

## B-line to the skyline. The fastest query in all the land.

This paper investigates skyline queries and how to optimise the speed in which these queries are resolved.

### What is a skyline query?

For the sake of simplicity let's assume that each node represents a stock and all we care about is its risk and return.

Given a set of nodes a skyline query will return nodes that are not dominated by any other nodes. For a node to dominate another node it has to be better on one dimension (risk) and better than or equal to on the other dimension (return). This list of nodes is also known as Pareto set and/or Pareto frontier.

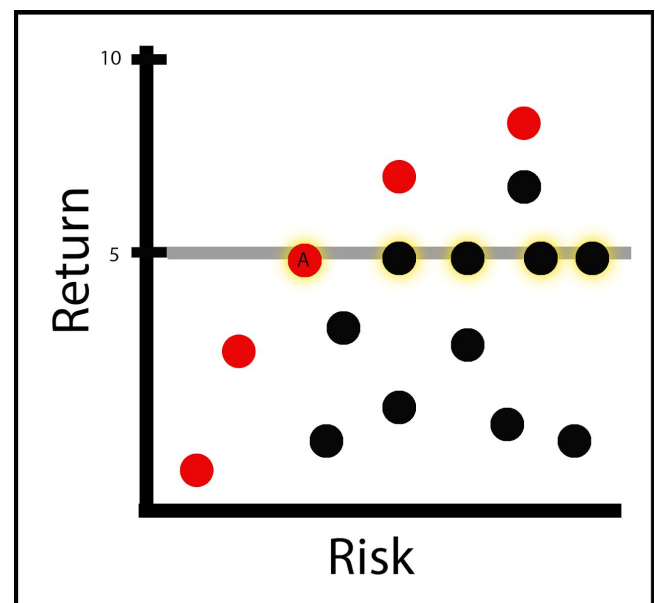
In the example on the right all the red stocks would be returned by a skyline query, the Pareto set. In this example the red nodes are the only stocks that a stock trader should be interested in. Let's say we want a stock that has a return of 5%, there are five stocks that meet this requirement. Out of these five stocks A has the lowest risk, if anyone was to buy the other highlighted stocks they would be taking on extra risk for no extra reward. Any stock that gives the optimal return compared to risk will be in the Pareto set.

### Why are Pareto set's useful and where are they used?

A skyline query has the ability to drastically reduce the set of data for any problem, without removing any optimal solutions. This

ability can be used to reduce the size of databases. It can also help with many multicriteria decision making applications. If each node represents a possible choice a decision making algorithm analysis needs to make, reducing the number of nodes will drastically speed up the algorithm. It can screen incoming data for real time decision making algorithms which are always time sensitive. As a skyline query gets faster and more efficient it also becomes more versatile. For most of the situations there will be a payload associated with each node and it is frequently required to be returned with each node.

Figure 1. Risk versus return, where the desirable stocks have high return and low



risk:

### Dumb algorithm

The first algorithm takes a collection of nodes as input. Each node had an x, y and payload value associated with it. X and y

are initially stored as ints and payload is stored as type string of length 4. This payload string could also be replaced to a pointer.

This simple algorithm loops through all of the input nodes. If the current node is not dominated by any nodes in *best* it is then added. Once a node is added to *best* it is compared to all other *best* nodes. If it dominates any other node the dominated node is removed. The algorithm returns *best* and finishes once it has iterated through the input without adding a new node to *best*.

### Pseudo Code:

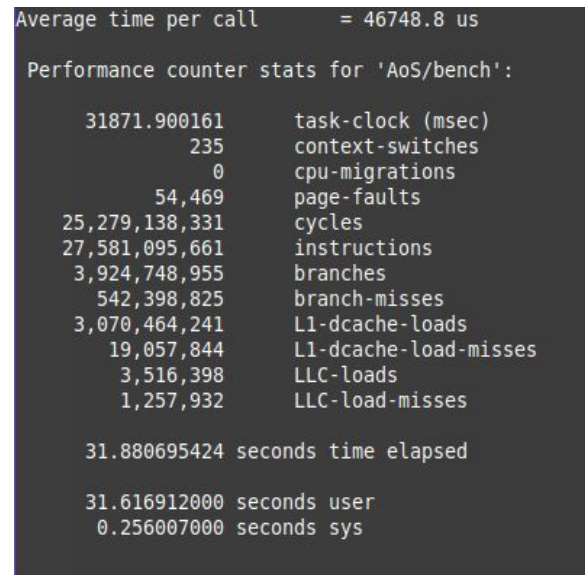
```
Node Best [ ]
Update = True
While( Update )
    For( i < input.size i++)
        Update = False
        Add = True
// If nothing dominates the input node add it to
best
        For(k< Best.size k++)
            If( Best[k] Dominates input[ i ]
            OR equal)
                Add = False
            if( Add )
                Best.add( input [ i ] )
                Update = True
// Remove anything the input node dominates
        For( j = 0 ; j < Best.size ; j++)
            if( input[ i ] Dominates Best[ j ] )
                Remove Best [ j ]

Return Best
```

### The Bottleneck

Perf is a performance analysis tool in Linux, it gives the user access to a lot of cpu counters.

