The VLDB Journal (2009) 18:345–362
DOI 10.1007/s00778-008-0104-3

REGULAR PAPER

Skyframe: a framework for skyline query processing in peer-to-peer systems

Shiyuan Wang · Quang Hieu Vu · Beng Chin Ooi · Anthony K. H. Tung · Lizhen Xu

Received: 24 September 2007 / Revised: 8 May 2008 / Accepted: 22 May 2008 / Published online: 24 June 2008

Abstract This paper looks at the processing of skyline queries on peer-to-peer (P2P) networks. We propose Skyframe, a framework for efficient skyline query processing in P2P systems, which addresses the challenges of quick response time, low network communication cost and query load balancing among peers. Skyframe consists of two querying methods: one is optimized for network communication while the other focuses on query response time. These methods are different in the way in which the query search space is defined. In particular, the first method uses a high dominating point that has a large dominating region to prune the search space to achieve a low cost in network communication. On the other hand, the second method relaxes the search space in order to allow parallel query processing to speed up query response. Skyframe achieves query load balancing by both query load conscious data space splitting/merging during the join/departure of nodes and dynamic load migration. We further show how to apply Skyframe to both the P2P systems supporting multi-dimensional indexing and the P2P systems supporting single-dimensional indexing. Finally, we have conducted extensive experiments on both real and synthetic data sets over two existing P2P systems: CAN (Ratnasamy in A scalable content-addressable network. In: Proceedings of SIGCOMM Conference, pp. 161–172, 2001) and BATON (Jagadish et al. in A balanced tree structure for peer-to-peer networks. In: Proceedings of VLDB Conference, pp. 661–672, 2005) to evaluate the effectiveness and scalability of Skyframe.

Keywords Framework · Skyline query processing · Peer-to-peer systems · Optimization · Load balancing

S. Wang (✉)
University of California, Santa Barbara, USA
e-mail: sywang@cs.ucsb.edu

Q. H. Vu · B. C. Ooi · A. K. H. Tung
National University of Singapore, Singapore, Singapore
e-mail: hieuvq@nus.edu.sg

B. C. Ooi
e-mail: ooibc@comp.nus.edu.sg

A. K. H. Tung
e-mail: atung@comp.nus.edu.sg

L. Xu
Southeast University, Nanjing, China
e-mail: lzxu@seu.edu.cn

1 Introduction

Skyline queries, which are useful for multi-criterion decision support, preference answering and continuous status monitoring, have been well studied in centralized systems [25]. 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. As an example, a user might want to find hotels with low price and yet close to beach. A hotel which is worse than another hotel in both of these criteria will be dominated and thus will not be considered as part of the answer.

In a P2P system where decisions have to be made based on distributed data and services, skyline computation can be very useful. Consider an online scientific data analysis system in which participating organizations that focus on different parts of the experiments publish and exchange their data. In this case, detecting the skyline will help scientists identify outstanding data and results. Since there may be large numbers of queries issued by different scientists in a short time, it is crucial to respond to skyline queries quickly. At the same time, we want to minimize the communication

bandwidth and balance the query processing loads among the peers. However, little work has been done on such efficient processing of skyline queries in a P2P context.

Adapting a centralized strategy for skyline computation to a P2P setting is not straightforward. Most, if not all centralized strategies, are not easily modified to prune visits to peers that are not involved in the skyline computation. In [37], the first attempt was made at progressive processing of skyline queries on the CAN [29] P2P network. The proposal controls query propagation based on the partial orders of CAN's zones. Even though the most irrelevant succeeding zones are very likely to be pruned, the search boundary of succeeding zones can only be defined by preceding zones, which may reduce parallelism. Moreover, its zone replication load balancing approach is not efficient because of the expensive cost of replication update, especially when the number of replicas is large. This can happen easily if there are many skyline queries that seek the results in the whole data space.

In this paper, we propose Skyframe, a framework for skyline query processing in structured P2P systems. Skyframe consists of two querying methods. The first method is called Greedy Skyline Search (GSS). This method prunes the query search space by using a high dominating point that has a large dominating region, as defined in [19]. Compared to existing methods, the advantage of GSS is that it limits the number of involved nodes, the number of query messages, and consequently the bandwidth consumed in the search process. However, similar to other existing methods, this method takes approximately $O(2 \cdot S)$ steps for processing a query, where S is the search time in the overlay network (i.e log N if CHORD [32] or BATON [20] is used, $d \cdot N^{1/d}$ if CAN [29] is used, where N is the number of nodes in the system, d is the data dimensionality). This is because S steps are taken to find a high dominating point and another S steps for processing the query after that.

In some systems, it is desirable to get the query results as soon as possible. For this purpose, we introduce a second method, called Relaxed Skyline Search (RSS). This method relaxes the skyline search space boundaries slightly to speed up query response. In particular, RSS first searches for all border nodes which contain skyline points with high probability. After that, adjacent nodes of these border nodes are further queried if necessary. In this way, the system can provide faster query response time at approximately $O(S + k)$ steps, where $k \ll S$. The disadvantage of RSS compared to GSS is that RSS may return more false skyline points, which are filtered later at the query initiator node. Consequently, RSS may consume more network bandwidth.

To achieve query load balancing, Skyframe uses two types of load balancing: static and dynamic. In static load balancing, the query loads are balanced based on a predetermined partitioning of the data space at node arrival and departure. On the other hand, in dynamic load balancing, nodes periodically sample loads from both neighboring and random nodes to detect if an imbalance has occurred and, if it has, load balancing is triggered. The procedure migrates data from a heavily loaded node to a lightly loaded node.

We show that Skyframe can be applied to not only the P2P systems that support multi-dimensional indexing but also the P2P systems that support single-dimensional indexing by implementing algorithms on both CAN [29] and BATON [20]. To apply Skyframe on BATON, we propose a method to convert regions in a multi-dimensional space to single dimensional values. This method 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 number of accessed peers and search messages during skyline query processing.

In summary, we make the following main contributions:

- We propose Skyframe, a framework consisting of two querying methods for efficient skyline processing on Peer-to-Peer systems. The first method, Greedy Skyline Search (GSS), is optimized for network communication while the second method, Relaxed Skyline Search (RSS), focuses on query response time.
- We introduce a method to balance the query loads among the peers in Skyframe through both load induced data space partitioning and dynamic load migration. For this, we propose a novel load sampling mechanism that balances the quality of the samples and the efficiency of the sampling process by combining direct and random sampling.
- We show how to apply Skyframe to the P2P systems that support multi-dimensional indexing (represented by CAN) and single-dimensional indexing (represented by BATON).
- We make a comparison between the two core search methods of Skyframe and conduct extensive experimental studies to evaluate the effectiveness and scalability of Skyframe.

This paper is an extended version of our previous paper [36]. In the previous paper, we present only the Greedy Skyline Search and its implementation on BATON. In this paper, we have extended it to become a framework, which can be adapted in different structured P2P systems. Additionally, we introduce a new method, Relaxed Skyline Search that is optimized for query response time. We present a discussion about the difference between these two methods in terms of advantages and disadvantages of each method compared to the other. Extensive experiments have been conducted in both methods to validate our claims.

The rest of the paper is organized as follows. Section 2 reviews related work. Section 3 describes the general framework. Section 4 presents the detailed implementation of the

framework on CAN and BATON. Section 5 shows our experimental study. Finally, Sect. 6 concludes the paper.

2 Related work

Skyline querying was first discussed in the database field in [5]. In this work, the authors introduce two basic algorithms *Block-Nested-Loops* (*BNL*) and *Divide-and-Conquer* (*D&C*) from which algorithms for computing skyline results were derived, using standard index structures *B-tree* and *R-tree*. In particular, *BNL* maintains a self-organized window for currently incomparable tuples, to compare with every incoming tuple. *D&C* partitions the entire data set into multiple subsets from which local skyline results are computed and then merged. These two algorithms, *BNL* and *D&C*, set up a basic foundation for skyline computing and have had a significant impact on later works [8,11,16,21,23,25–27,33,34,40].

The number of computations in *BNL* was optimized in SFS [11] by pre-sorting the input data into the topological order corresponding to the skyline criteria. The first work to progressively compute the skyline was proposed in [33] with two algorithms: *Bitmap* and *Index*. *Bitmap* encodes the input data into a bitmap structure so that the skyline points can be identified quickly using bitwise operations. *Index* transforms each data point into a single dimensional space and builds a B^+-tree index for each dimension such that skyline points are most likely to be found on top of the indexes. However, this method is not efficient when handling skyline points that are not particularly good in any dimension. As a result, a different approach was proposed in *Nearest Neighbors* (*NN*) [21] and *Branch-and-Bound Skyline* (*BBS*) [25] which convert skyline query to a nearest neighbor query towards the query reference point. These algorithms employ the *R-tree* index to recursively partition the data set and compute the skyline points. The algorithm in *BBS* has been proven to be I/O optimal if we adopt the best-first strategy to minimize *R-tree* node accesses and the size of necessary sub-partitions.

Subsequently, some works investigated the semantics of query by subspace analysis [26,27,34,40], the maintaining of skylines on data updates [38,39], the processing of skyline queries over data streams [23], in subspaces with range constraints [12], or in high dimensional spaces [8]. Some other works processed new kinds of skyline queries in more complicated data environments, such as $top-k$ skyline with the maximum number of dominated data points [24], spatial skyline considering a set of query reference points or neighborhood [22,30], skyline with partially ordered domains [7], skyline with dominance in k dimensions on high-dimensional data [9] and probabilistic skyline on uncertain data [28]. There was also a study to speed up the maximal vectors computation [16] combining the benefits of the non-indexing skyline algorithms BNL and SFS.

