Revisited Skyline Query Algorithms on Streams of Multidimensional Data

Alexander Tzanakas, Eleftherios Tiakas, Yannis Manolopoulos

Aristotle University of Thessaloniki, Thessaloniki, Greece

Abstract

This paper focuses on the analysis and evaluation of skyline query algorithms on streams of multidimensional data. It presents three algorithms and evaluates them with different types of datasets, number of dimensions and "*Sliding Window*" sizes. The results of these experiments are reviewed according to time performance, memory consumption and the size of the skyline.

Keywords: Skylines, Analysis, Experiments, Evaluation

1. Introduction

1.1. Overview of the skyline problem

Skyline queries are important for applications in which the preferences of the user are the ones that determine its result. One such example is of the tourist who wants to locate the hotels which better satisfy the preferences of his stay in an area. Each hotel is represented by two values: the distance from a specific point (E.g. a beach) and its price. The users wants to discover the hotels which have the lowest price and also the smallest distance from the beach. Skyline queries try to find items that "*dominate*" on other items. Hotel A "*dominates*" B, if it is at least as close to the beach as B but offers a better price. [1]

Finding the "*dominance*" essentially means finding the items with the lowest values in all the dimensions. Items that are dominating, cannot be dominated, because that would mean that another object has at least a dimension that is "better".

In *Fig. 1* one can see how the example's data are represented. The X-axis depicts the hotel's distance from the beach and the Y-axis depicts the price. The line that connects the points nearest to two axes represents

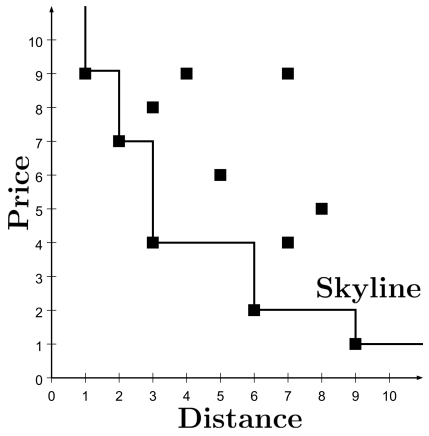
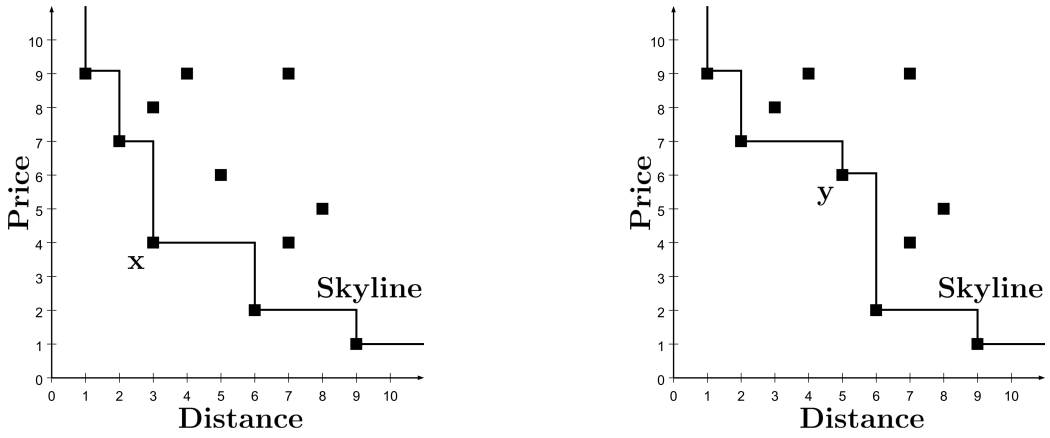


Figure 1: Skyline Example

the skyline. These points dominate all other points of the data-set but do not dominate each other. The skyline queries can be expanded to more dimensions, in which an object dominates another if it has lower values in all dimensions. Algorithms that are related with skyline discovery are divided into two categories.

The algorithms that inspect the data statically, namely there are no insertions or deletions while executing the algorithm and all of the data are available at the beginning. The second category algorithms are valuable in scenarios where a program has to return results based on the user's preferences. E.g. a server in the tourist example calculates which are the best hotels, when a user specifies a region and also when new businesses are inserted or deleted. In such scenarios static analysis of data has no meaning because the cases are practically innumerable. One more case in which continuous skyline evaluation algorithms are useful, is the stock market. Traders are interested in finding both the best value of a share and the number of the shares that are negotiating in a specific price. Since transactions are represented in the time space, the traders are interested only in actions that occurred in a specific time frame. For that reason a mechanism that allows transactions that are no longer a part of the trader's interests to be removed dynamically after a certain time, is required.

Fig. 2 depicts the change of skyline in time, after deleting an object. The form of the skyline changes dynamically, so it must be continuously calculated and updated.



(a) Skyline at time t=a (b) Skyline at time t=b

Figure 2: Dynamic Skyline

1.2. Algorithms for skyline calculation

In databases, skyline queries have been examined thoroughly in the past. Borzsonyi et al. [2] develop two techniques that are based on divide and conquer and *Block Nested Loop*. The divide and conquer algorithm, divides the data into parts that fit into the main memory. The skylines on all different partitions are calculated separately in the main memory and then joined in the final result. The "*BNL*" algorithm checks each tuple in the database with every other and returns it, given that it is not dominated by another. Another algorithm is the "*SFS*" [3], which sorts the data of the database using a monotone function. Then the skyline can be calculated using a sorted list. There are also algorithms that use binary operations found in databases. Tan et al. [4] provide one more method based on the relations of the skyline and the lowest coordinates of specific points. Papadias et al. [5] and Kossman et al. [6] discover skylines using nearest neighbor techniques. [7] Continuous skyline algorithms are reviewed in the next section of the paper.

