CS615: Group 13
APURNA: A Java Library For Skylines Over Incomplete And Private Data

Abhishek Maharana
Student Id 17
Roll Number 14111002
abhimaha@iitk.ac.in
Dept. of CSE

Banothu Raj Kumar
Student Id 18
Roll Number 14111007
rajb@iitk.ac.in
Dept. of CSE

Indian Institute of Technology, Kanpur

Final report
25th November, 2015

Abstract

Skyline queries help us to filter unnecessary information efficiently and provide us clues for various decision making tasks. Most of the existing skyline algorithms assume a dataset complete in all dimensions over which skylines need to be computed. This seems to be a very restrictive assumption. A more practical case is that of incomplete datasets (i.e tuples missing values in some dimensions). In this project, we present **APURNA** - a Java library that can be used directly in Java based projects to find skylines involving incomplete data. Our initial algorithms are Naive and Bucket algorithms for skyline "points" in incomplete datasets. With increased focus on privacy, the second algorithm that we chose to include is RBSSQ algorithm - a replacement based convex skyline set algorithm. Set skylines ensure that individual records of the dataset aren't revealed. Instead, they output s-set of tuples that provide summarized skyline information. We have taken care that our library solves all primary use cases related to incomplete and private data, is modular and very easy to use, and easily extensible. Also we have made it open source and released it on GitHub (https://github.com/vagabondtechie/apurna).

1 Introduction and Problem Statement

Consider database DB has a view table whose schema has following columns: ID, a_1, a_2,....,a_k, where ID is primary key attribute and a_j(j=1,....,k) are k-dimensional float attribute. The problem of skylines over incomplete data is to find skylines points for this database when tuples might be missing values in some dimensions. We include the Naive and Bucket algorithm for this purpose in our library.

The term s-sets mean sets of "s" objects of the database.

A virtual database S is set of all possible s-sets of a database.

In practice, database owners don't want to reveal the actual records of the database to others. Convex skyline set queries can represent aggregated skyline summary objects that do not disclose individual records of the database. We include the RBSSQ algorithm to compute convex skyline set queries over incomplete dataset. ***If RBSSQ is implemented as-is in [2], it might run into infinite recursion for higher dimensions. We have solved this problem very efficiently by storing normal vectors to the higher dimensional hyperplanes encountered during algorithm run.***

1.1 Related Material

We have followed the approach proposed in [1] and [2] throughout the project. The data sets are generated with varying number of dimensions and missing values in random dimensions of random objects. Each data file has varying number of entries depending on what parameter was being tested.

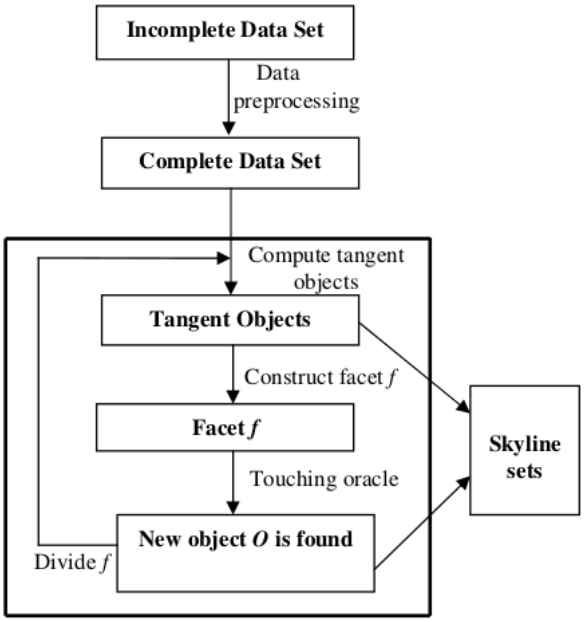

Figure 1: System architecture of RBSSQ method

2 Algorithms and Library Usage

Below, we present the algorithms available in our library.

2.1 RBSSQ

Replacement Based Sets Skyline Queries (RBSSQ) method consists of two phases: (1) Data Preprocessing phase (2) Skyline sets computation phase. Figure 2 shows overall methodology of RBSSQ.

2.1.1 Data Preprocessing

In data preprocessing phase, we first search the database for obtaining missing values of the records in each attribute. Then, we replace missing values of the records in an attribute with a value outside the domain values. As we have considered smaller values are better for an attribute, missing values are replaced with a value larger than the domain value. We have taken domain range [0.0 - 1.0] and filled the missing values of attribute with 99999.0.

2.1.2 Touching Oracle Function

Touching Oracle is a method to compute a point on the convex hull of a virtual database S without generating it. It computes the tangent point of the convex hull of S and a (k - 1) - hyperplane directly from DB. It takes normal vector to the hyperplane as input and computes the inner product of each record in database with normal vector. Then it chooses the top s inner products to compose the s-set tangent point.

