Continuous Skyline Queries for Multi-Dimensional Moving Objects

Yu-Ling Hsueh† and Roger Zimmermann‡
†Department of Computer Science & Information Engineering, National Chung Cheng University, Chiayi, Taiwan, ‡Department of Computer Science, National University of Singapore, Computing 1, 13 Computing Drive, Singapore 117417
hsueh@cs.ccu.edu.tw       rogerz@comp.nus.edu.sg

A continuous skyline query retrieves a set of d-dimensional moving object points, which are not dominated by any other points in space. Each dimension represents a unique criterion and the skyline query result reflects all possible trade-offs among the best candidate points. Skyline queries are of crucial importance with the recent proliferation of location-based and e-commerce mobile services. To support these applications we assume that both query and data objects are dynamic, in contrast to much of the prior work. In this paper we propose C-Sky, a grid-based, progressive algorithm to efficiently compute continuous skyline queries in multi-dimensional and dynamic environments. Objects are indexed with a regular grid consisting of \( n^d \) cells. A search order imposed on the grid cells is introduced to reduce unnecessary cell visits. Moving from one cell to the next in search order, C-Sky performs three consecutive operations for each cell that it processes: (1) a block-nested-loop algorithm to locally compute the skyline points, (2) a pruning operation to remove unpromising cells from the search space, and (3) a merge operation to combine the latest skyline points with results from previous cell visits. A set of overall skyline points is obtained after the algorithm terminates. Because of its pruning abilities, C-Sky significantly reduces the required processing. We also propose a lazy update approach that updates an object only when its activity circle, a boundary on its movement, intersects with the currently processed cell. This approach further decreases the I/O overhead. We evaluate C-Sky through rigorous simulations and compare it with an existing technique. Experimental results are presented that show the performance and utility of our novel approach.

Key Words: spatial databases, skyline processing, continuous queries

1. INTRODUCTION

Skyline query computations are important for multi-criteria decision making applications and they have been studied intensively in the context of spatio-temporal databases [1,2,4]. Towards an efficient skyline computation the following challenges must be addressed: effective indexing of the data points, a fast strategy to reduce the search space dimensionality, and efficient collection of the final skyline points. Skyline queries have been defined as retrieving a set of points which are not dominated by any other points in space. For example, a user may look for gas stations which do not have all the parameters - such as the gas price and the distance to the user - worse than any other gas station. Therefore, \( p_1 \) is preferable to \( p_2 \) (or in this paper, we say \( p_1 \) dominates \( p_2 \)) if and only if \( p_1.price \leq p_2.price \) and \( p_1.distance \leq p_2.distance \). The skyline query result reflects the subset of gas stations which provide the best trade-offs between the gas price and the distance.

Some of the prior work on skyline queries assumed that query and data objects were permanently settled in their positions [1,2,3]. Other approaches assumed that the skyline computation involved dynamic query objects, but static data objects [5]. In recent years, there has been increasing interest in repositories of objects that are in motion to support a proliferating number of location-based and e-commerce mobile services. In these new applications both query and data objects are mobile.
and dynamic. Therefore, the set of skyline points changes over time and this type of search is termed a *continuous skyline query*. Existing techniques are not well-suited to provide efficient solutions for continuous skyline queries, hence new approaches are needed.

Continuous skyline queries are applicable to, for example, a battle simulation or a game world. A query example might be “find a list of my skyline enemies based on their distance to me, their energy strength, their speed and their direction.” Another example is an emergency response system where a query might be “return a list of rescue trucks for which their distance to a rescue team (which is moving toward the disaster area), their emergency supply, their speed and their direction are not worse than any other truck.” In this scenario, the skyline computation is important to enable decision makers to optimally dispatch their limited rescue resources.

In this paper we propose C-Sky, a grid-based, progressive algorithm to efficiently compute continuous skyline queries in \(d\)-dimensional and dynamic environments. The moving objects are indexed by an \(n \times n \times ... \times n = n^d\) grid \(G\). Each cell is associated with a list of moving objects residing in it. This grid structure is a crucial factor in C-Sky’s efficiency and it results in two important properties.

