Q+Tree: An Efficient Quad Tree based Data Indexing for Parallelizing Dynamic and Reverse Skylines

Md. Saiful Islam#1, Chengfei Liu†2, Wenny Rahayu#3 and Tarique Anwar†*4
#La Trobe University, Melbourne, Australia
†Swinburne University of Technology, Melbourne, Australia
*Data61, CSIRO, Melbourne, Australia
{1m.islam5, 3w.rahayu}@latrobe.edu.au, {2cliu, 4tanwar}@swin.edu.au

ABSTRACT

Skyline queries play an important role in multi-criteria decision making applications of many areas. Given a dataset of objects, a skyline query retrieves data objects that are not dominated by any other data object in the dataset. Unlike standard skyline queries where the different aspects of data objects are compared directly, dynamic and reverse skyline queries adhere to the around-by semantics, which is realized by comparing the relative distances of the data objects w.r.t. a given query. Though, there are a number of works on parallelizing the standard skyline queries, only a few works are devoted to the parallel computation of dynamic and reverse skyline queries. This paper presents an efficient quad-tree based data indexing scheme, called Q+Tree, for parallelizing the computations of the dynamic and reverse skyline queries. We compare the performance of Q+Tree with an existing quad-tree based indexing scheme. We also present several optimization heuristics to improve the performance of both of the indexing schemes further. Experimentation with both real and synthetic datasets verifies the efficiency of the proposed indexing scheme and optimization heuristics.

Keywords

Quad Tree, Aggressive Partitioning, Dynamic Skyline, Reverse Skyline, Load Balancing and Parallel Computation.

1. INTRODUCTION

The skyline query has been first proposed by Börzsönyi et al. [2]. Since then, this query has received lots of attention among the community and is studied extensively in dominance based data retrieval ([12], [8], [16], [3], [10], [22], [21], [17] for survey). Given a dataset of objects P, the standard skyline query retrieves all data objects $p_1 \in P$ that are not dominated by any other data objects $p_2 \in P$. A data object p_1 dominates another data object p_2 *iff* it is as good as p_2 in every aspects of p_2, but better than p_2 in at least one aspect. Given P and a query object q, a dynamic skyline query [12] retrieves all data objects $p_1 \in P$ that are not *dynamically dominated* by another data object $p_2 \in P$ w.r.t. the query q. Unlike standard skyline queries where the aspects of p_1 is directly compared with the corresponding aspects of p_2 without considering any query object, the dynamic skyline query adheres to the around-by semantics where the absolute differences of the aspects of p_1 and the query q are compared with the corresponding absolute differences of the aspects of p_2 and the query object q in deciding the dominance between p_1 and p_2. Consider the dataset of products P given in Fig. 1(a), the standard skyline retrieves p_1 and p_2 from P (without considering any query) as no other objects in P dominate them. On the other hand, given a query $q =< 12, 12 >$, the dynamic skyline of q retrieves p_4, p_5 and p_6 from P as no other objects in P can dominate them w.r.t. q. Both the standard [2] and dynamic [12] skyline queries retrieve data objects from P considering the user's point of view, i.e., objects incomparably preferable to a user.

ID	Dim1	Dim2
p_1	6	6
p_2	4	18
p_3	6	20
p_4	9	15
p_5	12	18
p_6	16	14
p_7	14	6
p_8	16	6
p_9	20	8
p_{10}	20	20

(a) Products, P

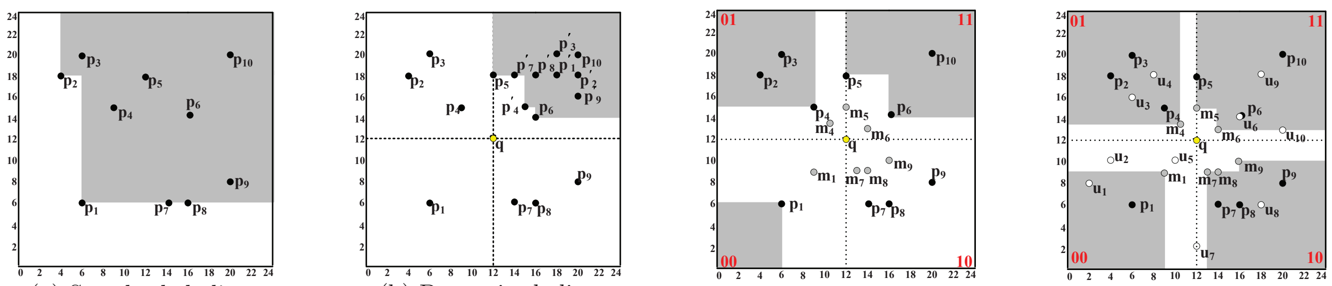
ID	Dim1	Dim2
u_1	2	8
u_2	4	10
u_3	6	16
u_4	8	18
u_5	10	10
u_6	16	14
u_7	12	2
u_8	18	6
u_9	18	18
u_{10}	20	13

(b) Users, U

Figure 1: A dataset of products and users

Dellis et al. [3] propose a new type of skyline query called, the reverse skyline query, which retrieves data objects from the database considering the manufacturer's point of view. Given a dataset of products P and a query q, the monochromatic reverse skyline query retrieves all products $p \in P$ that includes q in their dynamic skylines. Consider the dataset of products P given in Fig. 1(a) and a query $q =< 12, 12 >$, the monochromatic reverse skyline of q retrieves p_1, p_4, p_5 and p_6 as these objects include q in their dynamic skylines. Given datasets of products P, users U and a query q, a bichromatic reverse skyline [9] query retrieves all users $u \in U$ who find the query q in their dynamic skylines. Consider the dataset of products P and users U given in Fig. 1 and a query $q =< 12, 12 >$, the bichromatic reverse skyline of q retrieves u_2, u_5 and u_7 as they include q in their dynamic skylines. Like skyline queries [2][12], the reverse skyline queries also receive lots of attention in the community, specifically in influence-based processing of market research queries for measuring the attractiveness of a product among the users ([22], [4], [1], [20], [6], [5] for survey).

CIKM'16 , October 24-28, 2016, Indianapolis, IN, USA
© 2016 ACM. ISBN 978-1-4503-4073-1/16/10...$15.00
DOI: http://dx.doi.org/10.1145/2983323.2983764



(a) Standard skyline (b) Dynamic skyline (c) Monochromatic RSL (d) Bichromatic RSL

Figure 2: The (a) standard skyline (b) dynamic skyline (c) monochromatic reverse skyline and (d) bichromatic reverse skyline of the query $q = (12, 12)$

Due to the abundance of data in today's data intensive systems including dominance based data retrieval systems, there is a growing interest in parallelizing the skyline queries. Though, there are a number of works on parallelizing the standard skyline ([19], [24], [7] [11], [15] for survey), parallelizing dynamic and reverse skylines receives little attention among the community. The only work on parallelizing the dynamic and (monochromatic) reverse skylines based on quad-tree structure exists in [14] (probabilistic version [13]). The existing quad-tree based data indexing scheme, we call it QTree here, has performance bottleneck when there is a skewed data distribution. In addition of it, the optimal value of *split threshold* and ideal *sampling method* are hard to know for quad-tree based data indexing. This paper presents an advanced quad tree based data indexing scheme, called Q+Tree, which alleviates much of the aforementioned problems. We also present several optimization heuristics such as *aggressive partitioning* and *load balancing* to expedite the performance of the quad-tree based data indexing schemes for parallelizing the dynamic and reverse skyline queries. Our main contributions are as follows:

1. We present an efficient quad tree based data indexing scheme, called Q+Tree, for parallelizing the computations of dynamic skyline, monochromatic reverse skyline and bichromatic reverse skyline queries.

2. We present several optimization heuristics to expedite the performance of the quad-tree based indexing schemes for parallelizing all of the skyline queries.

3. We compare the efficiency of Q+Tree with an existing quad-tree based indexing scheme by conducting extensive experiments with both real and synthetic datasets.

The rest of the paper is organized as follows. Section 2 presents the preliminaries and the computing environment. Section 3 discusses the related work. Section 4 presents the quad tree based Q+Tree data indexing scheme for parallelizing the dynamic and reverse skyline queries. Section 5 presents the optimization heuristics proposed in this paper. Section 6 presents the experimental evaluation of all indexing schemes for processing the dynamic and reverse skyline queries in parallel. Finally, Section 7 concludes the paper.

2. BACKGROUND

This section provides the preliminaries, a background on the skyline queries and the computing enviorment.

2.1 Preliminaries

We assume that the dataset D consists of two different group of objects and these are: products P and users U. We consider each product $p \in P$, query q and user $u \in U$ as a d-dimensional data object. Without any loss of generality, we assume that each data object stores only numeric values in its dimensions. The i^{th} dimensional values of a product p, query q and user u are denoted by p^i, q^i and u^i, respectively. In general, we use o to denote any kind of data object in D.