1.3. Related work

Even though a lot of work has been done and many algorithms have been proposed for the calculation of the skyline, not much is done in the evaluation of the algorithms and their proposed benefits. This paper tries to address

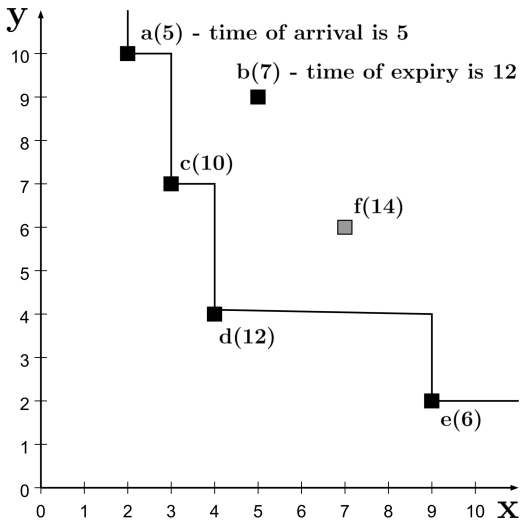


Figure 3: Skyline at time 14. A new tuple has arrived. The sliding window size is 5.

this issue by comparing three algorithms that are widely used by researchers to compare new algorithms.

2. Continuous skyline calculation algorithms

In this paper continuous skyline algorithms are presented and evaluated. Three different approaches are examined, the "*LookOut*" method presented in [1] and "*Lazy*" and "*Eager*" methods proposed by [7]. Certain aspects of the implementation of these three algorithms are given and then they are evaluated based on different metrics. These metrics include execution time, memory allocation and the size of the skyline.

2.1. The Lazy Algorithm

"*Lazy*" algorithm is presented in [7] by Yufei Tao and Dimitris Papadias. Changes in skylines can happen in two occasions:

1. a new tuple is inserted in the database
2. an object has "*expired*" and has to be removed

The "*expiration*" of the object is calculated as the time resulting from "*time of arrival + size of sliding window*". The lazy algorithm uses the pre-processing module ("**L-PM**") and the maintenance module ("**L-MM**").

When a tuple r is inserted in the system the "*L-PM*" module checks if it is dominated by another tuple in the current skyline. E.g. in *Fig. 3* the arrival of tuple f at time 14 does not affect the current skyline, because f is dominated by object d. For that reason f is saved in the database with the objects that are not being used currently in the skyline, but may appear in it later. The database which stores the "*inactive*" data is called "*DBrest*" and the database which stores the skyline is called "*DBsky*".

In the case which the incoming object dominates some of the skyline objects, it is stored in "*DBsky*" and the data that are dominated are deleted, because they will not appear again.

The algorithm also defines two regions of a tuple r:

1. the dominance region, called ***r.DR***
2. and the anti-dominance region ***r.ADR***

The $r.DR$ region has as starting point the coordinates of the object r and as ending the maximum coordinates that can appear. Contrary the $r.ADR$ region covers a region that spans from the star of the axes to the object itself. Figure *4* depicts the shape of $r.DR$ and $r.ADR$ in a 2-d example.

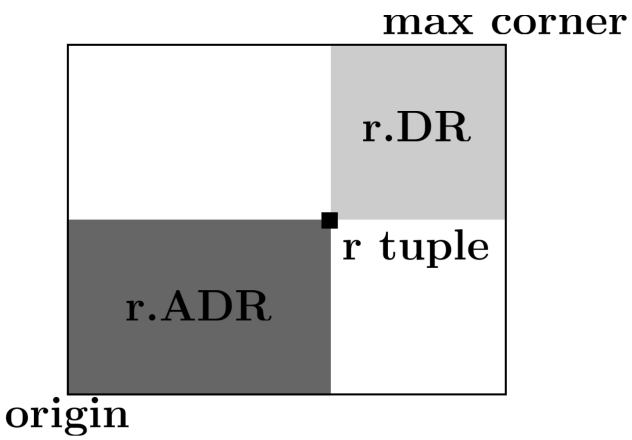


Figure 4: r.ADR and r.DR regions

When a tuple arrives a test is performed, to check if any of the objects that are already in the skyline are in the "$r.ADR$" region. In contrast, to find the objects that belong in the dominance region of the new tuple, the "*Lazy*"

Algorithm 1 L-PM algorithm

```
1: r = an incoming tuple
2: issue a d-dimensional emptiness test to check whether any point of DBsky
   is in r.ADR
3: if the test result is false (r.ADR contains objects) then
4:     insert r to DBrest and terminate
5: else
6:     issue a d-dimensional query to find the points of DBsky in r.DR
7:     for each tuple r' in the query result do
8:         output (-r', current time) and discard from system
9:     end for
10:    output(+r, current time) and add r to DBsky
11:    set T_exp^sky to the earliest expiry time of the tuples in DBsky
12: end if
```

algorithm, performs an *r.DR* query. If an object is found in the "*r.ADR*" region then the new tuple is stored in the "*DBrest*" database, where it will stay until it appears in the skyline, or expires. On the other hand, if there are objects in the "*r.DR* " region, they are expunged from the system and the new one is inserted in the skyline. The time of expiry for the skyline is set to the lowest value found in it. The pseudo code of "*L-PM*" module is shown in *Algorithm 1*

