_3$, $w_4$, …., $w_A$ are the different cardinal elements of the second attribute and there are data tuples with values $v_1$ and w*, then only w* will be placed in the next level with $v_1$ as the parent. This gives rise to a tree structure where each node has a parent and corresponding children as per the tuples in the compressed dataset. Each node will also be linked to its siblings if there are two or more children with the same parent. These nodes will also be linked to each other.

***Multi-way star-tree aggregation:***

Once we have obtained the star-tree structure, we begin to run the star-cubing algorithm on this structure. The algorithm starts with a top-down traversal of the tree, like DFS. DFS traversal on current node, a new star tree is built with root containing values of all its parents and all its child branches added to this root. Next recursive call is made to its first child. These trees are saved provided they pass the iceberg condition. This will ensure Apriori pruning of the star-tree structure.

When the recursive calls reach a leaf node, it begins to backtrack along the created child trees. As the algorithm backtracks, if there are no sibling nodes for the current node, then it will backtrack to the previous function call after outputting and destroy its aggregate child tree. However, if the current node during backtracking contains a sibling node, then its aggregate tree is not destroyed and further aggregated with the values of the sibling nodes. This way all possible combinations of the dataset, along with their aggregate values, can be effectively output while pruning the non-relevant datasets using star-cubing algorithm.

***Data pre-processing:***

This generic algorithm, while applicable to all datasets, requires the dataset itself to meet certain criteria. For example, in case of a numerical dataset, where there are different attributes with varying range of values, it is possible that no value in the given dataset will recur. This will lead to the creation of a large and useless star-tree data structure, where possibly none of the nodes pass the iceberg condition. It is also possible for incorrect data values to be present in some cells of the dataset. While nominal datasets do not have any problems regarding the range of values, they still suffer from incorrect or empty values in some cells. All this means that the given dataset needs to be pre-processed to remove incorrect values, fill up empty values, remove outliers and normalize the dataset values to a particular range and cardinality.

# V. Implementation

To simplify all the process involved in data cube generation, we decided to split the entire task into 4 smaller sub-tasks to achieve maximum efficiency. The subtasks are as follows:

- Dataset pre-processing
- Compressed Base Table creation
- Star tree and Star Table generation
- Generation of cubes

Before starting with these steps, we call method *readCSV()* or *readXLSX()* depending on the type of the input file to be parsed. At this initial level, these values are stored in an ArrayList of Dimension - Dimension being a java class encapsulating all an attribute's values along with its name. In a way, this structure can be considered as a modified version of 2D ArrayList. This structure makes the code more generic, regardless of the number of attributes present in the dataset.

***Dataset Pre-processing:***

If the given dataset contains errors/outliers, these need to be corrected. We have corrected such values by calling *normalizeList()* method. Within this method, we calculate the mean and replace the incorrect values with the mean. We then normalize the attribute to a pre-defined cardinality. This can be realized by the below example, where -200 refers to error data values collected by the sensor:

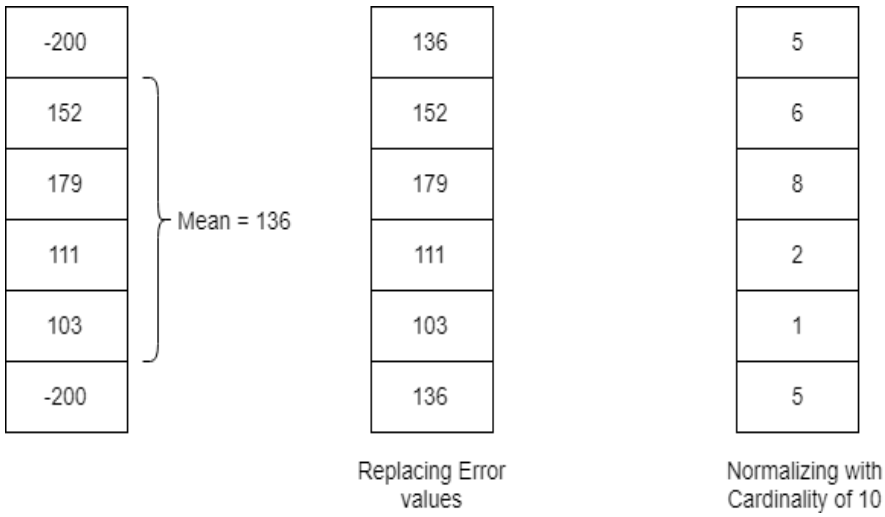



Fig.3: Error Minimization and Normalization

***Compressed Base Table creation:***

This task starts with replacing attribute values that do not follow Iceberg condition by '*'. This task is performed in *dimensionIceBCheck()* method which is defined inside 'Algorithm' class. There are still repeated tuples present in the normalized dataset, to generate Compressed Base Table, we rely on *.compressedBaseTable()* method. By using HashMap for counting the occurrence of each tuple, the execution time is immensely reduced (by a factor of O(n*n)) as it is completed in one iteration. Here, a tuple is the key and occurrence count as a value in HashMap. Compressed Base Table can be realized through this HashMap.

***Star-Tree Generation:***

Star-Tree can be generated after having the compressed base table. To make Star-Tree realizable and easier to access, we leverage on our own custom class, 'Cuboid' which has fields -

- *children* : HashMap of class 'Cuboid',
- *aggrVal*: It defines an aggregate value for current node and
- *attrValue*: this represents the actual attribute value of the current node.

Star Tree is built by method *cuboidTree()* residing inside *Algorithm* class. It starts iterating over the tuples of the compressed base table, and every attribute value within each tuple. Root is a dummy node. For each attribute a *Cuboid* node is created. Each node attribute in tuple is added in *children* map field of its previous attribute node recursively, first being added to root's children. While adding the child, the parent node's aggregate value *aggrVal* is incremented with the tuple occurrence count. When all attributes of tuple are added, next tuple is picked up. If the current attribute already exists in the parent node, then no new node is created and only the aggregate value of parent is increased. After exhausting all tuples of compressed base table, the star-tree is completely built.

***Multi-Way Star-Tree Aggregation:***

Using depth-first traversal, at each node a new star-tree is built with root node containing values of parent node attribute values by calling method *starCubing()* residing inside *Algorithm* class. Example, at level C in ABCDE star-tree if traversed from root→a6→b*, then root for new aggregate tree would be a6,b*,D,E/a6,b*. The children of current node are added to this root and its aggregate value is incremented correspondingly. Next, the first child is called recursively. At each call all children are added to the aggregated trees built previously and a new tree is also built. While back tracking, aggregate tree is outputted and destroyed if no siblings exist. Otherwise the tree is aggregated with the children of the siblings. Also sub-trees which do not satisfy iceberg condition and minimum support condition are pruned and not outputted.

Flow of code can be easily understood from example given below.