(Dynamic) Dominance. A data object o_1 dominates another data object o_2, denoted by $o_1 \prec o_2$, iff: (a) $\forall i \in [1...d]$, ${o_1}^i \leq {o_2}^i$ and (b) $\exists j \in [1...d]$, ${o_1}^j < {o_2}^j$. On the other hand, a data object o_1 dynamically dominates another data object o_2 w.r.t. a third data object o_3, denoted by $o_1 \prec_{o_3} o_2$, iff: (a) $\forall i \in [1...d]$, $|{o_1}^i - {o_3}^i| \leq |{o_2}^i - {o_3}^i|$ and (b) $\exists j \in [1...d]$, $|{o_1}^j - {o_3}^j| < |{o_2}^j - {o_3}^j|$.

Consider the datasets of products P given in Fig. 1(a) and the query $q =< 12, 12 >$. The product p_2 dominates the product p_3, i.e., $p_2 \prec p_3$, as (a) ${p_2}^1(= 4) < {p_3}^1(= 6)$ and also, (b) ${p_2}^2(= 18) < {p_3}^2(= 20)$. Now, the product p_4 dynamically dominates product p_8 w.r.t. q, i.e., $p_4 \prec_q p_8$, as (a) $|{p_4}^1 - q^1|(= 3) < |{p_8}^1 - q^1|(= 4)$ and (b) $|{p_4}^2 - q^2|(= 3) < |{p_8}^2 - q^2|(= 6)$.

Orthants and Midpoints. Given an object o and a query q, the orthant O of o w.r.t. q, denoted by $O_q(o)$, is computed as: $O^i_q(o) = 0$ *iff* $o^i \leq q^i$, otherwise $O^i_q(o) = 1$. A d-dimensional query q has 2^d orthants in total, e.g., the orthants of $q =< 12, 12 >$ are shown in Fig 2(c)-(d).

The midpoint m of a product p w.r.t. a query q is computed as: $m^i = (p^i + q^i)/2$. For example, the midpoints of p_1, p_4, p_5, p_6, p_7, p_8 and p_9 are shown in Fig 2(c)-(d).

2.2 Skyline Queries

(Dynamic) Skyline. Given a dataset of products P, the standard skyline, denoted by SKL, retrieves all products $p_1 \in P$ that are not dominated by other products $p_2 \in P$. Given a dataset of products P and a query q, the dynamic skyline of q, denoted by $DSL(q)$, retrieves all $p_1 \in P$ that are not dynamically dominated by any $p_2 \in P$ w.r.t. q, i.e., $\nexists p_2 \in P : p_2 \prec_q p_1$. Consider the dataset of P given in Fig. 1(a) and the query $q =< 12, 12 >$. The SKL of P consists of p_1 and p_2, shown in Fig. 2(a), as no other products in P dominate them. The $DSL(q)$ consists of p_4, p_5 and p_6, shown in Fig. 2(b), as no other products in P dynamically dominate them w.r.t. q. The $DSL(q)$ can be computed inefficiently by any SKL algorithm [2] having all $p \in P$ transformed into a new space where q is treated as the origin and the relative distances to q are used as the mapping functions as shown in Fig. 2(b). The mapping

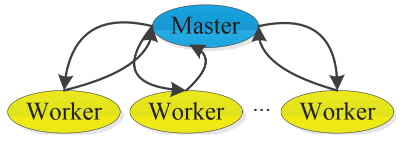


Figure 3: The simplified computing environment for parallelizing dynamic and reverse skylines

function f is defined as $f^i(p^i) = |p^i - q^i|$. The transformed p is denoted by p' in this paper.

Reverse Skyline. Given a dataset of products P and a query q, the monochromatic reverse skyline of q, denoted by $MRSL(q)$, retrieves all $p_1 \in P$ such that q is in the $DSL(p_1)$. Mathematically, the $MRSL(q)$ retrieves all $p_1 \in P$ such that $\nexists p_2 \in P$ and the following holds (a) $\forall i \in [1...d]$, $|p_2^i - p_1^i| \leq |q^i - p_1^i|$ and (b) $\exists i \in [1...d]$, $|p_2^i - p_1^i| < |q^i - p_1^i|$. On the other hand, given a dataset of products P, users U and a query q, the bichromatic reverse skyline of q, denoted by $BRSL(q)$, retrieves all $u \in U$ such that q is not dynamically dominated by any $p \in P$ w.r.t. u. Mathematically, the $BRSL(q)$ retrieves all $u \in U$ such that the following holds (a) $\forall i \in [1...d]$, $|q^i - u^i| \leq |p^i - u^i|$ and (b) $\exists i \in [1...d]$, $|q^i - u^i| < |p^i - u^i|$, $\forall p \in P$.

Consider the dataset of P and U given in Fig. 1 and the query $q =< 12, 12 >$. The $MRSL(q)$ consists of p_1, p_4, p_5 and p_6 as shown in Fig. 2(c) as q is in their DSLs. On the other hand, the $BRSL(q)$ consists of u_2, u_5 and u_7, shown in Fig.2(d), as no other $p \in P$ can dominate q w.r.t. them. We use $RSL(q)$ to denote any kind of reverse skylines of q.

Mid Skyline. The mid skyline of P w.r.t. q, denoted by $MSL(q)$, consists of all midpoints that are not dominated by any other midpoint w.r.t. q in the same orthant. For example, the MSL of $q =< 12, 12 >$ consists of m_1, m_4, m_5, m_6, m_7 and m_9 as they are not dominated by other midpoints w.r.t. q in the same orthant (see in Fig. 2(d)).

LEMMA 1. *A user $u \in U$ appears in $BRSL(q)$ iff $\nexists m \in MSL(q)$ such that (a) $O_q(u) = O_q(m)$ and (b) $m \prec_q u$.*

2.3 Computing Environment

We assume an oversimplified computing environment, as shown in Fig. 3, where a master processor is responsible for coordinating and managing the independent tasks carried out by the worker processors. The worker processor receives the necessary input data from the master and the task type, finish the task accordingly and finally, may send the processed result back to the master. The master processor may index and pre-process the input data before sending them to the workers. We assume that the communication and the synchronization between the master and the worker are integral parts of this environment. We also assume that the computing power of all workers are the same. This model can be simulated through Java Multithreading, MPIs and the state of the art MapReduce technology.

3. RELATED WORK

Parallelizing the Standard Skyline. There are a number of works on parallelizing the standard skyline. Vlachou et al.[19] exploit the hyperspherical coordinates of the data points to propose an angle-based space partitioning for parallelizing the standard skyline query processing. Zhang et al.[24] apply object-based space partitioning technique for processing skyline queries in parallel. Kohler et al.[7] present a hyperplane data projection technique, which is independent of the data distribution, for computing skyline in parallel. Mullesgaard et al.[11] presents a grid-based data partitioning scheme for computing the standard skyline in MapReduce. Pertesis and Doulkeridis [15] propose a novel technique for processing the skyline query in SpatialHadoop. Recently, Zhang et al.[23] propose a two-phase MapReduce approach for parallelizing the standard skyline query processing by applying filtering techniques and angle-based partitioning. None of these approaches are devoted to the parallel computation of dynamic and reverse skylines.

Table 1: The ratios of pruned areas in different data indexing schemes of the dataset given in Fig. 1

Data Indexing Scheme(s)	DSL	MRSL	BRSL
QTree	18.75%	14.06%	18.75%
Optimized (AVG) QTree	25.00%	25.17%	32.81%
Q+Tree	42.18%	28.13%	55.43%
Optimized (AVG) Q+Tree	48.44%	39.24%	69.49%

Parallelizing the Dynamic and Reverse Skylines. Though much of work are devoted to parallelizing the standard skyline, very few works exist on parallelizing the dynamic and reverse skylines [14], [13]. Park et al. [14] presents an approach for parallelizing the dynamic and reverse skylines based on quad-tree index (probabilistic version [13]). However, the quad-tree index is much dependent on the sampling method and the split threshold. The selection of optimum split threshold and the ideal sampling method are hard in reality. Also, the basic quad-tree based data indexing scheme QTree [14] has performance bottleneck if the underlying data distribution is skewed. Much of these problems of quad-tree based data indexing and the QTree [14] can be overcomed as evident from Table 1 (take these results granted until we reach Section 5), i.e., the ratios of pruned areas of the basic QTree are much less compared to the indexing scheme proposed in this paper.

Our work. We present an efficient quad-tree based data indexing scheme, called Q+Tree to alleviate much of the problems of the quad-tree based indexing schemes [14]. We also present several optimization heuristics for the quad-tree based indexing schemes to expedite the performance of computing dynamic and reverse skyline queries in parallel.