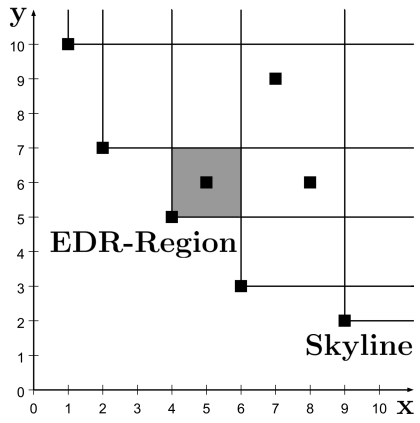


Figure 5: Exclusive dominance region for a 2-d example

The "*L-MM*" module is responsible for the maintenance of the data that already exist in the database. For that reason it is executed at the time specified by the "*L-PM*" module, which is when an object expires and has to be deleted from the skyline. The algorithm removes the specific object and also removes the objects that are stored in "*DBrest*" and have expired already. Then the skyline is recalculated, only for the objects that are dominated exclusively by a tuple r which is about to be deleted. In *Fig 5* the exclusive dominance region for a 2-d example is depicted. Then the algorithm defines the next execution time for the "*L-MM*" module, namely the time an object will be deleted from the system. The pseudo code is shown in *Algorithm 2*.

Algorithm 2 L-MM algorithm

1: r = the tuple in *DBsky* that expires
2: output(-r, T_{exp}^{sky}) and remove r from *DBsky*
3: r' = the first(oldest) tuple in *DBrest*
4: **while** $r'.t_{exp} < T_{exp}^{sky}$ **do** //r' has expired
5: expunge r' and set r' to the next tuple in *DBrest*
6: **end while**
7: compute the skyline for the set S of data in *DBrest* dominated exclusively by r, but not by any other skyline point
8: **for** each tuple r' in this skyline **do**
9: output(+r', T_{exp}^{sky}) and move r' from *DBrest* to *DBsky*
10: **end for**
11: set T_{exp}^{sky} to the earliest expiry time of the records in *DBsky*

2.2. The Eager algorithm

The "*Lazy*" algorithm has some disadvantages like the fact that is stores data that are obsolete and tuples that will not be used in the skyline at any time. Such reasons pushed its authors to consider a different approach, which led to algorithm "*Eager*" [7].

The "*Eager*" algorithm aims to resolve two main issues:

1. to lower the memory consumption by keeping only the tuples that are or will be part of the skyline
2. to lower the cost of the maintenance module, in this case the *"**E-MM** "*

It achieves these two goals, by doing more in the pre-processing module ("***E-PM***". In the "*E-PM* " module the "*influence time*" is calculated, which can predict at the time of arrival, if a tuple will be a part of the skyline at some future point. If there is no such time, the object can safely be ignored and not even stored in the database. The "*Eager*" algorithm uses a "***E****vent* ***L****ist*", in which the events are sorted in ascending order based on the time of their respective events. Such events are the expiration of an object, or the transfer from database to the skyline. Each tuple that is not a part of the skyline but, will be in the future, is *marked* and transferred to it at the proper time. Specifically in the "*E-PM* " module, for each incoming tuple, a query is made so that the tuples that are dominated by the incoming one are found. These tuples are then removed from the system. The new r tuple is inserted in the database and the "*influence time*" is calculated. The influence time is calculated by finding all the objects of the skyline that are in the "$r.ADR$ " region of the tuple r and then keeping the greatest expiry time of them. At that time point the tuple r will be transferred from the database to skyline. If the "*influence time*" calculated is equal to the arrival time, the tuple is inserted in the skyline directly and in the event list is marked with "*EX*" value. Otherwise it is stored in the database with the "*EL*" value. The pseudo code for "*E-PM*" module is shown in the *Algorithm 3*.

When the time for an event arrives the "*E-MM*" method is executed. This method is less complicated than its respective in "*Lazy*" algorithm, because more processing has been done in the "*E-PM*" module. Thus if the next event in the list is marked as "*EX*", then the tuple is simply removed from the system. Otherwise, the tuple is now a part of the skyline and a new event is stored in the event list to indicate the expiry time of the tuple. *Algorithm 4* contains the pseudo code of the "*E-MM*" module.

Algorithm 4 E-MM algorithm

e = the event with the minimum event time in EL
r = the record referenced by e
delete e from EL
if $e.tag = EX$ **then**
    output(-r, e.t) and expunge r from the system
else
    output(+r, $e.t$ and insert <r,$r.t_{sky}$, EX> into EL
end if

Algorithm 3 E-PM algorithm

1: r = the incoming tuple
2: issue a d-dimensional query for the $r.DR$ region to retrieve the tuples in DB dominated by r
3: **for** each tuple r' in the query result **do**
4: delete its event entry from EL, and discard r' from the system
5: if the event of r' is of type EX then output ($-r'$, current time)
6: **end for**
7: insert r into DB
8: issue a d-dimensional max search for the $r.ADR$ region to obtain the skyline influence time $r.t_{sky}$
9: **if** $r.t_{sky}$ equals the current time **then**
10: output($+r$, current time)
11: insert $<r$, $r.t_{exp}$, EX$>$ into EL
12: **else**
13: insert $<r$,$r.t_{sky}$, EX$>$ into EL
14: **end if**

3. The LookOut algorithm

The LookOut algorithm [1] connects each object of the database with a time interval for which it is valid. This time interval consists of the arrival time and the expiry time. The skyline can change in two occasions:

1. some skyline data are about to expiry
2. new data are inserted in the database

In the case of expiry, the entirety of the data has to be checked for tuples that where dominated by the expiring tuple i and should now be in skyline. These tuples must only be inserted in the skyline only if they are not dominated by others. In the case of insertion the skyline must be checked for tuples that dominate the incoming one. If this is not the case the new object is inserted in the skyline and every previous skyline object is checked if it is dominated by the new or not.

