Lecture 5: Top-1 and Skyline

CMSC 5705 Advanced Topics in Database Systems

Yufei Tao

Department of Computer Science and Engineering
Chinese University of Hong Kong

October 19, 2010

Definition (Monotonically increasing function)

Let p be a d-dimensional point in $\mathbb{R}^d$. Let $f : \mathbb{R}^d \to \mathbb{R}$ a function that calculates a score $f(p)$ for p. We say that f is *monotonically increasing* if the score never decreases when any coordinate of p increases.

For example, $f(x, y) = x + y$ is monotonically increasing but $f(x, y) = x - y$ is not.

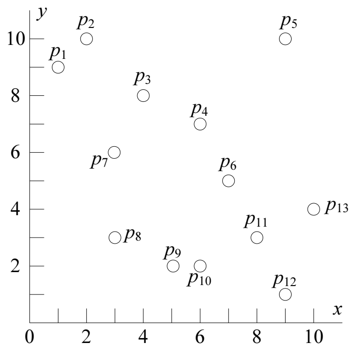
Definition (Top-1 search)

Let P be a set of d-dimensional points in $\mathbb{R}^d$. Given a monotonically increasing function f, a *top-1 query* finds the point in P that has the smallest score.

The problem can be extended to top-k search in a straightforward manner.

Example

If $f(x, y) = x + y$, then the top-1 is p_8.



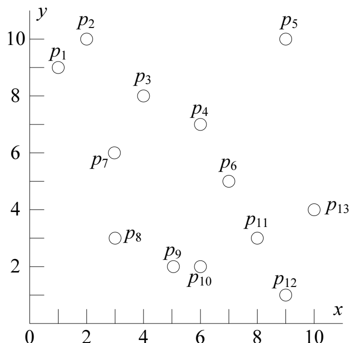
Assuming that the dataset P is indexed by an R-tree, we can answer a top-1 query by directly applying the nearest neighbor algorithm discussed in the last lecture. Specifically, the top-1 object is the NN of the origin of the data space according to the distance function f.

Think

What is the mindist of an MBR?

Drawback of top-1 search

In general, it is difficult to decide which distance function f should be used. For example, assume that the x-dimension corresponds to the price of a hotel and the y-dimension to its user rating (the smaller, the better). Why is $f(x, y) = x + y$ a good function to use? Why not $2x + y$, or something more complex like $\sqrt{x} + y^2$?



The skyline operator remedies the drawback of top-1 search with an interesting idea. Instead of reporting only 1 object, the operator reports a set of objects that are guaranteed to cover the result of any top-1 query (i.e., regardless of the query function, as long as it is monotonically increasing!).

Definition (Dominance)

A point p_1 *dominates* p_2 if the coordinate of p_1 is smaller than or equal to p_2 in all dimensions, and strictly smaller in one dimension.

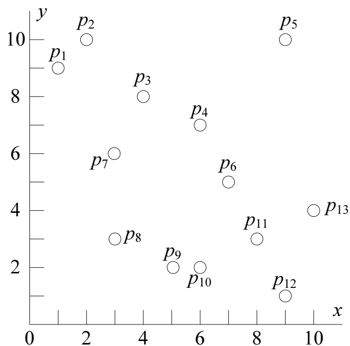
Note that p_1 has a smaller score than p_2 with respect to all monotonically increasing function.

Definition (Skyline)

Let P be a set of d-dimensional points in $\mathbb{R}^d$ such that no two points coincide with each other. The *skyline* of P contains all the points that are not dominated by others.

The skyline is also known as *pareto set*.

The skyline is $\{p_1, p_8, p_9, p_{12}\}$.



Theorem

For any monotonically increasing function, the top-1 point is definitely in the skyline. Conversely, every point in the skyline is definitely the top-1 of some monotonically increasing function.

The first statement is easy to prove. The establishment of the second statement is more involved, and not required in this course. The instructor will outline the basic idea of the proof.

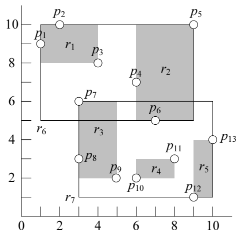
Next we will introduce two algorithms to solve the skyline problem. The first one assumes the existence of an R-tree on P, while the other does not assume any index on P.