First, by processing cells in a natural order, large parts of the space can be dismissed without further inspection. The natural order of computing skyline points starts from the cell with minimum \(\text{mindist}\) to the origin in the coordinate plane. The reason is that it results in the removal of the maximum number of cells when performing a pruning operation if there are skyline points found in the cell. Figure 1 shows a simple example where there is a set of moving objects and a \(4\times4\) grid \(G\). The computational sequence starts from cell \(c_{0,0}, c_{0,1}, c_{1,0}, c_{1,1}, \text{ etc.}\), based on the \(\text{mindist}\) to the origin in ascending order. Because \(a\) is a skyline point that was found in cell \(c_{0,0}\), C-Sky prunes cells \(c_{1,1}, c_{1,2}, c_{1,3}, c_{2,1}, c_{2,2}, c_{2,3}, c_{3,1}, c_{3,2}\) and \(c_{3,3}\) (the gray area) since any points in these cells are guaranteed to be dominated by \(a\).

The second important property of the grid division is that skyline points can be independently and locally computed for each cell. As a result, the skyline results sets of the cells are merged, rather than requiring to incrementally add all the moving points of each subsequently processed cell into the computation. Hence, the merge operation is very efficient and a final list of overall skyline points is progressively obtained as the computation moves forward. To illustrate, consider the continuation of our example in Figure 1. The rest of the cells, which have not been eliminated, are further processed by C-Sky. While processing cell \(c_{0,2}\), \(e\) is found as its skyline point. A merge operation is performed and it combines \(e\) only with the skyline point \(a\) from cell \(c_{0,0}\) and \(k\) from cell \(c_{0,1}\), rather than all the moving objects from these two cells. To compute the local set of skyline points in a cell, the block-nested-loops method (BNL) [1] is performed.

The contributions of the C-Sky algorithm are that it is a progressive algorithm that can return a subset of skyline points without scanning the entire data set. C-Sky adopts a grid-based index for moving objects, and it does not make any
assumptions about the movement of objects (e.g., velocity, or trajectories). In addition, a lazy update approach is introduced to decrease the number of updates issued by the moving objects. C-Sky algorithm introduces a general framework that is applicable to all combinations of query and data objects: (1) static query and data objects, (2) a static query object and dynamic data objects, (3) a dynamic query object and static data objects, (4) dynamic query and data objects. We experimentally evaluate C-Sky's performance in terms of overall CPU time and IO overhead in comparison to other R-tree-based skyline techniques.

The rest of this paper is organized as follows. In Section 2, we outline the related work. In Section 3 we formally define the problem of continuous skyline queries, introduce the notation we use and propose the C-Sky algorithm in detail. In Section 4, we discuss the performance by comparing our work with an existing approach, and in Section 5 we describe our conclusions and the future work.

2. RELATED WORK

Skyline queries in general can be classified into two categories: static skyline queries and continuous skyline queries. Most of existing work addressed static skyline queries. Borzsonyi et al. [1] proposed block-nested-loop and divide-and-conquer algorithms. The block-nested-loop (BNL) algorithm keeps a list of candidate skyline points in main memory. It scans the data set and compares each data point $p$ with the list. There are three cases: (1) If $p$ is dominated by any point in the list, it is discarded. (2) If $p$ dominates one or more points in the list, it is inserted into the list. Those points in the list dominated by $p$ will be removed. (3) Otherwise, $p$ is inserted into the list and may be part of the skyline. In case (3), if there is not enough memory in the list, $p$ is inserted into a temporary disk file. After scanning the whole data set, only points in the candidate list will be guaranteed and returned as the skyline result. Then BNL will be executed again with the temporary disk file as input. The whole procedure continues until all skyline points have been checked. The main problem of this algorithm is its restriction by the memory size. If the memory size of the candidate list is too small, many iterations will be incurred. As a result, this approach is inadequate for dynamic processing.