The LookOut algorithm takes advantage of two important observations in hierarchical spatial indexes, e.g. *R-Trees* [8] and *quadtrees* [9]:

1. If point p dominates all the corners of a node n, then p dominates all the objects of the node and its children.

2. If all the corners of a node n dominate a point p then all the objects and its children dominate that point.

Using these two observations "*pruning*" of nodes is possible, thus rejecting new objects is faster.

The pseudo code of "*LookOut*" algorithm is given in *Algorithm 5*. Each new object is inserted in the database (e.g. *R-Tree*) and then the expiry time is stored in a binary heap which contains all the expiry times sorted in ascending order. The object is checked if it belongs to the skyline by "***isSkyline***" algorithm. If an object must be removed then all candidates that may replace it in the skyline are calculated by "***MINI***" algorithm. Final insertion is only done if "*isSkyline*" algorithm returns true.

Algorithm 5 LookOut algorithm

```
1:  while  time!= endTime do
2:      ndp // new data point
3:      insert ndp into Tree and expiry time into Heap
4:      if isSkyline(Tree, ndp) is true then
5:          remove points from Skyline dominated by ndp
6:          add ndp to Skyline
7:      end if
8:      if top of Heap == currentTime then
9:          if point belongs to skyline then
10:             save point to variable DSP
11:         end if
12:         delete point from DB and Heap
13:     end if
14:     calculate new skyline points with MINI(DSP, Tree) and save them
        in NSP
15:     for every point p in NSP do
16:         if isSkyline(Tree, t) is true then
17:             add p to skyline
18:         end if
19:     end for
20:     update time
21: end while
```

"***isSkyline***" algorithm (shown in "*Algorithm 6*", uses a "*best-first*" search, namely nodes that have the lowest distance are inserted first in the heap. When expanding a node, if the lower left corner does not dominate the arriving tuple, it is not visited again and it is rejected. If the upper right corner of a child dominates the new tuple, the algorithm terminates with negative output and the tuple is not inserted in the skyline. If the node is a leaf, the tuple is compared with all the other tuples of the leaf, to check whether it dominates them or not. If there is such a leaf the incoming object is not inserted in the skyline, otherwise it is.

Algorithm 6 isSkyline algorithm

```
1: insert Tree into Bheap with distance 0
2: while BHeap isn't empty do
3:     Tree = top of Bheap
4:     if Tree is leaf node then
5:         if one of the entries of Tree dominates P_new then
6:             return false
7:         else
8:             continue
9:         end if
10:    end if
11:    if Child is part of the non-empty children of tree then
12:        if minimum corner of Child does not dominate P_new then
13:            continue
14:        end if
15:        if maximum corner of Child dominates P_new then
16:            return false
17:        end if
18:        insert Child into BHeap
19:    end if
20: end while
```

The "***MINI***" algorithm (shown in *Algorithm 7*), also uses a "*best-first*" search and a binary heap based on the distance from the axis origin to the coordinates of the point. An object that is about to be deleted is passed as an argument and returns the objects that are dominated by it. Moreover these objects are checked before insertion for domination by others, that have already been inserted. Following the same logic with "*isSkyline*" algorithm,

if the upper right corner is dominated by the object that is about to be deleted, the node is rejected, otherwise if it is an internal dominated node, it is inserted in the heap. If the node currently checked is a leaf, the local skyline is calculated and stored.

Algorithm 7 MINI algorithm

```
1: insert Tree into BHeap with distance 0
2: while doBHeap isn't empty
3:     if BHeap.top is a point then
4:         point = top of BHeap
5:         pIsDominated = FALSE
6:         for each element a in miniSkyline do
7:             if a dominates point then
8:                 pIsDominated = TRUE
9:             end if
10:            if pIsDominated is FALSE then
11:                insert point into miniSkyline
12:            end if
13:            pIsDominated = FALSE
14:        end for
15:    end if
16:    Tree = top of BHeap
17:    if P_sky dominates maximum corner of Tree then
18:        if Tree is a leaf node then
19:            find the local skyline of just Tree
20:            if point is part of the local skyline of Tree then
21:                if P_sky dominates point then
22:                    insert point into BHeap
23:                end if
24:            end if
25:        end if
26:    end if
27:    insert the children of Tree with their distance in BHeap
28: end while
```

4. Experimentation and Evaluation of the algorithms

4.1. Methodology

Data that can be adapted in different cases are needed for the experiments and the evaluation of the results. For that reason three data types were created and tested:

1. Correlated data
2. Anti-correlated data
3. Independent data

These categories enable the creation of datasets with different distribution of data in the hierarchical spatial indexes. In this paper R-Trees where used as the hierarchical spatial index [8]. R-Trees create nodes based on the data that were inserted in them. Every node, starting from the root contains data that are contained in the rectangle of the particular node. The node is created using the corner coordinates of the rectangle and as extension the data that are inside this node. This specific feature is important, because it enables algorithms to traverse the tree and prune nodes that are insignificant.