Unfortunately, the above centralized algorithms cannot be directly applied in distributed environments. As a result, efficient processing of a distributed skyline query is still a challenge. In [3], Balke et al. proposed to use both sorted and random access to reduce data access on vertically distributed data, in a web information system. Apparently, their strategies do not apply to distributed data that is horizontally partitioned. In [19], Huang et al. defined a filter tuple with maximum dominating region to prune non-skyline points transmitted on a peer-to-peer like mobile devices network. However, their approach is dedicated to small-scale distributed systems and still needs to visit all nodes to retrieve the skyline. Consequently, it is not scalable enough to be used on a large-scale P2P network.

Some recent works process skyline query variants on distributed networks. In [13], Deng et al. proposed a variant of skyline query, called multi-source skyline query, which has several query reference points instead of just one reference point. In particular, the authors focused on processing such relative skyline query on the road networks. Another work, SKYPEER [35], which supports skyline query in a super-peer P2P network, suggested that subspace skyline queries can be effectively answered by storing and scanning the super-set of skyline whose attribute set is the super-set of all subspace skylines. Similar to [19], the authors also proposed a threshold based algorithm to optimize local skyline computation at peers and reduce the amount of unnecessary data transmitting on the network. Instead of processing exact skyline queries, [17] relaxed the definition of skyline to include the data points within a distance to the actual skyline point, such that they reduced the network processing cost by using data summaries and approximate answers.

The closest work to ours is DSL [37]. It parallelizes the search for skyline and progressively returns skyline answers by enforcing a partial order on query propagation based on CAN. But the succeeding nodes have to wait for preceding nodes' completion to start their computation, so it slows down the query response time. Furthermore, since the query search boundary is not refined, DSL incurs unnecessary return overhead. Additionally, the system emphasizes 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. Instead, 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 region containing the skyline.

Another area of the related work consists of works that inspired us to map multi-dimensional data onto a single dimensional P2P network. MAAN [6] uses locality preserving hashing to map data values onto Chord [32] identifier

space. Mercury [4] 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 structures in MAAN and Mercury are not effective for processing skyline queries that implicitly specify the constraints among the attributes. Instead, we consider the whole data space in the structure. For this, some works use the space filling curves as the hash function to build range query services on Distributed Hash Table networks [2,10]. The works closest to ours in the multi-dimensional data indexing are Murk [15], SkipIndex [41] and ZNet [31]. 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 data space partitioning, they do not deal with the query load balancing problem that is present in skyline query processing.

Since the potential skyline may always exist in a small portion of the whole data space, balancing the query load among peers is important for the performance of frequent skyline queries in a P2P system. Many works have explored data/storage load balancing on P2P networks either by data replication [1] or data movement/re-partition [14]. The way they move the data is useful for load balancing in our framework, while the load we look at is not only induced by data, but also by the queries themselves.

3 Skyframe architecture

In this section, we will start with the problem definition, after which we will present two core components of our framework (1) the query processing manager, which is in charge of processing skyline queries, and (2) the query load-balancing manager, which is responsible for balancing the query load among participating nodes. We will summarize the framework at the end of the section.

3.1 Problem definition

Without loss of generality, we assume that the data values of each dimension are in the range [0, 1] and the whole d-dimensional data space {[0, 1], [0, 1], ..., [0, 1]} is distributed on a P2P network with N nodes. 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-cubes, each of which is confined by 2 points: a lower bound and an upper bound (e.g. lower left and upper right in a two-dimensional space). In this data space, our framework processes the skyline query in the form $SQ = (q_1, q_2, \ldots, q_d)$. q_i can take one of two values: $\{`Max', `Min'\}$. $q_i = `Min'$ if we require a minimal value on dimension $i$ and $`Max'$ otherwise.

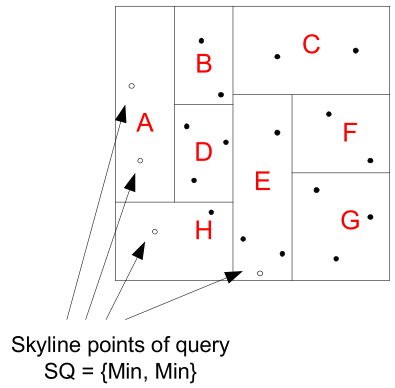


Fig. 1 An example of skyline query on two dimensional space

As an example, Fig. 1 shows a distribution of data regions in a 2-dimensional space with 8 nodes in the P2P system. In this example, a skyline query $SQ = \{`Min', `Min'\}$ should return 4 skyline points, which are hollow points. 2 points are stored at node A, 1 point is stored at node H, and 1 point is stored at node E. In the rest of the paper, if no other skyline query is specified, we also use this skyline query $SQ = \{`Min', `Min'\}$.

3.2 Query processing manager

The query processing manager provides two methods, Greedy Skyline Search and Relaxed Skyline Search, each of which is based on a different approach to prune the skyline search space. We define the *skyline search space* to be the ranges of the data space or the data regions that need to be examined to answer a skyline query.

3.2.1 Greedy skyline search

While it is costly to find the optimal skyline search space, it is possible to employ a greedy strategy to find a greedy search space, which is not optimal but is good, in a much lower cost. The basic idea of the Greedy Skyline Search (GSS) method is that we start the search from the node whose local results are guaranteed to be in the final skyline. We denote such a node as *SQ-Starter*. This node can be located by searching the most dominating boundary point that dominates all other points in the data space. For instance, we would take point (0.0, 0.0) in the two dimensional space if SQ is the $(`Min', `Min')$ skyline query. After finding the *SQ-Starter* node, we use local skyline points on *SQ-Starter* to delimit the skyline search boundaries when we continue to process

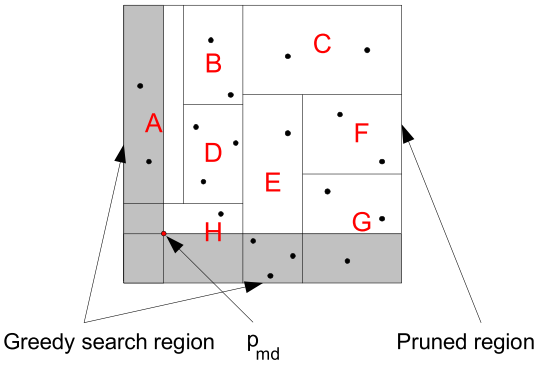


Fig. 2 Greedy Skyline search

the query.[1] In particular, we compute local skyline results and select the most dominating point that has the largest dominating region [19] from local data set on *SQ-Starter*. Let this point be $p_{\text{md}}(a_1, a_2, \ldots, a_d)$. Even though p_{md} is not the global most dominating point, p_{md} should be a high dominating point, which has a large dominating region, and hence it can be used to prune the search space to get a greedy skyline search space, which is defined in the following definition. Note that, to avoid incurring unnecessary network bandwidth, we use only p_{md} for pruning irrelevant nodes and refining the search space instead of using all intermediate skyline results as is done in [37].

Definition 1 Let p_{md} be the point that has the largest dominating region among the data points on *SQ-Starter*. The *greedy skyline search space* is the set of all data points that are not dominated by p_{md}. It is the union of d hyper-cubic search ranges (SR), each of whose boundary is limited only by one coordinate of p_{md} along one axis, such as $\{[0,1], \ldots, [0, a_i), \ldots, [0, 1]\}$.

We define a *dominated node* as follows:

Definition 2 A node is dominated (and should be pruned) i.f.f. the single ideal point with the best value in each dimension within its ranges, denoted as p_{best}, is dominated by p_{md}.

For the example query SQ, the shadowed part in Fig. 2 indicates its skyline search space. Here, the high dominating point p_{md} is stored at the *SQ-Starter* node H. Dominated nodes are B, C, D, and F.

Now we prove we can find the correct answers by visiting only the non-dominated (unpruned) nodes inside the skyline search space. Without loss of generality, we will formalize the proofs for the situation where the query requires minimal values on all dimensions.

Lemma 1 *All nodes outside the skyline search space are dominated nodes.*

[1] Note that, through the load balancing process, we can always guarantee that the *SQ-Starter* node should contain data for computation.

Proof Since the data space outside the greedy skyline search space (i.e. $\{(a_1, 1], (a_2, 1], \ldots, (a_d, 1])\}$) 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 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 3 *Any final skyline points cannot be dominated by any points of the dominated nodes.*

Proof For the sake of brevity, let us consider, w.l.o.g., a two dimensional space and $SQ = \{`Min', `Min'\}$. Suppose $p_s(s_1, s_2)$ is a skyline point and 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 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. This contradicts with the assumption, so p_s cannot be dominated by p_d.

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

Theorem 1 *The non-dominated (unpruned) nodes in the greedy skyline search space constitute the complete skyline set and only the skyline set.*

In general, the algorithm for GSS is executed in two phases. At first, the system looks up the *SQ-Starter* node. When the *SQ-Starter* node is found, that node computes the high dominating point from which the greedy search region is formed. The *SQ-Starter* node then routes the query to the nodes covering the search region. These nodes compute their local skyline points, refine the skyline search space and return the result to the query initiator. At the end of the process, the query initiator computes global skyline points from the local skylines points which it has received.

