INTRODUCTION
Skyline queries are important for several applications involving multi-criteria decision making, where it is less important to output all points of the result set and instead only interesting results are required. In this report, we discuss algorithms to deal with continuous k-dominant skyline queries.

Motivation
As an example, consider the following situation where an online customer is looking for laptops in a website. He will have several preferences in his mind about the various features his laptop should support like the price of the laptop, the battery life, weight of the laptop, processor speed, memory etc. It would be really cumbersome for that person to look at all laptops in the website. Instead if he issues a skyline query, most of the laptops are going to be pruned from his search space. Thus, only laptops that are not worse than any other laptop in all dimensions are contained in the skyline. From skyline, one can make decisions based on one’s personal preferences.

Skyline and its variants

A General Skyline query
Given a d-dimensional dataset, a point ‘p’ is said to dominate another point ‘q’ if it is not worse than ‘q’ in all dimensions. The skyline of P is the set of those points of P which are not dominated by any other point in P.

Assume that we have a database of N objects. Each database object p with d real-valued attributes can be conceptualized as a d-dimensional point \((p_1, \ldots, p_d)\) where \(p_i\) is the i-th attribute of \(p\). Figure 1 a illustrates a database of six objects \(P = \{a, b, c, d, e, f\}\) each representing the description of a hotel with two attributes: distance to beach and price.
Figure 1 b shows the corresponding points in the 2-dimensional space where $x$ and $y$ axes correspond to the range of attributes distance and price, respectively. It shows the skyline points $S = \{a,c,e\}$. These are the points which are not dominated by other points i.e., better than other points in atleast 1 attribute. In the following sections, we define key terms related to this problem thus leading to explaining continuous k-dominant spatial skyline queries based on the definition of the general skyline query.

*A k-dominant skyline query*

As the number of dimensions increases, the chance of one point dominating another becomes low. As a result, the number of skyline points become numerous to provide any helpful information for the person. In [3], the authors propose a new concept called k-dominant skyline query thus relaxing the idea of d-dominance to k-dominance.

A point ‘$p$’ is said to k-dominate another point ‘$q$’ if there are k dimensions in which ‘$p$’ is better than or equal to ‘$q$’ and better than ‘$q$’ in atleast one dimension [3].

<table>
<thead>
<tr>
<th></th>
<th>$d_1$</th>
<th>$d_2$</th>
<th>$d_3$</th>
<th>$d_4$</th>
<th>$d_5$</th>
<th>$d_6$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_1$</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>$p_2$</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>$p_3$</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>$p_4$</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>$p_5$</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>1</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

*Figure 2: An example dataset*

From this point, we will refer to the general skyline as a free skyline or a d-dominant skyline. In the table shown in figure 2, \{$p_1, p_2, p_3, p_5\} form a free (6-dominant) skyline, \{$p_1, p_2, p_3\} form a 5-dominant skyline while \{$p_1, p_2\} form a 4-dominant skyline.

Now let us look at this with a detailed example: Suppose we are interested in obtaining the results for 5-dominat skyline from the above dataset. Then any point which is a part of the result should be better than other in all 5-dimensions. i.e., all 5 dimensions for the above dataset would be

\{d_1,d_2,d_3,d_4,d_5\}
\{d_2,d_3,d_4,d_5,d_6\}
\{d_3,d_4,d_5,d_6,d_1\}
\{d_4,d_5,d_6,d_1,d_2\}
\{d_5,d_6,d_1,d_2,d_3\}
\{d_6,d_1,d_2,d_3,d_4\}

For any point ‘$p$’ to be a part of the 5-dominant skyline, it should be better than all other points in all the above combinations of 5-dimensions. A point ‘$p$’ is said to be better than another point ‘$q$’ if it is not worse than ‘$q$’ in all dimensions.
With the above definitions in mind, we clearly see that \( P_4 \) is not a part of 5-dominant skyline as it is dominated by other points in all dimensions. \( P_5 \) is not a part of 5-dominant skyline as it dominated by other points in all 5-dimensions not involving \( d_4 \).

In the case of k-dominant skyline, dominance relationships may be cyclic and may not be transitive. There might be situations in a k-dominant skyline where ‘A’ k-dominates ‘B’, ‘B’ k-dominates ‘C’ and ‘C’ k-dominates ‘A’. This is one of the major reasons why existing skyline algorithms in the literature cannot be readily extended to support k-dominance evaluation.

