A Parallel and Distributed Approach for Diversified Top-k Best Region Search

Hamid Shahrivari
Eindhoven University of Technology
h.shahrivari.joghan@tue.nl

Matthaios Olma
EPFL
matthaios.olma@epfl.ch

Odysseas Papapetrou
Eindhoven University of Technology
o.papapetrou@tue.nl

Dimitrios Skoutas
Athena R.C.
dskoutas@athenarc.gr

Anastasia Ailamaki
EPFL
anastasia.ailamaki@epfl.ch

ABSTRACT

Given a set of points, the *Best Region Search* problem finds the optimal location of a rectangle of a specified size such that the value of a user-defined scoring function over its enclosed points is maximized. A recently proposed top-k algorithm for this problem returns results progressively, while also incorporating additional constraints, such as taking into consideration the overlap between the set of selected top-k rectangles. However, the algorithm is designed for a centralized setting and does not scale to very large datasets. In this paper, we overcome this limitation by enabling parallel and distributed computation of the results. We first propose a strategy that employs multiple rounds to progressively collect partial top-k results from each node in the cluster, while a coordinator handles the aggregation of the global top-k list, dealing with overlapping results. We then devise a single-round strategy, where the algorithm executed by each node is enhanced with additional conditions that anticipate potential overlapping solutions from neighboring nodes. Additional optimizations are proposed to further increase performance. Our experiments on real-world datasets indicate that our proposed algorithms are efficient and scale to millions of points.

1 INTRODUCTION

The amount of geospatial data generated from social networks, sensors, smart phone applications, tracking devices and so on is constantly increasing [9]. Analyzing big geospatial data at scale is of paramount importance for numerous applications in various areas such as geomarketing, mobile advertisement, urban planning, tourism and logistics. In many cases, the analysis involves identifying areas where the intensity of a studied phenomenon is maximized. This involves, for example, finding *hot spots* of user check-ins, commercial activities, crime incidents, etc. Various traditional methods exist and have been used for such purposes, such as computing global and local spatial autocorrelation [2] or finding density-based clusters [10], while several recent studies have also focused on problems related to finding areas of interest according to certain keyword-based and size-based criteria and constraints [3, 4, 7, 8, 15].

In this context, several types of *optimal location selection* problems have been studied. Range aggregate queries [18] have been proposed for scenarios where users are interested in summarized information about objects in a given region. Such queries return an aggregate score over the objects enclosed within a given region, and can be efficiently processed using aggregate spatial indices. However, these methods can not be efficiently applied when the problem is to find where the best regions are located. A typical and widely studied formulation of this problem is the *MaxRS* (Maximizing Range Sum) problem [6], which, given a set of 2-dimensional weighted points, aims at finding the optimal location of a fixed-size rectangular region that maximizes the sum of weights of the points enclosed in it.

In this paper, we focus on the *Best Region Search* (BRS) problem [11], which is a generalization of the *MaxRS* problem. Similarly, the input is a dataset $\mathcal{D}$ of spatial objects represented as weighted points in a 2-dimensional space, and two parameters $a, b \in \mathbb{R}^+$, denoting the width and height of the region to be found. The goal is to identify the optimal location of an axis-aligned $a \times b$ rectangular region R that maximizes the value of a scoring function f computed over the enclosed points. Hence, the difference lies in the fact that in *MaxRS*, f is restricted to the sum of weights of the points within R, while in *BRS* it can be any submodular monotone function. The *MaxRS* problem and its extensions and variants have their roots in problems studied in the past by the computational geometry community [14, 16], and have received much attention recently by the database community [1, 4, 6, 11, 12, 20, 21].

The *BRS* problem has numerous applications related to location planning. For instance, assume a company that is searching for the optimal location to open a new store. In that case, an $a \times b$ region R can be used to approximate the area from which the new store is expected to draw its potential customers. Assuming that a suitable scoring function f is provided, which computes a utility score for each candidate region R based on its contents, the solution to the *BRS* problem indicates the optimal location for placing this new store. Similar examples can also be found, for instance, when recommending regions to travelers. Given the size of the region that a user is willing to explore, modeled by an $a \times b$ rectangle, and a function f quantifying the utility of a region with respect to the user's preferences (e.g., presence of museums, restaurants, shops, etc.), the *BRS* problem computes the best region to be recommended.