3.2.2 Relaxed Skyline search

Although GSS prunes the search space as much as possible, it is necessary to find the *SQ-Starter* node in the first phase before actually executing the skyline query in the second phase. To optimize the query response time, we introduce the second method, Relaxed Skyline Search (RSS) method, which speeds up the search process by relaxing the boundaries of the search space of GSS. RSS is based on the observation that most of the time, the greedy search space involves

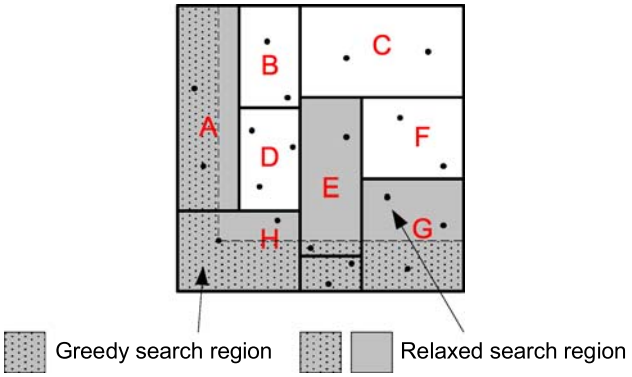


Fig. 3 Relaxed Search space

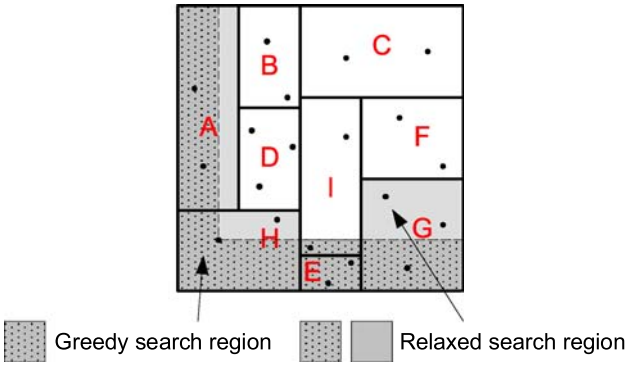


Fig. 4 Relaxed search space with additional steps

skyline border nodes, which are at the border of the skyline search space (we call them *SQ-Border* nodes).[2] As a result, we can directly search skylines at the *SQ-Border* nodes to save look up time, since *SQ-Border* nodes can be targeted directly from the skyline query. For example, in Fig. 3, if $SQ = \{`Min', `Min'\}$ is used, A, H, E, and G are the border nodes while if $SQ = \{`Max', `Max'\}$ is used, the border nodes are A, B, C, F and G. Now, we define the relaxed skyline search space as follows.

Definition 3 The relaxed skyline search space is the union of the regions covered by the border nodes and the greedy skyline search space.

From this definition, it is clear that the relaxed skyline search space covers the greedy skyline search space, and hence from Theorem 1, we can infer that the relaxed skyline search space returns the complete skyline set and only the skyline set.

In general, in RSS, instead of finding the p_{md} first and pruning the search space from that point as in GSS, we can find all *SQ-Border* nodes in parallel. These nodes process the query locally to find their local skyline points to return to the query initiator. Upon receiving results from the border nodes, the query initiator can compute the greedy skyline search space (since the *SQ-Starter* must also be a border node) and the regions covered by the border nodes. Two cases can happen here.

In the first case, the regions covered by the border nodes also cover the greedy search space. In this case, the query initiator simply computes the final answer from the skyline points that were returned in the previous step. For example, assume that we have a skyline query $SQ = \{`Min', `Min'\}$ in a two dimensional space as in Fig. 3. The system first finds nodes A, H, E, G since they are the border nodes. Since regions covered by these nodes also cover the greedy search region, the query initiator, upon receiving results from A, H, E, G, can compute the final set of skyline points. From this example, we can see that this method also expands the same number of nodes as the previous method. However, this only incurs S steps while the previous method incurred $2S$ steps for processing the query. This is because in this case, this method takes a maximum of S steps to find border nodes to process query and the process stops after that. On the other hand, the previous method required S steps to find the *SQ-Starter* node in the first phase and S steps to find other nodes in the search space in the second phase. Nevertheless, this method may return additional false skyline points, which are pruned later at the query initiator.

In the second case, the regions covered by the border nodes do not cover the greedy search space. Therefore, the system needs additional steps for processing the query. However, this does not take as many steps as in the previous method, where only the *SQ-Starter* is known after the first phase. Since these additional nodes, which need to process the skyline query, are all adjacent nodes of the border nodes, it takes only a number of k steps for further processing, where $k \ll S$. Consider the example shown in Fig. 4. There, in addition to processing the skyline query SQ at the border nodes A, H, E, and G, we also need to process the query at node I since the regions covered by the border nodes do not totally cover the greedy search space and the greedy skyline search space intersects with the region covered by I. However, only one additional step is needed since I is an adjacent node of a known node E. Furthermore, node I only needs to search skyline points in the shaded region bounded by the greedy search space. After this step, the process stops since all regions covered by A, H, E, G, and I have already covered the greedy skyline search space.

In conclusion, the overall relaxed skyline search also consists of two phases. In the first phase, the system searches for *SQ-Border* nodes. These nodes compute and return local skyline results to the query initiator. The first phase ends when the query initiator receives results from all *SQ-Border* nodes. At this time, the query initiator computes the high

[2] Note that, this statement holds true whenever p_{md} is not situated in the border of any dimension.

dominating point and the greedy query search region to determine if additional regions need to be searched. If there are additional search regions, the second phase starts in which the system routes the query to the nodes covering these regions via known border nodes. These nodes compute their local skyline points and return the result to the query initiator. When no further regions need to be searched, the query initiator computes the global result from received results and the process terminates.

3.3 Query load-balancing manager

In our framework, we assume that the query load of a node is proportional to the data load of that node.[3] To balance the load among nodes in the system, the following two steps are adopted. First, the data space is partitioned based on query load and node join/departure procedures are redesigned for load balancing. Second, a novel mechanism is proposed for sampling and dynamically balancing load during query processing. We start with the definition of query load.

Definition 4 The query load of a node is the sum of (1) the number of data points retrieved from local data set to answer a query and (2) the number of messages that are routed by the node but which do not lead to executing a query locally.

3.3.1 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 the best candidates to share load during the node join/departure process. We observe that allocating relatively smaller space to the "hot" regions facilitates load balancing. Therefore, we partition the data space by dividing a hyper-cubic region into two equally loaded sub-regions. Correspondingly, when necessary, we combine two buddy regions or the regions of adjacent nodes.

- **Node join**. We seek better query load balance during node join by selecting a node with the heaviest load among neighbor nodes to forward the join request. Additionally, instead of contacting only one node, we let the new node contact several nodes at different places and select the heaviest loaded node among the contacted nodes to join.
- **Node departure**. When a node leaves, two cases are considered. In the first case, if the departing node has a neighbor whose load is light, that neighbor node takes over the region of the departing node. On the other hand, if all neighbor nodes of the departure node have heavy load, the node tries to find a lightly loaded node L, which also has a lightly loaded neighbor node. Since there is a neighbor node of L, whose load is also light, L can pass the region it covers to that neighbor node and then take over the position of the departing node.

3.3.2 Dynamic query load balancing

The load balancing process during query processing works as follows. First, each node checks whether there is an imbalance in load. If yes, a data migration process is triggered to balance the load. To check the load imbalance, query loads are either periodically gathered by each node from the neighbor nodes in its routing tables and adjacent nodes or with query load changing notification from these nodes. The load imbalance is determined by the difference δ of the node's local load and sampled query loads. If δ is larger than a predefined threshold σ, an imbalance is detected.[4] 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 [4]. A number of $\log N$ nodes that are not linked to are sampled by sending probes that are attached through a small length limit of routing hops ($\log N$) 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.

Once the load imbalance is detected, load balancing can be performed in two ways. First, the node can simply try to balance its load with a neighbor node by repartitioning the data space between itself and its left/right neighbor nodes to make the query load on the neighboring nodes approximately equal. In particular, if the node is heavily loaded, it passes the extra data after data space repartition to a lightly loaded neighbor node while if the node is lightly loaded, it steals the extra data from a heavily loaded neighbor node. Second, if the first attempt cannot balance the load, the node needs to balance its load with a node from the sampling process. In this case, if the node is a lightly loaded node, the heaviest loaded node among sampled nodes is selected for the load balancing process. This node is considered as a heavily loaded node. On the other hand, if the node is a heavily loaded node, the lightest loaded node among sampled nodes is selected as a lightly loaded node for the load balancing process. To balance the load, the lightly loaded node will force-leave its position to rejoin next to the heavily loaded node.

3.4 Skyframe

To summarize, our general framework as described in Fig. 5, comprises three components built on top of the basic over-

[3] Our load balancing technique can be used together with replication techniques to avoid bottleneck at nodes holding popular items.

[4] The imbalance ratio is computed according to [14].

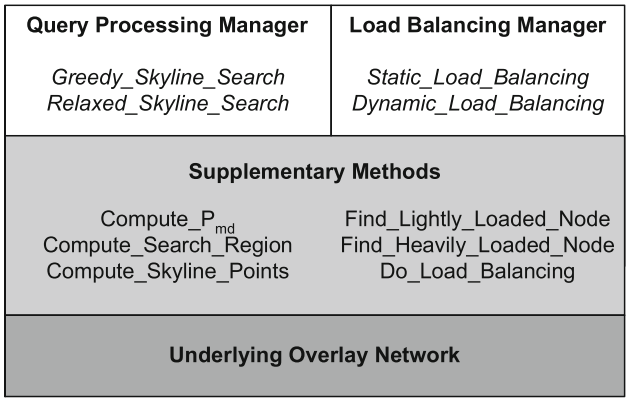


Fig. 5 The Skyframe

lay network. The first component contains supplementary methods, which are used by the other two components, the Query Processing Manager and the Query Load-Balancing Manager. These two components contain concrete methods for query processing and load balancing as discussed above.

There are not major differences to instantiate our framework on different structured P2P networks, as long as that overlay network has a way to represent a data region on a node, and utilizes its underlying routing structure to (1) forward the query to nodes in the search region once a skyline search region is defined and (2) look up the *SQ-Starter* node for the Greedy Skyline Search or border nodes for the Relaxed Skyline Search given a query.

In the following section, we will present the instantiation of Skyframe on two typical structured P2P networks CAN [29] and BATON [20].

4 Implementing Skyline query algorithms and instantiating Skyframe

In this section, we discuss in more details the implementation of the algorithms and how to instantiate Skyframe on a multi-dimensional indexing P2P structure (CAN [29]), and a single-dimensional indexing P2P structure (BATON [20]). We begin with a quick introduction to the original structure, and then present the implementation issues of the skyline query algorithms.