## VI. Example illustrating our implementation

We have demonstrated a small dataset example to illustrate the way in which our implementation works. Consider the following dataset (Table No.2) containing the quantity of different items purchased at a particular retail store between a particular time-range:

| Bread (# of loaves) | Milk (# of cartons) | Vegatables (Weight in kg) | Type of vegetables | Beer (# of cartons, each of six cans) | Small Household Items, Eg.- Knives, spoons, etc. (total # of different items purchased) |
|---|---|---|---|---|---|
| 1 | 2 | 6.4 | Potato | 2 | 1 |
| 1 | 1 | 2.1 | Tomato | 4 | 0 |
| 3 | 4 | 8.7 | Onion | 3 | 2 |
| 2 | 2 | 0.5 | Onion | 1 | 0 |
| 4 | 3 | 9.2 | Potato | 2 | 0 |
| 2 | 6 | 4.0 | Onion | 0 | 4 |
| 4 | 7 | 6.3 | Cabbage | 8 | 8 |
| 1 | 1 | 1.7 | Onion | 3 | 0 |
| 0 | 3 | 3.8 | Tomato | 0 | 1 |
| 0 | 2 | 4.0 | Potato | 2 | 8 |
| 2 | 5 | 7.2 | Tomato | 1 | 0 |
| 2 | 6 | 4.0 | Onion | 0 | 3 |

Table No.2: Given Example dataset

The following figure (Fig.4) gives an example .csv file created using the above dataset which will be parsed:

```
Bread;Milk;Vegetables;TypesOfVegetables;Beer;SmallHouseholdItems;;
1;2;6.4;Potato;2;1;;
1;1;2.1;Tomato;4;0;;
3;4;8.7;Onion;3;2;;
2;2;0.5;Onion;1;0;;
4;3;9.2;Potato;2;0;;
2;6;4.0;Onion;0;4;;
4;7;6.3;Cabbage;8;8;;
1;1;1.7;Onion;3;0;;
0;3;3.8;Tomato;0;1;;
0;2;4.0;Potato;2;8;;
2;5;7.2;Tomato;1;0;;
2;6;4.0;Onion;0;3;;
```

Fig.4: Example .csv file

In our implementation, this dataset given to us is first parsed and stored in memory as an ArrayList of dimension values. The rows store each of the tuples present in the dataset. *Util.readCSV()* function has been implemented to perform this functionality. The following figure (Fig.5) shows the data read from the dataset and stored in memory:

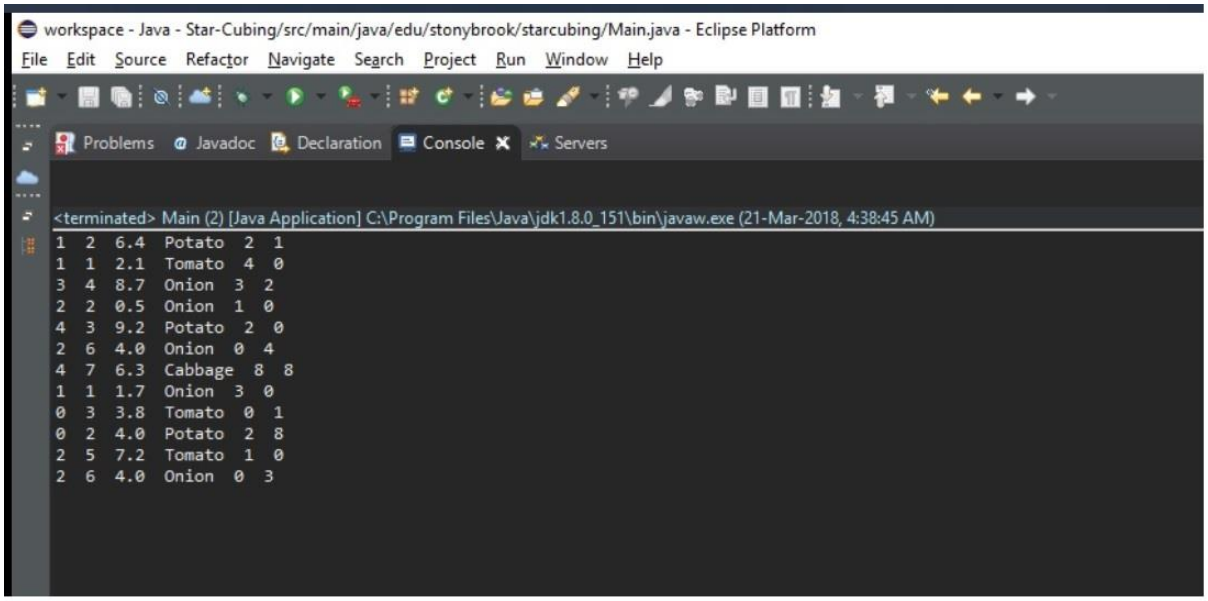



Fig.5: Data read and stored from .csv file

The given dataset is both numerical as well as nominal, and has a wide range of values for different attributes. As per our implementation, a numerical dataset will be first pre-processed so as to bring about a uniform range of values and to remove inconsistencies. This pre-processing for numerical dataset mainly involves normalization. In our example we have normalized all the attributes to a common range of 0-10. This has been performed by *DataPreProcessing.normalizeTable()* function. The nominal values are left untouched. Then the normalized values (using Min-Max Normalization) which get stored are shown in the following figure (Fig.6):

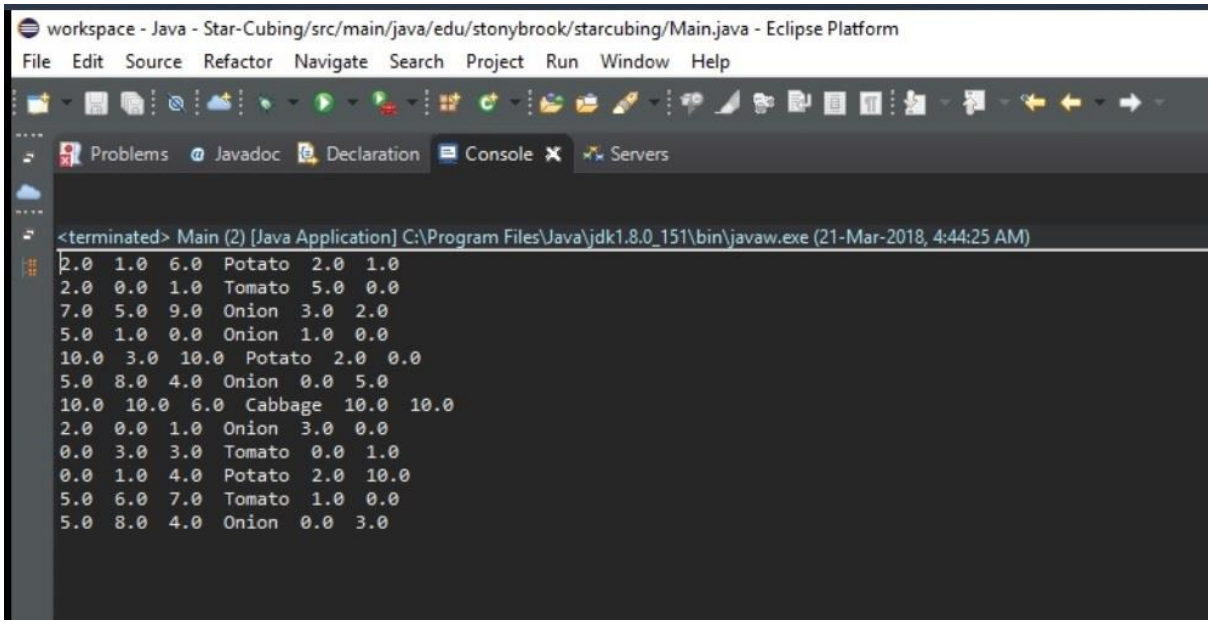



Fig.6: Given Example dataset after Normalization

These values are used for further processing. In the next step, we replace cells which do not pass Iceberg condition with <column_label>*. In our example, we have taken the count measure of 2 as the Iceberg condition. This means that, for every attribute, values occurring less than 2 times is replaced with <column_label>*. In our code, *Algorithm.replaceDimensions()* function is used to perform this functionality. The following figure (Fig.7) shows the table stored in memory after applying Iceberg condition:

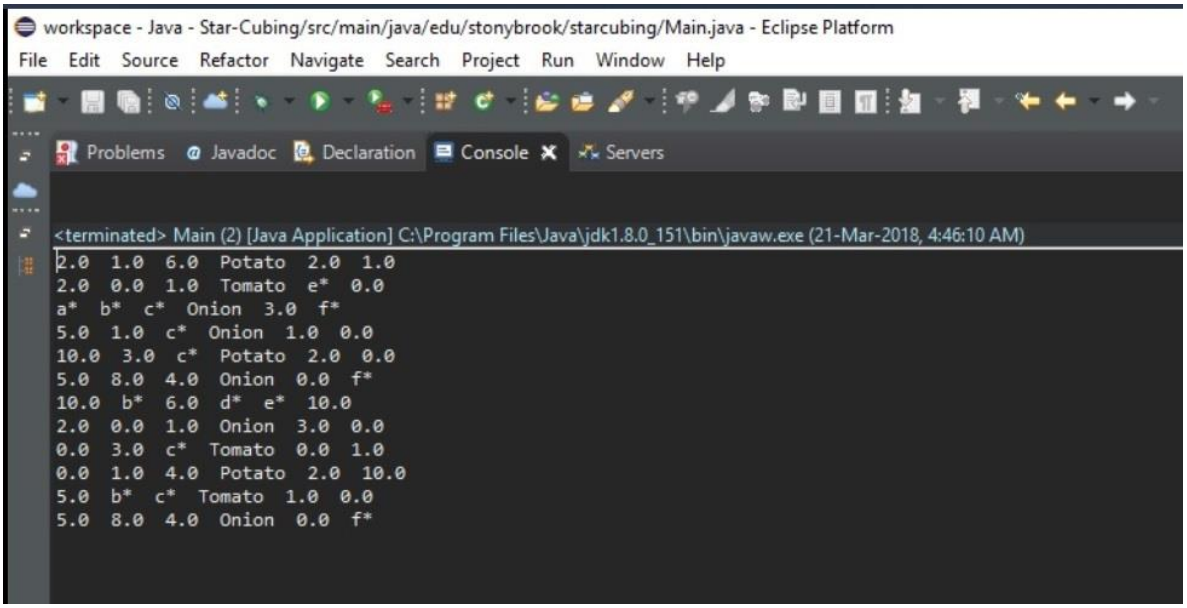



Fig.7: Dataset obtained after applying Iceberg condition

This List is used to obtain the compressed base star table. The compressed base star table is stored as *HashMap<String, Integer>()*. Within this HashMap, the map's key is a String obtained by combining attribute values of each tuple separated by a comma and the map's value is the count of occurrences of the tuple (row). *Algorithm.compressedBaseTable()* is used for performing this function. In our example, we have the following HashMap:

HashMap<Key, Value> for Compressed Base Table are:

<"2.0,1.0,6.0,Potato,2.0,1.0", 1>

<"2.0,0.0,1.0,Tomato,e*,0.0", 1>

<"a*,b*,c*,Onion,3.0,f*", 1>

:

:

<"5.0,8.0,4.0,Onion,0.0,f*", 2>

Since the tuples have been modified with <column_label>* (ex. a*) aggregate values, these tuples further can be compressed together for a higher aggregate value.

This base star table (HashMap) is used to create the base star-tree. The star tree structure has been implemented using *Cuboid()* class as tree nodes. This class has the following attributes – a string value denoting attribute value, an integer value to represent the aggregate, a map holding key-value pairs of the children nodes, and a variable pointing to the next sibling present, if any. *Algorithm.cuboidTree()* is used in our implementation to create this star-tree structure. Following our example, we obtain the following data structure in memory (as shown in Fig.8):

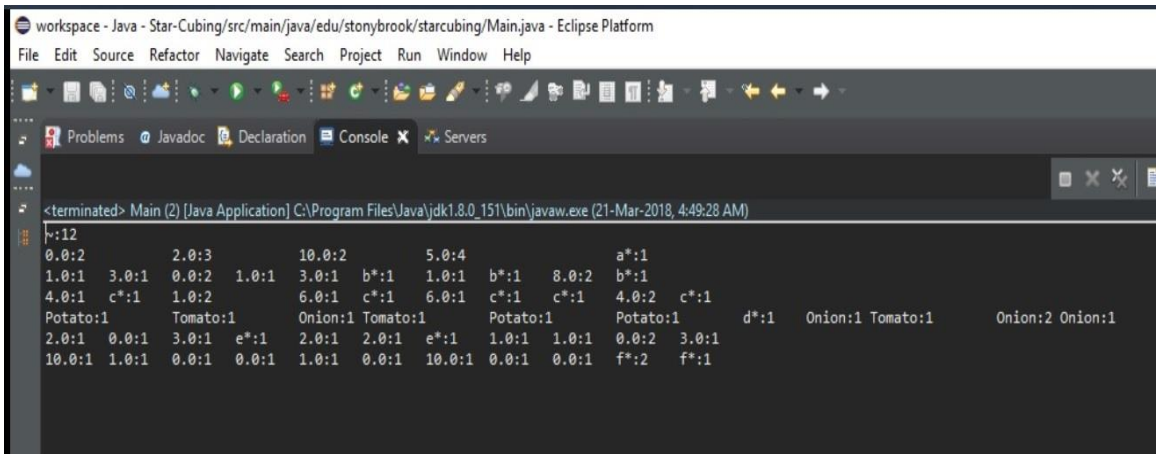



Fig.8: Star tree structure stored in memory

This star-tree structure is stored starting from the root node. In the figure, ~ represents the root, 0.0:2, 2.0:3, 10.0:2, etc. represent the nodes containing the attribute value and aggregate value. The nodes are displayed as per different levels of the star tree structure.

Following the creation of this star tree data structure, multi-way star-cubing algorithm is run using the *Algorithm.starCubing()* function. The following figure (Fig.9) displays the resultant roots of sub trees when min_sup value is kept at 2:

