# Efficient Skyline Query Processing on Peer-to-Peer Networks

Shiyuan Wang[1] *, Beng Chin Ooi[2] , Anthony K. H. Tung[2] , Lizhen Xu[1]

[1]Southeast University, China & [2]National University of Singapore
desiree_wsy@yahoo.com.cn, {ooibc, atung}@comp.nus.edu.sg, lzxu@seu.edu.cn

## Abstract

*Skyline query has been gaining much interest in database research communities in recent years. Most existing studies focus mainly on centralized systems, and resolving the problem in a distributed environment such as a peer-to-peer (P2P) network is still an emerging topic. The desiderata of efficient skyline querying in P2P environment include: 1) progressive returning of answers, 2) low processing cost in terms of number of peers accessed and search messages, 3) balanced query loads among the peers. In this paper, we propose a solution that satisfies the three desiderata.*

*Our solution is based on a balanced tree structured P2P network. By partitioning the skyline search space adaptively based on query accessing patterns, we are able to alleviate the problem of "hot" spots present in the skyline query processing. By being able to estimate the peer nodes within the query subspaces, we are able to control the amount of query forwarding, limiting the number of peers involved and the amount of messages transmitted in the network. Load balancing is achieved in query load conscious data space splitting/merging during the joining/departure of nodes and through dynamic load migration. Experiments on real and synthetic datasets confirm the effectiveness and scalability of our algorithm on P2P networks.*

## 1. Introduction

Peer-to-Peer (P2P) systems, as a popular medium for distributed information sharing and searching, have been gaining increasing interest in recent years. To provide P2P users efficient information exchange and rich query functions, existing studies have focused extensively on exact query [18, 20], range query [4, 2, 8, 19, 12] and keyword-based information retrieval [22].

Skyline queries are well studied in centralized systems. A skyline query returns a set of data points that are not dominated by any other points in a given data set. A point dominates another point if it is no worse in all concerning dimensions and better in at least one dimension. Examples are searching hotels with cheap price and yet close to beach and querying the overall best players in NBA seasons.

However, little work has concentrated on efficient skyline query processing in P2P context. The problem is that it is not straightforward and efficient to adopt a centralized algorithm and build a centralized index to compute skylines. [23] is the first attempt on progressive processing of skyline queries on a P2P network such as CAN [18]. The proposal controls query propagation based on the partial order of CAN's zones. Unfortunately, it focuses on constrained skyline queries [16, 23], and consequently, its load balancing approach has been designed to solve workload imbalance caused by skewed query ranges.

BestPeer [15] is a P2P platform that supports both structured and unstructured overlays. In this paper, we propose a solution called Skyline Space Partitioning (SSP) on the BestPeer's structured P2P network called BATON [12] to provide efficient processing of unconstrained skyline queries that search skyline points in the whole data space. The following desiderata are deemed essential for the efficient processing: (1) Small number of involved nodes. (2) Small number of search messages. (3) Query load balancing. Unlike exact, range or constrained skyline queries which might access data in any part of the space, the data access of an unconstrained skyline query is likely to be skewed towards the portion of the space that contain the skyline. Such hot spots must be relieved to avoid imbalance in query loads on the network.

Our solution satisfies the above desiderata in three folds. First, we organize the multi-dimensional data regions using a balanced BATON tree. We assign each region an ordered region number, in which each bit encodes a data space split whose detail is kept in the region split history. In such way, we can route a query by computing the target region number based on local split histories. Second, we define the skyline search space to limit the number of involved nodes and partition the search space into subspaces adaptively at each hop

*the work was done when the author was an intern at National University of Singapore in 2005

1-4244-0803-2/07/$20.00 ©2007 IEEE

to parallelize processing and control the number of search messages. Third, we balance the query loads in load balanced partitioning of data space during the joining/leaving of nodes. Furthermore, we dynamically sample load from both linked and random nodes and migrate data in case of load imbalance.

We make the following main contributions:

- We propose a novel approach called SSP that partitions and numbers the data space among the peer nodes such that the target subspace (region) number can be derived with good accuracy in order to control the peers accessed and search messages during skyline query processing. We formally prove the necessity and completeness of visiting non-dominated nodes within the delimited skyline search space.
- We balance the query loads among peers through both load balanced data space partition and dynamic load migration. We propose a novel load sampling mechanism that attends the quality of sampled load distribution and the efficiency of sampling process by combining direct and random sampling.
- We conduct extensive experimental study to evaluate the effectiveness and scalability of our algorithm.

**Organization**: Section 2 reviews related work. Section 3 describes the space partitioning strategy. Section 4 presents our skyline query algorithm. Section 5 discusses query load balancing, followed by experiments validation in Section 6. Finally, Section 7 concludes the paper.

## 2. Related Work

Our work is motivated by previous studies on both skyline query processing and multi-dimensional indexing on P2P, which are reviewed in this section.

### 2.1. Skyline Query Processing

Skyline computing was first introduced to the database field in [3] with *Block-Nested-Loops(BNL)*,*Divide-and-Conquer(D&C)* and B-tree algorithms. Later work [6] improved *BNL* by presorting. [21] proposed progressive *Bitmap* and *Index* algorithms. Subsequently, there are studies employing R-tree index to compute skyline [13, 16] faster and using external algorithm to compute the maximal vectors [9]. Some recent works investigate the semantics of skyline by subspace analysis [17, 24], processing of skyline over data streams [14] and in high dimensional spaces [5].