Referencing the 2-dimensional example in Figure 2, the divide-and-conquer (DC) approach works as follows: (1) compute the median $m_x$ of the data set on some dimension $x$ and divide the whole data set into two partitions $s_1$ and $s_2$. $s_1$ contains points whose values in dimension $x$ are smaller than $m_x$; $s_2$ contains all others. (2) Calculate skylines of $s_1$ and $s_2$, respectively. This is done by recursively applying the dividing step until there are one or very few points in one partition, making the computation of the skyline trivial. (3) Merge results of $s_1$ and $s_2$ as the overall skyline. In this step, first the algorithm computes the median $m_y$ of the data set on dimension $y$ and divides $s_1$ and $s_2$ further into $s_{11}, s_{21}, s_{12},$ and $s_{22}$. Because points in $s_{21}$ always have a smaller $y$ dimensional values than points in $s_{12}$, $s_{12}$ and $s_{22}$ are incomparable. As a result, we only need to merge results in $s_{11}$ and $s_{21}$, results in $s_{12}$ and $s_{22}$ (named as $R_1$), and results in $s_{11}$ and $R_2$. The DC algorithm cannot merge and return the overall skyline result until the partitioning phase completes. This phase needs continuous position updates for all moving objects.

The nearest neighbor (NN) method [8] indexes the input data set with an R-tree. It then utilizes the nearest neighbor query of the R-tree to find the result skyline. Taking the data
set of Figure 3(a) as a 2-dimensional example, NN invokes the nearest neighbor query and returns point \( g \), which is one point of the result skyline. Coordinates of \( g \) divide data space into four areas: 1, 2, 3, and 4 (as shown in Figure 3(b)). All other points located in area 4 need no further consideration because they are dominated by \( g \). Then NN inserts the remaining areas (1 and 3; 1 and 2) into a to-do list. This method continues to pop areas from the to-do list and repeats the above query-and-divide procedure. NN terminates when the to-do list is empty. The NN method can be generalized into higher dimensional data cases. Kossmann et al. [8] proposed hybrid approaches for duplicate elimination of overlapping partitions. However, construction of an R-tree for continuous moving objects may incur much communication cost for position updates.

Tan et al. [4] presented two progressive processing algorithms: the bitmap approach (Bitmap) and index method (Index). Bitmap encodes dimensional values of data points into bit strings. For a \( d \)-dimensional data point \( p = (p_1, p_2, ..., p_d) \), let \( k_i \) be the total number of distinct values on the \( i \)-th dimension and \( m = \sum_{i=1}^{d} k_i \). Taking Figure 3(a) as a 2-dimensional example, \( k_1 = 10 \) and \( k_2 = 10 \) and \( m = 20 \). Assume \( p_i \) is the \( j \)-th smallest value on the \( i \)-th dimension, it is represented by \( k_i \) bits, where the \( (k_i - j + 1) \) most significant bits are 1, and the remaining bits are 0. If we want to decide whether point \( c \) belongs to the skyline or not, first we find its most significant non-zero bits of all dimensions. Then we extract and compose corresponding bits of all points, respectively, for each dimension. Finally we apply bitwise AND to these bit strings. Only points with one 1 in the results of their bitwise AND computation can be skyline points. Bitmap requires exact positions of points for encoding, so it will incur lots of updates and communication cost for moving objects.

The Index method classifies a set of \( d \)-dimensional points into \( d \) lists. One point \( p = (p_1, p_2, ..., p_d) \) is assigned to list \( i \) if \( p_i \) is the minimum (this value is called minimal coordinate, \( minC \) for short) among all dimension values of \( p \). Points in each list are sorted in ascending order of their \( minC \) values. After construction of lists, Index continuously pops off the top batch of each list and processes them. Processing one batch needs to check (1) skyline points inside this batch and (2) points not dominated by current skyline result. Index terminates execution when all dimension values of the latest inserted skyline point are smaller than or equal to \( minC \) of following batches. Like Bitmap, Index requires exact point positions and will incur lots of updates and communication cost for moving objects.

The branch and bound (BBS) algorithm was proposed to efficiently find set of points composing skyline [2]. This algorithm utilizes an R-tree for indexing and a heap for sorting distances from the query point to all tree nodes. If the nearest node is an intermediate one, it is expanded; if it is a leaf node, it is added into node set of the skyline. However, this algorithm works on the assumption of static queries and data objects.
Zhang et al. [13] proposed ideas about strong skyline points, which limit the number of skyline points in subspaces, and algorithms for merging sub-results into a whole result. Yuan et al. [14] introduced SKYCUBE, which contains skylines of all possible data subspaces, and two algorithms, bottom-up and top-down, for efficient calculation of SKYCUBE. Pei et al. [3] presented the idea of a skyline group, which is the union of skyline data points in all subspaces, and decisive subspaces, which are subspaces containing data points in a skyline group. They also developed the Skyey algorithm for computing a skyline group and decisive subspaces. However, all these algorithms still concentrate on static skyline queries and data objects.