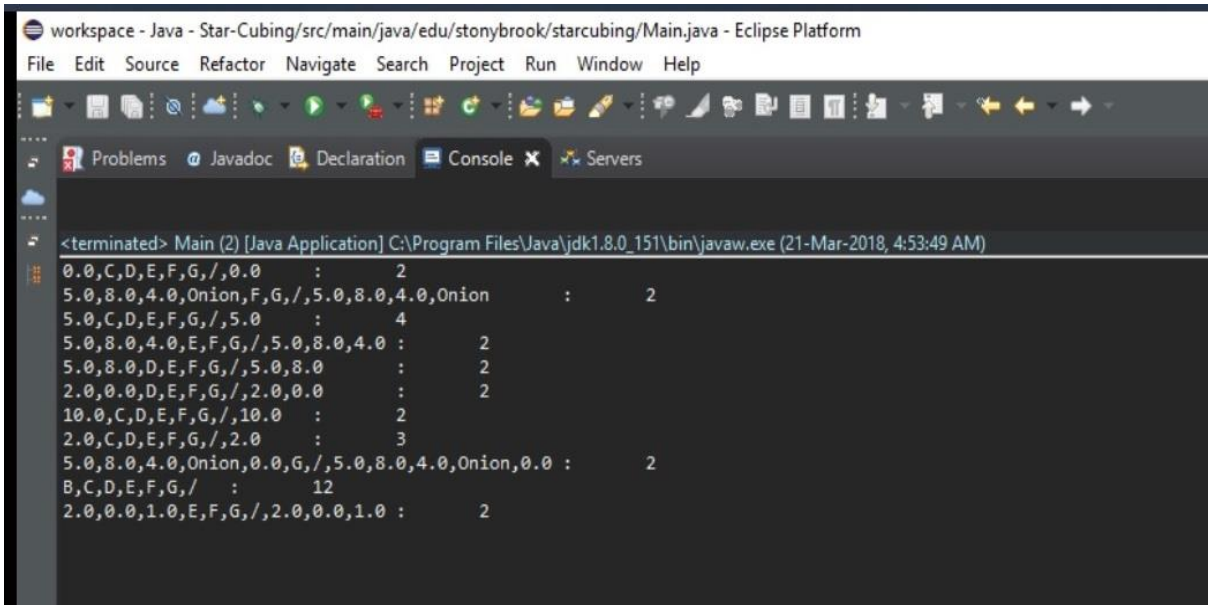



Fig.9: Star-Cubing results for min_sup = 2

Min_sup of 2 indicates that only those sun star-trees are displayed whose aggregate value has a minimum value of 2. Fig.10 displays the multi-way star-cubing resultant roots when the min_sup is kept at 1:

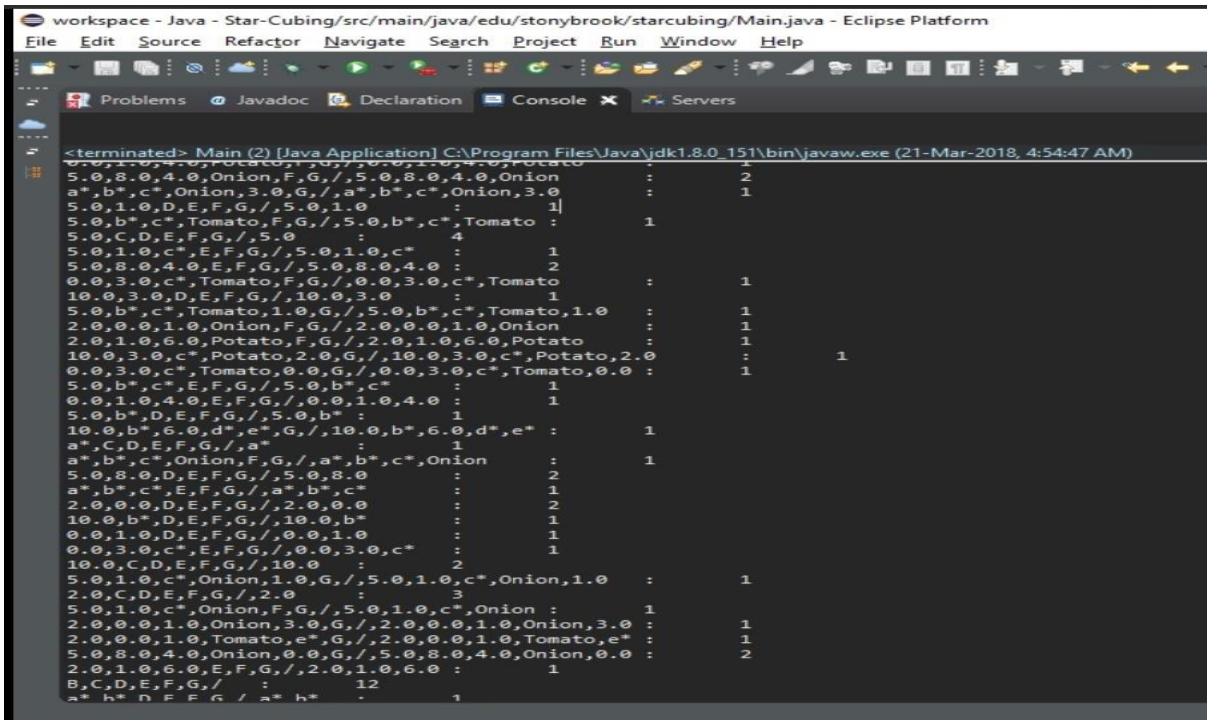



Fig.10: Star-Cubing results for min_sup = 1

As can be seen from the results, min_sup of 1 gives all the aggregate star-trees created. Each row of the result is of the form *A,B,C,D,E,…/A,B,C,D,E,… : N.*

For example: *0.0,3.0,c*,Tomato,F,G/0.0,3.0,c*,Tomato : 1* is a sample result for the case when min_sup=1. The meaning of this notation is that this aggregate value is a node at the fifth level from the root, i.e., for first attribute value of 0.0, second attribute value of 3.0, third attribute value of c*, and fourth attribute value of Tomato, there exists an aggregate count of one row in the dataset. If we compare the dataset in Fig.6 (dataset just after normalization) we find that there is only one row present with values 0.0 in the first attribute, 3.0 in the second attribute, and Tomato in the fourth attribute. Since the third attribute is c*, the value present in the table in Fig.6 has not passed the Iceberg condition and is thus not relevant. Observing the initial data, we can see that occurrence of 0 loaves of bread, 3 cartons of milk and Tomato as type of vegetable bought from the retail store occurs with only a frequency of 1 in the given dataset.

As a second example: *2.0,0.0,1.0,E,F,G/,2.0,0.0,1.0 : 2* is a sample result for the case when min_sup=2. This means that there are two rows with first attribute value 2.0, second attribute value 0.0, and third attribute value 1.0 in the table present in Fig.6. On verifying the table, we find that this is a correct result too. Checking the initial data, this means that occurrence of customers buying 1 loaf of bread, 1 carton of milk and nearly 2 kg of vegetables (1.7 & 2.1) has happened twice in the given dataset.

As a third example: B,C,D,E,F,G/ : 12 is a sample result for the case when min_sup=2. This means that the aggregate count obtained when all six values are considered is 12, which is the entire dataset. This is also true.

Thus, we observe that the output results can be validated with the initial given dataset and we can conclude that star-cubing algorithm has run successfully.

## VII. Extending Star-Cubing for "Linear Programming-Based Optimization for Robust Data Modelling in a Distributed Sensing Platform"

The paper "Linear Programming-Based Optimization for Robust Data Modelling in a Distributed Sensing Platform" provides an innovative approach to construct robust data models using samples acquired through a grid network of embedded sensing devices with limited resources. The paper has tested the scalability of the model by running simulations by using three network sizes of embedded nodes: 25, 64 and 100. We have considered the 25-node network model for our subsequent explanation on how we propose to implement the star-cubing algorithm on the dataset.

It is known that all the 25 sensing nodes are connected to one of the nodes, designated as the target point (TP), vis predefined path configurations. The TP receives the data packets, extracts temperature information, and builds the resultant thermal map. Loss, delay and error statistics are also extracted from the information received by the TP. This action performed is synonymous to file parsing, storage of dataset information in our file, and error correction actions performed in our implementation. The

sensor values received from the 25 nodes will contain multiple errors as described in the paper – data loss errors, time delay errors, path-induced errors, and variable lumping errors. Thus, every value received by the TP will be an aggregate of the original sensor data as well as the mentioned errors. As these error values can be extracted from the packets received from each node, we decide not to discard these error values since they would give considerable knowledge regarding the error flow in the network as well as they provide insight in improving efficiency.