Distributed skyline query was recently addressed in [1] for web information systems and in [11] against mobile devices. By visiting all nodes to retrieve skyline answers, the above approaches dedicated for small-scale distributed systems are not efficient to be applied on P2P networks. Recent work of skyline query processing on P2P [23] parallelizes search by enforcing a partial order on query propagation based on CAN. It emphasizes on constrained skyline queries that are posed within a query range, and consequently its skyline search method and zone replication approach have been designed for such constrained skyline queries, while we aim to solve skyline queries efficiently in the global range in which we are faced with a more serious *query load imbalance* along the portion in the skyline.

### 2.2. Multi-dimensional Indexing on P2P

Multi-dimensional indexing for supporting efficient search on structured P2P networks has been a hot research topic. MAAN [4] uses locality preserving hashing to map data values onto Chord identifier space. Mercury [2] attempts to support multi-attribute range queries by placing values of each attribute contiguously on a separate routing hub, while performing explicit load balancing for non-uniform data distribution. However, the separate attributes structure in MAAN and Mercury are not effective for processing skyline queries that implicitly specify the constraints among the attributes. The works closer to ours are Murk [8], SkipIndex [25] and ZNet [19]. Murk indexes multi-dimensional data partitions using the kd-tree. Like Murk, SkipIndex stores partition information in a binary tree, while ZNet splits data partitions in a quad-tree manner. Though SkipIndex and ZNet balance data loads during partition, they do not deal with the query load balancing problem present in skyline query processing.

## 3. Space Partitioning

In this section, we begin with the problem definition. We map the multi-dimensional data space on an existing P2P network called BATON [12] which is briefly introduced in Section 3.2. By numbering the data region and recording its split history, we can estimate the target region number for supporting efficient search. The regions are maintained in balanced query loads during node join / departure which is the first part of our query load balancing solution.

### 3.1. Problem Definition

Without loss of generality, we assume that the data values of each dimension are in the range $[0, 1]$. The whole $d$-dimensional data space $\{[0, 1], [0, 1], \ldots, [0, 1]\}$ is distributed on a P2P network with $N$ nodes, in which each node maintains a non-overlapping $d$-dimensional data region and the data points falling inside the region. A data region is constructed with one or more hyper-rectangles, each

of which is confined by its bottom left point (lower bound) and top right point (upper bound).

We process the skyline query in the form $SQ = (q_1, q_2, \ldots, q_d)$ , in which $q_i \in \{Max, Min\}$ $(1 \leq i \leq d$, $Max$, $Min$ indicates larger, smaller preference) for dimensionality $d$. All examples in the paper are based on the network shown in Figure 1, in which each node is identified by a region number and a physical node id attached within the parentheses. Example $SQ_1 = \{Min, Min\}$ is used in the discussion of skyline querying.

### 3.2. Background about BATON

BATON(BAlanced Tree Overlay Network) [12] is based on a binary balanced tree structure in which each node of the tree is maintained by a peer. The position of a node is defined by a (level, number) pair, in which level starts from 0 at the $root$, and number starts from 1 at the leftmost node at each level. Each tree node stores links to its parent, children, adjacent nodes, and selected nodes at the distance of power of two on its left / right side at the same level in left/right routing tables. BATON maintains the tree structure balanced by forcing each node to have both its left and right routing tables full before it has a child node, which is crucial for effective routing. It takes $O(logN)$ cost for joining / leaving of nodes and exact search.

### 3.3. Region Mechanism for Search

We employ the z-curve method for mapping the multi-dimensional data space onto the one-dimensional BATON. We number a data region according to the z-order and the position of the region in space split. Our search mechanism relies on the relationships of the regions and the operations of region numbers.

Figure 2 describes the mapping of the data regions in Figure 1. We see each tree node maintains a data region and the in-order traversal of the tree corresponds to the sequential visit of data regions in z-order. As shown in the right part of Figure 2, we keep the routing table structure of BATON, and add adjacent nodes in each routing table entry to facilitate data space merge check.

Each node maintains the following information for the region it owns: (1) **Region number**: a 0-1 string $RNum$ that identifies a region and is consistent with the z-order of the region, in which the bit values of 0, 1 at certain bit locations indicates the region is in a lower or upper part in the corresponding space split. (2) **Data range**: pairs of lower bound $LowerBound$ (bottom left point) and upper bound $UpperBound$ (top right point). (3) **Split history**: a list of entries of split value and dimension $(SplitPos, SplitDim)$. (4) **Next partition dimension**: a bit indicating the next split dimension. An example for the region information of node 6 is given in Figure 1.

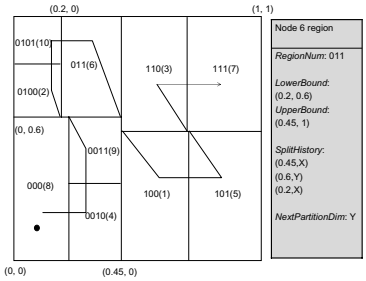



**Figure 1. Equal load partitions**

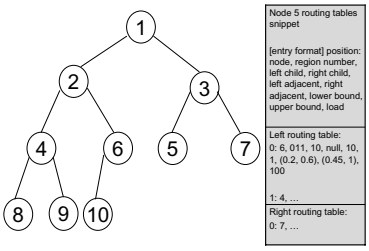



**Figure 2. Partitions mapping on BATON**

Note that "region" does not only represent a data partition held by a node, but also can refer to a region superset that a search range or target falls inside. Given a region $m$, let $\mathrm{RNum(m)}$ be its region number, $\mathrm{RNum(m).length}$ be the number of bits in $\mathrm{RNum(m)}$. We define the following relationships of regions based on the operations of region numbers.