Algorithm 8 BBS algorithm

```
1: sk =
2: insert all entries of the root R in the heap
3: while heap not empty do
4:     remove top entry e
5:     if e is dominated by some point in S then
6:         discard e
7:     else
8:         if e is an intermediate entry then
9:             for each child e_i of e do
10:                if e_i is not dominated by some point in S then
11:                    insert e_i into heap
12:                else
13:                    insert e_i into S
14:                end if
15:            end for
16:        end if
17:    end if
18: end while
```

Moreover "*BBS* " [5] algorithm (shown in *Algorithm 8*) was used for skyline computation, in all three algorithms. The "*BBS* " algorithm traverses from the root of the tree and expands each node, storing in ascending order the distances from the axes origin. In each iteration the node with the lowest distance is expanded or discarded. If the node is dominated by the existing skyline it is rejected, otherwise kept. When the algorithm finds a leaf, it inserts the data in the skyline, because they already have been checked.

In the case of the "*Lazy*" algorithm the "***E****xclusive* ***D****ominance* ***R****egion*" must be calculated, for "*L-MM*" algorithm to work. This is easily achieved in 2-d datasets. An array sorted in ascending order is needed for each dimension of the skyline points. Then finding the next value, after the point that is about to be deleted, creates a tuple that has the upper right corner of the exclusive dominance region. Using the coordinates of the point to be deleted, with the coordinates of the upper right corner, the "*EDR*" region is calculated. *Fig. 6* depicts this calculation.

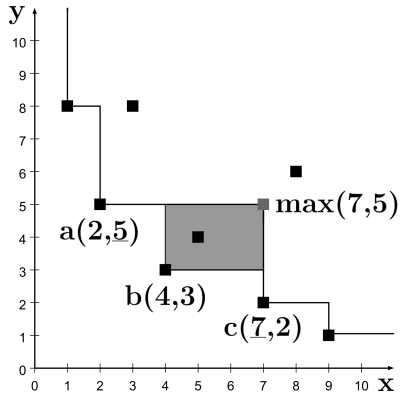


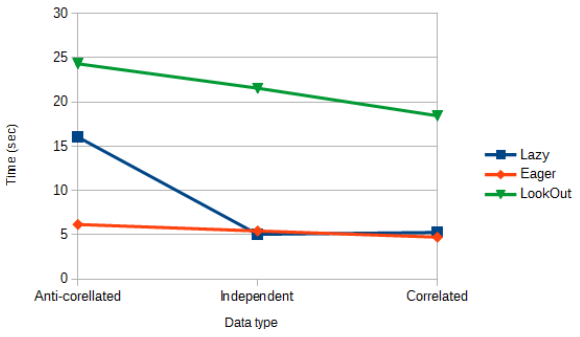
Figure 6: Calculating a 2-d EDR

On the other hand, in more than 2 dimensions the shape of the "*EDR*" becomes complicated and its calculation hard or even impossible [10]. The authors of [7] don't state, how the "*EDRs*" were calculated and if the datasets used on the experiments allowed the creation of "*EDRs*" that could be calculated easily like in the case of 2-d datasets. For that reason the experiments of this paper were conducted with the dominance region of a point, in more than 2 dimensions. This technique works for all dimensions.

4.2. Time performance of the algorithms

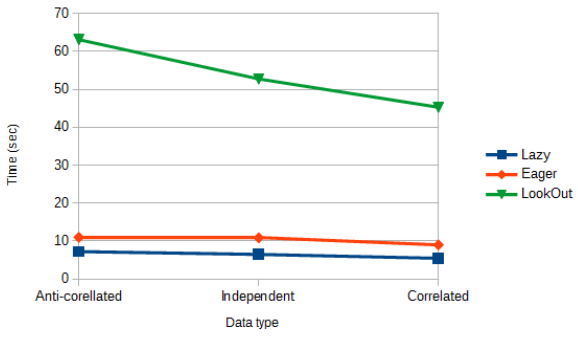
Various tests were conducted, based on the dimensions and the size of the "***S**liding **W**indow*". In all the tests the "*Eager*" algorithm achieves the best performance, as tuples are checked only once at the time of the arrival, if they belong in the skyline or not. What is more the "*Eager*" algorithm has a linear scaling in all dimensions and "*SW*" sizes. The "*Lazy*" algorithm has similar performance for 2-d datasets, but on more dimensions its performance is heavily compromised (*Fig. 10 - 15*). This is a result of the dominance region that is used in more than 2 dimensions. In this case the search region is far greater than in the "*EDR*" region, thus the number of tuples to be checked each time is also much greater.

"*LookOut*" algorithm is worse in all cases compared to the other two. For small "*SW*" sizes the difference is comparable, but when the size becomes greater than some hundreds the execution time is increased dramatically. One of the reasons of this behavior is caused by "*MINI*" sub-algorithm. For "*mini-skyline*" to be calculated, all the tuples that have not been "pruned" in the expansion phase, are possible insertions in the skyline and have to be checked. When the "*SW*" size is getting larger, more tuples are possible members of the skyline and must be checked with each other. Another issue of the "*LookOut*" algorithm, is after the execution of the "*MINI*", when the "*isSkyline*" has to be executed, so that the possible members are sorted and accordingly rejected, or inserted in the skyline.