4.1 Implementation of the greedy Skyline search

The algorithm for GSS is illustrated in Algorithm 1. As presented in Sect. 3.2.1, in the first phase ($p = 1$), whenever a node receives the query request, if it is not the *SQ-Starter* node, it forwards the query towards the *SQ-Starter* node via a neighbor node (*lines 14–15*). On the other hand, if the node is the *SQ-Starter* node, it first computes local skyline points,

Algorithm 1 GSS(node n, query q, search_region sr, phase p)

```
Define: RT(n) routing table of node n
Define: Region(n) the region maintained by node n
1: if p = 1 then
2:   if n is SQ-STARTER of q then
3:     local_skyline_points = Compute_Skyline_Points
4:     p_md = Compute_P_md
5:     SR = Compute_Search_Region
6:     Partition SR into a disjoint set of subSRs for neighbor nodes in
       RT(n)
7:     for all nodes m in RT(n) do
8:       if m is in charge of a subSR then
9:         GSS(m, q, subSR, 2)
10:      end if
11:    end for
12:    Return local_skyline_points, p_md, and the region the node is in
       charge of to the query initiator.
13:  else
14:    x = a node in RT(n) nearer towards SQ-STARTER
15:    GSS(x, q, null, 1)
16:  end if
17: else {p = 2}
18:  local_skyline_points = Compute_Skyline_Points
19:  Return local_skyline_points and the region the node is in charge
     of to the query initiator
20:  if Region(n) ⊉ sr then
21:    SR = sr \ Region(n)
22:    Partition SR into a disjoint set of subSRs for neighbor nodes
       in RT(n)
23:    for all nodes m in RT(n) do
24:      if m is in charge of a subSR then
25:        GSS(m, q, subSR, 2)
26:      end if
27:    end for
28:  end if
29: end if
```

calculates p_{md} from these points and determines the greedy search region from p_{md} (*lines 3–5*). After that, the search region is partitioned into sub search regions to forward to neighbor nodes, which process the skyline query in the second phase (*lines 6–11*). Finally, the node returns locally computed skyline points, p_{md}, and the region the node is in charge of, to the query initiator (*line 12*).

In the second phase ($p = 2$), whenever a node receives a query, it also receives a search region that is determined by the sender node. This node first computes the local skyline points and returns them together with the region the node is in charge of to the query initiator (*lines 18–19*). If the region the node is in charge of does not cover the whole specified search region, the node continues to partition the search space into smaller search regions and forwards the query towards the neighbor nodes corresponding to these subregions (*lines 20–28*).

When the query initiator receives all local skyline points from the greedy search regions, it recomputes these skyline points to get the final skyline set. Note that, in order to let the query initiator know when all local skyline points are returned to it, the *SQ-Starter* needs to return to the query initiator

Algorithm 2 RSS(node n, query q, dimension d, search_region sr, phase p)

```
Define: RT(n) routing table of node n
Define: Region(n) the region maintained by node n
Define: Border(q, d) the boundary of search region of query q at dimension d
 1: if p = 1 then
 2:   if n is QUERY INITIATOR then
 3:     for each dimension d in q do
 4:       x = a node in RT(n) nearer towards Border(q, d)
 5:       RSS(x, q, d, null, 1)
 6:     end for
 7:   else
 8:     if n is a Border Node at dimension d then
 9:       local_skyline_points = Compute_Skyline_Points
10:       Return local_skyline_points and the region the node is in charge of to the query initiator
11:       for all neighbor Border Node nn at dimension d do
12:         if nn does not exist in the search path from the QUERY INITIATOR to n then
13:           RSS(nn, q, d, null, 1)
14:         end if
15:       end for
16:     else
17:       x = a node in RT(n) nearer towards Border(q, d)
18:       RSS(x, q, d, null, 1)
19:     end if
20:   end if
21: else {p = 2}
22:   if n is not a Border Node then
23:     local_skyline_points = Compute_Skyline_Points
24:     Return local_skyline_points and the region the node is in charge of to the query initiator
25:   end if
26:   if Region(n) ⊉ sr then
27:     SR = sr\Region(n)
28:     Partition SR into a disjoint set of subSRs for neighbor nodes in RT(n)
29:     for all nodes m in RT(n) do
30:       if m is in charge of a subSR then
31:         RSS(m, q, d, subSR, 2)
32:       end if
33:     end for
34:   end if
35: end if
```

p_{md} so that the greedy search region can be determined at the query initiator. Additionally, all nodes which return local skyline points also send to the query initiator the region they are in charge of. Another important point to note is that, since sub search regions are disjoint regions and the query is always forwarded via neighbor nodes, we can guarantee that duplicate query messages are avoided in the system.

4.2 Implementation of the relaxed Skyline search

The algorithm for the relaxed skyline search is illustrated in Algorithm 2. There are also two phases for this algorithm. In the first phase, the query initiator creates a query for each dimension of the skyline query in which each query is responsible for finding a border node on one dimension (parameter d) (*lines 3–6*). After that, when a node receives a skyline query, if it is a border node of the specified dimension d of the query, it computes local skyline points and returns the result together with the region it is in charge of to the query initiator (*lines 9–10*). Additionally, the node forwards the query to neighbor border nodes on the specified dimension, which have not been visited in the search path from the query initiator to the node (*lines 11–15*). On the other hand, if the node receiving the query is not the border node, it continues to forward the query towards the search boundary of the query on the specified dimension (*lines 17–18*). In this way, the process of searching border nodes first starts at the query initiator, then propagates through the most direct path to the border nodes, which are perpendicular to the query initiator at different dimensions, and finally propagates from these first border nodes to the remaining border nodes.

When the query initiator receives local skyline points and search regions from the border nodes, it checks to see if it has received results from all border nodes or not by checking the regions which have been searched. Once the query initiator receives all results from the border nodes, it also knows the *SQ-Starter* node, p_{md} and the greedy search region. Therefore, the node can check to see if additional regions need to be searched. The border nodes, whose zone is adjacent to these additional search regions, are sent the query in the second phase of this algorithm to help locate the nodes in charge of the additional search regions. Finally, the system behaves in a similar way to the previous algorithm, and the process stops when no further nodes or search regions need to be explored (*lines 22–34*).

4.3 Instantiating Skyframe on CAN

4.3.1 Background of CAN

CAN (Content Addressable Network) [29] is based on a logical d-dimensional Cartesian coordinate space. Each data record is mapped into a point in the coordinate space using a hash function. Each node stores all the points in a distinct zone for which it is responsible, and a routing table to all the neighbors in the coordinate space. Two nodes are neighbors if their coordinate spans overlap along d-1 dimensions and abut along one dimension. To search data, the request is forwarded through neighbor nodes towards the node whose zone contains the data. When a new node joins the system, it tries to find a node whose zone is large. That node then gives half of its zone to the new node. When a node leaves the system, if there is a neighbor node whose zone can be merged with the zone of the departure node to produce a valid single zone, then it is done. If not, the zone is handled by the neighbor whose zone is the smallest among neighbor

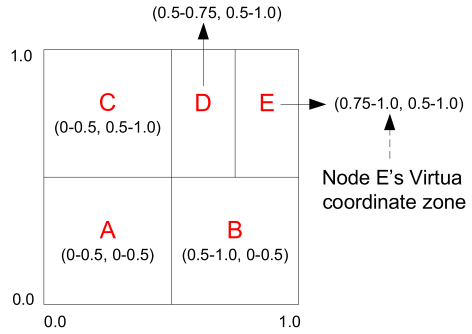


Fig. 6 A two dimensional space CAN system with 5 nodes

nodes' zones. Figure 6 shows an example of a CAN system with five nodes on a two dimensional space.

4.3.2 Implementing Skyline query algorithms on CAN

Using the coordinate space of CAN as the multi-dimensional data space naturally, implementing the proposed skyline query algorithms on CAN is quite straightforward according to Algorithms 1 and 2.

Theorem 2 *The maximum number of steps required to answer a skyline query based on CAN is* $O(2 \cdot d \cdot N^{1/d})$.

Proof In both solutions, the first phase takes maximum $O(d \cdot N^{1/d})$ steps while the second phase requires maximum $O(d \cdot N^{1/d})$ steps for processing. Adding up these costs, it is $O(2 \cdot d \cdot N^{1/d})$.

4.4 Instantiating Skyframe on BATON

Compared to a natural multi-dimensional indexing system like CAN, the challenges in instantiating Skyframe on a single-dimensional indexing P2P system like BATON [20] are mainly about the representation of a data region: (1) how to map data in a multi-dimensional space into a single-dimensional space, and (2) how to adapt the algorithms to process skyline queries on the single-dimensional space. While the advantage in BATON compared to CAN is smaller search hops to reach a target node for processing certain search regions. This section introduces a solution for the challenges. In particular, we employ the Z-curve method to map data in a multi-dimensional space to a single-dimensional space. For query processing purposes, we number the data region and record its split history. As a result, we can estimate the target region number for supporting efficient search.