**Definition 3.1** *(Region Succeed, $\succ$) Region $m$ "Succeed" $n$, or $m \succ n$, i.f.f.* $\mathrm{RNum(m)[b]} = 1, \mathrm{RNum(n)[b]} = 0$ *and* $\mathrm{RNum(m)[i]} = \mathrm{RNum(n)[i]}$ *for* $1 \leq b \leq \min(\mathrm{RNum(m).length}, \mathrm{RNum(n).length}), 1 \leq \mathrm{i} < \mathrm{b}$. *The precede or $\prec$ relationship is argued in the same way.*

**Definition 3.2** *(Region Cover, $\supseteq$) Region $m$ "Cover" $n$, or $m$ is the superset of $n$, or $m \supseteq n$, i.f.f.* $\mathrm{RNum(m).length} \leq \mathrm{RNum(n).length}$ *and* $\mathrm{RNum(m)[i]} = \mathrm{RNum(n)[i]}$ *for* $1 \leq i \leq \mathrm{RNum(m).length}$. *The covered or $\subseteq$ relationship is defined in the same way.*

To route a query, a node first delimits the region of the searched target by computing its region number based on local split history, and then passes the query to a linked node whose region is covered by or nearest to the delimited region. Algorithm 1 describes the process of computing the target region number. By "estimation", we mean we can only compute the region number of a superset of the target region. The accuracy depends on how many times the

**Algorithm 1** estimate_num($node\ n, target\ p, bit\ b$)

```
1: if (Region(p) ⊇ Region(n)) then
2:   for b to RNum(n).length do
3:     if (p[b%d] ∈ HistoryRange(n)[b]) then
4:       RNum(p)[b] ← RNum(n)[b]
5:     else
6:       RNum(p)[b] ← 1 − RNum(n)[b]
7:       break
```

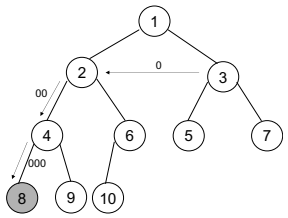



**Figure 3. Routing process**

searched point $p$ falls in the same range with local region after a history split (line 3–4). The computation terminates once $p$ falls out of a history range of the current node (line 6–7).

Suppose node 3 wants to locate a point in node 8 as shown in Figure 1, node 3 first computes its region number as 0, for it is in the lower part in the first space split on axis x. Then node 3 routes the query to its nearest left neighbor node 2 whose region the delimited target region covers. The same process is executed on node 2 and node 4 subsequently until the query is routed to the target node 8 as illustrated in Figure 3. The maximum number of routing hops is approximately the number of bits in the accurate region number of the searched target. For the uniformly distributed loads and corresponding equal space partitions, the average routing path length is $O(logN)$.

### 3.4. Query Load Balanced Partition

This section presents the first part of our query load balancing solution. We split load equally in data space partitions and select better candidates to share load in node join / departure.

We observe that allocating relatively smaller space to the "hot" regions should facilitate load balancing. So we partition data space by dividing a hyper-rectangular region into two equally loaded hyper-rectangular regions. Accordingly, we combine two buddy regions (such as region 2, 10 in Figure 1) or the regions of adjacent nodes in data space merge. An example of such partitions is demonstrated in Figure 1.

**Node Join**. We seek better query load balance during node join by selecting a node with heavier load among several known candidates at the same level instead of directly joining the first qualifying node that does not have enough children and has full routing tables as in [12]. Candidates

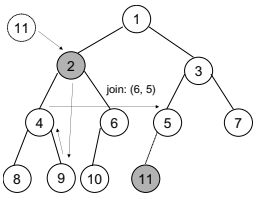



**Figure 4. Node join**

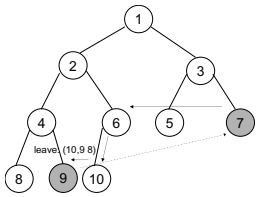



**Figure 5. Node Departure**

are picked amongst the neighbors in the routing tables.

For example, when node 4 receives the joining request of a new node 11 as the arrows show in Figure 4, it finds two qualifying neighbors: node 5 and node 6. Assume the load on node 5 is heavier than the load on node 6. Node 4 will make a decision and pass the new node to node 5. The process of candidate selection only adds constant terms of cost, thus the total cost for node join is still $O(logN)$.

**Node Departure**. When a node leaves, it first requests the adjacent node to take over its region. If the empty position caused by leaving does not incur tree imbalance and the two regions can be combined without bringing a heavy load, no further action needs to be taken. Otherwise, the procedure to search for a leaf node replacement is called on one of its adjacent nodes at the lower level.

For replacement, we choose a leaf node with lighter load whose region can be better combined with the region of its adjacent node from a list of known leaf node neighbors at the same level. As an example, when node 7 leaves in Figure 5, it has to find a replacement at the level of node 10. Among node 10 and its two neighbor nodes 9 and 8, if node 9 has a lighter load, it will be the best candidate for replacing node 7, besides its region can merge with the region of its adjacent node 4. The cost for replacement search is $O(logN)$ with constant cost addition for choosing candidate.

## 4. Skyline Query Processing

Efficient skyline query processing on P2P networks demands quick response and small number of involved nodes and search messages. Obviously, simple approaches of neither sequential scanning in the order of region number nor query flooding through linked nodes would be satisfactory. Effective strategies are needed to taken towards three key problems: 1) parallelized search; 2) irrelevant nodes pruning; 3) reduction of duplicate query forwarding.