For the scope of this paper it is fine to treat LLC as L2 cache. A third of all LLC loads Figure 2. Perf statistic running our Dumb algorithm:



```
Average time per call      = 46748.8 us

Performance counter stats for 'AoS/bench':

   31871.900161      task-clock (msec)
         235         context-switches
           0         cpu-migrations
        54,469       page-faults
25,279,138,331      cycles
27,581,095,661      instructions
 3,924,748,955      branches
   542,398,825      branch-misses
 3,070,464,241      L1-dcache-loads
   19,057,844      L1-dcache-load-misses
    3,516,398      LLC-loads
    1,257,932      LLC-load-misses

   31.880695424 seconds time elapsed

   31.616912000 seconds user
    0.256007000 seconds sys
```

are missed, shown in the image above (Figure 2). This paper will explore two ways to combat this poor performance. The first way is to implement a new algorithm focused on sorting the data, discussed in the next section. The second solution is to reduce the size of the working data.

### Sorting based algorithm

This algorithm sorts the input, once sorted it only needs to loop through the data a single time.

The input is sorted from best to worst based on x then y values. Given A (10, 2), B (3, 4), C (1,5) and D (10,3) the sorted order would be: C, B, A, D. Once the input is sorted the algorithm would simply keep any node that had a y value better then the current best y value. This works because all nodes to the right of any given nod will have a less valuable x and for it to not be

dominated by a right node it must have a more desirable value of y.

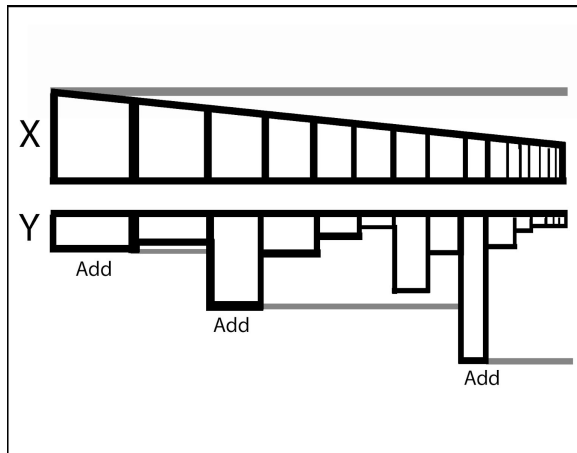


Figure 3. Image showing how the sort algorithm works where greater x and y values are preferable.

### Dumb algorithm optimisation using SoA

Both our dumb and sort based algorithm used an AoS data structure. AoS stands for array of structures. The AoS used upto this point contains :

- Integer x
- Integer y
- String payload

When computing the skyline solutions, only x and y values are used. The payload of the nodes is not needed for any computation, however in the original AoS approach all three values would need to be loaded into cache. By switching to an SoA, hot and cold data can be separated. This separation of data reduces the amount pulled in to cache. The SoA Node structure contains:

- Array x[ ] // x coordinates
- Array y[ ] // y coordinates
- String name[ ] // name of data

Unlike the AoS which will have many individual objects the SoA will have separate arrays for each x, y, and payload.

Figure 4. Structure of arrays:

X1	X2	X3	...	Xn
----	----	----	-----	----

Y1	Y2	Y3	...	Yn
----	----	----	-----	----

Payload 1	Payload2	...	Payload n
-----------	----------	-----	-----------

In theory the two optimizations mentioned above should have better performance.

### Experiment setup and benchmarking

Benchmarking is designed to test and stress the algorithms with realistic data. Each benchmarking test is run using randomly generated data multiple times, the results are then averaged and reported. When comparing two algorithms the same randomly generated dataset is used.

Most of the benchmarking is done with 20,000 points/records and 50 test instances although other setups were also tested. Both x and y coordinates are assigned a random integer from 0 to 2147483647 (RAND\_MAX) using a number generator. The corresponding payload is also randomly generated. The payload is string containing a random combination of 4 lowercase alphabetical characters. The size of the payload is small enough that it could be smaller or equal to the size of a pointer(size of pointer depends on OS). This is important because if the payload is of greater size the string could simply be

replaced with a pointer without changing the size of the node structure.