Considering that each sensor node would have four attributes – the original sensor value, the loss associated with the sensor node, delays in receiving the packet, and errors associated with the values received – our initial dataset that would be created using all the sensor node's values will contain 4 x 25 attributes, i.e., each sensor node will have 4 attributes associated with it. The losses, delays and error statistics can be extracted using the equations described in the paper. Basically, the error correcting function in our implementation will get additional methods in order to extract the relevant data required for our attributes.

We will consider the following implementation for our obtained dataset. Suppose, at a time t, the TP node collects data for all 25 sensors. The data required for our dataset will be extracted from these values and stored in our data structure as a tuple. Now, the same process is repeated at time $t+1$, $t+2$, …., $t+n$, where n is the sampling period we are considering. This will give us n tuples to work with to perform star-cubing algorithm. This n value can be increased or decreased to incorporate a larger/smaller time range of values to work with.

Once these parameters have been extracted and stored in our data structure (ArrayList of dimensions), we will be able to proceed with the star-cubing algorithm. We will consider the utilization rates - $\alpha$ for discarding sensed values, $\lambda$ for different lumping levels, $\beta$ for bandwidth values and Path for DCPs (data path configurations) – as Iceberg conditions for our obtained dataset. This will require modifications in the way the Iceberg condition is currently set in our implementation. Applying the Iceberg condition will result in the formation of the compressed base table.

After the compressed base table has been obtained, we can continue with the formation of star-tree data structures and star-cubes as per our implementation. The results of the sensor network can be calculated and stored for multiple sampling times n. The results that we obtain can provide us with valuable insights about how the sensor network functions. We can use the results to check how the temperature variations from one sensor affect the others, or how temperature variations in a particular area of the network (using a group of sensor value) affect other areas in the network. If similar patterns of values start arising in the sensor readings, we can use the dataset to predict the behavior of the sensor networks, like, if under some operating conditions, the temperature readings consistently show similar high values, we can predict that further occurrence of similar operations conditions will lead to similar high values. We can also obtain the path losses among data sent from different sensor nodes and try to find an optimum path to reduce such path losses by making use of cuboid trees. The developed star tree data structure can also be stored inherently in the memory of the sensors, in order to increase the speed of data processing.

Our design implementation suffers from a drawback in that we have not considered memory constraints. Our design assumes that memory provided will be sufficient to perform the in-processor star-cubing algorithm while the datasets will be stored in secondary memory. This drawback can be overcome up to certain limit by storing required cuboid tree in primary memory i.e. memory of the processor so that processing can be done quickly.

# VIII. Experimental Results

Each stage of algorithm execution provides an opportunity for experimentation. Below are the experiments implemented and analysed in this solution.

## 1. Normalization:

Dimensions/ attributes of a dataset could be of continuous-range i.e. with a large cardinality. As star-cubing is sensitive to attribute cardinality (usual being around 10), such attributes have to be normalized to a range with buckets to achieve small cardinality. If attributes are not normalized, the most values would not satisfy initial Iceberg condition and be replaced with '*'.
Normalizing the data set results in more multi-way aggregate star-trees as more attribute values contribute to compressed base star-tree.

Below table shows the total number resultant sub-trees after multi-way star-tree aggregation for both approaches.

| *Dataset* | *Without Normalization* | *With Normalization* |
|---|---|---|
| Above Example | 11 | 12 |
| Bank Marketing | 8709 | 52708 |
| Air Quality | 3309 | 5525 |
| Automobile | 598 | 663 |
| Bike Sharing | 502 | 613 |

Table No.3: Star Tree Count With and without Normalization

Following graph represents difference in count of resultant Sub-Trees from Table No. on logarithmic scale.

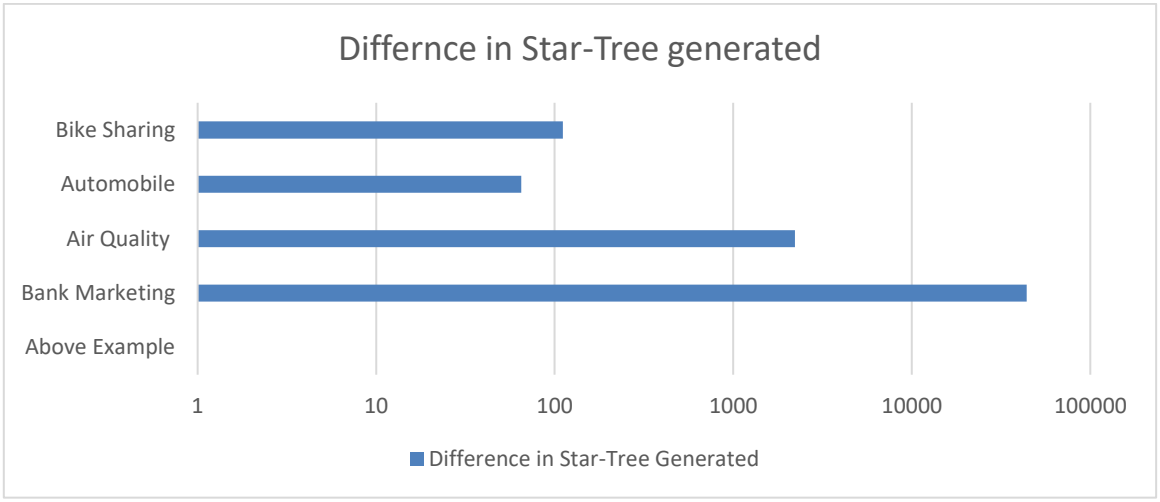



Fig. 11: Difference in count of resultant Sub-Trees from Star Tree generated

## 2. Range-Based Normalization:

Here, the cardinal range of an attribute is decided based on its original range compared to other attributes. Example, assume A1 with original cardinality 180 and A2 with 360. Then A2 is normalized on 0-15 range and A1 on 0-14. This results in lesser child sub-tree apriori pruning and

more and better multi-way star-cube aggregation i.e. more sub-trees passing iceberg and min_sup condition.

Below table shows the total number resultant sub-trees after multi-way star-tree aggregation for both approaches.

| *Dataset* | *Without Range-Based Normalization* | *With Range-Based Normalization* |
|---|---|---|
| Above Example | 11 | 12 |
| Bank Marketing | 52529 | 53388 |
| Air Quality | 13125 | 16970 |
| Automobile | 597 | 663 |
| Bike Sharing | 511 | 613 |