To the best of our knowledge, the problem of continuous skyline queries for moving objects has not been well studied to provide efficient solutions. In this paper, we present a progressive algorithm that utilizes some existing techniques and provides a general framework which is applicable to all mutability combinations of query and data objects.

3. SKYLINE POINT SEARCH ALGORITHM

The formal definition of skyline points in \(d\)-dimensional space is a set of distinct objects, where any two tuples of objects \(a\) and \(b\), \(t_a = (id, cor_{1a}, ..., cor_{na})\), \(t_b = (id, cor_{1b}, ..., cor_{nb})\) in the set satisfy the condition that if any \(k\), \(cor_{ka} < cor_{kb}\), there exists at least one \(m\) that satisfies \(cor_{ma} > cor_{mb}\). Each coordinate represents a spatial or non-spatial value, for example, the distance between a moving object and a query object, the maximum speed, and the current direction, etc. The number of features for a moving object is more than one and the features are dynamically changing over time. The general setup of the problem consists of a dynamic query object \(q\) and a set of dynamic data objects with a dimension \(d\). A regular grid \(G\) consisting of \(n \times n \times \ldots \times n = n^d\) cells is used to index the moving objects and the origin \(O\) represents the query object \(q\). At any time instance \(t\), each cell \(c_{i,j}\) consists of indices of a list of moving objects (denoted \(M_{i,j}\)) and a list of skyline points (denoted \(S_{i,j}\)) residing within it. Constructing the index requires scanning through the objects and inserting each object into a corresponding cell. The grid-cell index structure is shown in Figure 4. Moving objects in grid \(G\) can freely move from cell to cell.

An update from each object is issued to a centralized server periodically and its tuple \(\langle id, cor_{1old}, cor_{2old}, ..., cor_{nold} \rangle\) is changed to \(\langle id, cor_{1new}, cor_{2new}, ..., cor_{nnew} \rangle\). Thus, in case that a moving object moves out of its current cell \(c_{i,j}\) into \(c_{i+1,j}\), the index of this object will be removed from the \(M_{i,j}\) and inserted into \(M_{i+1,j}\).

C-Sky processes a cell in \(G\) by utilizing the block-nested-loops method (BNL) [2], and a set of local skyline points is returned. The algorithm then combines these local skyline points with the skyline points from some other cells in order to obtain an overall set of skyline points. The C-Sky algorithm uses a grid of cells that provides a partitioning approach [9,10,11], such that skyline computations consider the minimal set of cells in order to retrieve the skyline points efficiently. The sequence of computing skyline points is important to avoid unnecessary cell visits. It can be easily observed that a natural processing order of skyline computations starts from the cell \(c_{i,j}\) with minimum \(mindist\) to the query point (or the origin), since it prunes a maximum number of cells, the moving points of which are guaranteed to be dominated by any skyline points in cell \(c_{i,j}\). Given a dynamic query object and a cell, \(mindist\) is defined as a possible minimum distance.
between the query point and a cell rectangle boundary. Efficient methods for the computation of $mindist$ have been discussed in previous work [6,7].

![Figure 4. Data structure of grid-based index.](image)

![Figure 5. Grid $G$ and corresponding binary search tree.](image)

We define the search order as $mindist$-first traversal. In this paper, we construct a binary search tree and perform an inorder traversal to visit the tree nodes in sorted search key order. Figure 5 shows a grid $G$ and its corresponding binary search tree where each node stores an index of a cell and its $mindist$ value to the origin. The search order with respect to $mindist$ in ascending order is $c_{0,0} \rightarrow c_{0,1} \rightarrow c_{1,0} \rightarrow c_{1,1} \rightarrow c_{0,2} \rightarrow c_{2,0} \rightarrow c_{1,2} \rightarrow c_{2,1} \rightarrow c_{2,2}$, and this order is stored into a search order list. Section 3.1 describes the core C-Sky algorithm and Section 3.2 discusses the lazy update approach of handling updates from moving objects.