4. THE Q+TREE

The quad-tree indexing scheme for computing dynamic and reverse skylines is an indexing scheme in which the given data space D are recursively divided into 2^d partitions in each orthant of a query q, until the number of objects in it meets a certain threshold, ρ. The query q is the root and represents the whole data space D. Each internal node representing a partition has exactly 2^d children. The range of values covered by a node n is denoted by $region(n)$. Each node n is assigned an id consisting of $k \cdot d$ bits, i.e., $id(n) = a_1a_2...a_{k\cdot d}$, where k is the depth of n in the tree. The first $(k-1) \times d$ bits of $id(n)$ come from its parent node and the remaining d bits are $a_{(k-1)\cdot d+1}, a_{(k-1)\cdot d+2}, ..., a_{k\cdot d}$ where $a_{(k-1)\cdot d+i} = 0$ (or 1) if the i^{th} dimensional range of the $region(n)$ is the first half (or the second half) of its parent's i^{th} dimensional range. Only a subset of products of P (e.g., reservoir sampling [18]) is used to build the index.

Main Idea: As we discuss in Section 1 and Section 3, the limitation of the basic quad tree based indexing scheme QTree[14] is that its pruned regions are largely dependent on the sampling method and the setting of the split threshold, ρ. We may end up having different trees with different pruning capabilities for different samples and settings of ρ. Also, we know that the ideal sampling method and the optimum value of ρ are hard to predict. Much of this limitation can be mitigated by exploiting a property of the *pruned* nodes in the quad-tree. Hence, we present an advancement of the basic QTree, called Q+Tree, which extends the *pruned* node regions based on node dominances. Unlike the basic QTree [14], the regions of the children of an internal node in Q+Tree may not be equal in areas. The pseudocode of the quad-tree based data indexing schemes is given in Algorithm 1. The extension of *pruned* node regions and the node-dominances in Q+Tree for all skylines as well as their parallel computations are discussed in the following sections.

4.1 Q+Tree Index for DSL

To construct the quad tree index for $DSL(q)$, [14] applies the following: a node n_2 is marked as pruned *iff* $\exists n_1 \in QTree$ such that $\forall p_2 \in n_2$, $\exists p_1 \in n_1$ and $p_1 \prec_q p_2$. If such $p_1 \in n_1$ exists for n_2, we say p_1 dominates n_2 w.r.t. q and is denoted by $p_1 \prec_q n_2$. We also say n_1 dominates n_2 w.r.t. q and is denoted by $n_1 \prec_q n_2$. However, checking this pairwise node dominance $n_1 \prec_q n_2$ by checking pairwise object dominance [14] is inefficient. An alternative to decide the node dominance efficiently is given in the following lemma:

Lemma 2. *If $\exists p_1 \in n_1$ such that p_1 dynamically dominates all of the 2^d corners of n_2 w.r.t. q, then $p_1 \prec_q n_2$.*

Proof. We know that all $p_2 \in n_2$ is bounded by the 2^d corners of node n_2. Therefore, $p_1 \in n_1$ dominates any $p_2 \in n_2$ w.r.t. q iff p_1 dominates all of the 2^d corners of n_2 w.r.t. q, i.e., $p_1 \prec_q n_2$. Hence the lemma. □

The children of a *pruned* node are set to *null*.

DQTree: The QTree index for $DSL(q)$ [14], we call it DQTree, of the products P given in Fig. 1(a) and $q =< 12, 12 >$ with samples $\{p_1, p_2, p_4, p_6, p_7, p_9, p_{10}\}$ is shown in Fig. 4(a) by setting ρ to 1. The gray regions are *pruned* nodes. Here, node with *id* 0101 is marked as *pruned* as p_4 of node with *id* 0110 and p_6 of node with *id* 1100 dominate all of its four corners (marked by green circles) w.r.t. q.

DQ+Tree: To extend the region of a *pruned* leaf node n_2 in a DQTree further, denoted by $region+(n_2)$, we first gather all objects p that dynamically dominate the corners of n_2 w.r.t. q. Then, we insert these p in a min heap $\mathcal{H}_q$[1] and repeatedly retrieve the root p_1 until $p_1 \in region(parent(n_2))$ or $p'_1 \in region(parent(n_2))$. Finally, we readjust the regions of the children of $parent(n_2)$ considering p_1 (or p'_1 if not in the same orthant as of n_2) as the new splitting point. We call the above tree as DQ+Tree here. The DQ+Tree of the products P given in Fig. 1(a) with samples $\{p_1, p_2, p_4, p_6, p_7, p_9, p_{10}\}$ for $q =< 12, 12 >$ is shown in Fig. 4(b). The gray patterned regions are the new pruned areas. Here, we redistribute the regions of the children of the node with *id* 01 considering p_4 as the new splitting point. Similarly, we redistribute the regions of the children of the nodes with *id*s 10 and 11 considering p'_4 as the new splitting point.

The master constructs the DQ+Tree, which is shared by all workers. The node dominances of DQ+Tree is given in Algorithm 2. For DQTree, the lines 8-18 of Algorithm 2 are not executed and the *for* loop in line 6 can terminate as soon as an o_2 is found such that $o_2 \prec_q n_1$ and n_1 is *pruned*.

[1]To compare two objects for the min heap H_q, we use the euclidean distances of o_1 and o_2 to the given query q.

Algorithm 1: Q+Tree

```
Input  : query $q$, split threshold $\rho$, samples $\mathcal{S}$
Output: Q+Tree
1  begin
2      $root$ ← new Node("root", $q$, null); // (node type,
       center-of-split, parent node)
3      for $each\ o \in \mathcal{S}$ do
4          isSplitted←false; // a boolean flag
5          $root$ ←insert($o$, root, null); // start from root node
6          if $isSplitted$ then
7              testDominances(root); // check node dominances

8  insert($Object\ o,\ Node\ n,\ Node\ parent$)
9  begin
       // the current node is not a leaf node
10     if $n!{=}null\ and\ !n.getType().equals(\text{“leaf”})$ then
           // set query to compute child's index
11         o.setQuery($n$.getQuery());
12         index← $o$.getOrthant();
           // insert the point in the child node
13         n.children[index]← insert($o$, $n$.children[index], $n$);
       // the current node is a leaf node
14     else if $n!{=}null\ and\ n.getType().equals(\text{“leaf”})$ then
15         $n$.add($o$);
           // split the leaf node if exceeds the threshold
16         if $n.size() > \rho\ and\ !n.isPruned()$ then
17             $n$ ← split($n$);        // only leaf node is splitted
18             isSplitted←true; // triggers node dominance test
19     return $n$;

20 split($Node\ n_1$)
21 begin
22     Object center←getCenter($n_1$.getRanges());
23     $n_2$ ← new Node("internal", center, $n_1$.getParent());
24     $n_2$.setRanges($n_1$.getRanges());
       $n_2$.setOrthant($n_1$.getOrthant());
25     Object $objects$[] ← $n_1$.getPoints();
26     Node $children$[] = $n_2$.getChildren();
       // redistribute the objects among the children
27     for $each\ o \in objects$ do
28         o.setQuery(center);
29         children[$p$.getOrthant()].add($o$);
30     return $n_2$;
```

4.1.1 DSL in Parallel with DQ+Tree

The steps of computing $DSL(q)$ in parallel are as follows:

-(1) Firstly, the master divides P into several chunks $P_j \subset P$ (such that $\cup P_j = P$) and then sends these chunks P_j as well as the DQ+Tree to its workers.

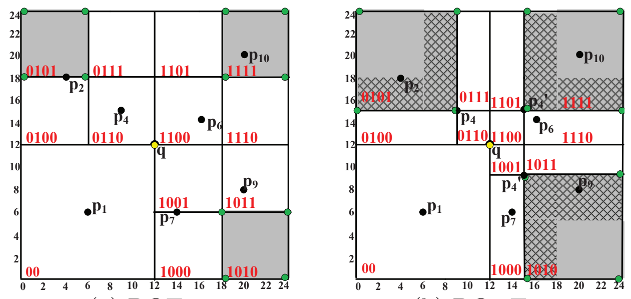
-(2) A worker does the followings: (a) $\forall p \in P_j$ finds its node *id* in the DQ+Tree, if it is from the *pruned* node, then it is ignored, otherwise, it is inserted into the min heap $\mathcal{H}^j_q$; (b) initializes DSL_j to $\emptyset$, repeatedly retrieves the root product p_1 from $\mathcal{H}^j_q$ and adds it to DSL_j iff $\nexists p_2 \in DSL_j$ such that $p_2 \prec_q p_1$; and (c) sends the local DSL_j to master.