Algorithm 1 Data Preprocessing

Input: Incomplete Data Set S, value $value_j$ that will replace missing values in attribute j. Here, j = 1 to k
Output: Complete Data Set T

```
repeat
read point p_i, 1≤i≤n from input S
for each p_i do
check every dimension j, j=1 to k
replace each missing value with value_j.
end for
until end of input S
```

2.1.3 Skyline Sets Computation

Skyline sets computation module computes skyline sets from the processed data by utilizing touching oracle function. First of all, we have computed initial k tangent points by using touching oracle function with initial k vectors θ_x = (θ_1, θ_2, ..., θ_k) where θ_i = -1 if i = x, otherwise θ_i = 0 for each x = 1, ..., k. Then we have computed the initial facet, the hyperplane joining the initial points. Next, we have computed the normal vector of the initial facet and invoked the touching oracle to obtain a tangent point which is a point on the convex hull. We expand the initial facet into k new facets if the new point lies outside the facet. We have recursively computed the tangent points for each expanded facet. If we find new point outside the facet we expand the facet further. We continuously adopt the recursive operation while we can find new tangent point outside the facet.

2.2 Bucket Algorithm

Assume we assign each object a bitmap. The values of the bits are assigned as follows - if the object has data available in i^{th} dimension, i^{th} bit in bitmap is 1, else 0.

Bucket algorithm divides all incoming points into distinct buckets where all points in each bucket have the same bitmap representation. Then we apply the traditional BNL algorithm to find the local skylines within each bucket by ignoring incomplete dimensions. All these local skylines are later pushed into candidate skyline list. Then we perform exhaustive pairwise comparison among all points to get the query answer. Bucket algorithm is as shown in Algorithm 2.

Algorithm 2 Bucket algorithm

Input: Tuples o_j of database D
Output: Candidate skyline set S

1: **while** $\exists$ object o_j **do**
2: Push o_j into bucket with appropriate bitmap.
3: **end while**
4: Compute local skylines for each bucket
5: Aggregate local skylines into candidate skyline list
6: Perform BNL over candidate skyline list and get set of skyline points S.
7: **return** S

2.3 Library Usage Sample

Now that we have seen what algorithms "APURNA" includes, let us see how easy it is to use in any Java project. A sample code listing is shown in Fig. 2.

```
1  import in.ahmrkb.prj.SkylineAlgorithm.ALGOTYPE;
2
3  public class MainDriver {
4
5      public static void main(String[] args) {
6          Database.configureDatabase(new DBConfig(5)); // No of dimensions
7
8          Database.readValuesIntoDB("/data/ip.txt"); // Input file name
9
10         SkylineAlgorithm algorithm = SkylineAlgorithmFactory
11                 .getSkylineAlgorithm(ALGOTYPE.NAIVE);
12
13         algorithm.getSkylineTuples();
14     }
15 }
```

Figure 2: Sample Code Usage

(1) Line 6 shows how to configure the database. We need to pass a DBConfig object with dimensionality of the dataset as parameter.
(2) Line 8 reads values from the input source into the database.
(3) Line 10 lets you choose a skyline algorithm.
(4) Line 11 gets you the skylines over your incomplete(and/or private) dataset.

Just four lines of easily understandable code should suffice to demonstrate the ease of usage of our library.

3 Results

3.1 Performance Test Setup and Graphs

Note: *For each data-point on the graphs that follow, we ran the code 5 times on the same dataset and took average of all the times taken.*

First, we compare our Bucket and Naive algorithm implementation for checking the effect of dimensionality of tuples on running time of algorithm. We fixed the dataset size to 10^5 and varied the dimensionality from 2 to 6. The results are plotted as a graph in Fig. 2.

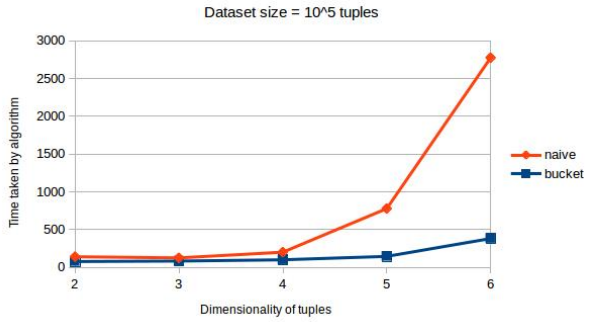

Figure 3: Time v/s Dimensionality of tuples

As the number of dimensions increase, the difference in running times of Bucket and Naive algorithms becomes noticeable with Bucket algorithm giving very good running time.

Second, we test the effect of dataset size on the running time. We fixed the dimensionality to 6D and varied the dataset size in 5000, 10000, 50000 and 100000. The results are plotted in Fig. 3.