4.4.1 Background of BATON

BATON (BAlanced Tree Overlay Network) [20] 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

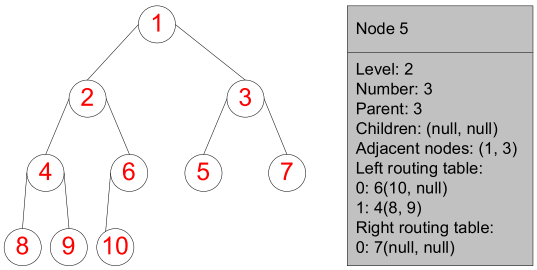
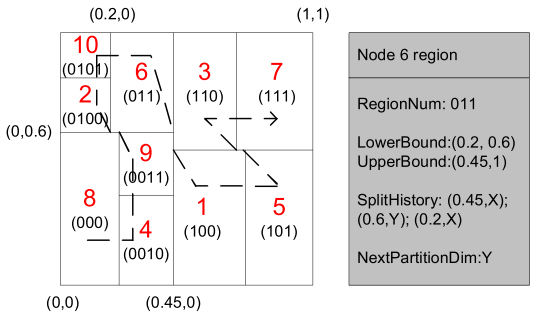
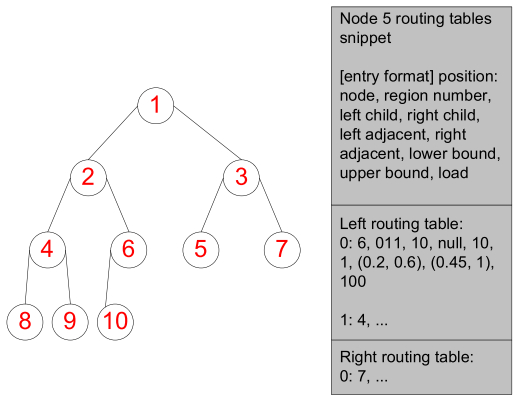


Fig. 7 BATON structure



(a) Nodes and covered regions in a 2-dimensional space



(b) Positions of nodes in BATON structure

Fig. 8 Mapping nodes on BATON

(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 $O(\log N)$ cost for joining/leaving of nodes and exact search. An example of BATON structure is shown in Fig. 7.

4.4.2 Multi-dimensional data region mapping for search

We employ the Z-curve method to map the multi-dimensional data space onto the one-dimensional BATON. Figure 8b

describes the mapping of the data regions in Fig. 8a, in which each tree node maintains a data region.

A data region is represented by the following meta data: (1) *Region number*: a binary string $RNum$ that identifies the region and is consistent with the Z-order of the region, in which the bit values of 0, 1 at certain bit locations indicate the region is in a lower or upper part in the corresponding space split. (2) *Data range*: pairs of $Lower\,Bound$ (bottom left point) and $Upper\,Bound$ (top right point). (3) *Split history*: a list of entries of split value and dimension ($Split\,Pos$, $Split\,Dim$). (4) *Next partition dimension*: a bit indicating the next split dimension. An example for the region information of node 6 is given in Fig. 8a.

Our search mechanism relies on the relationships of the regions and the operations of region numbers. So the term "region" not only represents 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 RNum(m) be its region number, RNum(m).length be the number of bits in RNum(m), and RNum(m)[i] be the ith bit of RNum(m) ($1 \leq i \leq$ RNum(m).length). We define the following region operations.

Definition 5 (Region Succession, $\succ$) Region m "Succeeds" n, or $m \succ n$, i.f.f. RNum(m)[b] = 1, RNum(n)[b] = 0 and RNum(m)[i] = RNum(n)[i] for $1 \leq i < b$, $1 \leq b \leq$ min(RNum(m).length, RNum(n).length). The "precedes" or $\prec$ relationship is defined similarly.

Definition 6 (Region Cover, $\supseteq$) Region m "Covers" n, or m is the superset of n, or $m \supseteq n$, i.f.f. RNum(m).length $\leq$ RNum(n).length and RNum(m)[i] = RNum(n)[i] for $1 \leq i \leq$ RNum(m).length. The "isCovered" or $\subseteq$ relationship is defined in the same way.

To route a query on the mapped BATON, a node 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 is nearest to, the delimited region. Algorithm 3 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