-(3) Finally, the master collects all local DSL_js and insert them into the min heap $\mathcal{H}_q$. Then, the master computes the global $DSL(q)$ by following the same technique as given in step 2(b) for the worker.

Unlike [14], we do not transform the objects $p \in P$ into a new space w.r.t. the given query q, we establish the node dominances and compute $DSL(q)$ in the original data space and thereby, scan the product dataset P only once. The standard skyline SKL can also be computed in parallel using the DQ+Tree considering the given query q at *zero*.

4.1.2 Correctness of DQ+Tree

The following lemma proves the correctness of $DSL(q)$ computed using the DQ+Tree.



(a) DQTree (b) DQ+Tree

Figure 4: The quad tree indices for $DSL(q)$ of the dataset given in Fig. 1(a) and $q =< 12, 12 >$

Algorithm 2: testDominancesDQ+Tree

```
Input : Node root
begin
    $\mathcal{N}$ ←retrieveNodes(root);
    for each $n_1 \in \mathcal{N}$ do
        if $n_1$.isPruned() then
            continue; // already pruned
        // compute the objects that dominate node $n_1$
        for each $n_2 \in \mathcal{N}$ do
            $\mathcal{S}$ ←computeDominatingObjects($n_1$, $n_2$);
        // dominating objects are found for $n_1$
        if $\mathcal{S}$.size()!=0 then
            $n_1$.pruned ← true; // change node status
            $\mathcal{H}_q$ ←insert($\mathcal{S}$); // initialize the min-heap
            Node parent← $n_1$.getParent();
            while !$\mathcal{H}_q$.isEmpty() do
                center ← $\mathcal{H}_q$.retrieveRoot(); // root entry
                center′ ←mirror(center, root.getQuery());
                // extend regions with center or center′
                if parent.isCovered(center) then
                    extendRegions(parent, center);
                else if parent.isCovered(mirroredCenter) then
                    extendRegions(parent, center′);
```

LEMMA 3. *Assume that n is a pruned node in DQ+Tree and $p \in region+(n)$. Then, $p \notin DSL(q)$.*

PROOF. According to the construction of DQTree, $\exists p_1 \in n_1$ such that p_1 dominates all 2^d corners of $region(n)$ w.r.t. q (Lemma 2). Now, we insert all these p_1 into $\mathcal{H}_q$ and select the p_1 that has the least distance to q and $p_1 \in region(parent(n))$ (or $p_1' \in region(parent(n))$). We get $region+(n)$ in DQ+Tree by redistributing the regions of the children of $parent(n)$ including n considering p_1 (or p_1') as the new splitting point in DQ+Tree. Since p_1 dominates all 2^d corners of the $region+(n)$ w.r.t. q and p is bounded within $region+(n)$, we get $p_1 \prec_q p$. Hence, the Lemma. □

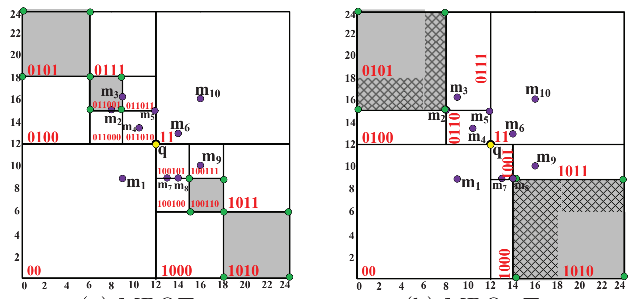
4.2 Q+Tree Index for MRSL

The *pruned* regions in the quad tree index for computing the $MRSL(q)$ are established as per the following [14].

LEMMA 4. *A node n_2 is marked as pruned iff $\exists m_1, m_2$ such that (a) $O_q(m_1) = O_q(n_2)$; (b) $O_q(m_2) = O_q(n_2)$; (c) $m_1 \prec_q n_2$ and (d) $m_2 \prec_q n_2$.*

The children of a *pruned* node are set to *null*.

MRQTree: The basic quad tree index for computing $MRSL(q)$, we call it MRQTree here, of the dataset P given in Fig. 1(a) for the midpoints of the samples $\{p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8, p_9, p_{10}\}$ and the query $q =< 12, 12 >$ is shown in Fig. 5(a) by setting $\rho = 2$ ($\rho \geq 2$ as per Lemma 4).



(a) MRQTree (b) MRQ+Tree

Figure 5: The quad tree indices for $MRSL(q)$ of the example dataset given in Fig. 1(a) and $q =< 12, 12 >$

MRQ+Tree: To extend the *pruned* region of a *leaf* node n_2 further, denoted by $region+(n_2)$, firstly we gather all midpoints m that dynamically dominate the corners of n_2 w.r.t. q in the same orthant as of n_2. Then, we insert these m in a min heap $\mathcal{H}_q$ and retrieve top-2 midpoints m_1 and m_2 such that $m_1 \in region(parent(n_2))$ and also, $m_2 \in region(parent(n_2))$. Finally, we readjust the regions of the children of $parent(n_2)$ by considering $m_{1,2}^{min}$[2] as the new splitting point. We call it MRQ+Tree here. The MRQ+Tree of the dataset P given in Fig. 1(a) for the midpoints of the samples $\{p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8, p_9, p_{10}\}$ and $q =< 12, 12 >$ is shown in Fig. 5(b). The gray patterned regions are the new pruned areas. Here, we redistribute the regions of the children of the node with id 01 considering $m_{2,4}^{min}$ as the new splitting point. Similarly, we redistribute the regions of the children of the node with id 10 considering $m_{7,8}^{min}$ as the new splitting points, respectively.

The master constructs the MRQ+Tree, which is shared by all workers. The node dominances of MRQ+Tree is given in Algorithm 3. For MRQTree, the lines 8-21 of Algorithm 3 are not executed and the *for* loop in line 6 can terminate as soon as two objects o_1 and o_2 is found in the same orthant as of n_1 such that $o_1 \prec_q n_1$ and $o_2 \prec_q n_1$, finally n_1 is *pruned*.

4.2.1 MRSL in Parallel with MRQ+Tree

The parallel steps of computing $MRSL(q)$ with MRQ+Tree are listed as follows:

-(1) Firstly, the master divides P into several chunks $P_j \subset P$ (such that $\cup P_j = P$) and then sends these chunks P_j as well as the MRQ+Tree to its workers.

-(2) A worker does the followings: (a) construct $\mathcal{X}_j = \bigcup p \cup m$, $\forall p \in P_j$; (b) $\forall o \in \mathcal{X}_j$ finds its node *id* in the MRQ+Tree, if it is from the *pruned* node, then it is ignored, otherwise, it is inserted into the min heap $\mathcal{H}_q^j$; (c) initializes $MRSL_j$ to $\emptyset$, repeatedly retrieves the root o_1 from $\mathcal{H}_q^j$ and adds it to $MRSL_j$ iff $\nexists m_2 \in MRSL_j$ such that $O_q(o_1) = O_q(m_2)$ and $m_2 \prec_q o_1$; and (d) sends $MRSL_j$ to the master.

-(3) Finally, the master does the followings: (i) collects all local $MRSL_j$s from its workers and inserts them into the min heap H_q; and (ii) computes the $MRSL(q)$ by following the same technique as given in step 2(c) for the worker. Only $p \in MRSL(q)$ are reported as the MRSL of the query q.

4.2.2 Correctness of MRQ+Tree

The following lemma proves the correctness of $MRSL(q)$ computed using the MRQ+Tree.

[2]A point that is dominated by both m_1 and m_2 w.r.t. q. The coordinates of $m_{1,2}^{min}$ come from m_1 and/or m_2.