We propose our solution called Skyline Search Space Partitioning (SSP) in this section. It delimits a skyline search space to solve the second problem, and adaptively partitions the search space to solve the first and the third problems.

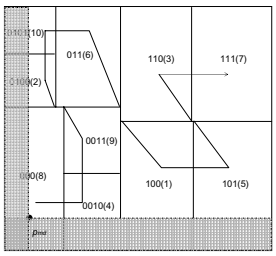



**Figure 6. Skyline search space**

### 4.1. Skyline Search Space Definition

We start search from the node whose local results are guaranteed to be in the final skyline. We denote such a node as $SQ\text{-}Starter$. It can be located by searching the most dominating boundary point that dominates all other points in the data space, such as point (0,0) on node 8 in Figure 1.

We now formalize the skyline search space. Practically, we compute local skyline results on $SQ\text{-}Starter$ and select the most dominating point that has the largest dominating region [11]. Let this point be $p_{md}(a_1, a_2, \ldots, a_d)$. To avoid waste network bandwidth, we use only $p_{md}$ for pruning irrelevant nodes and refining the search space instead of using all intermediate skyline results as [23] does. Thereby we define skyline search space as follows.

**Definition 4.1** *Skyline search space is the set of all data points that are not dominated by $p_{md}$. It is the union of $d$ hyper-rectangle search ranges ($SR$), each of whose boundary is limited only by one coordinate of $p_{md}$ along one axis, such as $\{[0,1], \ldots, [0,a_i), \ldots, [0,1]\}$.*

For the example query $SQ_1$, the shadowed part in Figure 6 indicates its skyline search space, which is composed of $SR = \{[0,a_1), [0,1]\}$ and $SR = \{[0,1], [0,a_2)\}$.

**Definition 4.2** *A node is dominated and should be pruned, i.f.f. the single ideal data point with the best value in each dimension within its ranges, denoted as $p_{best}$, is dominated by $p_{md}$.*

Now we prove we can find the correct answers by visiting only the non-dominated (unpruned) nodes inside the skyline search space.

**Lemma 4.1** *All nodes outside the skyline search space are dominated nodes.*

**Proof:** Since the data space outside the skyline search space like $\{(a_1,1], (a_2,1], \ldots, (a_d,1])\}$ has the worse values in all dimensions than $p_{md}$, $p_{best}$ of any node within this space must be dominated by $p_{md}$. Hence, any node in this space is dominated by $p_{md}$.

**Lemma 4.2** *Any dominated node cannot contain skyline points.*

**Proof:** Given any dominated node, any of its data points must have at least one inferior attribute value compared to $p_{best}$. Since $p_{best}$ is dominated by $p_{md}$, any data point of the node is dominated by $p_{md}$, and cannot be a skyline point.

**Lemma 4.3** *Any final skyline points cannot be dominated by any points of the dominated nodes.*

**Proof:** For the sake of brevity, let's see $SQ1 = \{Min, Min\}$. Suppose a skyline point $p_s(s_1, s_2)$ is dominated by a point $p_d(v_1, v_2)$ on a certain dominated node, then we have $s_1 \geq v_1, s_2 \geq v_2$. According to Definition 4.2, $p_d$ is dominated by $p_{md}$, $v_1 \geq a_1, v_2 \geq a_2$, thus we have $s_1 \geq a_1, s_2 \geq a_2$, meaning $p_s$ is dominated by $p_{md}$ and cannot appear as a final skyline point. It contradicts with the fact, so $p_s$ cannot be dominated by $p_d$.

Based on the above lemmas, we draw the following conclusion:

**Theorem 4.1** *The non-dominated (unpruned) nodes in the skyline search space return the complete skyline set and only the skyline set.*

### 4.2. Optimizing Skyline Search Space

Given a skyline search space constructed with hyper-rectangle search ranges ($SR$) in Definition 4.1, we divide it into separate search subspaces or subranges ($subSR$) for parallel query forwarding paths. Nodes within each $subSR$ only forward query to the node whose region is covered by or nearest to the unsolved part of $subSR$. As such, a node is rarely revisited except for routing a $subSR$ that cannot directly reach a covered node.

As in computing the region number for a search target, we partition an $SR$ into $subSRs$ based on the history ranges stored in local split history. By assigning each $subSR$ a region number, we can compare its position with the regions of the linked nodes to decide outward query forwarding.

Algorithm 2 depicts the procedure of partitioning the search space and computing region number for the partitioned subspaces. Similar to Algorithm 1, we scan the split history sequentially (line 1), compute the next unknown bit of the region number for $SR$ (line 3), and partition $SR$ into two $subSR$s on each history split position not exceeding $SR$ (line 4–8). The computing stops once the current node cannot partition $SR$ further. This happens when the updated $SR$ falls out of a certain history range of the current node (line 10–11).

As for the search space of $SQ_1$, after subtracting the range of $SQ\text{-}Starter$ as shown in Figure 7(a), we refine the left part $SR$ into a single $SR$ estimated as 01, for it falls outside the history range of region 8 in the second time split.

**Algorithm 2** partition($node\ n, range\ SR$)

**Define:** $subSRSet$ disjoint sub search range set of $SR$
1: **for** $b$ from RNum($SR$).$length$ to RNum($n$).$length$ **do**
2: **if** ($SR \subseteq$ HistoryRange($n$)[$b$]) **then**
3: RNum($SR$)[$b$] $\leftarrow$ RNum($n$)[$b$]
4: **if** (HistoryRange($n$)[$b$] is lower part) **then**
5: UpperBound($SR$)[$b$] $\leftarrow$ SplitPos($n$)[$b$]
6: **else**
7: LowerBound($SR$)[$b$] $\leftarrow$ SplitPos($n$)[$b$]
8: Add $SR$'s buddy $subSR$ to $subSRSet$
9: **else**
10: RNum($SR$)[$b$] $\leftarrow 1 -$ RNum($n$)[$b$]
11: break

The right part $SR$ is partitioned into two parts on the first time history split: the one overlapping region 1 and region 5 is assigned an estimated region number 1, and the one overlapping region 4 is estimated as 001.