3.1 C-Sky Computation Algorithm

We observe the following lemmas and present a proof of correctness for C-Sky algorithm.

**Lemma 1**: The pruning operation: In case of a 2-D world, if there exists a skyline point at cell $c_{i,j} = G[i,j]$, prune $C = \{G[i+1...n,j+1...n]\}$ cells.

**Proof**: The proof is straightforward, since each cell in $C$ has a larger value of coordinate than that of cell $c_{i,j}$ on any axis. Any moving points in $C$ cells must have been dominated by any skyline points in cell $c_{i,j}$, if there exists a set of skyline points. Therefore, prune each cell in $C$ by marking it “delete.” If no skyline points are found in cell $c_{i,j}$, prune $c_{i,j}$ by marking it “delete.”

As an example consider Figure 6(a), with a set of moving objects and a $4 \times 4$ grid $G$. Since there is no skyline point found in cell $c_{0,0}$, it is eliminated. Next, based on the search order, C-Sky processes $c_{0,1}$ and then cell $c_{1,0}$. Two sets of skyline points $\{b\}$ and $\{a,c\}$ are found in cell $c_{0,1}$ and $c_{1,0}$, respectively. Therefore, cells $c_{1,2}, c_{1,3}, c_{2,1}, c_{2,2}, c_{2,3}, c_{3,1}, c_{3,2}, c_{3,3}$ (the gray area) are pruned and C-Sky will not access these cells to retrieve skyline points at the time instant. This mechanism can greatly reduce the number of cell accesses to retrieve a skyline result set.

**Lemma 2**: The merge operation: Merge a cell $c_{i,j} = G[i,j]$ only with $C = \{G[0...i-1,j], G[i,0...j-1]\}$ cells.

**Proof**: (1) Intuitively, $c_{i,j}$ needs to merge with all the cells $G[0...n, 0...n]$ for an overall set of skyline points, except for $c_{i,j}$ itself. Based on Lemma 1, C-Sky needs to perform a merge operation on $c_{i,j}$ only when it is not pruned by any of these cells in $C_{lower}(i,j) = G[0...i-1,0...j-1]$. In other words, there are no skyline points found in any cell of $C_{lower}(i,j)$, such that cell $c_{i,j}$ remains “undeleted.” Since every cell in $C_{lower}(i,j)$ is marked “delete,” the merge operation of the C-Sky algorithm skips these cells. (2) The merge operation also ignores $C_{upper}(i,j) = \{G[0...n,j+1...n], G[i+1...n,0...j]\}$ cells. Since any...
moving object in any cell of $C_{upper}(i,j)$ has at least one coordinate value worse than that of $c_{i,j}$, it can never dominate those moving objects in $c_{i,j}$. Therefore, based on (1) and (2), the merge operation only needs to merge cell $c_{i,j}$ with $C = \{G[i0...i-1,j], G[i0...j-1]\}$ cells.

Figure 6(b) shows an example of the merge operation. When BNL($c_{2,2}$) returns a set of local skyline points $S_{local} = \{a,b,c\}$ for cell $c_{2,2}$, a merge operation is executed to combine $S_{local}$ with other skyline points from cells $c_{0,2}, c_{1,2}, c_{2,0}$, and $c_{2,1}$. Cells in area $A$ are ignored since no skyline points are found in these cells, and they are marked “delete.” The merge operation also skips those cells in area $B$ since the moving points in any cell of area $B$ and cell $c_{2,2}$ can never dominate each other.

**Lemma 3:** An extension of Lemma 2. During a merge process, a point $h$ of the local skyline points in cell $c_{i,j} = G[i,j]$ is pruned if and only if any skyline point in $C = \{G[i0...i-1,j], G[i0...j-1]\}$ cells dominate it.