Algorithm 3 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
8:     end if
9:   end for
10: end if
```

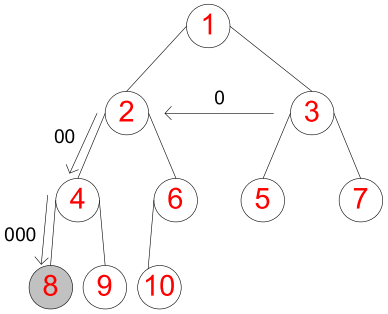


Fig. 9 Routing process

times the searched point p falls in the same range with local region after a history split (*lines 3–4*).

For example, suppose node 3 wants to locate a point in node 8 as shown in Fig. 8a. Node 3 first computes the target point's region number as 0, because it is in the lower part in the first space split on the x-axis. 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 Fig. 9. 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 $O(\log N)$.

4.4.3 Implementing Skyline query algorithms on BATON

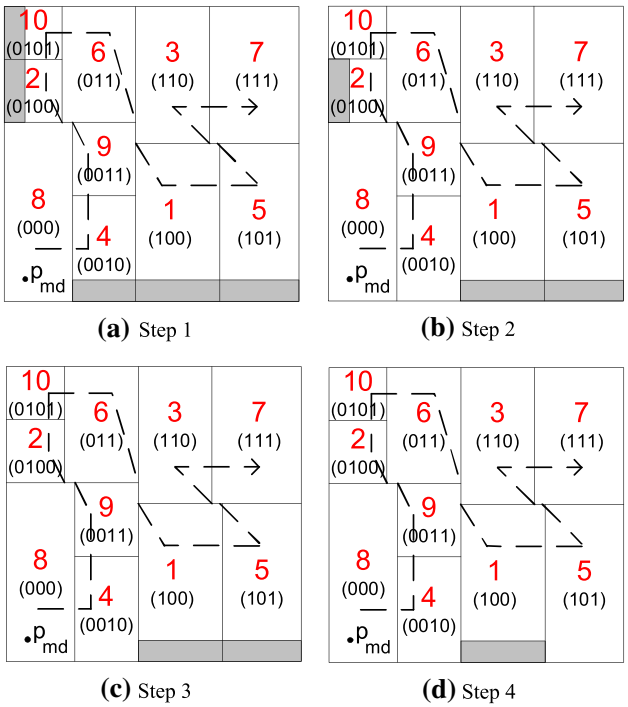
Implementing the proposed skyline query algorithms on BATON needs to take care of two subtle aspects: partitioning a search region SR and routing a query, both of which are based on the above mapping mechanism.

The partitioning of an SR into $subSRs$ is based on the history ranges which are 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. This procedure is detailed in Algorithm 4. Similar to Algorithm 3, 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 (*lines 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 (*lines 10–11*).

For example, as for the search space of SQ, after subtracting the range of *SQ-Starter* as shown in Fig. 10a, we refine the remaining part SR into a single SR estimated as 01, for it falls out of the history range of region 8 in the second time split. The right part SR is partitioned into two parts upon the first space split: the one overlapping region 1 and

Algorithm 4 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* ⊆ HistoryRange(*n*)[*b*]) **then**
3: RNum(*SR*)[*b*] ← RNum(*n*)[*b*]
4: **if** (HistoryRange(*n*)[*b*] is lower part) **then**
5: UpperBound(*SR*)[*b*] ← SplitPos(*n*)[*b*]
6: **else**
7: LowerBound(*SR*)[*b*] ← SplitPos(*n*)[*b*]
8: **end if**
9: Add *SR*'s buddy *subSR* to *subSRSet*
10: **else**
11: RNum(*SR*)[*b*] ← 1 − RNum(*n*)[*b*]
12: break
13: **end if**
14: **end for**



(a) Step 1 **(b)** Step 2 **(c)** Step 3 **(d)** Step 4

Fig. 10 Skyline query solving process

region 5 is assigned an estimated region number 1, and the one overlapping region 4 is estimated as 001.

Routing a query associated with a search region on BATON is different from that on CAN because of the tree structure. As described in Algorithm 5, each *subSR* is forwarded to a non-dominated neighbor, child or adjacent node whose region is covered by *subSR* (*lines 7–11* in Algorithm 5). 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 (*lines 13–14*), or just passed to the adjacent node (*line 16*). Skyline search at each of the next hops (*line 21*) is parallelized.

For the example of answering *SQ*, in the first step (Fig. 10a), we promote *subSR* estimated as 001 to node

Algorithm 5 GSS (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*) ⋂ *SR* ≠ ∅) **then**
2: Compute local skyline not dominated by p_{md} and report to *SQ-Starter*
3: *SR* ← Range(*SR*) − Range(*n*)
4: **end if**
5: *subSRSet* ← partition(*n*, *SR*)
6: **for all** sub-range *subSR* in *subSRSet* **do**
7: **if** (LowerBound(*subSR*) > UpperBound(*n*)) **then**
8: *m* ← NodeWhoseRegionCoveredBy (Region(*subSR*)) in RRT(*n*)
9: **if** ((*m* not exist or Dominated(*m*)) and (RChild(*n*) not processed *subSR*) and (Region(*subSR*) ⊇ Region(RChild(*n*)))) **then**
10: *m* ← RChild(*n*)
11: **end if**
12: **if** ((*m* not exist || Dominated(*m*)) and (RAdj(*n*) not processed *subSR*) and (Region(*subSR*) ⊇ Region(RAdj(*n*)))) **then**
13: *m* ← RAdj(*n*)
14: **end if**
15: **if** (*m* not exist || Dominated(*m*)) **then**
16: **if** (Region(*subSR*) ≻ Region(FarthestNodeInRRT(*n*))) **then**
17: *m* ← FarthestNodeInRRT(*n*)
18: **else**
19: *m* ← RAdj(*n*)
20: **end if**
21: **end if**
22: Map *m* to *subSR* in *subSRSet*
23: **else**
24: // A similar process executes towards the left
25: **end if**
26: **end for**
27: **for all** (*m*, *subSR*) in *subSRSet* **do**
28: GSS(*m*, *SQ*, *subSR*)
29: **end for**

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 Fig. 10b, node 10 refines *subSR* 01 to *subSR* 0100(*line 3*), then promotes it to the left adjacent node 2 who can fully resolve it, and passes *subSR* 1 to the right adjacent node 6 (*line 16*). Figure 10c 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 Fig. 10d, we can see node 1 is the last hop for processing *subSR* 1. Figure 11 demonstrates the corresponding process of query forwarding.

Theorem 3 *Skyframe based on BATON answers a skyline query in* $O((1 + 2d(1 - 1/\sqrt[d]{N}))\log N)$ *steps for uniform*

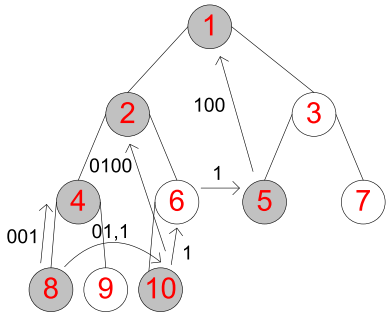


Fig. 11 Skyline query

load distribution, and $O((1 + (m + d)/2)m)$ *steps in the worst case.*

Proof The total number of steps for both solutions consists of the routing hops of locating a *SQ-Starter*/*SQ-Border* in phase 1 and processing an SR in phase 2. Let m be the number of bits in the region number of the farthest region from *SQ-Starter*. Then, phase 1 takes $O(\log N)$ hops for uniform load distribution and $O(m)$ in the worst case. For phase 2, let's consider an SR whose boundary is limited by the coordinate of p_{md} on dimension i. 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 by 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 an SR is the arithmetic progression of the former, namely $O(d + 2d + \cdots + |m/d|d)$ 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 $O(\log N)$, the average number of steps for processing SR is $O(\log N + \frac{1}{2}\log N + \cdots + \frac{1}{2}^{\log N/d} \log N)$, namely $O(2(1 - 1/\sqrt[d]{N})\log N)$. By adding the number of steps in phase 1 and phase 2 we get $O((1 + 2d(1 - 1/\sqrt[d]{N}))\log N)$ for uniform load distribution and $O((1 + (m + d)/2)m)$ for the worst case.

5 Experiment evaluation

We evaluate our proposed framework by instantiating it on both CAN [29] and BATON [20]. We compare our systems to the distributed skyline query algorithm DSL [37], which is implemented on CAN, in terms of network size, dimensionality, cardinality and query load balance. The performance measures are the number of nodes involved in the search process, the number of skyline search messages, the number of search hops, the amount of consumed bandwidth measured by the number of points transmitted in a query following the definition in [37], and the query load distribution.

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	$(2^{10}, \ldots, 2^{14}) \times 100$	$2^{14} \times 100$

We conduct simulation experiments on a Linux box with Intel Xeon 3.0 GHz processor and 2 GB of RAM. The response time is tested in real deployment of 80 peers on a cluster consisting of 20 computing nodes, each of which has an Intel Xeon 3.0 GHz processor and 4 GB 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 [18] that represents a typical correlated data distribution in practice. The other two are synthetic independent and synthetic anti-correlated datasets with a maximal data size of 1,638,400 multi-dimensional points. Note that (1) since many real datasets are correlated in practice and the experimental results of them are similar to those of synthetic correlated datasets, we just use this real NBA dataset instead of a synthetic correlated dataset in our experiments to demonstrate that our methods would work as well in practice as in synthetic data environment; (2) for the synthetic datasets, we show only one set of results in cases where both exhibit similar performance. The experimental settings are summarized in Table 1. Each experiment is repeated 10 times and the average is taken. Each test issues 1,000 skyline queries, which have the same dimensionality as the data space, and the average cost is taken. Each query prefers a 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.

5.1 Effects of network size