*Continuous k-dominant skyline query*

Continuous k-dominant skyline operation involves data points that are continuously being added are removed from the k-dominant skyline. Each data point has an arrival time and a validity time or expiry time beyond which the data points moves from its current location and hence becomes invalid. For example, people might query details regarding flight prices. The travel agents may wish to submit prices for flights that are valid only across a period of time after which they change and therefore become invalid.

**Our contributions**

Our contributions in this paper are as follows:

1. In this project, we discuss a technique to support Continuous k-dominant Spatial Skyline queries.
2. We also prove that the algorithm produces correct and complete results. We also show that the improved approach is better than the naïve approach in terms of the time taken to perform k-dominance.

**RELATED WORK**

As location-aware mobile services are rapidly growing, efficient processing of advanced, location-based skyline queries has become extremely important. Much of the existing work assumes that the attributes associated with the data and query points are static. Skyline queries are important for various applications including customer information systems, decision support and data visualization.

Inspired by multivariate optimization, maximum vectors and contour problems, [4] proposed the skyline operator in the context of relational database and introduced *Block Nested Loop* (BNL) and *Divide-and-Conquer* (D&C) algorithms. In these algorithms, they basically showed how B-Tree and R-Tree can be used to process skyline queries. BNL scans the dataset sequentially and compares each new point to all skyline candidates kept in memory. D&C, recursively breaks up large datasets into smaller partitions. This is continued till each smaller partition of the dataset fits in the main memory.

[5] proposed a progressive processing algorithm *Nearest Neighbor* (NN) based on the depth-first nearest neighbor search by assuming the presence of some index structure like R-tree. In this, the skyline points are identified by repeated application of a nearest neighbor search technique on the data points, using a suitably defined \( L_1 \) distance norm.
Papadias et al [6,8] proposed an improved algorithm named Branch-and-Bound Skyline (BBS) based on the best-first nearest neighbor search. By accessing only nodes that contain skyline points, BBS incurs optimal node access and so far is the most efficient skyline algorithm in static settings. In [7] Tan et al, propose an algorithm which plane sweeps along each of the \( d \) dimensions. One advantage of this technique is that it can detect early termination.

In [1], Sharifzadeh et al exploit geometric concepts of the space to develop their algorithms. In their paper, the authors also propose a voronoi-based technique which can efficiently handle continuous queries. But the algorithm has been designed to handle \( d \)-dominant skyline queries considering all dimensions. Exploiting geometric concepts of the space becomes non-trivial and difficult to apply when \( k \)-dominant skyline needs to be evaluated. While the approach discussed in this paper doesn’t exploit geometric concepts, it still handles continuous evaluation of \( k \)-dominant skyline.

In [2], the authors propose a novel kinetic-based data structures and an associated efficient query processing algorithm. In their paper, the authors basically investigate the spatio-temporal coherence of the continuous skyline evaluation problem and propose an elegant strategy to handle moving objects. In [9], the authors propose a best-first strategy to prune the search space while assuming the presence of index trees like R-Trees. While [2,9] can handle continuous queries, they become difficult to extend for handling continuous \( k \)-dominant queries.

In [3], the authors proposed the idea of \( k \)-dominant skyline queries where the authors propose three algorithms to handle such queries. But the algorithms assume that the attributes associated with data points are static. In our project, we overcome this assumption by supporting continuous \( k \)-dominant skyline queries.

**PROPOSED APPROACH**

The task at hand is to compute the \( k \)-dominant skyline continuously on the objects that are valid at any given time. From this point, skyline and \( k \)-dominant skyline will be used interchangeably. Also, we will refer to the \( d \)-dimensional skyline as free skyline henceforth. Each arriving point comes with two times, arrival time (AT) and validity time (VT) i.e., the time for which the attributes associated with the data point will remain valid. For example, air ticket agencies keep changing the flight ticket prices and provide different deals at different times of the day. With these times, it is quite trivial to determine the expiry time (ET) as \( ET = AT + VT \).

A continuous \( k \)-dominant skyline may change due to one of the following:
1. A new data point ‘\( p \)’ may arrive at a time ‘\( t \)’.
2. An existing data point ‘\( q \)’ may expire and therefore become invalid.

Now let us look at a naïve method that can handle continuous \( k \)-dominant skyline. We will then overcome some of the problems associated with this naïve approach and see how this improved algorithm prunes the search space effectively and reduces memory consumption thereby giving quick, correct and complete results. In both the naïve and
improved approaches described below, we use two heaps, one storing the set of k-
dominant skyline points and another storing the set of non sky line points.