Algorithm 3: testDominancesMRQ+Tree

```
   Input : Node root
1  begin
2      $\mathcal{N}$ ←retrieveNodes(root);
3      for each $n_1 \in \mathcal{N}$ do
4          if $n_1$.isPruned() then
5              continue; // already pruned
6          for each $n_2 \in \mathcal{N}$ do
7              $\mathcal{S}$ ←computeDominatingObjects($n_1$, $n_2$);
8          if $\mathcal{S}$.size()≥ 2 then
9              $n_1$.pruned ← true; // change node status
10             $\mathcal{H}_q$ ←insert($\mathcal{S}$);
11             Node parent ← $n_1$.getParent();
12             while !$\mathcal{H}_q$.isEmpty() do
13                 $o_1$ ← $\mathcal{H}_q$.retrieveRoot();
14                 if parent.isCovered($o_1$) then
15                     break; // parent region contains $o_1$
16             while !$\mathcal{H}_q$.isEmpty() do
17                 $o_2$ ← $\mathcal{H}_q$.retrieveRoot();
18                 if parent.isCovered($o_2$) then
19                     break; // parent region contains $o_2$
20             center ←computeVirtualMin($o_1$, $o_2$);
21             extendRegions(parent, center);
```

LEMMA 5. *Assume that n is a pruned node in MRQ+Tree and $p \in region+(n)$. Then, $p \notin MRSL(q)$.*

PROOF. According to the construction of MRQTree, $\exists m_1$, m_2 in the same orthant as of n such that m_1 and m_2 dominate all 2^d corners of $region(n)$ w.r.t. q (Lemma 4). Now, we select the pair m_1 and m_2 that has the least Euclidean distance to q by inserting them into $\mathcal{H}_q$ and $m_1 \in region(parent(n))$, $m_2 \in region(parent(n))$. We get $region+(n)$ by redistributing the regions of the children of $parent(n)$ including the node n considering $m_{1,2}^{min}$ as the new splitting point. Since $m_{1,2}^{min}$ dominates all 2^d corners of $region+(n)$ w.r.t. q and p is bounded within $region+(n)$, we get $p_1 \prec_p q$ or $p_2 \prec_p q$, i.e., $q \notin DSL(p)$. Hence, the Lemma. □

4.3 Q+Tree Index for BRSL

The pruned regions in the quad tree index for computing the $BRSL(q)$ are established as per the following lemma:

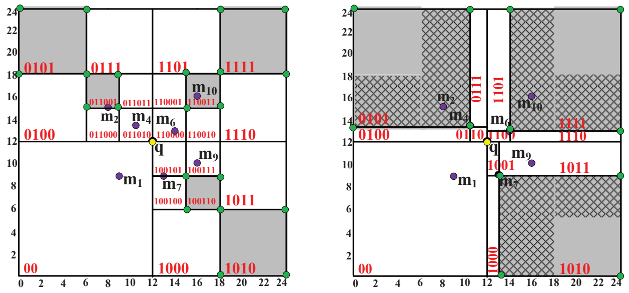
LEMMA 6. *A node n_2 is marked as* pruned *iff $\exists m_1$ such that (a) $O_q(m_1) = O_q(n_2)$ and (b) $m_1 \prec_q n_2$.*

PROOF. We know that if $\exists m_1$ such that $m_1 \prec_q n_2$ (condition (b)), then m_1 dominates all 2^d corners of node n_2. As all users $u \in n_2$ is bounded by its 2^d corners, $m_1 \prec_q u$, $\forall u \in n_2$. Since, m_1 is also in the same orthant as of n_2 w.r.t. q (condition (a)), any user $u \in n_2$ cannot be in $BRSL(q)$. Hence the lemma. □

The children of a *pruned* node are set to *null*.

BRQTree: The basic QTree index for computing $BRSL(q)$, we call it BRQTree, of the dataset P given in Fig. 1(a) for the midpoints of the samples $\{p_1, p_2, p_4, p_6, p_7, p_9, p_{10}\}$ and query $q =< 12, 12 >$ is shown in Fig. 6(a) by setting $\rho = 1$.

BRQ+Tree: To extend the *pruned* region of a *leaf* node n_2, denoted by $region+(n_2)$, firstly we gather all midpoints m that dynamically dominate the corners of a node n_2 w.r.t. q in the same orthant as of n_2. Then, we insert these m into the min heap $\mathcal{H}_q$ and retrieve the root m_1 such that $m_1 \in region(parent(n_2))$. Finally, we readjust the regions of the children of $parent(n_2)$ by considering m_1 as the new splitting point. We call it BRQ+Tree here. The BRQ+Tree of the dataset P given in Fig. 1(a) with midpoints of the samples $\{p_1, p_2, p_4, p_6, p_7, p_9, p_{10}\}$ for $q =< 12, 12 >$ is shown in Fig. 6(b). The regions with gray patterns are the new pruned areas. Here, we redistribute the regions of the children of the node with id 01 considering m_4 as the new splitting point. Similarly, we redistribute the regions of the children of the nodes with ids 10 and 11 considering m_7 and m_6 as the new splitting points, respectively.



(a) BRQTree (b) BRQ+Tree

Figure 6: The quad tree indices for $BRSL(q)$ of the dataset given in Fig. 1 and $q =< 12, 12 >$

Algorithm 4: testDominancesBRQ+Tree

```
   Input : Node root
1  begin
2      $\mathcal{N}$ ←retrieveNodes(root);
3      for each $n_1 \in \mathcal{N}$ do
4          if $n_1$.isPruned() then
5              continue; // already pruned
6          for each $n_2 \in \mathcal{N}$ do
7              $\mathcal{S}$ ←computeDominatingObjects($n_1$, $n_2$);
8          if $\mathcal{S}$.size()!=0 then
9              $n_1$.pruned ← true; // change node status
10             $\mathcal{H}_q$ ←insert($\mathcal{P}$);
11             Node parent← $n_1$.getParent();
12             while !$\mathcal{H}_q$.isEmpty() do
13                 center ← $\mathcal{H}_q$.retrieveRoot();
14                 if parent.isCovered(center) then
15                     extendRegions(parent, center);
```

The master constructs the BRQ+Tree, which is shared by all workers. The node dominances of BRQ+Tree is given in Algorithm 4. For BRQTree, the lines 8-15 of Algorithm 3 are not executed and the *for* loop in line 6 can terminate as soon as an object o_2 is found in the same orthant as of n_1 such that $o_2 \prec_q n_1$ and n_1 is *pruned*.

4.3.1 BRSL in Parallel with BRQ+Tree

The parallel steps of computing $BRSL(q)$ with BRQ+Tree in two-rounds are listed as follows:

-(1) In the first round, the master divides P into several chunks $P_j \subset P$ (such that $\cup P_j = P$) and sends these chunks P_j as well as the BRQ+Tree to its workers.

-(2) A worker does the followings: (a) $\forall p \in P_j$ convert p to its midpoint m, finds the node *id* of m in the BRQ+Tree, if it is from the *pruned* node, then it is ignored, otherwise, it is inserted into the min heap $\mathcal{H}_q^j$; (b) initializes MSL_j to $\emptyset$, repeatedly retrieves the root m_1 from $\mathcal{H}_q^j$ and adds it to MSL_j iff $\nexists m_2 \in MSL_j$ such that $O_q(m_1) = O_q(m_2)$ and $m_2 \prec_q m_1$; and (c) sends the MSL_j to the master.

-(3) Then, the master does the followings: (i) collects all MSL_js from its workers and insert them into a min heap

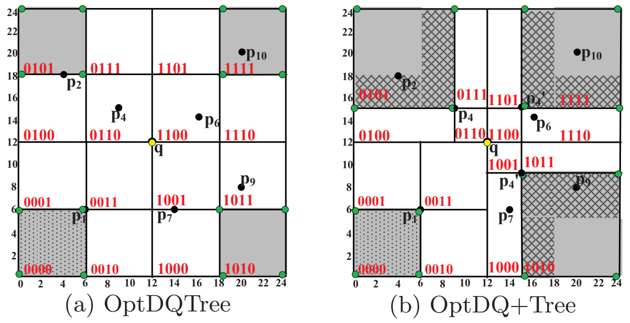


Figure 7: The optimal quad tree indices for $DSL(q)$ of the dataset given in Fig. 1(a) and $q =< 12, 12 >$

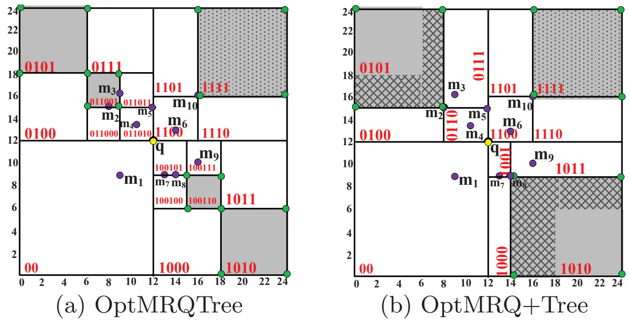


Figure 8: The optimal quad tree indices for $MRSL(q)$ of the dataset given in Fig. 1(a) and $q =< 12, 12 >$

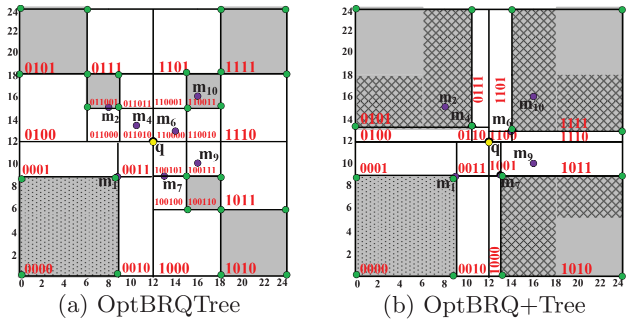


Figure 9: The optimal quad tree indices for $BRSL(q)$ of the dataset given in Fig. 1(a) and $q =< 12, 12 >$

$\mathcal{H}_q$; and (ii) computes the $MSL(q)$ by following the same technique as given in step 2(b) for the worker.