Data Type \ Algorithm	Lazy	Eager	LookOut
Anti-correlated	16.05	6.17	24.32
Independent	5.06	5.44	21.56
Correlated	5.28	4.74	18.45

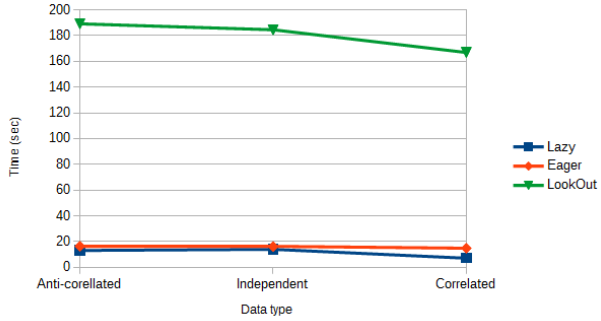
Figure 7: 1M 2-d data with sliding window of 100



Data Type \ Algorithm	Lazy	Eager	LookOut
Anti-correlated	7.26	11.01	63.11
Independent	6.52	10.89	52.75
Correlated	5.48	9.03	45.29

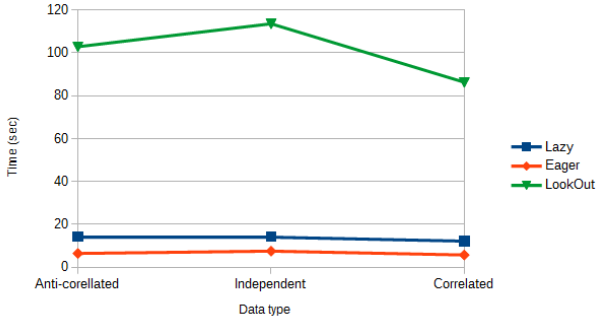
Figure 8: 1M 2-d data with sliding window of 1K

One more observation lies in the fact that all 3 algorithms seem to perform better in "Correlated" data type. This probably is due to different "*R-Trees*" structure and better "*MRBs*" creation , across different data types, which results in faster traversals of the tree. Figures *7* - *15* depict all these observations.



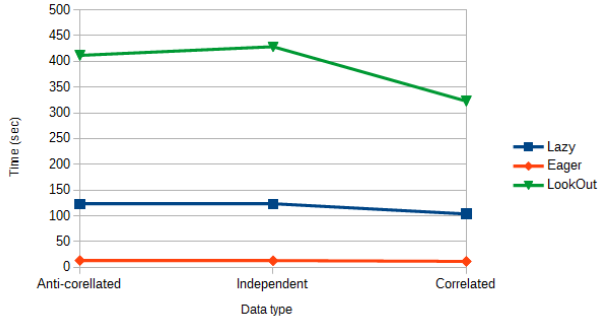
Data Type \ Algorithm	Lazy	Eager	LookOut
Anti-correlated	13.09	16.41	189.27
Independent	13.99	16.18	184.63
Correlated	7.12	14.85	166.96

Figure 9: 1M 2-d data with sliding window of 10K



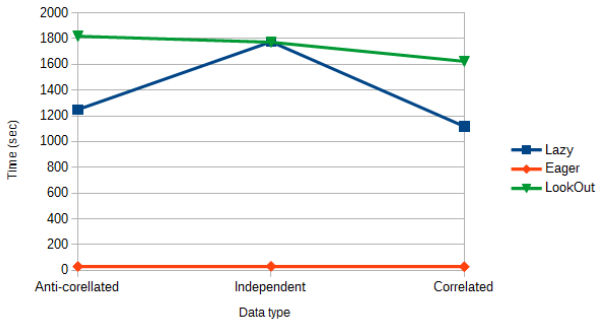
Data Type \ Algorithm	Lazy	Eager	LookOut
Anti-correlated	14.19	6.49	102.87
Independent	14.18	7.56	113.59
Correlated	12.20	5.76	86.32

Figure 10: 1M 4-d data with sliding window of 100



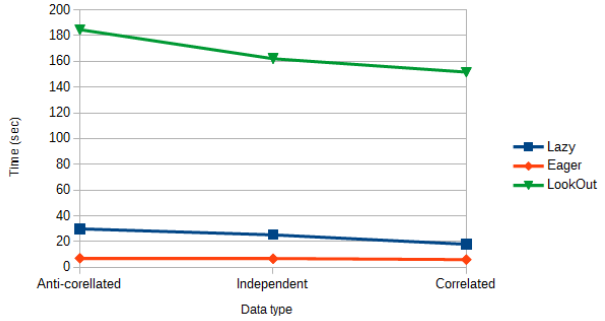
Data Type \ Algorithm	Lazy	Eager	LookOut
Anti-correlated	122.89	13.29	411.80
Independent	123.09	12.93	428.73
Correlated	103.70	11.62	323.01

Figure 11: 1M 4-d data with sliding window of 1K



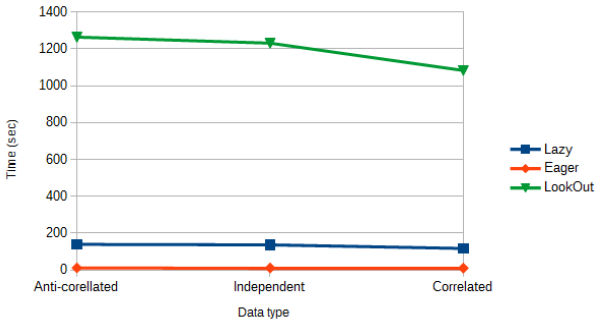
Data Type \ Algorithm	Lazy	Eager	LookOut
Anti-correlated	1249.9	30.09	1819.39
Independent	1776.30	31.15	1773.00
Correlated	1118.71	28.10	1624.72