Now, let's consider an $SR$ whose boundary is limited by the coordinate of $p_{md}$ on dimension $i$. Let $m$ be the number of bits in the region number of the farthest region from $SQ$-$Starter$ that overlaps $SR$. Then the number of split history entries for dimension $i$ is around $|m/d|$. So the number of the partitioned $subSR$ is $|m/d|$, and the known number of bits in the region number of $subSR$ decreases $d$ each time when a $subSR$ falls out of the history range of $SQ$-$Starter$. Since the maximum number of hops to solve a $subSR$ is the number of bits in its region number, the maximum number of hops to process a $SR$ is the arithmetic progression of the former, namely $O(d{+}2d{+}\ldots{+}|m/d|d)$ and asymptotically $O(m(1+m/d)/2)$. In uniform load distribution, the query range is always partitioned in halves, so the cost for processing a $subSR$ always reduces in halves. Since the average routing path length to locate the farthest region that $subSR$ overlaps is $O(logN)$, the average number of steps for processing $SR$ is $O(logN + \frac{1}{2}logN + \cdots + \frac{1}{2}^{logN/d}logN)$, namely $O(2(1 - 1/\sqrt[d]{N})logN)$.

### 4.3. Skyline Query Algorithm

Based on the above, we present our skyline query algorithm in Algorithm 3 which is for local processing at each hop after $SQ$-$Starter$ defines the skyline search space.

Each $subSR$ is forwarded to a non-dominated neighbor, child or adjacent node whose region is covered by $subSR$ (line 7–11 in Algorithm 3). If such a node is not found, the $subSR$ is forwarded to the farthest neighbor in the case it succeeds (precedes) the rightmost (leftmost) neighbor (line 13–14), or just passed to the adjacent node (line 16). Skyline search at each of the next hops (line 21) is parallelized.

**Algorithm 3** search_skyline($node\ n$, $query\ SQ$, $search\_range\ SR$)

**Define:** RRT($n$) right routing table of node $n$
**Define:** RChild($n$) right child of node $n$
**Define:** RAdj($n$) right adjacent of $n$
**Define:** $subSRSet$ disjoint search range set of $SQ$
**Define:** Dominated($m$) if node $m$ is dominated by $p_{md}$
1: **if** (Range($n$) $\bigcap SR \neq \emptyset$) **then**
2: Compute local skyline not dominated by $p_{md}$ and report to $SQ$-$Starter$
3: $SR \leftarrow$ Range($SR$) $-$ Range($n$)
4: $subSRSet \leftarrow$ partition($n, SR$)
5: **for all** sub-range $subSR$ in $subSRSet$ **do**
6: **if** (LowerBound($subSR$) $>$ UpperBound($n$)) **then**
7: $m \leftarrow$ NodeWhoseRegionCoveredBy (Region($subSR$)) in RRT($n$)
8: **if** (($m$ not exist or Dominated($m$)) and (RChild($n$) not processed $subSR$) and (Region($subSR$) $\supseteq$ Region(RChild($n$)))) **then**
9: $m \leftarrow$ RChild($n$)
10: **if** (($m$ not exist || Dominated($m$)) and (RAdj($n$) not processed $subSR$) and (Region($subSR$) $\supseteq$ Region(RAdj($n$)))) **then**
11: $m \leftarrow$ Radj($n$)
12: **if** ($m$ not exist || Dominated($m$)) **then**
13: **if** (Region($subRQ$) $\succ$ Region(FarthestNodeInRRT($n$))) **then**
14: $m \leftarrow$ FarthestNodeInRRT($n$)
15: **else**
16: $m \leftarrow$ RAdj($n$)
17: Map $m$ to $subSR$ in $subSRSet$
18: **else**
19: // A similar process executes towards the left
20: **for all** $(m, subSR)$ in $subSRSet$ **do**
21: search_skyline($m, SQ, subSR$)

To answer $SQ_1$, in the first step (Figure 7(a)), we promote $subSR$ estimated as 001 to node 4 that can fully resolve it, pass $subSR$ 01 to the covering right neighbor node 10 (line 9), and also send $subSR$ 1 to the rightmost neighbor node 10 for further forwarding (line 14). In the second step shown in Figure 7(b), node 10 refines $subSR$ 01 to $subSR$ 0100 by local data (line 3) and partition(line 4), then promotes it to the left adjacent node 2 that can fully resolve it, and passes $subSR$ 1 to the right adjacent node 6 (line 16). Figure 7(c) illustrates the skyline search space after local processing of node 2. The only remaining $subSR$ 1 is then forwarded to the covering right neighbor node 5 (line 9), which solves the part 101 and sends the updated $subSR$ to its left adjacent node 1. From Figure 7(d), we can see node


(a) Step 1 (b) Step 2

(c) Step 3 (d) Step 4

**Figure 7. Skyline query solving process**

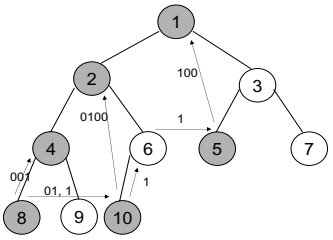



**Figure 8. Skyline query**

1 is the last hop for processing $subSR$ 1. Figure 7 and Figure 8 demonstrate the process of search space refining and query forwarding respectively.