**Proof:** A naïve approach is to consider all moving objects in $C$ cells for merging. Assume a cell $c_{i,j}$ in $C$, a set of non-skyline points $NS_{i,j}$, and a set of skyline points $S_{i,j}$. If a local skyline point $h$ in cell $c_{i,j}$ is dominated by a non-skyline point in $NS_{i,j}$, $h$ must also be dominated by a skyline point in $S_{i,j}$, since each non-skyline point in $NS_{i,j}$ is dominated by a skyline point in $S_{i,j}$. Therefore, it is not necessary to examine all moving objects in $C$ to obtain a final list of skyline points for $c_{i,j}$.

As Figure 6(b) shows, $S_{local} = \{a,b,c\}$ for cell $c_{2,2}$, and C-Sky compares these points with $S_{0,2} = \{h,e\}$ and $S_{2,0} = \{d,f\}$ and $S_{2,1} = \{g\}$ to output a final list of skyline points of cell $c_{2,2}$. As a result, $a$, $b$, $c$ are dominated, so $S_{2,2} = \emptyset$.

Algorithm 1 presents the pseudo-code for C-Sky. In line 2, each data object is periodically updated before the skyline computations. An entry of the search order is retrieved in line 3 and line 5 computes a set of local skyline points for a cell with respect to the entry. A pruning operation and merge operation are performed in line 9 and lines 11-15, respectively. Line 16 stores the overall skyline points.

**Algorithm 1. C-Sky Algorithm – Periodic Updates.**

**Input:** Grid $G$, search order list $T$, query object $q$ and data objects $D$.

**Output:** A Skyline Point Set $S$.

1. for (every processing cycle) do
2.   Update $D$ in $G$.
3.   for (each $t$ in $T$) do
4.     Let cell $c_{i,j}$ map into a cell in $G$ by index $t$.
5.     if $c_{i,j}$ is not marked "delete" then
6.       Let $S_{local} = BL(c_{i,j})$ be a local set of skyline points in $c_{i,j}$.
7.       if (Number of $S_{local} > 0$) then
8.         /*Pruning Procedure*/
9.       for each cell $c_{i,j}$ in $G[i0...i-1,j]$ and $G[i0...j-1]$ do
10.          if $c_{i,j}$ is not marked "delete" then
11.             merge skyline points in $c_{i,j}$ with those in $S_{local}$ and store the final skyline points in $S_{i,j}$.
12.         end if
13.       end for
14.     end if
15.     Insert $S_{i,j}$ to $S$.
16.   end for
17. else
18.     $c_{i,j}$ is marked "delete"
19.   end if
20. end for
21. end for

Proceeding of the third International Conference on Emerging Databases (EDB 2011)
3.2 Lazy Updates

Since C-Sky computes dynamic skyline points over moving objects in multiple dimensions, the system receives periodic updates from each moving object in order to maintain the skyline point result set correctly. The periodic update approach, however, results in high communication cost between the centralized server and each moving object. The drawback can be improved by applying a lazy updates method. The motivation is based on the search order list \( T \) and the pruning operation proposed earlier in this paper. Since the pruning operation avoids visiting unpromising cells, it is not necessary to update the moving objects in these pruned cells because they are not considered as part of skyline points. Thus, moving objects do not need to issue updates periodically, but issue updates only when C-Sky algorithm accesses them for skyline computations. To guarantee the correctness of the skyline point result set, we define an activity circle for each moving object. An activity circle is defined as a movement boundary of an object and it is a circle area with a center point \( P = (x_{tk}, y_{tk}, ..., w_{tk}) \) and radius \( R = (r_x \times \Delta t, r_y \times \Delta t, ..., r_w \times \Delta t) \), where \( r_i \) is a maximum speed for the \( i \) coordinate, \( \Delta t (=t_{current} - t_k) \) is the elapsed time period since \( t_k \), \( t_k \) is the latest update time and \( t_{current} \) is the current time. An activity circle covers the maximum moving range of a moving object. C-Sky needs to change the corresponding indexes as the activity circle grows. Figure 7(a) shows a set of moving objects at \( t_k \). Figure 7(b) shows the activity circle of each moving object after some time \( t \). For example at \( t_k \), object \( b \) was originally indexed at cell \( c_{0,0} \), and object \( g \) was indexed at cell \( c_{1,0} \). At \( t_{current} \), \( b \) is indexed at the cells \( c_{0,0} \) and \( c_{0,1} \) and \( g \) is indexed at the cells \( c_{0,0}, c_{0,1}, c_{1,0} \) and \( c_{1,1} \), because these cells are covered by the activity circles of \( b \) and \( g \).