Figure 12: 1M 4-d data with sliding window of 10K



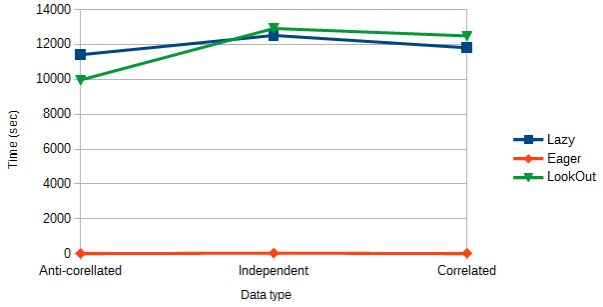
Data Type \ Algorithm	Lazy	Eager	LookOut
Anti-correlated	29.91	7.09	184.71
Independent	25.24	6.72	162.12
Correlated	17.96	6.00	151.75

Figure 13: 1M 6-d data with sliding window of 100



Data Type \ Algorithm	Lazy	Eager	LookOut
Anti-correlated	139.78	12.49	1264.03
Independent	137.27	11.79	1231.35
Correlated	117.59	10.93	1083.14

Figure 14: 1M 6-d data with sliding window of 1K

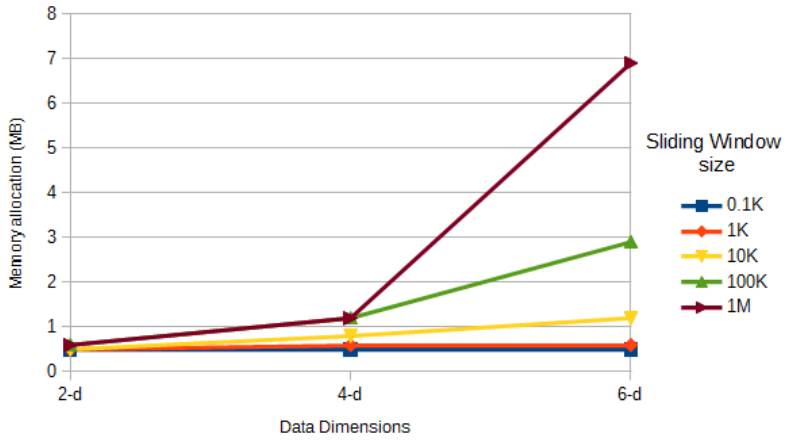


Data Type \ Algorithm	Lazy	Eager	LookOut
Anti-correlated	11415.70	27.59	9969.60
Independent	12522.34	47.05	12920.20
Correlated	11814.40	38.53	12492.90

Figure 15: 1M 6-d data with sliding window of 10K

4.3. Memory consumption of the algorithms

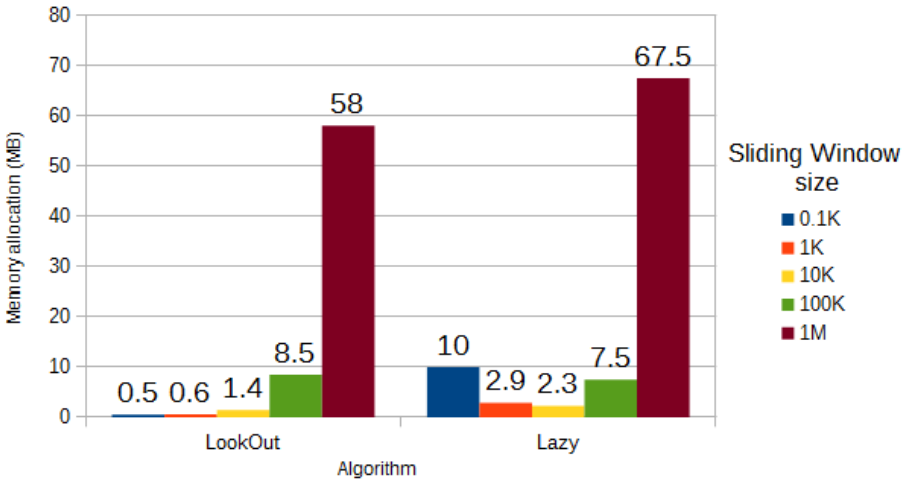
Authors of [7] state that "*Eager*" algorithm was developed to consume less memory than "*Lazy*". This is verified by the experiments, because even in the 6-d datasets and the largest "SW", the algorithm consumes less than 10MB of memory - *Fig. 16.*



"SW" size \ Data dimensions	2-d	4-d	6-d
0.1K	0.5	0.5	0.5
1K	0.5	0.6	0.6
10K	0.5	0.8	1.2
100K	0.6	1.2	2.9
1M	0.6	1.2	6.9

Figure 16: Memory consumption of "*Eager*" algorithm in MB

On the other hand the "*Lazy*" and the "*LookOut*" algorithms have higher memory consumption, since they exceed in some cases hundreds of MB even in 2-d datasets. "*LookOut*" algorithm displays fluctuations in the memory allocation, that are not proportionate to the size of the "*Sliding Window*" or the data dimensions. "*Lazy*" algorithm has normal fluctuations, but when the "SW" size is 1M the memory consumption reaches 50MB - *Fig. 17.*