Table No. 4: Range based Normalization

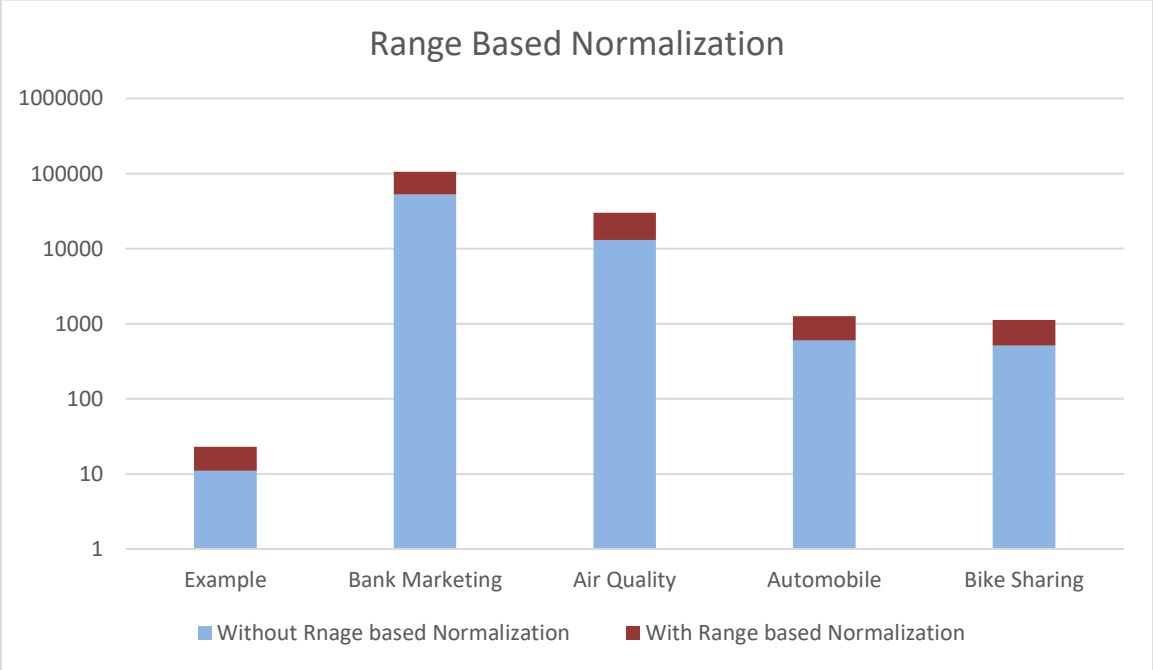



Fig. 12: Range based Normalization

Functionality of code was primarily tested on 'Bank Marketing' dataset. Further in this section we will provide discussion about results obtained from another dataset as well.

## 3. Iceberg based on Average Occurrence (IBAO):

Iceberg condition for each attribute is based on the average number of occurrences of attribute values. Given an input percentage (Iceberg percentage) we calculate Iceberg condition for each attribute.

Example, A1 has 3 types of values: a1, a2, and a3.

- a1 occurs 9 times, a2 occurs 11 times and a3 occurs 13.
- Then average occurrences AO = (9 + 11 + 13) /3
- Iceberg condition = (Iceberg %) × AO

This provides more attribute specialized compression. Below table provides number of tuples in compressed base table for both approaches.

| *Dataset* | *Without IBAO* | *With IBAO (IB%=0.25)* |
|---|---|---|
| Bank Marketing | 32656 | 32656 |
| Air Quality | 9208 | 9357 |
| Automobile | 173 | 187 |
| Bike Sharing | 706 | 727 |

Table No. 5 Iceberg condition

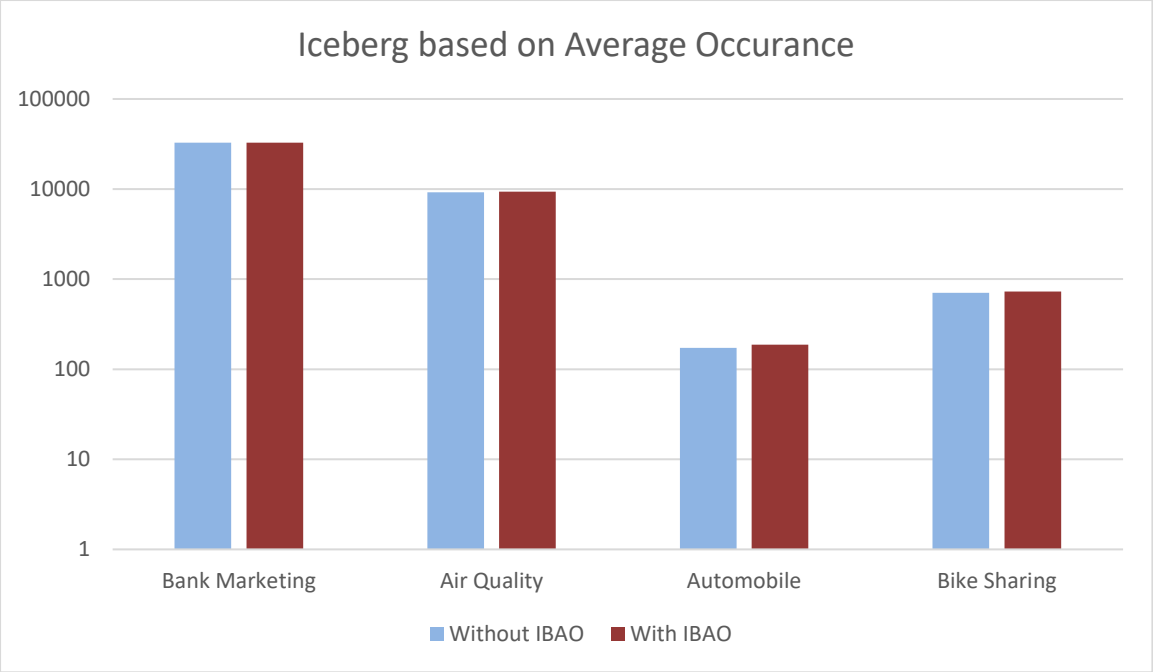



Fig. 13: Iceberg based on Average occurrences

## 4. Dimension Ordering:

Star-Cubing is sensitive to the order of the dimensions. The goal of ordering dimensions is to prune the trees as early as possible. The internal nodes whose aggregate values do not satisfy the iceberg condition by the biggest value should be processed earlier.

So before the compressed table and base star-tree are built the dimensions/ attributes are ordered based on their cardinalities in descending order.

Below table provides total number resultant sub-trees after multi-way star-tree aggregation for both approaches.

| *Dataset* | *Without Dimension Ordering* | *With Dimension Ordering* |
|---|---|---|
| Bank Marketing | 57034 | 57029 |
| Air Quality | 3485 | 3485 |
| Automobile | 898 | 722 |
| Bike Sharing | 848 | 599 |

Table No. 6: Comparison between with Dimension ordering and without dimension ordering


Fig. 14: Dimension Ordering

## IX. Results

Below is the table providing general results for each execution steps for given datasets. With Iceberg % = 0.25 and min_sup = 2.

| *Dataset(dimensions)* | *Initial No. of Instances* | *Compressed Base Table size* | *No of resultant sub-trees* |
|---|---|---|---|
| Bank Marketing(17) | 45211 | 32656 | 57029 |
| Air Quality(15) | 9358 | 9208 | 3485 |
| Automobile(26) | 205 | 173 | 722 |
| Bike Sharing(16) | 731 | 706 | 599 |