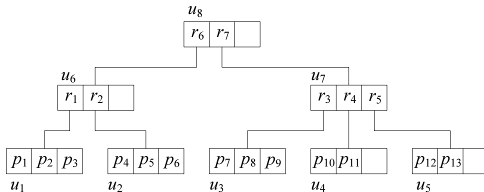
BBS example

Assuming an R-tree on P, the *branch and bound skyline (BBS)* algorithm can be thought of a variation of the BF algorithm in the previous lecture. Specifically, it accesses the nodes of the R-tree in ascending order of the mindists from the origin to their MBRs. The novelty is that if an MBR is dominated by a skyline point already found, it can be pruned. Next let us get the idea from an example.

BBS example (cont.)

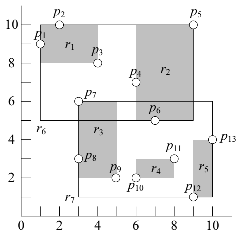
First, we access the root, and put the MBRs there in a min-heap H, namely, $H = \{(r_7, \sqrt{10}), (r_6, \sqrt{26})\}$.

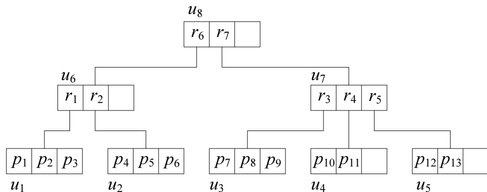




BBS example (cont.)

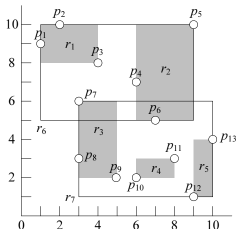
Next, the algorithm visits node u_7, after which the heap becomes: $H = \{(r_3, \sqrt{13}), (r_6, \sqrt{26}), (r_4, \sqrt{40}), (r_5, \sqrt{82})\}$.

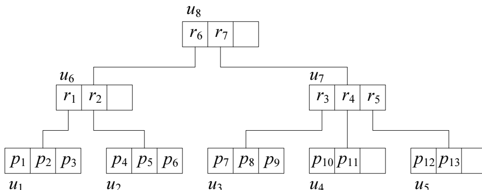




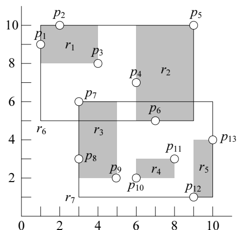
BBS example (cont.)

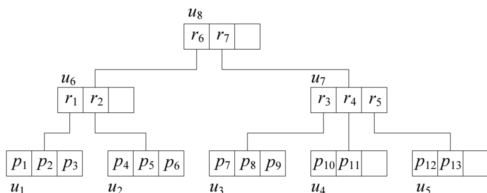
We now visit u_3 which is a leaf node. Among the points there, p_7 is dominated by p_8 and hence discarded. The other points p_8, p_9 cannot be ruled out yet. So our current result is $SKY = \{p_8, p_9\}$. At this time, $H = \{(r_6, \sqrt{26}), (r_4, \sqrt{40}), (r_5, \sqrt{82})\}$.





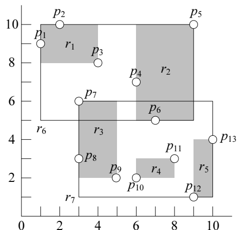
Access u_6, and update the heap to $H = \{(r_4, \sqrt{40}), (r_2, \sqrt{61}), (r_1, \sqrt{65}), (r_5, \sqrt{82})\}$. The top of H, r_4, can be pruned because its lower left corner is dominated by p_9 in the current result. In other words, no point in r_4 can possibly belong to the skyline. For the same reason, r_2 can also be pruned.

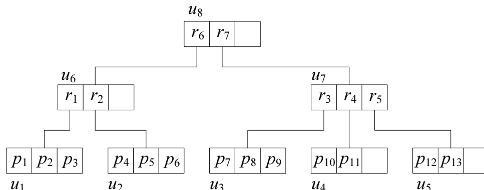




BBS example (cont.)