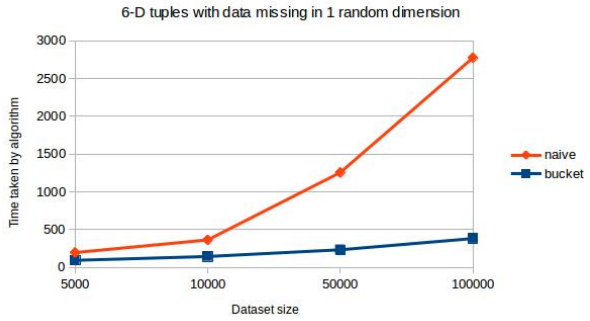

Figure 4: Time v/s Dataset Size

As the dataset size increases, time taken by both - Bucket and Naive algorithm increase too. But, while Bucket algorithm's time varies almost linearly with dataset size, Naive algorithm's running time just explodes.

Next, we test the effect of size of s-set on the running time of RBSSQ. We fixed the dimensionality to 2D, number of tuples to 10000 and varied the s-set size from

2 to 5. The results are plotted in Fig. 5.

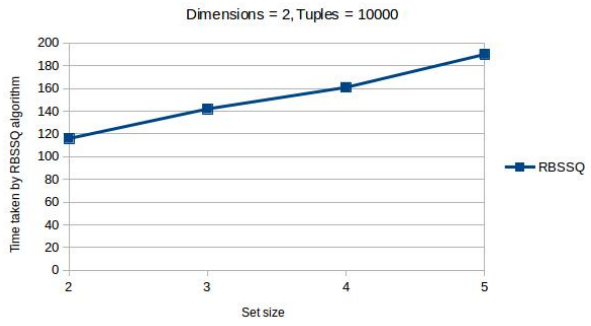

Figure 5: Time v/s s-set Size

As the s-set size increases, time taken by RBSSQ increases too (obviously).

And finally, we test the effect of data set size on the running time. We fixed the dimensionality to 2D, s-set size to 3 and varied the dataset size from 10^2 to 10^5 by factor of 10. The results are plotted in Fig. 6.

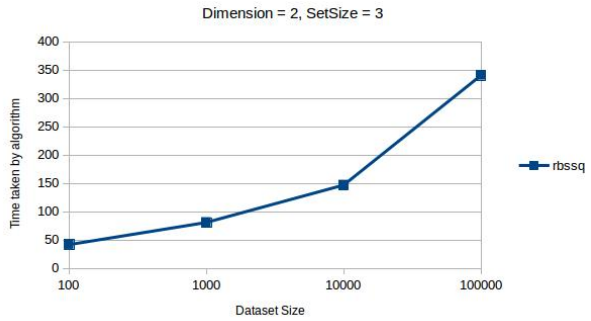

Figure 6: Time v/s Dataset Size

As the dataset size increases, RBSSQ too, expectedly, takes more time.

3.2 Reasoning About the Results

Bucket algorithm fares better compared to Naive for decent sized datasets due to following reasons-

(1) The size of the candidate list in the Bucket algorithm is much less than the window size in replacement based BNL algorithm, thus exhaustive pairwise comparison is cheaper.

(2) Applying a traditional replacement based BNL algorithm several times for few data items in each bucket is cheaper than applying it once over all data items.

4 Conclusions

In this project, we have presented our library "APURNA" that deals with skyline computation over incomplete and private data. Our library solves the infinite recursion problem in [2] which we face on as-is implementation of RBSSQ as detailed in paper. As seen from previous sections, our library is very easy to use (3-4 lines of code). It is also easily extensible and very modular. It provides all basic functionalities related to incomplete data. In privacy aware environments where we have to hide individual values and are only allowed to disclose aggregated values of the objects, our implemented RBSSQ algorithm can be a very helpful in decision-making. In cases of computing skyline "points" over incomplete data, our BucketAlgorithm (and in some smaller cases, the NaiveAlgorithm) method can be used as efficient functions.

4.1 Future Work

Our library (APURNA) is fairly modular, good to be used for small and medium projects. But we would like to make it industrial strength. Also, dominance testing can be improved along the lines of [3] which will result in overall better skyline computation performance. We are open to collaboration with students as well as professional programmers as we have released the source code on GitHub (https://github.com/vagabondtechie/apurna)

References

[1] M.E. Khalefa, M.F. Mokbel, and J.J. Levandoski. *Skyline query processing for incomplete data. In ICDE, 2008*

[2] Arefin, M.S., Morimoto, Y.: *Skyline sets queries for incomplete data. Int. J of Computer Science and Information Technology 4(5), 67-80 (2012)*

[3] S.R. Cho et. al. *VSkyline: Vectorization for Efficient Skyline Computation. In SIGMOD Rec., 2010*