We first study the effects of network size using both independent and anti-correlated datasets. Figures 12 and 13, respectively, illustrate the results on the independent and anti-correlated datasets. Note that in Fig. 12a, b, the y-axis is displayed in log-scale. In these figures, Greedy-Can (Baton) and Relaxed-Can (Baton), respectively, represent the Greedy Skyline Search (GSS) and Relaxed Skyline Search (RSS) algorithms on CAN (BATON). The results on both datasets show that in both systems, our algorithms outperform DSL in all aspects. That is because our algorithms prune the search space more efficiently than DSL does (DSL only prunes the search space naively using the top right corner of the

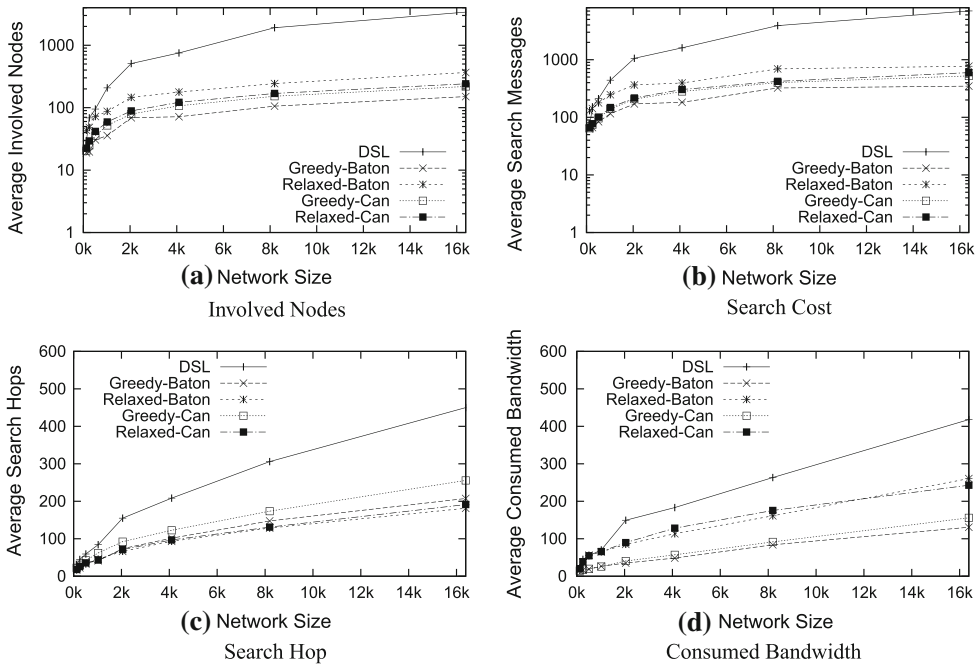


Fig. 12 Effect of network size on independent dataset

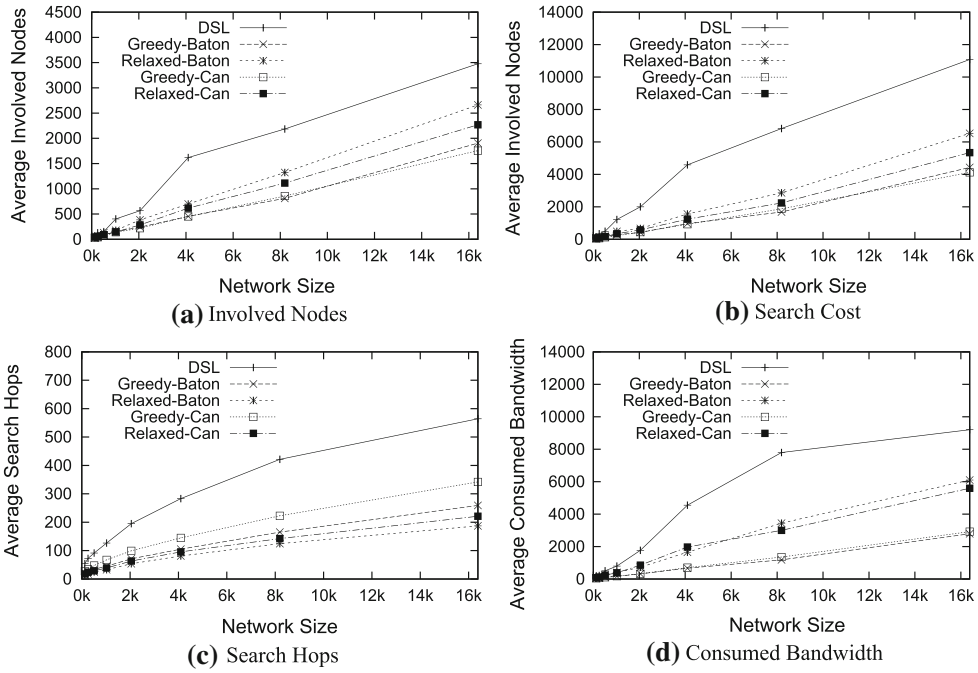


Fig. 13 Effect of network size on anti-correlated dataset

SQ-Starter node). In all testing aspects, our algorithms only incur approximately half of the cost required by DSL. In particular, since the number of skyline points in the independent dataset is much less than those in the anti-correlated dataset, our algorithms can significantly prune the search space to bring a big improvement in the number of involved nodes and the number of search messages used in query processing compared to DSL (see Fig. 12a, b). On the other hand, since DSL does not define the search space at the *SQ-Starter* node, skyline points have to be carried in the query for pruning at subsequence nodes. As a result, in the anti-correlated dataset, since the number of skyline points is enormous, DSL consumes much more bandwidth compared to our algorithms (Fig. 13d).

Making a comparison between our algorithms, GSS and RSS, we observe that in both CAN and BATON, RSS has a faster query response time than GSS (the response time in this case is simply measured by the number of search hops), while GSS is better than the RSS in all remaining aspects. This is because, in order to achieve a faster query response

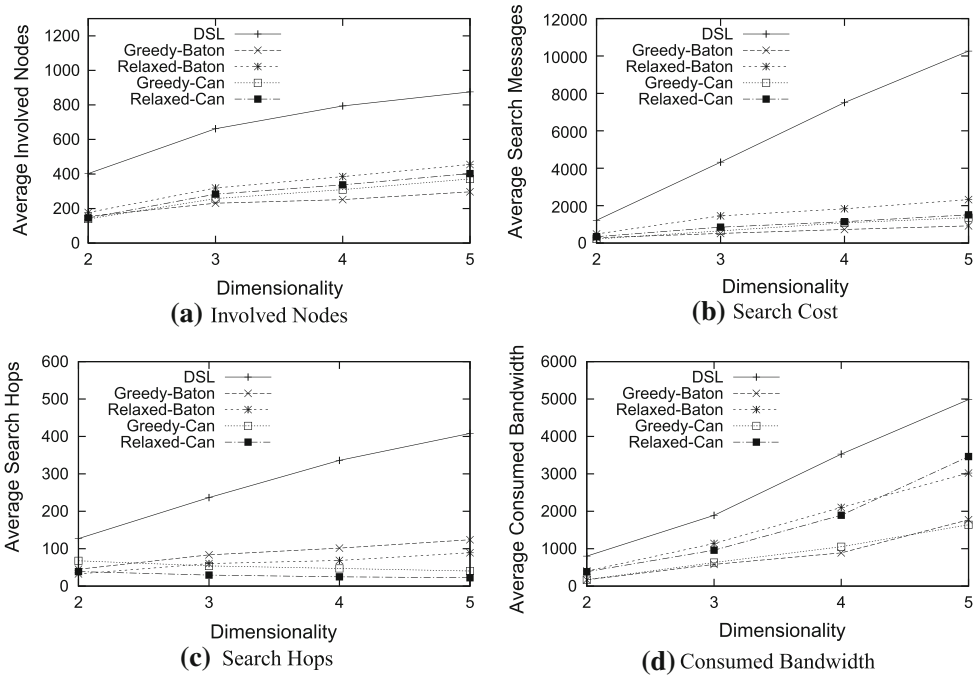


Fig. 14 Effect of dimensionality on anti-correlated dataset

time, RSS parallelizes the search process in the first phase by replacing the process of finding the *SQ-Starter* node with independent processes at the border nodes. This processing strategy makes skyline queries affect a slightly larger region. As a result, RSS produces more false skyline points, which consumes more bandwidth, than GSS does. Additionally, by parallelizing the border node search process, a small number of additional nodes are involved. This causes RSS to have a slightly higher number of involved nodes and search messages. Nevertheless, these extra costs are insignificant compared to the overall costs because the number of involved nodes and the number of search messages required for query processing mainly depend on the number of nodes involved in the search region, which is approximately equal in most of cases in both methods even though search regions in these methods may slightly differ.

5.2 Effects of dimensionality

We investigate the effects of dimensionality by fixing the number of peers and the cardinality while varying the dimensionality from 2 to 5 [25]. The results are displayed in Fig. 14. Figure 14a shows that the number of nodes involved in the DSL method increases drastically with the increase of the dimensionality. In particular, the system visits more than three quarters of nodes for a dimensionality larger than 4, and incurs more than 10,000 messages in 5 dimensions (Fig. 14b). In contrast, the performance of our algorithms remains steady and is far better than DSL in larger dimensions, owing to its effective controlling and partitioning of the skyline search space. It is interesting to realize that when our algorithms are instantiated on CAN, they inherit a special property of CAN: the reduction of query search hops when the dimensionality of the space increases. This is due to the fact that, when the dimensionality increases, the routing path length on CAN decreases. Note that even though DSL is also implemented on CAN, it seems that this algorithm does not benefit from the reducing of query search hops.

5.3 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 1,024,00 to 1,638,400, in which the average data size per node increases from 100 to 1,600. Figure 15 witnesses that all systems are not very sensitive to data size, but our algorithms are more stable than DSL. Furthermore, except consumed bandwidth cost, we see all other costs taken by our systems decline a little bit in the larger data. 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. The consumed bandwidth cost does not decrease as other costs simply because with more data points, there are more skyline points, which need to be transferred in the search results.

5.4 Results on the real dataset

We now test the three systems on the real dataset of NBA players's season statistic from 1949 to 2003 distributed in a small network of 128 peers. For this dataset, we identify

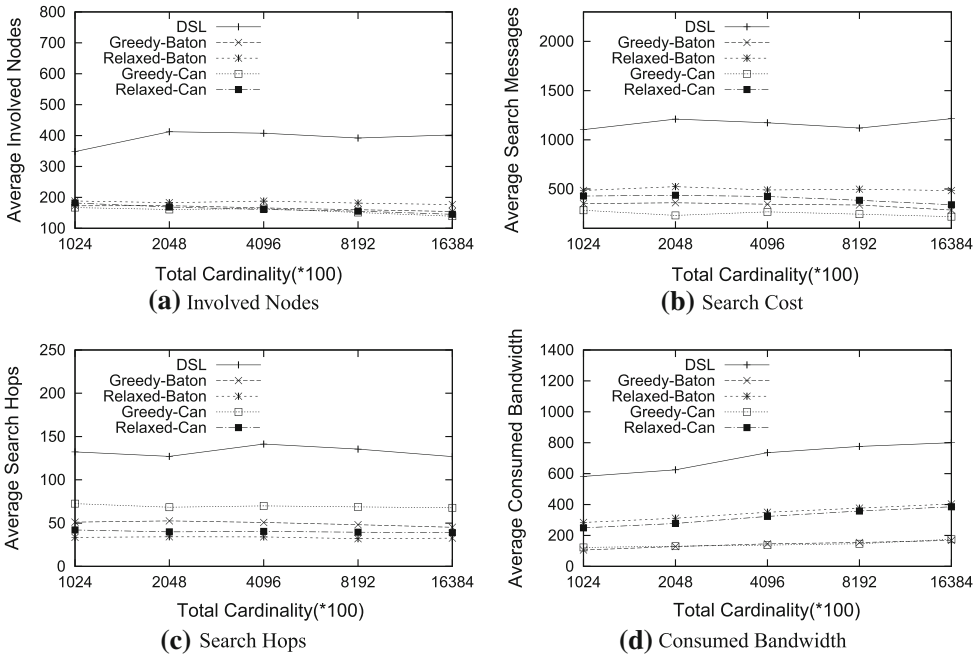


Fig. 15 Effect of anti-correlated data size

Table 2 Results on the real NBA dataset

Metrics	DSL	Greedy-Baton	Relaxed-Baton	Greedy-Can	Relaxed-Can
Average involved nodes	110	84	81	88	79
Average search messages	2491	272	313	265	297
Average search hops	78	51	41	45	36
Average bandwidth per node	204	13	23	14	25

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. The results confirm again that our algorithms are better than DSL in all aspects. Note that here we see the large portion of involved nodes in the result. This is caused by (1) the large dimensionality specified in skyline query and (2) 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.

5.5 Query load balancing

To evaluate the different effects of load balanced partitioning and the dynamic balancing discussed in Sect. 3.3, we compare our systems with both load balancing mechanisms enabled (Scheme-1) and dynamic balancing disabled (Scheme-2) to DSL on default settings. We only demonstrate results on the independent dataset because query load is easier to be balanced on the anti-correlated dataset. Furthermore, to make figures clear, we only show results of the GSS on BATON and RSS on CAN since the effects are similar to the others. The result is shown in Fig. 16b in which GB1, GB2, RC1, RC2, respectively, represent GSS on BATON using scheme 1 and 2 and RSS on CAN using scheme 1 and 2. The results show that query load in DSL has a larger variance than in any of the other systems. Moreover, some of its nodes hold a very small number of data points and have no query visiting. The advantage of our dynamic balancing process may not be clearly shown in this figure because the figure only displays the results at the end of the experiment. However, during the experiment, the advantage of the dynamic load balancing is shown by two factors. On the one hand, the sorted load distribution of nodes in the system with dynamic load balancing has not changed as much as that of nodes in the system without dynamic load balancing. On the other hand, the maximum imbalance load ratio, which is the load ratio between the heaviest loaded node and the lightest loaded node in the system, is much smaller in the system with dynamic load balancing than in the system without it.

5.6 Response time

We evaluate the progressiveness and the response time of our algorithms in real deployment using both independent and anti-correlated datasets. Figure 17 presents the returning time of executing a three-dimensional skyline queries in these two datasets. While the systems return all answers within 1

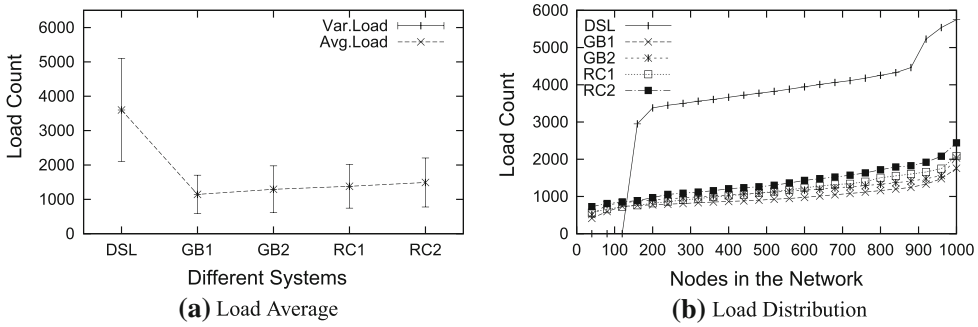


Fig. 16 Effect of load balancing on independent dataset

(a) Load Average **(b)** Load Distribution

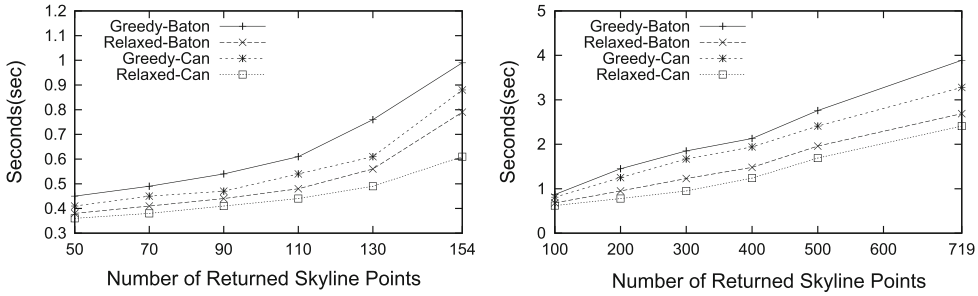


Fig. 17 Response time

(a) Real progressiveness on independent dataset **(b)** Real progressiveness on anti-correlated dataset

Table 3 Time average and variance

		Dimension = 2				Dimension = 3				Dimension = 4				Dimension = 5			
		GB	RB	GC	RC	GB	RB	GC	RC	GB	RB	GC	RC	GB	RB	GC	RC
D1	Average	0.43	0.33	0.51	0.35	0.58	0.40	0.53	0.36	0.63	0.51	0.52	0.36	1.04	0.80	0.54	0.40
	Variance	0.17	0.24	0.25	0.22	0.19	0.21	0.27	0.18	0.24	0.29	0.25	0.22	0.32	0.26	0.22	0.28
D2	Average	2.17	1.82	2.19	1.68	3.81	2.87	3.36	2.53	4.53	3.12	4.18	2.97	5.67	3.84	5.53	3.58