-(4) In the second round, the master divides U into several chunks $U_j \subset U$ (such that $\cup U_j = U$) and then sends these chunks U_j and the $MSL(q)$ to its workers.

-(5) A worker does the followings: (a) $\forall u \in U_j$ finds its node *id* in the BRQ+Tree, if it is from the *pruned* node, then it is ignored, otherwise, it is inserted into the local $BRSL_j$ iff $\nexists m \in MSL(q)$ such that $O_q(u) = O_q(m)$ and $m \prec_q u$; and (b) sends the $BRSL_j$ to the master.

-(6) As a final step, the master collects all local $BRSL_j$s from its workers into the global $BRSL(q)$.

4.3.2 Correctness of BRQ+Tree

The following lemma proves the correctness of $BRSL(q)$ computed using the BRQ+Tree.

LEMMA 7. *Assume that* n *is a pruned node in BRQ+Tree and* $u \in region+(n)$. *Then,* $u \notin BRSL(q)$.

PROOF. According to the construction of BRQTree, $\exists m_1$ in the same orthant as of n such that m_1 dominates all 2^d corners of $region(n)$ w.r.t. q (Lemma 6). Now, we insert all these m_1 into $\mathcal{H}_q$ and retrieve the m_1 that has the least Euclidean distance to q and $m_1 \in region(parent(n_2))$. We get $region+(n)$ in BRQ+Tree by redistributing the regions of the children of $parent(n)$ including the node n considering m_1 as the new splitting point. Since u is bounded within $region+(n)$ and m_1 dominates all 2^d corners of $region+(n)$, we get $m_1 \prec_q u$, i.e., $q \notin DSL(u)$. Hence, the Lemma. $\square$

5. OPTIMIZATION HEURISTICS

This section presents index specific heuristics to improve the performance of all of the aforementioned indexing schemes.

5.1 Aggressive Partitioning

In QTree and Q+Tree indexing schemes, we stop partitioning a node if the number of samples in it is below the threshold, ρ. However, the sample objects selected by the adopted sampling method may not represent the node space well. Therefore, the basic stopping criteria may not prune sufficient number of objects to expedite the parallelization of dynamic and reverse skyline queries. To overcome this limitation, we apply the following heuristic to repartition a *non-empty* and *unpruned* node n aggressively as given as: if $\Delta(n) > \delta$, where $\Delta(n)$ denotes the *area* of n. For simplicity, we propose to repartition a *unpruned* node n only once. For each QTree/Q+Tree, the repartitioning is conducted as:

-**DQTree/DQ+Tree**: We insert all p of the node n into a min-heap $\mathcal{H}_q$, retrieve the root p_1 from $\mathcal{H}_q$ and then, partition n considering p_1 as the center of split. The child node of n dominated by p_1 w.r.t. q is *pruned*. The optimal DQTree and DQ+Tree of the dataset in Fig. 1(a) and query $q =< 12, 12 >$ with sample products $\{p_1, p_2, p_4, p_6, p_7, p_9, p_{10}\}$ is shown in Fig. 7.

-**MRQTree/MRQ+Tree**: We insert all m of n into a min-heap $\mathcal{H}_q$, retrieve the top-2 midpoint objects m_1 and m_2 from $\mathcal{H}_q$ and then, partition n considering $m_{1,2}^{min}$ as the center of split. The child node of n dominated by $m_{1,2}^{min}$ w.r.t. q is *pruned*. The optimal MRQTree and MRQ+Tree of the example dataset given in Fig. 1(a) and query $q =< 12, 12 >$ with the midpoints of the samples $\{p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8, p_9, p_{10}\}$ is shown in Fig. 8.

-**BRQTree/BRQ+Tree**: We insert all m of n into a min-heap $\mathcal{H}_q$, retrieve the root m_1 from $\mathcal{H}_q$ and then, partition n considering m_1 as the center of split. The child node of n dominated by m_1 w.r.t. q is *pruned*. The optimal BRQTree and BRQ+Tree of the example dataset given in Fig. 1(a) and query $q =< 12, 12 >$ with midpoints of the samples $\{p_1, p_2, p_4, p_6, p_7, p_9, p_{10}\}$ is shown in Fig. 9.

To apply aggressive partitioning on node n, we need at least one sample object $p \in n$ for DQTree/DQ+Tree, two midpoints $m_1 \in n$ and $m_2 \in n$ for MRQTree/MRQ+Tree and one midpoint $m \in n$ for BRQTree/BRQ+Tree. Now, we propose two different heuristics for δ as follows: (a) $AVG\{\Delta(n)|n.pruned = \text{“true”}, \forall n \in \text{QTree/Q+Tree}\}$ and (b) $MIN\{\Delta(n)|n.pruned = \text{“true”}, \forall n \in \text{QTree/Q+Tree}\}$.

The MIN heuristic assumes that an unpruned node is not represented well by the sampling method if its area is greater than the area of a pruned node and is suitable for uniformly distributed data space. On the other hand, the AVG heuristic assumes that an unpruned node is not represented well by the sampling method if its area is greater than the areas of all pruned nodes in average and is suitable for skewed data distribution in correlated and anticorrelated data space. The Table 1 shows the ratio of pruned areas of different quad-tree

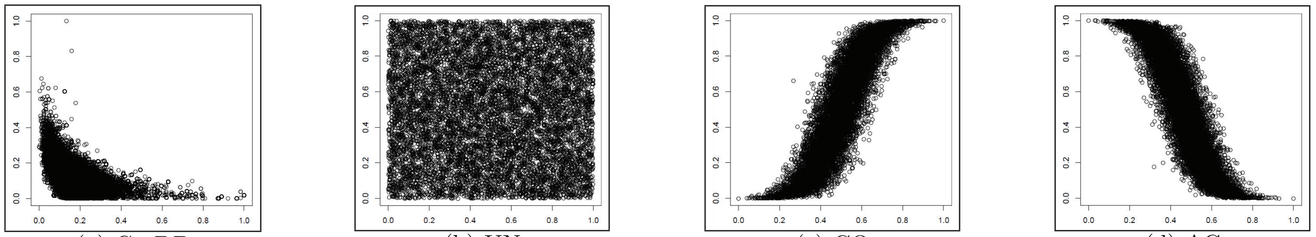

(a) CarDB (b) UN (c) CO (d) AC

Figure 10: Data distribution in tested two-dimensional real CarDB and synthetic UN, CO and AC datasets

Table 2: Settings of parameters

Parameter	**Values**
Tested Datasets	Real (CarDB), Synthetic (UN, CO, AC)
Data Cardinality	100 Thousands $\sim$ 10 Millions
Dimensionality	$2 \sim 6$
No. of Threads	$2 \sim 15$ (1 thread per processor)
No. of Samples	$200 \sim 1000$ Objects
Split Threshold	$20 \sim 50$ Objects

based data indexing schemes for parallelizing the dynamic and reverse skylines of the example dataset given in Fig. 1(a) and the query $q =< 12, 12 >$, where we set δ to AVG.

5.2 Load Balancing

Consider the final round of dynamic skyline computation carried out by the master processor. Assume that the size of the accumulated local dynamic skyline objects in the master is T. Now, the master needs to perform $\mathcal{O}(T^2)$ pairwise dominance checkings in the worst case to finalize the global dynamic skyline. This worst-case time complexity may dominate the overall efficiency. To mitigate this performance bottleneck, we propose to parallelize the dominance checkings among the workers until the size of the accumulated local dynamic skyline objects in the master becomes below a threshold τ. For reverse skyline queries, we also propose to parallelize the midpoint skyline objects for each orthant.

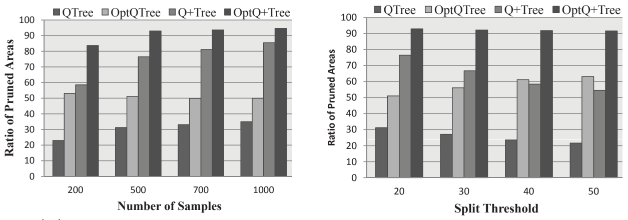
6. EXPERIMENTS

Here, we compare the efficiencies of different quad-tree based data indexing schemes: QTree, OptQTree, Q+Tree and OptQ+Tree for parallelizing dynamic skyline, monochromatic reverse skyline and bichromatic reverse skyline queries.