**Theorem 4.2** *SSP answers a skyline query in* $O((1 + 2d(1 - 1/\sqrt[d]{N}))logN)$ *steps for uniform load distribution, and* $O((1 + (m + d)/2)m)$ *steps in the worst case.*

**Proof:** This proof is straightforward given the routing hops of locating $SQ\text{-}Starter$ and processing an $SR$. Locating $SQ\text{-}Starter$ takes $O(logN)$ hops for uniform load distribution, $O(m)$ in the worst case; processing each $SR$ requires $O(2(1 - 1/\sqrt[d]{N})logN)$ steps for uniform load distribution, $O(m(1+m/d)/2)$ steps in the worst case. By adding these costs together, we get $O((1 + 2d(1 - 1/\sqrt[d]{N}))logN)$ for uniform load distribution and $O((1+(m+d)/2)m)$ for the worst case.

## 5. Query Load Balancing

To balance query load, the following two steps are undertaken. First, data space is partitioned based on query load, and node join/departure are redesigned for balancing load. Second, a novel mechanism is proposed for sampling and dynamically balancing load during query processing. As the first step has been presented in Section 3.4, we shall only describe the second step in this section. We start with the definition of query load.

**Definition 5.1** *Query load of a node is the sum of the number of skyline retrieving towards its local data records, and the number of messages that are routed by the node but do not lead to a local query.*

The load balancing process during query processing works as follows. First, each node checks whether there is an imbalance. If yes, data migration process is established to balance the load.

Query loads are either periodically gathered by each node from the neighbor nodes in its routing tables and from adjacent nodes or with query load changing notification from these nodes. The imbalance is determined by the difference $\delta$ of local load and sampled query load. If $\delta$ is larger than a predefined threshold $\sigma$, an imbalance is detected [1]. However, load sampled from the linked nodes may not reflect the global load distribution. To compensate, we start random sampling when the first step does not detect imbalance and yet local load is heavier than the average of the gathered loads. Our random sampling strategy is similar to [2]. A number of $logN$ nodes that are not linked to are sampled by sending probes that are attached a small length limit of routing hops ($logN$), and following an existing link randomly at each hop. The hop where the probe terminates sends back its load and the load data it has gathered directly.

To balance the query load, a leaf node finds a lightly loaded leaf node to drop and forces it to rejoin as one of its children. A non-leaf node attempts to share load with adjacent nodes [12].

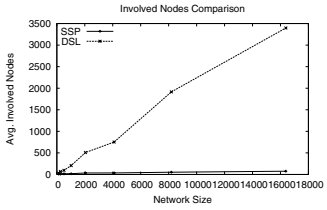
## 6. Experiment Evaluation

We evaluate our skyline search approach SSP by comparing it with the distributed skyline query algorithm DSL [23] in terms of network size, dimensionality, cardinality and query load balance. The performance measures are the number of involved nodes, the number of skyline search messages and the query load distribution. We conduct simulation experiments on a Linux box with Intel Xeon 3.0GHz processor and 2GB of RAM. The response time is tested in real deployment on a cluster consisting of 20 nodes, each of which has an Intel Xeon 3.0GHz processor and 4GB of RAM. We use three kinds of datasets: one is a real dataset of NBA players' season statistics from 1949 to 2003 containing 19,000 records downloaded from [10] that resembles a correlated data distribution; the other two are synthetic independent and synthetic anti-correlated datasets with a maximal data size of 1,638,400. For the synthetic datasets, we show only one set of results in cases where both exhibit similar performance.

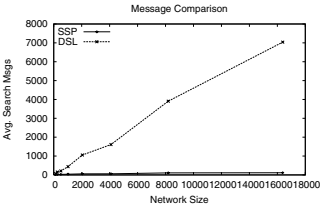
[1] The imbalance ratio is computed according to [7]

**Table 1. Experimental settings**

| Parameter | Domain | Default |
|---|---|---|
| Number of peers | $2^7, \ldots, 2^{10}, \ldots, 2^{14}$ | $2^{10}$ |
| Dimensionality | 2, 3, 4, 5 | 2 |
| Cardinality | $(2^{10}, \ldots, 2^{14}) \times 100$ | $2^{14} \times 100$ |




(a) Involved Nodes



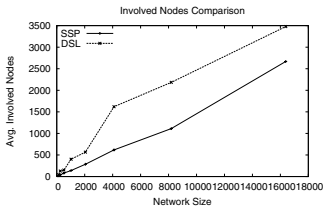

(b) Search Cost

**Figure 9. Effects of network size on independent dataset**

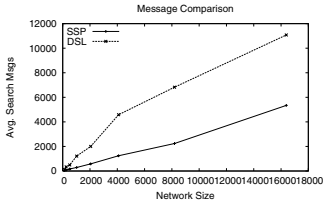
The experimental settings are summarized in Table 1. Each experiment is repeated 10 times and the average is taken. Each test issues 1000 skyline queries and the average cost is taken. Each query prefers smaller or larger value randomly in each interested dimension. It starts from a random node and asks for the answers in the whole data space.