The *k-BRS* problem has been introduced in [19], presenting an algorithm that can compute top-k best regions *progressively*. While doing so, we have also observed that in many real-world datasets, where spatial objects are typically not uniformly distributed in space, these top-k results tend to highly overlap, even for very large values of k. This is expected, since by slightly shifting a region R horizontally and/or vertically it is possible to obtain a new region R' that covers almost the same area as R and thus achieves very similar utility score to it. Yet, results of such type offer essentially no new insight over the explored data. Hence, to tackle this problem, we have introduced an additional constraint that either completely prohibits overlaps among the

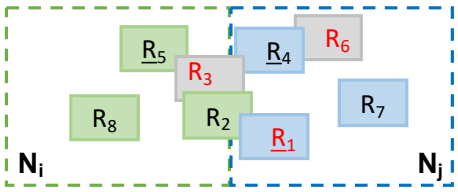

Figure 1: Illustrative example. The green and blue regions are those returned by N_i and N_j, respectively. The underlined results are those finally selected by the coordinator. The red ones are the correct top-3 results.

returned results or reduces the degree of overlap by allowing but penalizing partial overlaps among the returned top-k regions.

Being able to retrieve top-k results progressively, instead of simply computing the overall best region, as well as avoiding overlapping results that provide near-duplicate information, has many advantages in data exploration scenarios. Still, the state-of-the-art solution proposed in [19] operates in a centralized setting and cannot scale as the input dataset size grows; indicatively, it needs around 30 seconds to identify the top-10 non-overlapping regions of size 200×200 m^2 even on moderate-size datasets – around 100 thousand points.

To efficiently address the *k-BRS* problem when dealing with big geospatial data, an approach is needed that can scale out, i.e., distribute and parallelize the computation over multiple workers (computing nodes) in a cluster. In this paper, we address this problem by proposing different methods for parallel and distributed top-k best region search while taking into account the criterion of overlap among the returned results. In particular, we consider a setting where the input dataset is partitioned across several workers that perform local computations in parallel, while a coordinator node is assigned the task to coordinate the execution and merge local results to produce the global top-k result list. Most contemporary Big Data platforms, e.g., Spark, are naturally represented by the aforementioned setting.

Before we describe our approach, we explain the main challenge to this problem, which arises when non-overlapping (or only partially overlapping) results are requested. Specifically, the challenge arises from the fact that, under this constraint, the global top-k results are not contained in the union of the local top-k results computed at each node. Therefore, simply computing the local top-k results at each node and merging these top-k lists at the coordinator can lead to incorrect results in many cases. We illustrate this with the following example.

Example 1. *Consider the example illustrated in Fig. 1, assuming that the data is split in two partitions, each one assigned to a different worker, represented as N_i and N_j. Assume that the top-8 regions with highest scores in both partitions (regardless of overlaps) are those shown in the figure, with the index of each region indicating its rank (i.e., R_1 having the highest score). Suppose that each region belongs to (i.e., is computed by) the worker that contains its center.*

Assume that the user requests the top-3 non-overlapping results. Node N_i will progressively compute its results, and will return to the coordinator the regions R_2, R_5 and R_8. R_3 is skipped since it overlaps with R_2 which has a higher rank. Similarly, N_j will return to the coordinator the results R_1, R_4 and R_7. Again, R_6 is skipped as it overlaps with R_4. Based on these two sets of local top-3 results, the coordinator will compute the global top-3 list excluding overlapping regions, thus returning R_1, R_4 and R_5.

However, the above result is incorrect. The correct global top-3 results should have been R_1, R_3 and R_6. As we can see, R_3 is a false negative – it was missed because N_i skipped it, as it was overlapping with R_2; however, R_2 was eventually discarded by the coordinator because it was overlapping with a better result R_1 from the neighboring node. R_4 and R_5 are false positives – their respective nodes considered them as valid results; however they should have been discarded because eventually a better result R_3 that overlaps with both of them should have been present. R_6 is again a false negative, for reasons similar to R_3.

As revealed by the above example, the root cause of false results is the fact that the admission of a *local* top-k region in the *global* top-k list is precluded by the existence of a higher ranked region that overlaps with it. Since the latter may arise from a different worker than the former, a worker cannot independently reason regarding the validity of its own results, i.e., which of these results will eventually "survive" at the coordinator. To make matters worse, candidates that were rejected locally due to the existence of a better local overlapping result R, may actually have a place in the global top-k list if R is later dismissed at the coordinator.