	Variance	0.52	0.47	0.72	0.62	0.48	0.69	0.63	0.71	0.93	0.73	0.81	0.64	1.67	1.08	1.25	0.97

second for the independent dataset, it takes a longer time (3–4 s) to return all answers for the anti-correlated dataset simply because this dataset has more skyline points than the previous dataset. In particular, the systems return the first 100 results at around 0.5 s (for the independent dataset) and 0.7 s (for the anti-correlated datasets). Table 3 further shows the total response time varying with dimensionality, in which the response time generally increases with the size of skyline in both datasets. In this table, D1 and D2 respectively represent the independent dataset and anti-correlated dataset; GB, RB, GC and RC respectively represent GSS on BATON, RSS on BATON, GSS on CAN and RSS on CAN; Time is measured in seconds.

6 Conclusion

In this paper, we have addressed the efficient processing of traditional centralized skyline querying on P2P networks by introducing Skyframe, a framework consisting of two methods for efficient and effective skyline query processing. The first method uses the high dominating point to prune the search space to achieve a low cost for network communication. The second method, on the other hand, processes skyline queries in parallel at the border nodes to increase the query processing speed. Besides the two query processing methods, we have also devised approaches for effective query load balancing in the framework. We have presented the instantiation of the Skyframe framework on both CAN [29] and BATON [20]. In particular, for the implementation using BATON we have proposed a method to map the multi-dimensional space into one-dimensional space. By numbering the data regions and keeping their split histories, we can efficiently infer the target node of the search region. The correctness and effectiveness of our proposed framework were formally proved and validated by experiments.

References

1. Aberer, K., Datta, A., Hauswirth, M.: Multifaceted simultaneous load balancing in dht-based p2p systems: a new game with old balls

and bins. In: Self-star Properties in Complex Information Systems, pp. 373–391 (2005)
2. Andrzejak, A., Xu, Z.: Scalable, efficient range queries for grid information services. In: Peer-to-Peer Computing, pp. 33–40 (2002)
3. Balke, W.T., Güntzer, U., Zheng, J.X.: Efficient distributed skylining for web information systems. In: Proceedings of EDBT Conference, pp. 256–273 (2004)
4. Bharambe, A.R., Agrawal, M., Seshan, S.: Mercury: supporting scalable multi-attribute range queries. In: Proceedings of SIGCOMM Conference, pp. 353–366 (2004)
5. Börzsönyi, S., Kossmann, D., Stocker, K.: The skyline operator. In: Proceedings of ICDE Conference, pp. 421–430 (2001)
6. Cai, M., Frank, M.R., Chen, J., Szekely, P.A.: Maan: A multi-attribute addressable network for grid information services. In: GRID, pp. 184–191 (2003)
7. Chan, C.Y., Eng, P.-K., Tan, K.-L.: Stratified computation of skylines with partially-ordered domains. In: Proceedings of SIGMOD Conference, pp. 203–214 (2005)
8. Chan, C.Y., Jagadish, H.V., Tan, K.-L., Tung, A.K.H., Zhang, Z.: On high dimensional skylines. In: Proceedings of EDBT Conference, pp. 478–495 (2006)
9. Chan, C.Y., Jagadish, H.V., Tan, K.-L., Tung, Anthony K.H., Zhang, Z.: Finding k-dominant skylines in high dimensional space. In: SIGMOD Conference, pp. 503–514 (2006)
10. Chawathe, Y., Ramabhadran, S., Ratnasamy, S., LaMarca, A., Shenker, S., Hellerstein, J.M.: A case study in building layered dht applications. In: SIGCOMM, pp. 97–108 (2005)
11. Chomicki, J., Godfrey, P., Gryz, J., Liang, D.: Skyline with presorting. In: Proceedings of ICDE Conference, pp. 717–816 (2003)
12. Dellis, E., Vlachou, A., Vladimirskiy, I., Seeger, B., Theodoridis, Y.: Constrained subspace skyline computation. In: CIKM, pp. 415–424 (2006)
13. Deng, K., Zhou, X., Shen, H.T.: Multi-source skyline query processing in road networks. In: Proceedings of ICDE Conference (2007)
14. Ganesan, P., Bawa, M., Garcia-Molina, H.: Online balancing of range-partitioned data with applications to peer-to-peer systems. In: Proceedings of VLDB Conference, pp. 444–455 (2004)
15. Ganesan, P., Yang, B., Garcia-Molina, H.: One torus to rule them all: Multidimensional queries in p2p systems. In: Proceedings of WebDB workshop, pp. 19–24 (2004)
16. Godfrey, P., Shipley, R., Gryz, J.: Maximal vector computation in large data sets. In: Proceedings of VLDB Conference, pp. 229–240 (2005)
17. Hose, K., Lemke, C., Sattler, K.-U.: Processing relaxed skylines in pdms using distributed data summaries. In: CIKM, pp. 425–434 (2006)
18. http://basketballreference.com
19. Huang, Z., Jensen, C.S., Lu, H., Ooi, B.C.: Skyline queries against mobile lightweight devices in manets. In: Proceedings of ICDE Conference (2006)
20. Jagadish, H.V., Ooi, B.C., Vu, Q.H.: Baton: A balanced tree structure for peer-to-peer networks. In: Proceedings of VLDB Conference, pp. 661–672 (2005)
21. Kossmann, D., Ramsak, F., Rost, S.: Shooting stars in the sky: An online algorithm for skyline queries. In: Proceedings of VLDB Conference, pp. 275–286 (2002)
22. Li, C., Tung, A.K.H., Jin, W., Ester, M.: On dominating your neighborhood profitably. In: VLDB, pp. 818–829 (2007)
23. Lin, X., Yuan, Y., Wang, W., Lu, H.: Stabbing the sky: efficient skyline computation over sliding windows. In: Proceedings of ICDE Conference, pp. 502–513 (2005)
24. Lin, X., Yuan, Y., Zhang, Q., Zhang, Y.: Selecting stars: The k most representative skyline operator. In: ICDE, pp. 86–95 (2007)
25. Papadias, D., Tao, Y., Fu, G., Seeger, B.: An optimal and progressive algorithm for skyline queries. In: Proceedings of SIGMOD Conference, pp. 467–478 (2003)
26. Pei, J., Jin, W., Ester, M., Tao, Y.: Catching the best views of skyline: A semantic approach based on decisive subspaces. In: Proceedings of VLDB Conference, pp. 253–264 (2005)
27. Pei, J., Fu, A.W.-C., Lin, X., Wang, H.: Computing compressed multidimensional skyline cubes efficiently. In: ICDE, pp. 96–105 (2007)
28. Pei, J., Jiang, B., Lin, X., Yuan, Y.: Probabilistic skylines on uncertain data. In: VLDB, pp. 15–26 (2007)
29. Ratnasamy, S., Francis, P., Handley, M., Karp, R.M., Shenker, S.: A scalable content-addressable network. In: Proceedings of SIGCOMM Conference, pp. 161–172 (2001)
30. Sharifzadeh, M., Shahabi, C.: The spatial skyline queries. In: VLDB, pp. 751–762 (2006)
31. Shu, Y., Tan, K.L., Zhou, A.: Adapting the content native space for load balanced indexing. In: Proceedings of DBISP2P Workshop, pp. 122–135 (2004)
32. Stoica, I., Morris, R., Karger, D.R., Kaashoek, M.F., Balakrishnan, H.: Chord: a scalable peer-to-peer lookup service for internet applications. In: Proceedings of SIGCOMM Conference, pp. 149–160 (2001)
33. Tan, K.L., Eng, P.K., Ooi, B.C.: Efficient progressive skyline computation. In: Proceedings of VLDB Conference, pp. 301–310 (2001)
34. Tao, Y., Xiao, X., Pei, J.: Subsky: efficient computation of skylines in subspaces. In: ICDE, 65 pp. (2006)
35. Vlachou, A., Doulkeridis, C., Kotidis, Y., Vazirgiannis, M.: Skypeer: efficient subspace skyline computation over distributed data. In: Proceedings of ICDE Conference (2007)
36. Wang, S., Ooi, B.C., Tung, A.K.H., Xu, L.: Efficient skyline query processing on peer-to-peer networks. In: Proceedings of ICDE Conference, pp. 1126–1135 (2007)
37. Wu, P., Zhang, C., Feng, Y., Zhao, B.Y., Agrawal, D., Abbadi, A.E.: Parallelizing skyline queries for scalable distribution. In: Proceedings of EDBT Conference, pp. 112–130 (2006)
38. Wu, P., Agrawal, D., Egecioglu, Ö., El Abbadi, A.: Deltasky: optimal maintenance of skyline deletions without exclusive dominance region generation. In: ICDE, pp. 486–495 (2007)
39. Xia, T., Zhang, D.: Refreshing the sky: the compressed skycube with efficient support for frequent updates. In: SIGMOD Conference, pp. 491–502 (2006)
40. Yuan, Y., Lin, X., Liu, Q., Wang, W., Yu, J.X., Zhang, Q.: Efficient computation of the skyline cube. In: Proceedings of VLDB Conference, pp. 241–252 (2005)
41. Zhang, C., Krishnamurthy, A., Wang, R.Y.: Skipindex: towards a scalable peer-to-peer index service for high dimensional data. Technical report, Princeton University (2004)