Currently $H = \{(r_1, \sqrt{65}), (r_5, \sqrt{82})\}$. Both MBRs need to be accessed. SKY is updated accordingly with the points found in the leaf nodes of those MBRs. Now that H is empty, the algorithm terminates.



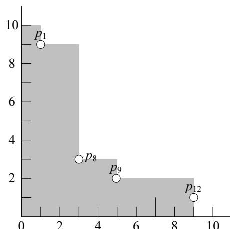


Pseudocode of BBS

```
algorithm BBS
1.  insert the MBR of the root into the min-heap H
    /* MBRs in H are organized by their mindists to the origin */
2.  SKY = ∅ /* current result */
3.  while H is not empty do
4.      remove the MBR r from the top of H
5.      if the node u of r is a leaf node then
6.          update SKY using the points in u
7.      else
8.          if no point in SKY dominates the lower-left corner r then
9.              visit u and insert each MBR there into H
```

Optimality of BBS

As with BF, BBS is optimal, i.e., it incurs the least I/Os among all algorithms that correctly finds the skyline using the same R-tree. To prove this, let us define the *search region* as the union of the points in $\mathbb{R}^d$ that are not dominated by any skyline point. For example, in our previous example, the search region is the shaded area below:



It is easy to see that any correct algorithm must access all the nodes whose MBRs intersect the search region.

Optimality of BBS (cont.)

We can show that BBS accesses *only* the nodes whose MBRs intersect the search region. Assume, for contradiction, that the algorithm needed to visit a node u whose MBR r is disjoint with the region.

- It follows that a skyline point p dominates the lower-left corner of r. Let u' be the leaf node containing p, and r' the MBR of u'.
- It is easy to see that r' has a smaller mindist to the origin than r. Hence, u' was accessed before u.
- However, the visit to u' immediately led to the discovery of p, which should have allowed BBS to prune u at Line 8 of Slide 17.

Recall that, if there is no index on the underlying dataset, range search and nearest neighbor search are not interesting, because they can be trivially solved with a single scan of the dataset, and it is not possible to do any better. This is not the case, however, for the skyline problem. As we will see in the next slide, a trivial algorithm (in the absence of any index) would have to take time quadratic to the dataset size. Therefore, it is important to explore alternative faster solutions.

Naive algorithm

algorithm `naive`

1. $SKY = \emptyset$
2. **for** each point $p \in P$
3. $\quad SKY \leftarrow$ the skyline of $SKY \cup \{p\}$
4. **return** SKY

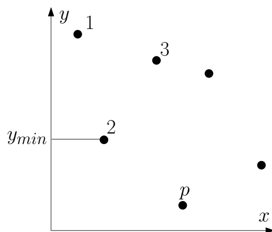
Next we will explain how to solve the skyline problem in $O(n \log n)$ time in 2-d and 3-d spaces, when the entire dataset fits in memory. In other words, we are considering the RAM computation model (as opposed to the external memory model).

2-d

Assume that P has been sorted in ascending order of their x-coordinates (which can be done in $O(n \log n)$ time). In case two points have the same x-coordinate, rank the one with a smaller y-coordinate first. Consider any point $p \in P$. Let S be the set of points that rank before P. Observe:

- No point that ranks *after* p can possibly dominate p.
- Some point in S dominates p, if and only if the smallest y-coordinate of the points in S is no greater than the y-coordinate of p.

2-d (cont.)



Pseudocode of the 2-d algorithm

algorithm `2d-skyline`
1. sort the dataset P as described in Slide 23
2. $SKY = \emptyset, y_{min} = \infty$
2. **for** each point $p \in P$ in the sorted order
3.   **if** the y-coordinate $p[y]$ of p is smaller than y_{min}
4.   add p to SKY, and $y_{min} = p[y]$
5. **return** SKY

Line 1 takes $O(n \log n)$ time, whereas Lines 2-4 essentially scan the entire P only once in $O(n)$ time. Hence, the overall cost is $O(n \log n)$.