6.1 Datasets, Queries and Environment

Datasets: We evaluate the performance of all indexing schemes for parallelizing the dynamic and reverse skylines using real data, namely CarDB[3], consisting of 2×10^5 car objects. This is a six-dimensional dataset with attributes: *make*, *model*, *year*, *price*, *mileage* and *location*. We consider only the three numerical attributes *year*, *price* and *mileage* in our experiments. The dataset is also normalized into the range $[0, 1]$. We randomly select half of the car objects as products and the rest as the user data for bichromatic reverse skyline. We also present experimental results based on synthetic data: uniform (UN), correlated (CO) and anti-correlated (AC), consisting of varying number of products, users and dimensions. The cardinalities of these datasets in products and users are 100 thousands (K) $\sim$10 millions (M). The dimensionality (d) varies in 2$\sim$6. The data distributions of the above tested datasets for $d = 2$ are shown in Fig. 10.

Test (Skyline) Queries: For all experiments, we run a



(a) Effect of Samples (b) Effect of ρ

Figure 11: Effect of (a) samples and (b) ρ on pruned areas for dynamic skyline query in indexing schemes

number of queries generated (synthetic) and selected (CarDB) randomly by following the distribution of the tested datasets.

Computing Environment: We execute all of our algorithms in Swinburne HPC system [4] with 2$\sim$15 processors and 4GB main memory. The master-worker is simulated with Java multi-threading. Table 2 summarizes the values of different parameters used in our experimentation.

6.2 Data Indexing Evaluation

This section evaluates all indexing schemes in terms of the ratios of *pruned* areas. Firstly, we build data indices for 100 skyline queries (queries follow the distribution of the dataset) using 500 samples for each dataset, where we set product cardinality $|P|$ to 100K, dimensions d to 2 and split threshold ρ to 20. Table 3 shows the average ratios of the *pruned* areas in different data indexing schemes. It is evident from Table 3 that the proposed Q+Tree outperforms the basic QTree indexing scheme in terms of the ratios of *pruned* areas. Also, the proposed *aggressive pruning* heuristic optimizes the pruned areas for both QTree and Q+Tree indexing schemes. Secondly, we conduct two experiments using the same data settings for DSL queries: (a) varying #samples $= 200 \sim 1000$ with $\rho = 20$ and (b) varying ρ=20 $\sim$ 50 with 500 samples in CarDB dataset. We set δ to AVG for both experiments. The results are shown in Fig. 11. It is evident from Fig. 11 that both #samples and ρ play an important role in quad-tree based data indexing schemes. We see that Q+ Tree is less susceptible to the settings of #samples and ρ than the QTree. The optimized Q+Tree and QTree data indexing schemes are mostly tolerant to the settings of them.

6.3 Efficiency Study Results

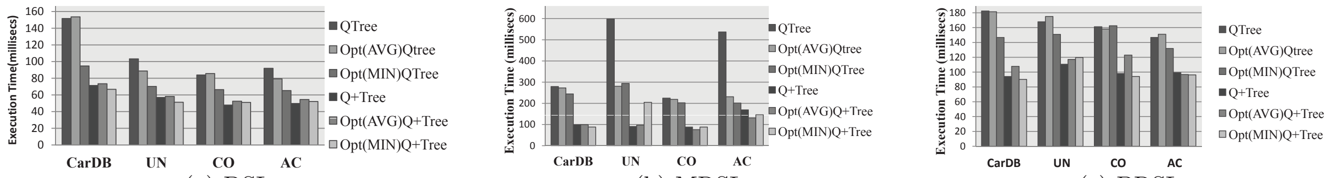
Here, we present the efficiency study results of all indexing schemes for computing the DSL, MRSL and BRSL queries in parallel. Firstly, we experiment for 100 skyline queries using 500 samples for each dataset, where we set product cardinality $|P|$ and user cardinality $|U|$ to 100K, dimensions d to 2, split threshold ρ to 20 and threads to 5. We also set τ to 100 for DSL queries. Fig. 12 shows the average of the

[3]https://autos.yahoo.com/

[4]http://www.astronomy.swin.edu.au/supercomputing/

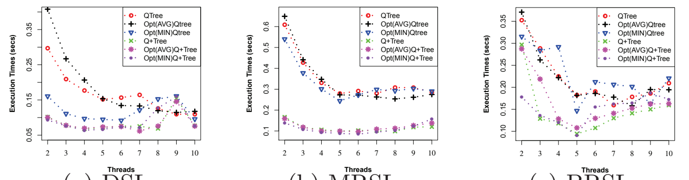
Table 3: Ratios of pruned areas in different quad-tree based data indexing schemes on different datasets

Scheme(s)	CarDB			Uniform (UN)			Correlated (CO)			Anticorrelated (AC)		
	DSL	MRSL	BRSL	DSL	MRSL	BRSL	DSL	MRSL	BRSL	DSL	MRSL	BRSL
QTree	31.24%	27.60%	27.62%	56.72%	38.89%	38.89%	38.95%	29.80%	29.83%	38.56%	29.42%	29.49%
Opt(AVG)QTree	51.06%	43.08%	45.38%	67.96%	39.42%	39.49%	57.35%	41.52%	43.42%	54.34%	38.14%	41.87%
Opt(MIN)QTree	54.74%	44.26%	47.10%	82.24%	44.59%	46.49%	52.65%	35.01%	36.81%	49.98%	34.03%	35.59%
Q+Tree	76.48%	76.68%	77.48%	90.34%	89.59%	92.77%	73.49%	72.74%	75.33%	70.40%	71.08%	73.25%
Opt(AVG)Q+Tree	92.95%	90.97%	93.83%	90.38%	89.69%	92.77%	85.38%	84.22%	87.86%	82.36%	79.65%	85.13%
Opt(MIN)Q+Tree	95.18%	92.77%	96.04%	94.09%	94.05%	96.81%	77.80%	75.43%	79.21%	84.60%	73.46%	75.84%



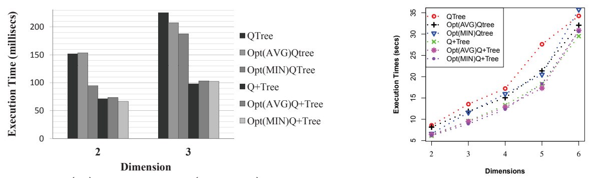
(a) DSL (b) MRSL (c) BRSL

Figure 12: Execution times of all indexing schemes for (a) DSL (b) MRSL and (c) BRSL queries in all datasets



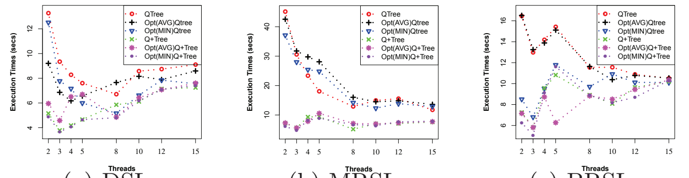
(a) DSL (b) MRSL (c) BRSL

Figure 13: CarDB (100K): threads vs. efficiency



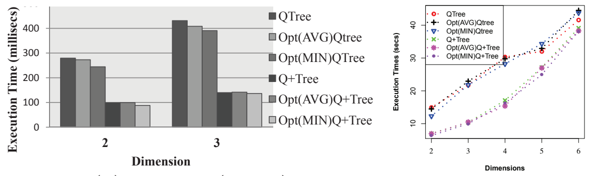
(a) CarDB (100K) (b) UN (3M)

Figure 15: DSL: dimensionality vs efficiency



(a) DSL (b) MRSL (c) BRSL

Figure 14: UN (3M): threads vs efficiency



(a) CarDB (100K) (b) UN (3M)

Figure 16: MRSL: dimensionality vs efficiency

execution times of all skyline queries. We see that the proposed Q+Tree indexing scheme outperforms the basic QTree indexing scheme in parallelizing all types of skyline queries. The aggressive partitioning heuristic improves the efficiency of both Q+Tree and QTree indexing schemes in most cases for all datasets. However, we do not observe any significant improvement of applying aggressive partitioning on the efficiencies of parallelizing the skyline queries in some cases. This indicates that searching the *pruned* regions of the data objects in the tree is sometimes more costly (due to tree depth) than performing the pairwise dominance check for them. Therefore, we advocate to use Q+Tree indexing only if there is no significant improvement in the ratios of pruned regions after applying aggressive partitioning for them. The following sections study the effect of different parameters.

6.3.1 *Effect of threads*

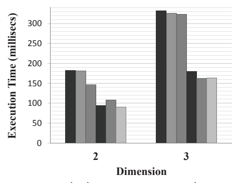
This section investigates the effect of #workers, i.e., Java threads, on the execution time of processing skyline queries in parallel. We run experiments with CarDB and UN datasets. For CarDB dataset, we set $|P| = 100K$, $|U| = 100K$, #samples to 500, $\rho = 20$, $d = 2$ and vary #threads from 2 to 10. For UN dataset, we set $|P| = 3M$, $|U| = 1M$, samples to 1000, $\rho = 50$, $d = 2$ and vary #threads from 2 to 15. For both CarDB and UN datsets, we also set τ to 100 for balancing loads in DSL queries. The average results for 100 skyline queries are shown in Fig. 13 and Fig. 14. It is evident that Q+Tree indexing offers the best efficiency with less threads than the basic QTree for all skyline queries. We observe that increased #threads may not improve the efficiency at all as the overhead of maintaining threads also gets increased.