### 6.1. Effects of Network Size

We first study the effects of network size using both independent and anti-correlated datasets. Figure 9 illustrates the results on the independent dataset. When the number of peers increases from $2^7$ to $2^{14}$, the number of involved nodes and search messages for SSP go up rather slowly, outperforming DSL in an increasing order of magnitude. For the anti-correlated dataset, Figure 10 demonstrates that the costs for both SSP and DSL follow approximately a linear curve, yet the involved nodes for SSP are only half of the involved nodes for DSL, and the search messages for SSP are between one quarter and half of the messages for DSL. The reduced number of involved nodes and search messages for SSP are due to our approaches of delimiting skyline search space and forwarding a query within the partitioned search subspaces.




(a) Involved Nodes




(b) Search Cost

**Figure 10. Effects of network size on anti-correlated dataset**

### 6.2. Effects of Dimensionality

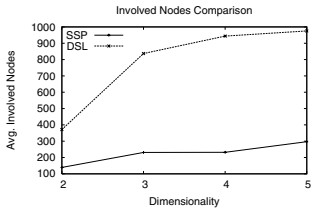
We next investigate the effects of dimensionality by fixing the number of peers and the cardinality while varying the dimensionality from 2 to 5 [16]. In Figures 11, DSL presents a drastic increase in the number of involved nodes. It visits more than three quarters of nodes for a dimensionality larger than 3, and incurs more than 10000 messages in 5 dimensions. Recall that the routing path length on CAN reduces when increasing the dimensionality of the space, yet it seems DSL does not benefit from larger dimensions. In contrast, the performance of SSP remains steady and is far better than DSL in larger dimensions owing to its effective controlling and partitioning of skyline search space.

### 6.3. Effects of Data Size

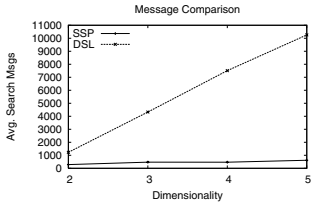
In this experiment, we would like to study the effects of data size. We fix the other parameters and change the total cardinality from 102400 to 1638400, in which case the average data size per node increases from 100 to 1600. Figure 12 witnesses that both SSP and DSL are not very sensitive to data size, but SSP is more stable than DSL in terms of both involved nodes and search messages. We see the costs taken by SSP decline in the larger data size. This is because the average load per node is more likely to be distributed uniformly when increasing data size without changing the number of queries, in which case it is possible to get the skyline answers by visiting fewer nodes.

### 6.4. Real Data Results

We now test the two methods on the real dataset distributed in a small network of 128 peers. For this dataset,

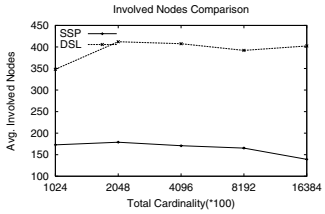



(a) Involved Nodes

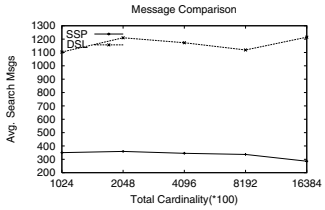



(b) Search Cost

**Figure 11. Effects of dimensionality on anti-correlated dataset**




(a) Involved Nodes



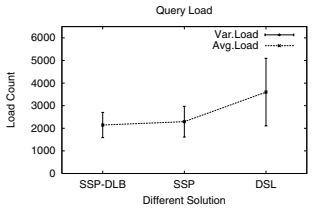

(b) Search Cost

**Figure 12. Effects of anti-correlated data size**

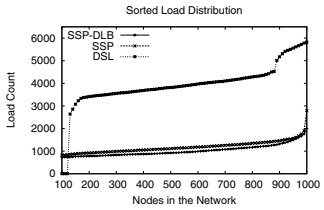
we identify 6 attributes, such as playoff, gained points, assists, etc. The conditions of all skyline queries are set the same according to the real meanings of these attributes. The comparison results are summarized in Table 2, in which the bandwidth is measured by the number of points transmitted in a query following the definition in [23]. SSP costs much less than DSL in average bandwidth per node, for we only transmit $p_{md}$ for pruning instead of using all skyline points. The large portion of involved nodes are due to the large dimensionality specified in skyline query and frequent invocation of the query load balancing process which aims to assuage the high load skew caused by large amounts of the same skyline queries.

**Table 2. Real data results**

| Metrics | SSP | DSL |
|---|---|---|
| avg. involved nodes | 84 | 110 |
| avg. search msgs | 272 | 2491 |
| avg. bandwidth per node | 13 | 204 |




(a) Load Average & Variance




(b) Load Distribution

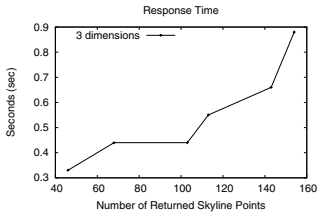
**Figure 13. Effects of load balancing on independent dataset**

## 6.5. Query Load Balancing

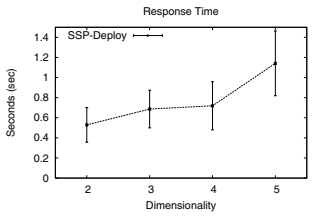
To evaluate the different effects of load balanced partitioning and the dynamic balancing discussed in Section 5, we compare SSP with both mechanisms enabled (SSP-DLB) to SSP with dynamic balancing disabled and to DSL on default settings. We only demonstrate results on the independent dataset because query load is easier to be balanced on an anti-correlated dataset. As Figure 13 shows, query load in DSL has a larger variance than in SSP or SSP-DLB. Moreover, some of its nodes hold a very small number of data points and has no query visiting, which suggests data partitioning of CAN that implicitly assumes a uniform data distribution is not appropriate for load balance. The advantage of our dynamic balancing process is not clearly shown, but its sorted load distribution has not changed as much as in SSP, especially in the heavier load side in Figure 13(b).