**Naïve Approach:**

**Case 1:**
When a new point ‘p’ arrives, it needs to be checked if it is a k-dominant skyline point.
For this, ‘p’ needs to be compared against all the points in the existing k-dominant skyline and also the heap containing the non-skyline points. If it were a normal skyline, it would have been sufficient to compare ‘p’ with just the skyline points. We can’t do this for k-dominant skyline because k-dominance is cyclic while d-dominance is transitive.

If ‘p’ k-dominates any existing point ‘q’ in the skyline, these k-dominated points are removed from the skyline and inserted into the heap containing non-skyline points.

If ‘p’ is k-dominated by some points in the skyline, then it clearly can’t be a part of the skyline. So, it is added to the set of non-skyline points.

If ‘p’ is not k-dominated by any existing point ‘q’ in the skyline, we still cannot add ‘p’ to the skyline as it may be k-dominated by some point in the set of non-skyline points. This is where k-dominance differs from the problem of d-dominance i.e., free skyline. In the case of free skyline, dominance relationships are transitive while k-dominance relationships may be cyclic. Because of this property, ‘p’ has to be checked for k-dominance with non-skyline points also.

**Case 2:**
If an existing data point ‘p’ that is a part of the k-dominant skyline expires, then ‘p’ needs to be discarded from the skyline as it is no longer valid. But, in addition to this, all the points in the non-skyline set that were k-dominated by ‘p’ need to be rechecked to see if they can now become a part of the k-dominant skyline because the k-dominating point ‘p’ has expired. After checking if each of the points is a skyline point, they are added to the skyline. Also, if there are any points in the skyline that this point dominates, they also need to be removed. Note that this step is required because of the possibility of cyclic dominance with respect to k-dominant skylines.

Also every ‘T’ seconds, the non-skyline heap is checked to remove all obsolete and expired points by checking their ET.

As we can see above, the number of dominance checks that need to be done in k-dominant skyline is much more than that required for the free skyline case. Clearly, the above approach is naïve and cannot be used especially when the points are arriving continuously.

So in the following section, we will try to minimize the number of k-dominance checks that need to be done by using the time attributes, AT and ET associated with every data point. We introduce a new metric return_skyline time i.e., the time a point may come to the skyline after all its dominating points expire out of the skyline.
Improved Approach:

Case 1:
When a new point ‘p’ arrives, it needs to be checked if it is a k-dominant skyline point. For this, ‘p’ needs to be compared against all the points in the existing k-dominant skyline and the heap containing non-skyline points.

If ‘p’ k-dominates any existing point ‘q’ in the skyline, ‘q’ is removed from the skyline and inserted into the heap containing non-skyline points if its expiry time is bigger than that of ‘p’. Otherwise ‘q’ is simply discarded. When ‘q’ is inserted into the non-skyline heap, the return_skyline time of ‘q’ is set to the expiry time of ‘p’. This means that ‘q’ cannot become a part of the skyline unless the k-dominating point ‘p’ expires. Also, the return_skyline time of all points that are k-dominated by ‘p’ in the non-skyline heap are also updated to the expiry time of ‘p’.

If ‘p’ is k-dominated by some points in the skyline or non-skyline, then ‘p’ clearly can’t be a part of the skyline. So, it is inserted into the heap of non-skyline points with return_skyline time set as the maximum expiry time of all its k-dominating points. Also, while inserting all points in the non-skyline heap that ‘p’ k-dominates are discarded if their expiry time is lesser than that of ‘p’.

Case 2:
When an existing point ‘q’ that is a part of the k-dominant skyline expires, the point can just be discarded from the existing skyline. One should note that all points that are k-dominated by ‘q’ will automatically be brought back to skyline because their return_skyline time has been set accordingly while these points arrived. So, we don’t have to perform any k-dominance checks to determine the set of non-skyline points that ‘q’ is dominating. This is the improvement that significantly reduces the number of k-dominance checks that would have otherwise been necessary in the naïve approach. Also, as we discard the points that are k-dominated and as a result can never come back to the skyline, we reduce the memory required to store the points.

As k-dominance checks are quite expensive, while dealing with continuously arriving points, one is particularly interested in an approach that would minimize the number of dominance checks that needs to be done while checking if a point is skyline or not. The improved approach is clearly more efficient as it prunes the search space while checking for k-dominance by setting the return_skyline time.