The same benchmarking and timing script is used for each measurement in order to avoid discrepancies. This paper also outlines the result of modifying the size of the dataset and its effect on the overall speedup.

### Overall comparison

The benchmarking script is used to determine the amount of time each solution takes. Changing the number of data points/node i.e(x,y,name) and the number of test cases, helps to further stress the algorithms. In theory, the SoA dumb algorithm should always take the least amount of time.

Figure 5. AoS Dumb Algorithm:

```
Average time per call      = 46748.8 us
Performance counter stats for 'AoS/bench':

   31871.900161      task-clock (msec)
         235        context-switches
           0        cpu-migrations
       54,469       page-faults
  25,279,138,331    cycles
  27,581,095,661   instructions
   3,924,748,955   branches
   542,398,825    branch-misses
  3,070,464,241   L1-dcache-loads
   19,057,844     L1-dcache-load-misses
    3,516,398     LLC-loads
    1,257,932     LLC-load-misses

   31.880695424 seconds time elapsed

   31.616912000 seconds user
    0.256007000 seconds sys
```

From figures 5-7 is a significant decrease in time between each. Cache loading also decreases this can be seen in *L1-dcache-loads* and *LLC-loads*.

Figure 6. AoS Sort-Based:

```
Average time per call      = 2862.44 us
Performance counter stats for './bench':

  29005.117812      task-clock (msec)
           39       context-switches
            1       cpu-migrations
       39,785       page-faults
  23,020,443,519    cycles
  25,380,518,799   instructions
   3,407,953,825   branches
   501,444,594    branch-misses
  2,513,234,333   L1-dcache-loads
    12,682,698    L1-dcache-load-misses
    1,998,408     LLC-loads
    635,242       LLC-load-misses

   29.007459755 seconds time elapsed

   28.826150000 seconds user
    0.179988000 seconds sys
```

Figure 7. SoA Dumb Algorithm:

```
Average time per call      = 22936.5 us
Performance counter stats for 'AoS/bench':

  30506.301691      task-clock (msec)
           41       context-switches
            2       cpu-migrations
       54,470       page-faults
  24,207,463,299    cycles
  26,281,351,987   instructions
   3,636,273,417   branches
   524,980,754    branch-misses
  2,741,913,339   L1-dcache-loads
   26,061,431     L1-dcache-load-misses
    3,564,606     LLC-loads
    933,794       LLC-load-misses

   30.509091005 seconds time elapsed

   30.251432000 seconds user
    0.256029000 seconds sys
```

Varying the size of the dataset will allow for testing L1 cache utilization, LLC cache utilization, and RAM utilization. The solutions are benchmarked with 50 test instances using 10 to 100,000 data points. The following table and graph contain the results.

Table 1 Time measured in microseconds( $\mu$ ).

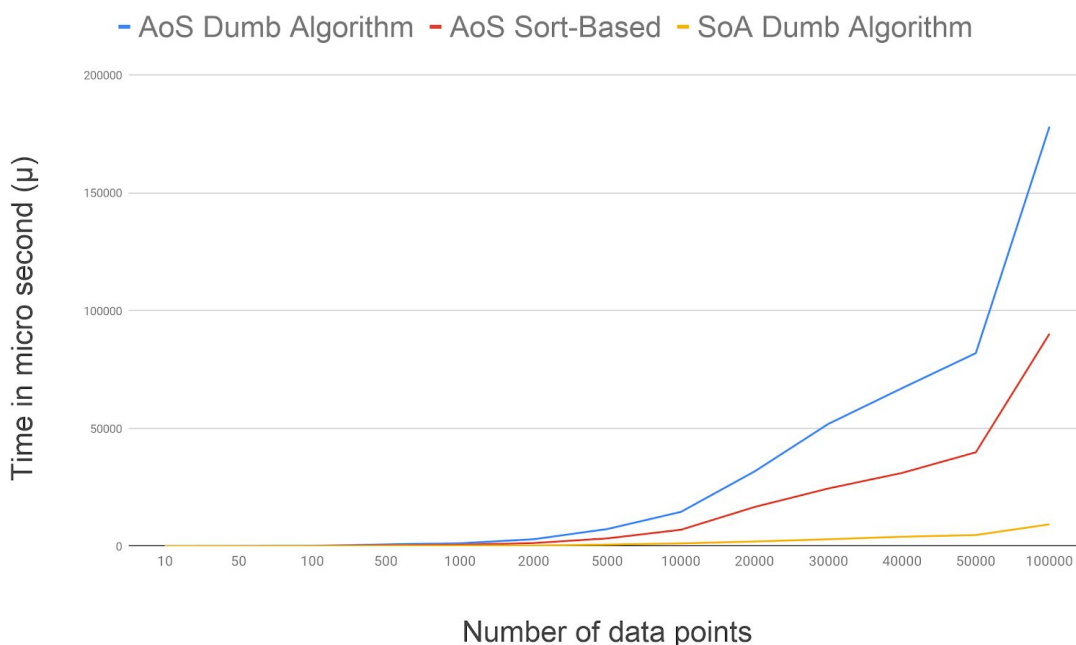
Number of data points	AoS Dumb Algorithm	AoS Sort-Based	SoA Dumb Algorithm
10	12.82	5.32	6.52
50	56.56	23.94	15.22
100	117.64	48.76	23.62
500	727.38	342.92	73.66
1000	1189.86	599.64	131.78
2000	2911.66	1299.7	238.14
5000	7239.12	3273.1	656.18
10000	14532.3	6944.02	1149.5
20000	31724.1	16611.5	1957.38
30000	51879.1	24467.7	2908.52
40000	67058.2	31069.2	3977.64
50000	81917	39828.8	4698.4
100000	178080	90170	9247.04