## 6.6. Response Time

We evaluate the progressiveness and the response time of our algorithm by distributing 1,638,400 independent data points in real deployment. Figure 14(a) presents the returning time of a three-dimensional skyline query that includes 154 skyline points. The first 46 results are reported at around 0.3 seconds, and all answers are returned within 0.9




(a) Real progressiveness




(b) Time Average & Variance

**Figure 14. Response time on independent dataset**

seconds. Figure 14 shows the total response time varying with dimensionality, in which the response time generally increases with the size of skyline. We can answer a two-dimensional skyline query in a time as short as 0.19 seconds and a five-dimensional skyline query with more than 2000 results in no more than 1.6 seconds.

## 7. Conclusion

In this paper, we have addressed the efficient processing of traditional centralized skyline querying on P2P networks. Based on a tree structured network BATON [12], we have proposed a skyline processing algorithm to partition the skyline space adaptively to control query forwarding behavior effectively. Consequently, we have been able to significantly reduce the number of visited nodes and search messages. We have also devised approaches for effective query load balancing. The correctness and effectiveness of our proposed algorithm were formally proved and validated by experiments. As for future research, we will investigate efficient approximate algorithm for high dimensional data querying in the P2P settings.

## References

[1] W. T. Balke, U. Güntzer, and J. X. Zheng. Efficient distributed skylining for web information systems. In *EDBT*, pages 256–273, 2004.

[2] A. R. Bharambe, M. Agrawal, and S. Seshan. Mercury: supporting scalable multi-attribute range queries. In *SIGCOMM*, pages 353–366, 2004.

[3] S. Börzsönyi, D. Kossmann, and K. Stocker. The skyline operator. In *ICDE*, pages 421–430, 2001.

[4] M. Cai, M. R. Frank, J. Chen, and P. A. Szekely. Maan: A multi-attribute addressable network for grid information services. In *GRID*, pages 184–191, 2003.

[5] C. Y. Chan, H. V. Jagadish, K.-L. Tan, A. K. H. Tung, and Z. Zhang. On high dimensional skylines. In *EDBT*, pages 478–495, 2006.

[6] J. Chomicki, P. Godfrey, J. Gryz, and D. Liang. Skyline with presorting. In *ICDE*, pages 717–816, 2003.

[7] P. Ganesan, M. Bawa, and H. Garcia-Molina. Online balancing of range-partitioned data with applications to peer-to-peer systems. In *VLDB*, pages 444–455, 2004.

[8] P. Ganesan, B. Yang, and H. Garcia-Molina. One torus to rule them all: Multidimensional queries in p2p systems. In *WebDB*, pages 19–24, 2004.

[9] P. Godfrey, R. Shipley, and J. Gryz. Maximal vector computation in large data sets. In *VLDB*, pages 229–240, 2005.

[10] http://basketballreference.com.

[11] Z. Huang, C. S. Jensen, H. Lu, and B. C. Ooi. Skyline queries against mobile lightweight devices in manets. In *ICDE*, 2006.

[12] H. V. Jagadish, B. C. Ooi, and Q. H. Vu. Baton: A balanced tree structure for peer-to-peer networks. In *VLDB*, pages 661–672, 2005.

[13] D. Kossmann, F. Ramsak, and S. Rost. Shooting stars in the sky: An online algorithm for skyline queries. In *VLDB*, pages 275–286, 2002.

[14] X. Lin, Y. Yuan, W. Wang, and H. Lu. Stabbing the sky: Efficient skyline computation over sliding windows. In *ICDE*, pages 502–513, 2005.

[15] W. S. Ng, B. C. Ooi, and K.-L. Tan. Bestpeer: A self-configurable peer-to-peer system. In *ICDE*, page 272, 2002.

[16] D. Papadias, Y. Tao, G. Fu, and B. Seeger. An optimal and progressive algorithm for skyline queries. In *SIGMOD Conference*, pages 467–478, 2003.

[17] J. Pei, W. Jin, M. Ester, and Y. Tao. Catching the best views of skyline: A semantic approach based on decisive subspaces. In *VLDB*, pages 253–264, 2005.

[18] S. Ratnasamy, P. Francis, M. Handley, R. M. Karp, and S. Shenker. A scalable content-addressable network. In *SIGCOMM*, pages 161–172, 2001.

[19] Y. Shu, K. L. Tan, and A. Zhou. Adapting the content native space for load balanced indexing. In *DBISP2P*, pages 122–135, 2004.

[20] I. Stoica, R. Morris, D. R. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM*, pages 149–160, 2001.

[21] K. L. Tan, P. K. Eng, and B. C. Ooi. Efficient progressive skyline computation. In *VLDB*, pages 301–310, 2001.

[22] C. Tang, Z. Xu, and M. Mahalingam. psearch: information retrieval in structured overlays. *Computer Communication Review*, 33(1):89–94, 2003.

[23] P. Wu, C. Zhang, Y. Feng, B. Y. Zhao, D. Agrawal, and A. E. Abbadi. Parallelizing skyline queries for scalable distribution. In *EDBT*, pages 112–130, 2006.

[24] Y. Yuan, X. Lin, Q. Liu, W. Wang, J. X. Yu, and Q. Zhang. Efficient computation of the skyline cube. In *VLDB*, pages 241–252, 2005.

[25] C. Zhang, A. Krishnamurthy, and R. Y. Wang. Skipindex: Towards a scalable peer-to-peer index service for high dimensional data. Technical report, Princeton University, 2004.