Table No. 7: General characteristics

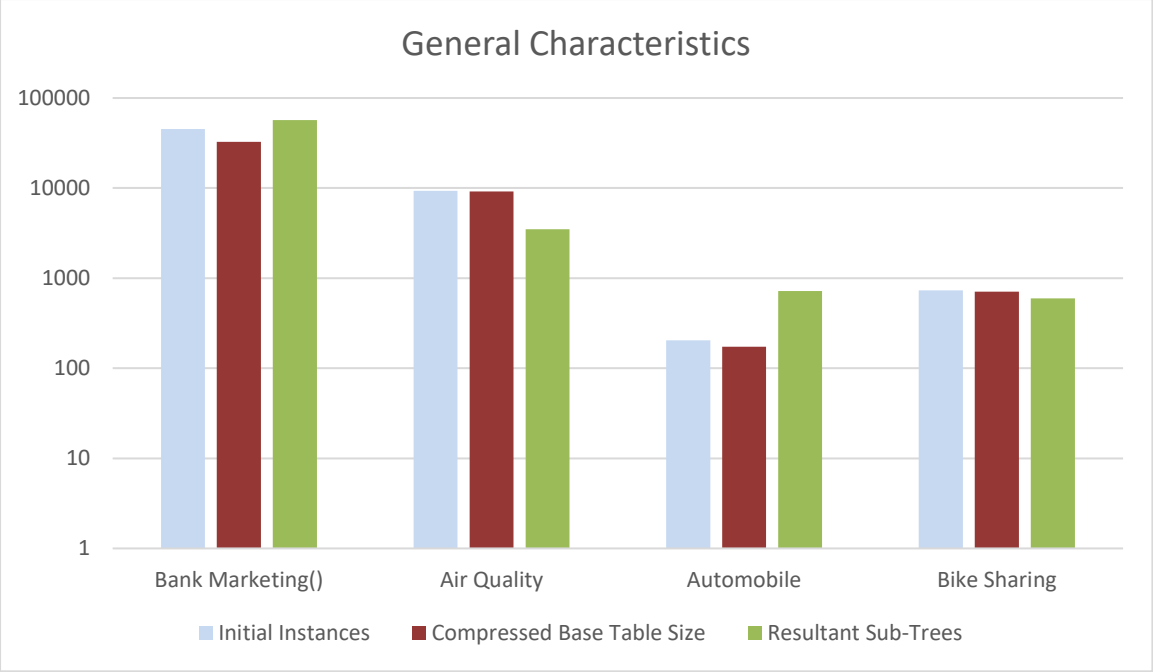



Fig. 15: General characteristics

## X. Conclusion

Top-down and Bottom-up approach based Star-Cubing, gives an advantage over other traditional approaches. Our experimental results have suggested that, use of HashMap while counting occurrences improves the processing speed compared to other techniques. Along with this, we have observed that dimension ordering gives a plus by pruning children at initial stage of algorithm and hence reducing the computations required. We have also observed that normalizing the raw dataset improves accuracy of reduced dataset as number of trees generated are more in case of normalization and hence one will not loose on important data tuples.

## XI. Bibliography

[1] Dong Xin, Jiawei Han, Xiaolei Li, Benjamin W. Wah “Star-Cubing: Computing Iceberg Cubes by Top-Down and Bottom-Up Integration”, VLDB’03.

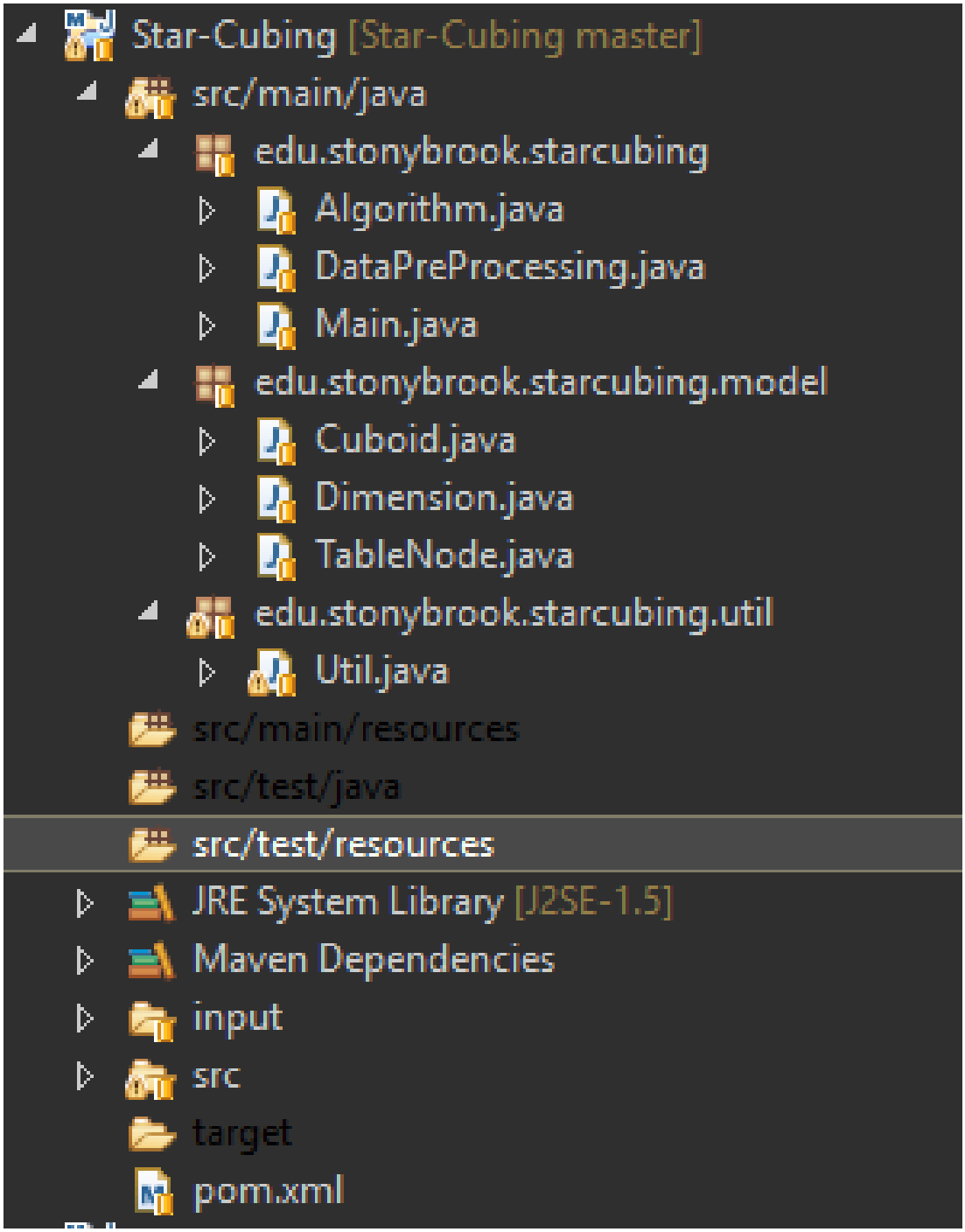
[2] A. Umbarkar, V. Subramanian, A. Doboli, ``Linear Programming-based Optimization for Robust Data Modelling in a Distributed Sensing Platform'', IEEE Transactions on CADICS.

[3] Jiawei Han, Micheline Kamber, “Data Mining Concepts and Techniques”, 2nd edition.

[4] Dataset sources

- https://archive.ics.uci.edu/ml/datasets/Bike+Sharing+Dataset
- https://archive.ics.uci.edu/ml/datasets/Air+Quality
- https://archive.ics.uci.edu/ml/datasets/Bank+Marketing
- https://archive.ics.uci.edu/ml/datasets/Automobile

## Appendix 1: Package Structure



## Appendix 2: Source Code

Please find the entire source code attached in the zip folder shared.