In order to support all the above operations, heaps are used to maintain details about the skyline and non-skyline points. All heaps are min-heaps where the expiry time (ET) associated with a point serves as the key in case of skyline heap and the return_skyline time serves as the key for non-skyline heap. To support the various events, such as deleting a point when it hits the expiry time, inserting a point into the skyline heap when a point hits the return_skyline time, we need an event manager that would call the relevant events at appropriate time instances. The event manger is again a list containing a set of events that need to be fired arranged in the increasing order of time. Each event may have details like the point on which the event has to be fired, the time instance at
which the event should be fired and the name of the event that has to be fired. For example an entry may look like ‘p₄,DIS,5’ - Discard the point p₄ as it expires at the time instance ‘5’. An event ‘p₆,RET,44’ would mean that the point p₆ needs to be inserted into the skyline heap at time instance 44 as all its dominating points would have expired from the skyline heap.

Let us look at an example to understand the naïve and improved approach better. Here the values in the brackets denote the AT and ET of points.

Let A (2,22), B(4,24), C(6,26), D(8,28), E(10,30), F(12,32), G(14,34) be the points in the k-dominant skyline. Let ‘H’ be a new point with AT = 16. Suppose the points {A,C,D,E,F} k-dominate the newly arriving point ‘H’. So, the maximum expiry time of all k-dominating points, in this case the expiry time of ‘F’, 32 is assigned as the return_skyline time of ‘H’ and ‘H’ is inserted into the non-skyline heap.

At the next time instance, the event manager fires the discard event of ‘A’ as ‘A’ expires at 22. If we are using a naïve approach, then all points that this expiring point k-dominates would have to be brought back from the non-skyline heap by performing k-dominance checks with respect to all the points in the non-skyline heap. But in case of the improved approach, we need not look into the non-skyline heap as all points will be inserted into the skyline at their return_skyline time.

Now consider a point ‘I’ arrives with AT = 18. Suppose ‘C’ is the only point k-dominating ‘I’ and also imagine ‘I’ k-dominates ‘H’. So, the expiry time of ‘C’ i.e., 26 is assigned as the return_skyline time of ‘I’ and inserted into the non-skyline heap. We also discard the point ‘H’ as its expiry time is less than that of ‘I’ and also ‘I’ k-dominates ‘H’. When the time instance 26 is reached, the event manager fires an event which would insert the point ‘I’ into the skyline heap.

**VALIDATION OF PROPOSED APPROACH**

In this section, we validate our approach by providing some intuitions through illustrative examples. We have also theoretically proved its correctness and completeness.

*Correctness:*
In order to show that the results returned by the improved approach is indeed correct, we will show that a point returned as a k-dominant skyline point is indeed a k-dominant skyline point and a point that is not k-dominant is actually not.

A point is declared as a skyline point only under the following two circumstances:

1. When it is not k-dominated by points both in the skyline and non-skyline heap.
2. When the time instance equals return_skyline time. This is correct because at return_skyline time all its k-dominating points would have expired.

A point is declared as a non-skyline point only under the following circumstances:

1. When it is k-dominated by some points in the skyline or non-skyline heap.
Completeness:
In order to show that the results returned by the improved approach are complete, we need to show that none of the skyline points are missed by the algorithm.

A point is discarded only in the following circumstances:
1. When the expiry time of a point I reached.
2. When it is k-dominated by some incoming point and its expiry time is less than that of the incoming point.

Thus, none of the actual skyline points are missed out from the results. Hence the results produced are complete.

EXPERIMENTAL ANALYSIS
We also did a simple experiment to analyze how the improved approach performs when compared to the naïve approach in terms of the time taken to do a k-dominance check. We generated data synthetically with 100, 200, 300, 400 and 500 data points with 6 dimensions and found the time taken to do k-dominance checks. As we can see from the below graph, the improved approach seems to perform better in terms of the response time when compared to the naïve approach. There is a fall in the time at x = 13 where the naïve approach seems to do better. We tried investigating the reason. We believe that there is some bug in our implementation which is causing this.

Clearly, the increase in performance comes from the fact that, a whole lot of k-dominance checks are saved when a point expires as it doesn’t have to look for the points it was dominating till then. This is facilitated by assigning the return_skyline_time whenever a new point is dominated.
CONCLUSIONS AND FUTURE WORK
There is always this curse of dimensionality, i.e., as the number of dimensions increases, the idea of dominance loses importance and more and more points get returned as a part of the skyline. Thus, in order to avoid the users from getting more points, it is desirable to get points that are more interesting than skyline points. One such thing that makes the skyline points more interesting are the k-dominant skyline points. We plan to investigate the applicability of k-dominant skyline in road networks. As we did very simple experiments due to limited time, in future, we plan to investigate the performance through implementation using some tools like network-object generator which generate objects continuously. By doing so, we can also estimate performance metrics other than response time like memory consumption, CPU time and I/O costs.

REFERENCES