Since overlapping regions between two neighboring workers occur at the borders, an intuitive – yet insufficient – approach to overcome this problem would be to replicate the contents of two neighboring nodes around the borders. In that case, in the example of Fig. 1, it could be possible for N_i to know, for instance, that R_1 overlaps with R_2 and has higher score than it. However, it may turn out that R_1 is disqualified in N_j, if it overlaps with an even better result not located in the border with N_i, in which case N_i would still have no knowledge of this. Even replicating the whole contents of N_j to N_i would not suffice, since the results of N_j may similarly be affected by its other neighbors as well.

In this paper, we explore different solutions to overcome this problem. First, we propose a *multi-round* algorithm (*MR*), where the problem of dealing with conflicts is mainly handled by the coordinator. The advantage of *MR* is that workers can readily exploit the local *k-BRS* algorithm with minor adaptations, since the decisions for forming the global top-k list are made at the coordinator level. The downside is that when overlapping results are identified, and hence discarded, at the coordinator, a new round has to be executed, where the relevant workers are informed about these results and are requested to compute new results accordingly. To overcome this drawback, we devise a *single-round* algorithm (*SR*). This requires formulating appropriate conditions to reason about the uncertainty of the validity of the local results. In *SR*, each worker proactively anticipates which results may be disqualified because of overlaps with regions from other nodes, and what the respective effects are in each case, and produces a *sufficiently extended* local set of results to guarantee a single round of communication. However, the computation of these extended results imposes additional overhead on each worker, implying potentially significantly higher execution time of the local processes. Eventually, we propose a hybrid algorithm (*HY*), which strikes a balance between the pros and cons of the multi-round and single-round strategies.

In the example described above, we assumed that the proposed regions must not overlap. Other conditions are also possible, e.g., the user might accept overlapping regions as long as their overlap is below a threshold, or even score the overlapping regions differently. Our methods support arbitrarily complex scoring functions, enabling different configurations concerning the overlap.

Summarizing, the paper makes the following contributions:

- We study the top-k best region search problem, identifying and pointing out the challenges that arise when attempting to compute non-overlapping (or partially overlapping) top-k results in a distributed and parallel setting.
- We explore the solution space of the problem, and propose two different distributed algorithms for efficiently solving it. The first is a multi-round algorithm, with the number of rounds bounded by the value of parameter k. The second is a single-round algorithm, which guarantees that the top-k best regions can be identified within a single round of interactions.
- The above two algorithms represent the two extremes in terms of number of rounds, and come with different performance tradeoffs. We thus proceed to explore the configuration space between them, leading to different optimizations, and to a hybrid, fixed-rounds algorithm that combines the benefits of the two.
- We implement all proposed algorithms in Spark and we thoroughly evaluate them using real-world datasets. Our evaluation results demonstrate the scalability and benefits of the proposed algorithms, and analyze their tradeoffs on queries and datasets of different properties.

The rest of the paper is structured as follows. Section 2 summarizes related work. Section 3 presents the problem definition and background. We introduce the multi-round and single-round algorithms in Sections 4 and 5, respectively. Section 6 presents our experimental evaluation, and Section 7 concludes the paper.

2 RELATED WORK

The *Best Region Search* problem has its roots in computational geometry problems relating to intersecting rectangular regions. An intersection graph of rectangles in the 2-dimensional space with sides parallel to the axes is defined in [14]. It is constructed by representing each rectangle with a vertex and connecting two vertices by an edge if the corresponding rectangles intersect. Then, finding the connected components and the maximum clique on this graph is investigated. Moreover, given a set of points in a 2-dimensional space, the problem of finding the placement of a rectangular region P of specified size such that it encloses the maximum or minimum number of points is investigated in [16]. The proposed algorithm is based on an interval tree.

More recently, the problem has attracted interest in spatial databases. Specifically, the *MaxRS* problem has been defined in [6] as follows: given a set of *weighted* points, and a rectangular region R of specified size, find the placement of R that maximizes the *sum of the weights* of all enclosed points. An external-memory algorithm is proposed that is optimal in terms of I/O complexity, based on an external version of the plane-sweep algorithm [13]. Furthermore, the (1-ϵ)-approximate *MaxRS* problem has been studied in [20], which returns a rectangle whose covered weight is at least(1-ϵ) m^*, where m^* is the optimal covered weight and ϵ is an arbitrarily small constant between 0 and 1.

