4. SKYLINE

A traditional in-memory two-dimensional skyline algorithm [33] uses a divide and conquer approach where all points are initially sorted by their x coordinates and divided into two subsets of equal size separated by a vertical line. Then, the skyline of each half is computed recursively and the two skylines are merged to compute the final skyline. To merge two skylines, the points of the left skyline are scanned in a non-decreasing x order, which implies a non-increasing y order, and each one is compared to the left most point of the right skyline. Once a point on the left skyline is dominated, it is removed along with all subsequent points on the left skyline and the two lists of remaining points from both skylines are concatenated together. The skyline operator is not natively supported in database management systems. Yet, it is of considerable interest in the database literature, where the focus is mainly on disk-based algorithms (e.g., see [7, 31]) with a *non-standard* SQL query.

```
SELECT * FROM points
SKYLINE OF d1 MAX, d2 MAX;
```

In this section, we introduce our two skyline algorithms for Hadoop and SpatialHadoop, while using input dataset in Figure 1(c) as an illustrative example.

4.1 Skyline in Hadoop

Our Hadoop skyline algorithm is a variation of the traditional divide and conquer skyline algorithm [33], where we divide the input into multiple (more than two) partitions, such that each partition can be handled by one machine. This way, the input needs to be divided across machines only once ensuring that the answer is found in one MapReduce iteration. Similar to our Hadoop polygon union algorithm, our Hadoop skyline algorithm works in three steps, *partitioning*, *local skyline*, and *global skyline*. The *partitioning* step divides the input set of points into smaller chunks of 64MB each and distributes them across the machines. In the *local skyline* step, each machine computes the skyline of each partition assigned to it, using the traditional algorithm, and outputs only the non-dominated points. Finally, in the *global skyline* step, a single machine collects all points of local skylines, combines them in one set, and computes the skyline of all of them. Notice that skylines cannot be merged using the technique used in the in-memory algorithm as the local skylines are not separated by a vertical line, and may actually overlap. This is a result of Hadoop partitioning which distributes the points randomly without taking their spatial locations into account. The global skyline step computes the final answer by combining all the points from local skylines into one set, and applying the traditional skyline algorithm. Readers familiar with MapReduce programming can refer to Appendix A.2 for pseudocode.

This algorithm significantly speeds up the skyline computation compared to the traditional algorithm by allowing multiple machines to run independently and in parallel to reduce the input size significantly. For a uniformly distributed dataset of size n points, it is expected that the number of points on the skyline is $O(\log n)$ [4]. In practice, for a partition of size 64MB with around 700K points, the skyline only contains a few tens of points for both real and uniformly generated datasets. Given this small size, it becomes feasible to collect all those points in one single machine that computes the final answer.

4.2 Skyline in SpatialHadoop

Our proposed skyline algorithm in SpatialHadoop is very similar to the Hadoop algorithm described earlier, with two main changes. First, in the *partitioning* phase, we use the SpatialHadoop partitioner when the file is loaded to the cluster. This ensures that the data is partitioned according to an R-tree instead of random partitioning, which means that local skylines from each machine are non overlapping. Second, we apply an extra *filter* step right before the local skyline step. The *filter* step, runs on a master node, takes as input the minimal bounding rectangles (MBRs) of all partitioned R-tree index cells, and prunes those cells that have no chance in contributing any point to the final skyline result.

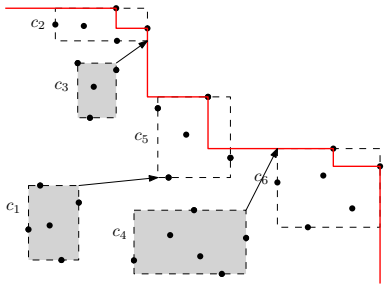

Figure 4: Skyline in SpatialHadoop

The main idea of the new *filter* step is that a cell c_i dominates another cell c_j if there is at least one point in c_i that dominates all points in c_j, in which case c_j is pruned. For example, in Figure 4, cell c_1 is dominated by c_5 because the bottom-left corner of c_5 (i.e., worst point) dominates the top-right corner of c_1 (i.e., best point). The transitivity of the skyline dominance relation implies that any point in c_5 dominates all points in c_1. Similarly, c_4 is dominated by c_6 because the top-left corner of c_6 dominates the top-right corner of c_4. This means that any point along the top edge of c_6 dominates the top-left corner of c_6, and hence, dominates all points in c_4. As the boundaries of a cell are minimal (because of R-tree partitioning), there should be at least one point of P on each edge. We can similarly show that cell c_3 is also dominated by c_2. So, our pruning technique in the *filter* step is done through a nested loop that tests every pair of cells c_i and c_j together. We compare the top-right corner of c_j against three corners of c_i (bottom-left, bottom-right, and top-left). If any of these corners dominates the top-right corner of c_j, we prune c_j out from all our further computations, and do not assign it to any node. Hence, we will not compute its local skyline, nor consider it in the global skyline step.

It is important to note that applying this filtering step in Hadoop will not have much effect, as the partitioning scheme used in Hadoop will not help in having such separated MBRs for different cells. The SpatialHadoop skyline algorithm has much better performance than its corresponding Hadoop algorithm as the *filtering* step prunes out many cells that do not need to be processed. Interested readers can refer to Appendix A.2 for the pseudocode of the filter step.

5. CONVEX HULL

The convex hull shown in Figure 1(e) can be computed as the union of two chains using Andrew's Monotone Chain algorithm [3]. First, it sorts all points by their x coordinates and identifies the left-most and right-most points. Then, the upper chain of the convex hull is computed by examining every three consecutive points p, q, r, in turn, from left to right. If the three points make a non-clockwise turn, then the middle point q is skipped as it cannot be part of the upper chain and the algorithm then considers the points p, r, s, where s is the successor of r; otherwise the algorithm continues by examining the next three consecutive points q, r, s. Once the rightmost point is reached, the algorithm con-