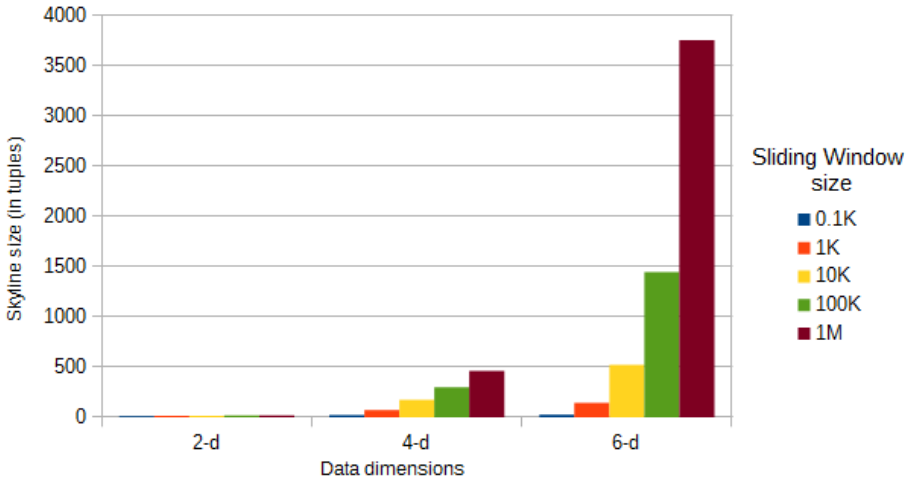


"*SW*" size \ Algorithms	LookOut	Lazy
100	0.5	10
1K	0.6	2.9
10K	1.4	2.3
100K	8.5	7.5
1M	58.0	67.5

Figure 17: Memory consumption of "*LookOut*" and "*Lazy*" algorithms in MB

4.4. Skyline size

In this section the size of skyline is examined. The dimensions and the size of the "*SW*" affect the size of skyline. The fewer dimensions the dataset has, the smaller the size of the skyline is. This is logical, as the possibility of dominance in more dimensions is reduced, thus increasing the skyline. The same principle stands for the size of the "Sliding Window" as shown in *Fig. 18*. The size of it, affects the size of the skyline. In a bigger time frame, more tuples are possible members of the skyline.



Data dimensions / "*SW*" size	**2-*d***	**4-*d***	**6-*d***
0.1K	11	22	28
1K	16	74	143
10K	17	174	522
100K	19	298	1447
1M	20	463	3754

Figure 18: Skyline size for different data dimensions and "SW" sizes

5. Future work

Skyline queries are an interesting and well studied area of scientific research. But there are a lot of challenges and questions that need answers. What is more, skyline queries are an integral part of other fields. As stated in the previous sections skyline queries can be used for finding the best results based on a user's preferences or even monitoring the stock prices.

This paper presented three skyline query algorithms that are used by many researchers, so as to compare their algorithms. Experiments established the fact that dimensions and the size of the "SW" are the main factors that affect the performance and the effectiveness of an algorithm, something that is not clearly visible in small datasets.

In the future, the research must focus on the development of algorithms that calculate the skyline even more effectively, because information is constituted by more and more data. This means that the dimensions needed to represent information, are also growing in size. But this has a negative im-

pact in the time complexity of the algorithms. Even algorithms like "*Eager*" in big datasets need a considerable time to calculate the skyline, something that is not consistent with the users' needs. Researchers have already started to work on algorithms that take advantage of the raw power of multi-core systems [11] and high-end GPU configurations [12], that improve the skyline computation time considerably. But that is not enough, as distributed systems become widespread and the services based on them are also increasing. Skyline algorithms that can operate in such environments are needed. One such example is the algorithm presented in [13], but there are a lot that can be done in this field.

6. References

[1] M. Morse, J.M. Patel, and W.I. Grosky. Efficient continuous skyline computation. *Information Sciences 177*, pages 3411–3437, 2007.

[2] S. Borzsonyi, D. Kossmann, and K. Stocker. The skyline operator. *IEEE 17th International Conference on Data Engineering*, pages 421–430, 2001.

[3] J. Chomicki, P. Godfrey, J. Gryz, and D. Liang. Skyline with presorting. *IEEE 19th International Conference on Data Engineering*, pages 717–719, 2003.

[4] K.-L. Tan, P.-K. Eng, and B.C. Ooi. Efficient progressive skyline computation. *27th International Conference on Very Large Data Bases*, pages 301–310, 2001.

[5] D. Papadias, Y. Tao, G. Fu, and B. Seeger. An optimal and progressive algorithm for skyline queries. *ACM SIGMOD international conference on Management of data*, pages 467–478, 2003.

[6] D. Kossmann, F. Ramsak, and S. Rost. Shooting stars in the sky: An online algorithm for skyline queries. *28th international conference on Very Large Data Bases*, pages 275–286, 2002.

[7] Yufei Tao and Dimitris Papadias. Maintaining sliding window skylines on data streams. *IEEE Transactions On Knowledge And Data Engineering*, 18:377–391, 2006.

[8] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The r*-tree: an efficient and robust access method for points and rectangles. *ACM SIGMOD international conference on Management of data*, pages 322–331, 1990.

[9] H. Samet. The quadtree and related hierarchical data structures. *ACM Computing Surveys (CSUR)*, 16:187–260, 1984.

[10] Ping Wu, Divyakant Agrawal, Omer Egecioglu, and Amr El Abbadi. Deltasky: Optimal maintenance of skyline deletions without exclusive dominance region generation. *IEEE 23rd International Conference on Data Engineering*, pages 486 – 495, 2007.

[11] Sungwoo Park, Taekyung Kim, Jonghyun Park, Jinha Kim, and Hyeonseung Im. Parallel skyline computation on multicore architectures. *IEEE 25rd International Conference on Data Engineering*, pages 760 – 771, 2009.

[12] Kenneth S. Bgh, Sean Chester, and Ira Assent. Work-efficient parallel skyline computation for the gpu. *Proceedings of the VLDB Endowment*, 8:962–973, 2015.

[13] Joo B. Rocha-Junior, Akrivi Vlachou, Christos Doulkeridis, and Kjetil Nrvg. Efficient execution plans for distributed skyline query processing. *ACM 14th International Conference on Extending Database Technology*, 8:271–282, 2011.