6.3.2 *Effect of dimensionality*

Here, we study the effect of dimensionality on the efficiencies of processing all types of skyline queries in parallel by experimenting with CarDB and UN datasets. For CarDB dataset, we set $|P| = 100K$, $|U| = 100K$, #samples to 500, $\rho = 20$, #threads to 5 and vary the dimensionality d from 2 to 3. For UN dataset, we set $|P| = 3M$, $|U| = 1M$, #samples to 1000, $\rho = 50$, #threads to 10 and vary d from 2 to 6. For both CarDB and UN datsets, we also set τ to 100 for balancing loads in DSL queries. The average results for 100 skyline queries are shown in Fig. 15, Fig. 16 and Fig. 17. We see that Q+Tree data indexing scheme offers better efficiencies than the basic QTree scheme for all types of skyline queries. We also observe that the optimization heuristics improve the efficiencies in most cases for both datasets.

6.3.3 *Effect of data cardinality*

This section examines the effect of data cardinality on the efficiencies of processing all skyline queries in parallel by experimenting with million of objects in UN dataset. Here,


(a) CarDB (100K)

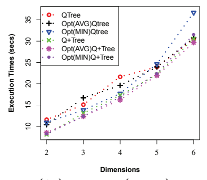
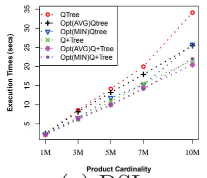
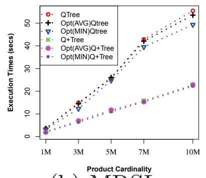

(b) UN (3M)

Figure 17: BRSL: dimensionality vs efficiency


(a) DSL


(b) MRSL

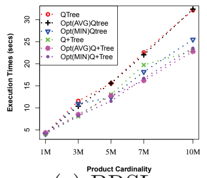
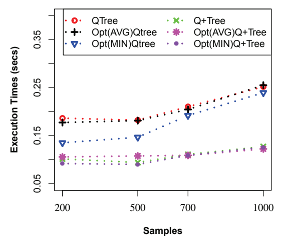

(c) BRSL

Figure 18: UN: product cardinality vs efficiency

we set $|U| = 1$M, #samples to 1000, $\rho = 50$, #threads to 10, $d = 2$ and vary $|P|$ from 1M to 10M. We also set $\tau = 100$ for DSL queries. The average results for 100 queries of all skyline types are shown in Fig. 18. We see that the proposed Q+Tree scheme scales well and outperforms the basic QTree scheme for parallelizing all types of skyline queries.

6.3.4 *Effect of samples and split threshold*

This section studies the effect of the settings of #samples and the split threshold, ρ, on the efficiencies of processing BRSL queries in parallel by experimenting with CarDB dataset. Firstly, we set $|P| = 100$K, $|U| = 100$K, #threads to 5, $d = 2$ and vary #samples from 200 to 1000. The average results of 100 queries are shown in Fig. 19(a). In the second experiment, we set #samples to 500 and vary ρ from 20 to 50. The average results are shown in Fig. 19(b). We see that the proposed Q+Tree indexing is more adaptive with the settings of both #samples and ρ than the basic QTree indexing. However, the setting of the above parameters as well as #worker processors (Java threads) in higher dimensions and data cardinalities (i.e., multi-parameter optimization) is an open challenge for future research.


(a) Samples vs. Efficiency

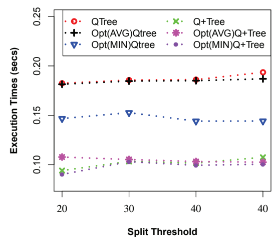

(b) ρ vs. Efficiency

Figure 19: CarDB: effect of samples and ρ on efficiency of bichromatic reverse skyline queries

7. CONCLUSION

This paper presents an efficient quad-tree based data indexing scheme, called Q+Tree, for parallelizing the dynamic and reverse skylines. We also present several optimization heuristics to improve the performance of the quad-tree based data indexing schemes. We conduct extensive experiments with both real and synthetic datasets and demonstrate the efficiency of the proposed Q+Tree data indexing scheme by comparing the results with its existing counterpart.

Acknowledgment: The work is partially supported by the ARC discovery grants DP140103499 and DP160102412.

8. REFERENCES

[1] A. Arvanitis, A. Deligiannakis, and Y. Vassiliou. Efficient influence-based processing of market research queries. In *CIKM*, pages 1193–1202, 2012.
[2] S. Börzsönyi, D. Kossmann, and K. Stocker. The skyline operator. In *ICDE*, pages 421–430, 2001.
[3] E. Dellis and B. Seeger. Efficient computation of reverse skyline queries. In *VLDB*, pages 291–302, 2007.
[4] P. M. Deshpande and D. Padmanabhan. Efficient reverse skyline retrieval with arbitrary non-metric similarity measures. In *EDBT*, pages 319–330, 2011.
[5] M. S. Islam and C. Liu. Know your customer: computing k-most promising products for targeted marketing. *The VLDB J.*, 25(4):545–570, 2016.
[6] M. S. Islam, R. Zhou, and C. Liu. On answering why-not questions in reverse skyline queries. In *ICDE*, pages 973–984, 2013.
[7] H. Köhler, J. Yang, and X. Zhou. Efficient parallel skyline processing using hyperplane projections. In *SIGMOD*, pages 85–96, 2011.
[8] C. Li, B. C. Ooi, A. K. H. Tung, and S. Wang. DADA: a data cube for dominant relationship analysis. In *SIGMOD*, pages 659–670, 2006.
[9] X. Lian and L. Chen. Monochromatic and bichromatic reverse skyline search over uncertain databases. In *SIGMOD*, pages 213–226, 2008.
[10] X. Lin, Y. Yuan, Q. Zhang, and Y. Zhang. Selecting stars: The k most representative skyline operator. In *ICDE*, pages 86–95, 2007.
[11] K. Mullesgaard, J. L. Pederseny, H. Lu, and Y. Zhou. Efficient skyline computation in mapreduce. In *EDBT*, pages 37–48, 2014.
[12] D. Papadias, Y. Tao, G. Fu, and B. Seeger. An optimal and progressive algorithm for skyline queries. In *SIGMOD*, pages 467–478, 2003.
[13] Y. Park, J. Min, and K. Shim. Processing of probabilistic skyline queries using mapreduce. *PVLDB*, 8(12):1406–1417, 2015.
[14] Y. Park, J.-K. Min, and K. Shim. Parallel computation of skyline and reverse skyline queries using mapreduce. *PVLDB*, 6(14):2002–2013, 2013.
[15] D. Pertesis and C. Doulkeridis. Efficient skyline query processing in spatialhadoop. *Information Systems*, 54:325–335, 2015.
[16] M. Sharifzadeh and C. Shahabi. The spatial skyline queries. In *VLDB*, pages 751–762, 2006.
[17] Y. Tao, L. Ding, X. Lin, and J. Pei. Distance-based representative skyline. In *ICDE*, pages 892–903, 2009.
[18] J. S. Vitter. Random sampling with a reservoir. *ACM Trans. Math. Softw.*, 11(1):37–57, 1985.
[19] A. Vlachou, C. Doulkeridis, and Y. Kotidis. Angle-based space partitioning for efficient parallel skyline computation. In *SIGMOD*, 2008.
[20] G. Wang, J. Xin, L. Chen, and Y. Liu. Energy-efficient reverse skyline query processing over wireless sensor networks. *IEEE Trans. Knowl. Data Eng.*, 24(7):1259–1275, 2012.
[21] T. Wu, D. Xin, Q. Mei, and J. Han. Promotion analysis in multi-dimensional space. *PVLDB*, 2(1):109–120, 2009.
[22] X. Wu, Y. Tao, R. C.-W. Wong, L. Ding, and J. X. Yu. Finding the influence set through skylines. In *EDBT*, pages 1030–1041, 2009.
[23] J. Zhang, X. Jiang, W.-S. Ku, and X. Qin. Efficient parallel skyline evaluation using mapreduce. *IEEE Trans. Parallel Distrib. Syst.*, 2016.
[24] S. Zhang, N. Mamoulis, and D. W. Cheung. Scalable skyline computation using object-based space partitioning. In *SIGMOD*, pages 483–494, 2009.