Main.java

```
public class Main {


	private final static String PATH = "./input/bank.csv";

	public static void main(String[] args) {
		executeAlgoSteps();
	}

	private static void executeAlgoSteps() {
		List<Dimension> list;
		if (PATH.contains("csv"))
			list = Util.readCSV(PATH);
		else
			list = Util.readXLSX(PATH);
		// Util.print(list);

		/* Set Iceberg Condition and Minimum Support */
		Algorithm.setIBC(2);
		Algorithm.setMinSup(2);

		/*Sort dimensions based on cardinality*/
		Algorithm.sortDimensions(list);

		/* Normalize all columns */
		DataPreProcessing.normalizeTable(list, true);
		// Util.print(list);

		/* replace Dimension/attribute values with **/
		Algorithm.replaceDimensions(list);
		// Util.print(list);

		/*Sort dimensions based on cardinality*/
		Algorithm.sortDimensions(list);

		/* Get compressed base table after star reduction */
		Map<String, Integer> baseTable = Algorithm.compressedBaseTable(list);
		// Util.printMap(baseTable);
		//System.out.println(baseTable.size());

		/*Create star-tree and get root*/
		Cuboid root = Algorithm.cuboidTree(baseTable);
		//Util.bfs(root);

		/* Last step to generate sub-trees */
		Map<String, Cuboid> map = Algorithm.starCubing(root, list.size());
		//Util.printAllBFS(map);
```

```
            //Util.printSubTreesFromMap(map);
            System.out.println(map.size());
        }
}
```

Algorithm.java

```
public class Algorithm {

        static double iBC = 2;
        static double min_sup = 2;

        public static void setIBC(double iBC){
                Algorithm.iBC = iBC;
        }

        public static void setMinSup(double min_sup){
                Algorithm.min_sup = min_sup;
        }

        public static void sortDimensions(List<Dimension> list) {
                Collections.sort(list, new Comparator<Dimension>(){

                        @Override
                        public int compare(Dimension o1, Dimension o2) {
                                return o2.cardinality - o1.cardinality;
                        }

                });
        }

        public static Map<String, Integer> compressedBaseTable(List<Dimension> list) {
                return Util.getOoccurrenceCount(getTuples(list));
        }

        public static void replaceDimensions(List<Dimension> list) {
                for (Dimension attribute : list) {
                        if (!attribute.values.isEmpty()) {
                                Algorithm.dimensionIceBCheck(attribute);
                        }
                }
        }

        public static Cuboid cuboidTree(Map<String, Integer> baseTable) {
                Cuboid root = new Cuboid();
                for (Map.Entry<String, Integer> entry : baseTable.entrySet()) {
                        String[] attrs = entry.getKey().split(",");
                        Cuboid lastChild = null,
                                        parent = root;
                        for (String attr : attrs) {
                                parent.aggrVal += entry.getValue();
                                if (!parent.children.containsKey(attr)) {
                                        parent.addChild(new Cuboid(attr));
                                }
                                parent = parent.children.get(attr);
                                lastChild = parent;
                        }
```

```
                if (null != lastChild)
                        lastChild.aggrVal += entry.getValue();
        }
        return root;
    }

    public static Map<String, Cuboid> starCubing(Cuboid root, int attributes) {
        Map<String, Cuboid> map = new HashMap<String, Cuboid>();
        StringBuilder sb = new StringBuilder();
        char attrRepresent = 'B';
        for (int i = 0; i < attributes; i++) {
                sb.append(attrRepresent);
                sb.append(',');
                attrRepresent++;
        }
        sb.append('/');
        for (Cuboid cuboid : root.children.values()) {
                starCubing(cuboid, map, sb.toString(), 0, new ArrayList<Cuboid>());
        }
        childPruning(map);
        return map;
    }

    private static void childPruning(Map<String, Cuboid> map) {
        Iterator<Map.Entry<String, Cuboid>> iterator = map.entrySet().iterator();
        while (iterator.hasNext()) {
                Map.Entry<String, Cuboid> entry = iterator.next();
                if (entry.getValue().aggrVal < min_sup) {
                        iterator.remove();
                        continue;
                }
                boolean prune = true;
                for (String string : entry.getKey().substring(0,
entry.getKey().indexOf("/")-1).split(",")) {
                        if (!string.contains("*")/* && !string.equals("/")*/) {
                                prune = false;
                                break;
                        }
                }
                if (prune)
                        iterator.remove();
        }
    }

    private static void starCubing(Cuboid cNode, Map<String, Cuboid> map, String
curTree, int level, List<Cuboid> addToList) {
        Cuboid levelCube = increaseAggrVal(cNode, map, curTree);
        String nextCurTree = nextTreeName(curTree, level, cNode.attrValue);
        for (Cuboid cuboid : cNode.children.values()) {
                List<Cuboid> nextAddToList = new ArrayList<Cuboid>();
                Cuboid c = increaseAggrVal(cuboid, levelCube.children,
cuboid.attrValue);
                nextAddToList.add(c);
                for (Cuboid prevCub : addToList) {
                        c = increaseAggrVal(cuboid, prevCub.children,
cuboid.attrValue);
                        nextAddToList.add(c);
                }
```

```
                starCubing(cuboid, map, nextCurTree, level + 1, nextAddToList);
            }
        }

        private static <T> Cuboid increaseAggrVal(Cuboid node, Map<String, Cuboid> map,
String curTree) {
            Cuboid levelCube = null;
            if (map.containsKey(curTree)) {
                levelCube = map.get(curTree);
            } else {
                levelCube = new Cuboid(curTree);
                map.put(levelCube.attrValue, levelCube);
            }
            levelCube.aggrVal += node.aggrVal;
            return levelCube;
        }

        private static String nextTreeName(String treeName, int level, String cur) {
            String[] str = treeName.split(",");
            str[level] = cur;
            StringBuilder sb = new StringBuilder();
            for (String string : str) {
                sb.append(string);
                sb.append(",");
            }
            sb.append(cur);
            return sb.toString();
        }

        private static <T> List<String> getTuples(List<Dimension> list) {
            List<String> concaList = new ArrayList<String>();
            for (int i = 0; i < list.get(0).values.size(); i++) {
                StringBuilder str = new StringBuilder();
                str.append(list.get(0).values.get(i));
                for (int j = 1; j < list.size(); j++) {
                    str.append("," + list.get(j).values.get(i));
                }
                concaList.add(str.toString());
            }
            return concaList;
        }

        private static void dimensionIceBCheck(Dimension attribute) {
            Map<String, Integer> countMap = Util.getOoccurrenceCount(attribute.values);
            attribute.cardinality = countMap.size();
            for (int i = 0; i < attribute.values.size(); i++) {
                if (countMap.get(attribute.values.get(i)) < iBC) {
                    String str = attribute.getLowerCLabel() + "*";
                    attribute.values.set(i, str);
                }
            }
        }
}
```

Cuboid.java

```
public class Cuboid {

	public String attrValue;
	public int aggrVal;
	public Map<String, Cuboid> children;

	public Cuboid(){
		children = new HashMap<String, Cuboid>();
		aggrVal = 0;
		attrValue = "~";
	}

	public Cuboid(String attrValue){
		children = new HashMap<String, Cuboid>();
		this.attrValue = attrValue;
		aggrVal = 0;
	}


	public void addChild(Cuboid child){
		children.put(child.attrValue, child);
	}

	@Override
	public boolean equals(Object o){
		if(null == o) return false;
		if(!Cuboid.class.isAssignableFrom(o.getClass())) return false;
		Cuboid co = (Cuboid) o;
		if (!co.attrValue.equals(this.attrValue)) return false;
		return true;
	}

	@Override
	public int hashCode() {
		return attrValue.hashCode();
	}
}
```