Figure 8. How number of node effects performance:

Figure 8 shows how changing the size of the dataset affects performance. When comparing AoS Dumb and AoS Sort, there is a significant increase at two points.

The first point represents when the combined size of all the nodes can no longer fit in L1 cache. The second increase comes when the node no longer fits in LLC. Each time the nodes move to a slower memory storage location resulting in an overall slower time.

Both algorithms have to load the entire node into cache and RAM. This is why the rate of change in time increases at specific data points. Since AoS Sort has better reference of locality (latter explained), it will have better performance.



SoA dumb is not affected with data points less than 50,000. The rate of time increase at data point 50,000 is significantly less compared to the other algorithms.

Evidence to support the optimisation claims

The table below represents the order of cache loading and unloading that was computer over several iteration.

Table 2. Cache performance ratio

	AoS Dumb Algorithm	AoS Sort-Bas ed	SoA Dumb Algorithm
L1 Load	3	2.7	2.5
LLC Load	3.5	3.5	2.0
L1 Miss	1.9	2.6	1.2
LLC Miss	1.2	0.9	0.6
Time (micro secs)	~50	~20	~0.2

Table 2 shows a correlation between cache loads decreases and algorithm overall speed ups. Each solution performs better than the previous one, this is the case even while varying the size of the dataset. The reduced time spent in loading cache memory is allowing for the overall decrease in runtime.

#### *Dumb Algorithm vs Sorted Algorithm*

The dumb algorithm operates in  $O(N^2)$ , this is because it loops through the entire dataset each time. Thus forces the cache to be fully flushed out multiple times resulting in back locality. The sorted

version operates in  $O(n) + O(n\log n)$ . As explained above, this algorithm sorts the input vector. The  $y$  value of the most recently added node in *best* will be compared to the input. This creates better temporal cache locality. The sorted solution also only loops through the input one time after it is sorted, this is good because once data is evicted from cache it will never need to be reloaded.

The sorted input vector is accessed sequentially therefore there is an improvement in spatial cache locality. Overall the sorted algorithm accounts for locality of reference which in turn decreases the amount of loading and unloading of data in/from cache. This can be seen in Table 2.

#### *SoA vs AoS*

The SoA data structure will reduce the number of cache loaded since the payload will not be loaded. This decrease in size of data allows for more relevant data to be stored in each cache line resulting in more nodes being loaded for each load. Table2 shows that the amount of data being loaded into cache is significantly reduced when using an SoA. This reduction in memory load translates into a decrease in overall runtime. The SoA approach on average gives a 16.3 times speed up of.

Note: The dumb algorithm itself is not changed. The time complexity is still  $O(N^2)$ .

#### *Future Improvements*

1. Due to the time restriction for the first phase of this project, none of the proposed solutions are fully optimized . To further improve cache locality, tiling may be used in

the for loops to decrease percentage of cache miss.

2. Another method could be using the sort based algorithm with a SoA data structure. While this approach is hard to implement, it could potentially be the fastest solution.

## Conclusion

In conclusion, it is clear that the three different approaches described in this paper to solve the skyline problem perform differently based on how cache memory is utilized. A sort based approach to solve this problem can provide 2x speedup compared to a nested loop approach. Changing data structure to using one node with three arrays can further increase performance to an average of 16x speedup. As the size of the dataset increases, AoS's performance will also increase compared to SoA. For the scope of this assignment, the requirements being to attain a 5x speed up which the proposed solution does reach, there is room for additional optimization as mentioned above.

By Akash Charitar (V00875728)  
and Ben Shapland(V00819775)  
Monday February 24, 2020