Once C-Sky computes the skyline points on cell \( c_{0,0} \), any points whose activity circles are covered with cell \( c_{0,0} \) are updated before the skyline point computation. In this example, object \( b \) and \( g \) are updated is shown in Figure 7(c). Due to the updates, the activity circles of \( b \) and \( g \) are cancelled. The cell index of \( b \) remains at cell \( c_{0,0} \), but the cell index of \( g \) is changed from cell \( c_{1,0} \) to cell \( c_{1,1} \). Any moving points that are not accessed during the computation at \( t_{current} \) remain un-updated and their activity circles keep growing until C-Sky accesses them in the future. Figure 7(d) illustrates the activity circles updates at \( t_{current+1} \).

4. EXPERIMENTAL EVALUATION

We evaluated the performance of C-Sky algorithm by comparing it with BBS [2]. For verifying the effectiveness and efficiency of the lazy update approach, we
implemented two versions of C-Sky: C-Sky with periodic updates (C-Sky-PD) and C-Sky with lazy updates (C-Sky-LZ). In accordance with a prior study [12], our data sets are generated based on the random walk theory. Each output data object moves with a constant velocity until its expiration time. The velocity is then replaced by a new velocity with an expiration time. Figure 8 shows an example of a moving object moving based on the random walk approach. At the initial position \( p_0 \), the moving object \( p \) moves with a constant velocity. After reaching its expiration time, \( p \) arrives at the position \( p_1 \). The length of movement path \( l \) is calculated by \( p\text{.velocity} \times p\text{.expTime} \). A new velocity and a new expiration time are assigned to \( p \) and it keeps moving accordingly with its new velocity.

We generated 100K to 1M uniform and correlated distributed data with a dimension in the range of 2 to 5. Experiments were conducted on a system with an Itanium 1 GHz CPU and 8 GB of memory. The result sets of the skyline points are evaluated at every time instance. Our experiments are using several factors to examine the performance of C-Sky. In particular, in Section 4.1, we evaluate the effect of grid sizes ranging from 2 to 30 for each axis; in Section 4.2, we study the effect of cardinality and dimensionality by comparing C-Sky with the BBS algorithm; in Section 4.3, the effect of the lazy update approach is analyzed. In the BBS algorithm, R*-trees are used for indexing the moving objects. We use the R*-trees library implemented in Java by [15]. A page size of 4KB is deployed, resulting in node capacities between 94 (\( d=5 \)) and 204 (\( d=2 \)).

### 4.1 Grid Size

In this section we measure the overall CPU time of C-Sky by varying the grid size, which is one of the important parameters for the C-Sky performance. If the grid size is too small (e.g., less than 5 per dimension), it incurs a higher CPU cost. Because each cell (with larger cell size) may contain a large data set and a BNL computation works efficiently only with a small set of data, it suffers many iterations and the performance deteriorates. Furthermore, when a merge operation is performed on cells containing large data sets, it results in many unnecessary computations. If the grid size is too large, each cell is under-utilized for containing only a few data points. Particularly in the lazy update approach, as an activity circle of a moving object grows, it is likely to cover more cells such that it results in expensive index update costs.

Figure 9 illustrates the result for grid sizes ranging from \( 2 \times 2 \times 2 \) to \( 30 \times 30 \times 30 \) for uniform and correlated data sets. The performance is improved when applying a grid size of 10, but it is degraded when a larger grid size is used. Figure 9 suggests that the optimal grid size is 10, so this size is chosen for the rest of our experiments.