3-d

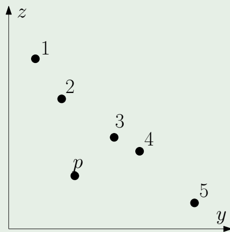
Again, sort P in ascending order of their x-coordinates. Break ties by putting the point with a smaller y-coordinate first, and if there is still a tie, the point with a smaller z-coordinate ranks first. Consider any point $p \in P$. Let S be the set of points that rank before P. Observe:

- (Same as 2-d) no point that ranks *after* p can possibly dominate p.
- Let $SKY_{yz}(S)$ be the skyline of the projections of (the points of) S in the y-z plane. Some point in P dominates p in the x-y-z space, if and only if a point of $SKY_{yz}(S)$ dominates p in the y-z plane.

3-d (cont.)

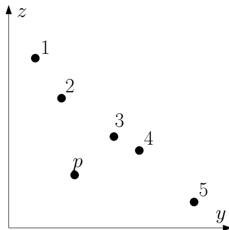
Example

Assume $SKY_{yz}(S)$ includes points 1, 2, 3, 4, 5. As no point of $SKY_{yz}(S)$ dominates p in the y-z plane, we can assert that p is definitely in the skyline (of the original space).



3-d (cont.)

$SKY_{yz}(S)$ is a 2-d skyline. In general, a 2-d skyline is a staircase. Namely, if we walk along the skyline towards the direction of ascending x-coordinates, the y-coordinates of the points keep decreasing.



3-d (cont.)

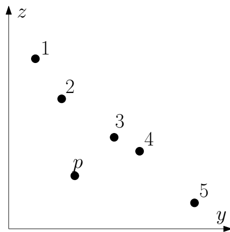
Let us index the points of $SKY_{yz}(S)$ by their y-coordinates using a binary tree (or a B-tree with a constant $B \geq 4$). Two operations can be done efficiently:

- Detect if a point p is dominated by any point in $SKY_{yz}(S)$ (in the y-z plane).
- Remove all points of $SKY_{yz}(S)$ dominated by p (in the y-z plane).

We will show that each detection can be done in $O(\log n)$ time, while removal in $O(k \log n)$ time, where k is the number of points removed.

3-d (cont.)

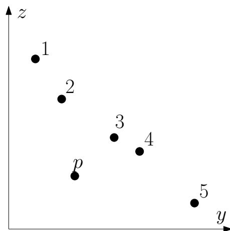
Detection is based on the observation that p is dominated by some point in $SKY_{yz}(S)$ if and only if p is dominated by the predecessor of p in $SKY_{yz}(S)$ on the y-dimension. For example, the predecessor is point 2 in the figure below.



Finding the predecessor takes $O(\log n)$ time using the binary tree.

3-d (cont.)

To remove the points of $SKY_{yz}(S)$ dominated by p (in the y-z plane), we first find the successor, say p', of p in $SKY_{yz}(S)$ on the y-dimension. If p dominates p', remove p' and set p' to its own successor. Repeat this until p no longer dominates p'. In the figure below, p' iterates through points 3 and 4.



Finding a successor and removing a point take $O(\log n)$ time.

Pseudocode of the 3-d algorithm

algorithm `3d-skyline`
1. sort the dataset P as described in Slide 26
2. $SKY = \emptyset$
3. let T be the binary tree as mentioned in Slide 29
4. **for** each point $p \in P$ in the sorted order
5.   **if** p is not dominated by any point of T in the y-z plane **then**
6.   add p to SKY
7.   remove from T all points dominated by p in the y-z plane
8. **return** SKY

The detection at Line 5 is performed n times, and thus, requires $O(n \log n)$ time in total. On the other hand, each point is inserted and removed in T at most once. Hence, all the insertions and deletions entail $O(n \log n)$ time.

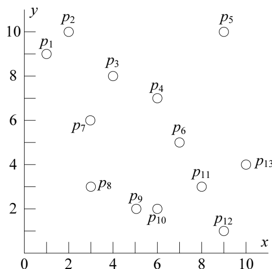
Remark

In general, the skyline problem can be settled in $O(n \log^{d-2} n)$ when the dimensionality d is at least 3. The algorithm for $d \geq 4$, however, is quite theoretical, and may not be as efficient as the heuristic algorithm introduced next.