![Figure 8. Random walk example.](image)

![Figure 9. Grid size.](image)
4.2 Cardinality and Dimensionality

In this section we compare the C-Sky-PD and BBS algorithms in terms of cardinality and dimensionality. For the purpose of cardinality evaluation we use uniform and correlated distributed data sets with a dimension of 3 ($d = 3$). The cardinality of these data sets ranges from 100K to 1M. Figure 10 shows the overall CPU time of C-Sky and BBS with respect to the cardinality for the uniform (Figure 10(a)) and correlated (Figure 10(b)) data sets. The overall CPU time consists of the update CPU time (e.g., data updates, index updates, or R*-tree re-builds) and the skyline query evaluation CPU time. Both BBS and C-Sky update the locations of moving objects periodically. C-Sky greatly outperforms BBS in the overall CPU time and the difference increases with cardinality. The reason is that the cost of updating the R*-tree as well as the cost of evaluating skyline queries are much higher than that for C-Sky-PD.

The effect of the dimensionality for uniform and correlated data sets is shown in Figures 11(a) and 11(b), respectively. The overall CPU time of BBS increases greatly with the dimensionality. The degradation of BBS is caused by the poor performance of R*-trees in high dimensions. C-Sky-PD has better performance with the correlated data set than with the uniform data set, since this can be considered as a best case for C-Sky. A correlated data set represents an environment where a data point is good/worse in one dimension and also good/worse in the other dimension. Therefore, C-Sky can prune a large number of unnecessary cells. BBS, on the other hand, has a stable performance with both data sets.

4.3 Lazy Updates

Lazy updates have been defined earlier in this paper as a mechanism where the location of a moving object is updated only when its activity circle intersects with the currently processed cell, in contrast to a periodic update approach where the entire set of moving objects are updated periodically. However, with the lazy update approach, if the activity circle of a moving object is not updated, the boundary of its movement is increasing such that it may cover many cells. As a result, it incurs many index updates and costs more CPU time. Therefore, we set up a threshold for the lazy update approach. When an activity circle of a moving object covers more than a certain percentage of the entire number of cells (e.g., 10%), the system updates the moving object, even though its activity circle does not intersect with the currently processed cell. Figures 12(a) and 12(b) compare the number of updates and CPU time versus the update rates from 0% to 20% for the uniform and
correlated data sets. It can be observed that the choice of update rate is a trade-off between the number of updates and the CPU time. A high lazy update rate results in a lower number of updates but incurs a higher CPU time. In the following experiments we use the mean of these rates, namely 10\%, as our lazy update rate.

Figures 13(a) and 13(b) measure the effect of the two update approaches for the uniform and correlated distributed data sets. The number of updates is greatly reduced by utilizing the lazy update approach. Figures 14(a) and 14(b) present the overall CPU time. C-Sky-LZ costs more CPU time than C-Sky-PD. The CPU time is increased because C-Sky-LZ handles more index updates. Since we assume C-Sky adopts by client-server architecture, the overall update cost has two parts: the communication cost between moving objects and the server, and the local update cost including data updates and index updates on the server. Therefore, if the number of updates issued by moving objects is decreased, the communication cost is reduced as well. In our future work, we plan to further investigate the measurements of the communication cost, and a fair comparison of C-Sky-PD and C-Sky-LZ with respect to their overall update cost.

5. CONCLUSIONS

We propose a progressive C-Sky algorithm of continuous skyline queries for moving objects. The applications for continuous skyline queries are addressed to support its utilization. A search order imposed on the grid cells is introduced to reduce unnecessary cell visits. Based on the grid-based index structure, we provide a symmetric approach to compute the skyline points. The core method of C-Sky consists of three operations, a block-nested-loop (BNL) operation, a pruning operation, and a merging operation, which enable C-Sky to compute the skyline
points independently and locally from each cell. An optimal grid size is suggested based on our experiments. The results show the performance and utility of our novel approach. In addition, a lazy update approach is proposed which can greatly reduce the number of updates in contrast to a traditional update approach. A threshold value for a lazy update rate is determined to avoid over bounded activity circles, which incur more index updates. For future work, it would be interesting to explore the estimation of the communication cost and provide an estimation cost of adopting the lazy update approach.

ACKNOWLEDGEMENTS

We would like to acknowledge that this research was carried out in part at the Centre of Social Media Innovations for Communities (COSMIC), sponsored and supported by the Singapore National Research Foundation and Interactive & Digital Media Program Office, MDA.

REFERENCES


Proceeding of the third International Conference on Emerging Databases (EDB 2011)