Now let us return to the external memory model, where the memory has a finite size M (words), each disk block has size B ($M \geq 2B$), and the objective is to minimize the number of I/Os. We will describe an algorithm called *sort first skyline (SFS)* that is efficient in practice, but is heuristic in nature (i.e., it does not have an attractable worst-case performance bound).

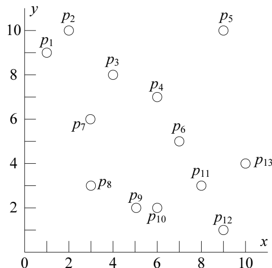
SFS example

We will use the following dataset as an example, assuming that each block can store at most 2 points, and our memory has two blocks.



First, sort all the points according to an arbitrary monotonically increasing function, e.g., $f(x, y) = x + y$. In case of a tie, the point with a smaller x-coordinate goes first. Note that a point can only be dominated by points that rank before it.

$P = \{(p_8, 6), (p_9, 7), (p_{10}, 8),$
$(p_1, 9), (p_7, 9), (p_{12}, 10), (p_{11}, 11),$
$(p_2, 12), (p_3, 12), (p_6, 12), (p_4, 13),$
$(p_{13}, 14), (p_5, 19)\}$



SFS example (cont.)

Next we will scan the sorted list. A memory block needs to be assigned as the buffer for this purpose. So only 2 points can be kept in memory at any time.

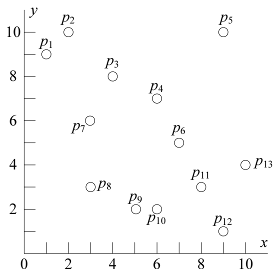
SFS example (cont.)

Read the first block (i.e., two points) of P into memory.

- memory $= \{(p_8, 6), (p_9, 7)\}$

What do we know about them? First, p_8 is definitely in the skyline. Second, since p_9 is not dominated by p_8, it is also in the skyline.

$P = \{(p_8, 6), (p_9, 7), (p_{10}, 8), (p_1, 9), (p_7, 9), (p_{12}, 10), (p_{11}, 11), (p_2, 12), (p_3, 12), (p_6, 12), (p_4, 13), (p_{13}, 14), (p_5, 19)\}$

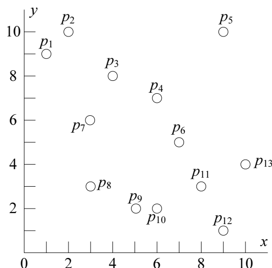


SFS example (cont.)

Read the next block of P into memory. p_{10} is dominated by p_9 and hence, can be discarded. On the other hand, since p_1 is not dominated by p_8 or p_9, it is in the skyline.

- memory $= \{(p_8, 6), (p_9, 7), (p_{10}, 8), (p_1, 9)\}$

$P = \{(p_8, 6), (p_9, 7), (p_{10}, 8),$
$(p_1, 9), (p_7, 9), (p_{12}, 10), (p_{11}, 11),$
$(p_2, 12), (p_3, 12), (p_6, 12), (p_4, 13),$
$(p_{13}, 14), (p_5, 19)\}$

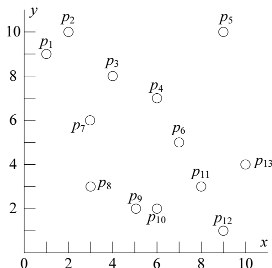


SFS example (cont.)

Now the memory is full. We must empty a memory block to read the next block of P. So p_1 is flushed to an overflow list in the disk.

- memory $= \{(p_8, 6), (p_9, 7)\}$
- overflow list $= \{(p_1, 9)\}$

$P = \{(p_8, 6), (p_9, 7), (p_{10}, 8), (p_1, 9), (p_7, 9), (p_{12}, 10), (p_{11}, 11), (p_2, 12), (p_3, 12), (p_6, 12), (p_4, 13), (p_{13}, 14), (p_5, 19)\}$

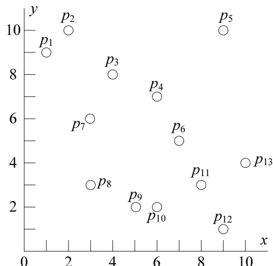


SFS example (cont.)

Read the next block of P. p_7 can be discarded. How about p_{12}? Note that it *cannot* be confirmed as a skyline point because it has not been compared to p_1 (in the overflow list) yet. Hence, we keep it into the overflow list for later processing.

- memory $= \{(p_8, 6), (p_9, 7), (p_7, 9), (p_{12}, 10)\}$
- overflow list $= \{(p_1, 9), (p_{12}, 10)\}$

$P = \{(p_8, 6), (p_9, 7), (p_{10}, 8),$
$(p_1, 9), (p_7, 9), (p_{12}, 10), (p_{11}, 11),$
$(p_2, 12), (p_3, 12), (p_6, 12), (p_4, 13),$
$(p_{13}, 14), (p_5, 19)\}$

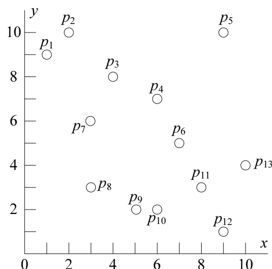


SFS example (cont.)

Similarly, the next block is loaded. p_{11} is pruned but p_2 enters the overflow list.

- memory $= \{(p_8, 6), (p_9, 7), (p_{11}, 11), (p_2, 12)\}$
- overflow list $= \{(p_1, 9), (p_{12}, 10), (p_2, 12)\}$

$P = \{(p_8, 6), (p_9, 7), (p_{10}, 8), (p_1, 9), (p_7, 9), (p_{12}, 10), (p_{11}, 11), (p_2, 12), (p_3, 12), (p_6, 12), (p_4, 13), (p_{13}, 14), (p_5, 19)\}$

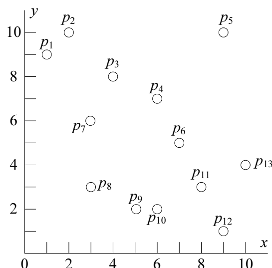


SFS example (cont.)

SFS proceeds in the way as described. When it finishes scanning the entire P, the contents of the memory and overflow list are as follows:

- memory $= \{(p_8, 6), (p_9, 7)\}$
- overflow list $= \{(p_1, 9), (p_{12}, 10), (p_2, 12)\}$

$P = \{(p_8, 6), (p_9, 7), (p_{10}, 8), (p_1, 9), (p_7, 9), (p_{12}, 10), (p_{11}, 11), (p_2, 12), (p_3, 12), (p_6, 12), (p_4, 13), (p_{13}, 14), (p_5, 19)\}$



SFS example (cont.)

The algorithm outputs the points p_8, p_9 that remain in the memory.

- memory $= \{(p_8, 6), (p_9, 7)\}$
- overflow list $= \{(p_1, 9), (p_{12}, 10), (p_2, 12)\}$

Now it remains to find the skyline of the points in the overflow list. Run another iteration of SFS again by treating the list as a new dataset. Repeat until the overflow list is empty.

Pseudocode of SFS

algorithm `SFS`

```
1.  sort the dataset P as described in Slide 36
2.  overflowList ← ∅
3.  while P ≠ ∅
    /* start a new iteration */
4.      while P has not been exhausted in this iteration do
5.          read the next block S of P
6.          delete the points of S dominated by a point currently in memory
7.          if memory is full then
            /* less than a block of memory is vacant */
8.              flush some least-recently-read points to the end of
                overflowList so that a block of memory becomes available
9.      output the points in memory
10.     P ← overflowList
```

Analysis of SFS

- Each iteration of SFS outputs at least $M - B$ skyline points, except possibly the last iteration.
  - In each iteration, all the points in memory when the first flush (to the overflow list) happens are guaranteed to be in the skyline.
- Hence, there can be at most $O(\frac{N}{M-B}) = O(N/M)$ iterations.
- Each iteration performs $O(N/B)$ I/Os.

Hence, the total cost is $O(N^2/(MB))$.

Remark

The $O(N^2/(MB))$ asymptotical performance of SFS is not impressive at all, because the same bound can be achieved by the naive algorithm in Slide 21. The strength of SFS, however, is that it runs much faster on practical data than the worst-case bound.

Playback of this lecture

- Top-1 search.
- Skyline.
- BBS
- Non-indexed solutions
  - 2-d and 3-d algorithms in memory.
  - SFS for external memory.