The *MaxRS* problem has also been investigated in a streaming setting. In [1], an algorithm that exploits a graph in a grid index is proposed, using also an upper-bounding technique to avoid unnecessary update computation. A sweep-line based algorithm has also been proposed for the continuous detection of *Bursty Regions* [12]. This is a variation of the continuous *MaxRS* problem, where the burst score of a region is defined over two consecutive sliding windows, and spatial objects in different windows contribute differently to the burst score. Moreover, the continuous maintenance of *range-sum heat maps* over dynamically updating data objects has been studied in [17].

The *Best Region Search* problem generalizes the *MaxRS* problem by allowing the objective score function used to quantify a rectangle's score to be any submodular monotone function over the enclosed points, instead of the sum of their weights [11]. Each point is represented by a fixed-size rectangle centered at it. Then, the best region is identified by finding the maximal intersections of these rectangles. To this end, the input space is partitioned in vertical *slices* that run parallel to the y-axis. Each slice is processed by executing a bottom-up scan over it using a horizontal sweep line to identify so-called *maximal slabs*. The maximal slabs are then processed using a vertical sweep line to identify *maximal regions*. The best region is guaranteed to be centered inside one of those maximal regions. A progressive algorithm for top-k Best Region Search has been proposed in [19]. This algorithm is used in this paper to retrieve local top-k results at each node; hence, it is described in more detail in Section 3.2.

Variants of the *MaxRS* problem have also been considered in road networks. The *length-constrained maximum-sum region* query is introduced in [4]. An approximation algorithm is proposed, utilizing a technique that scales node weights into integers, as well as a heuristic and a greedy algorithm. A unified framework that addresses three variants of *optimal location* queries in road networks is presented in [22]. Given a set of existing facilities and a set of clients, these queries compute the location for a new facility that optimizes a certain cost metric defined based on the distances between the clients and the facilities. Finally, continuous *Best Region Search* in spatial data streams in road networks has been addressed in [5], proposing several pruning strategies and a branch-and-bound algorithm.

3 PRELIMINARIES

In this section, we first provide a formal definition of the problem addressed in this paper, and then we briefly outline the k-BRS algorithm [19], which is exploited in our algorithms to compute the local results at each node.

3.1 Problem definition

Assume a set $\mathcal{D}$ of points in a 2-dimensional space. Let f be a scoring function that assigns a score $f(R)$ to any axes-aligned rectangle R, based on its enclosed points. We assume monotone scoring functions, such that if the contents of a rectangle R' are a superset of those of R, then $f(R') \geq f(R)$. Based on these, the problem of computing the *top-k overlap-aware best regions* can be formally defined as follows:

Problem 1 (Top-k Overlap-Aware Best Regions). *Given a two-dimensional point dataset $\mathcal{D}$, a monotone scoring function f, width and height parameters w, h, and an integer k, the goal is to compute a ranked list of k axis-aligned rectangles R_i with dimensions $w \times h$, such that for each i, j, $1 \leq i < j \leq k$, it holds that:*

- *$f(R_i) \geq f(R_j)$, i.e., the rectangles are ranked in decreasing order of their score,*
- *$R_i \cap R_j = \emptyset$, i.e., R_j does not overlap with any higher ranked result R_i, and*
- *any other rectangle R' either has score $f(R') \leq f(R_k)$, or there exists another rectangle R_i in the top-k list such that $f(R_i) \geq f(R')$ and R' overlaps with R_i.*

Note that this definition can be relaxed to allow results that *partially* overlap, up to a user-specified threshold. In fact, this boolean condition can be generalized even further, by allowing overlaps but penalizing the score of a region by a factor depending on its degree of overlap with a higher-ranked region, as described in [19] using the notion of *marginal gain*. To simplify presentation, in this paper we focus on the boolean case, but it is straightforward to adapt the proposed algorithms to handle cases where the score of a region is penalized based on its degree of overlap with previous results.

Our goal is to compute top-k overlap-aware best regions in a *parallel and distributed* setting. A progressive algorithm for top-k overlap-aware best regions has been presented in [19]. However, it operates in a centralized setting and does not scale to big geospatial datasets containing millions of points. Transferring this solution to a distributed environment is not straightforward, because, as shown in Section 1, deriving global top-k results from the union of local top-k ones leads to incorrect answers. Hence, in this paper, we focus on overcoming this problem.

3.2 The k-BRS algorithm

Next, we briefly outline the *k-BRS* algorithm [19], which is used in this paper as the basis for retrieving local top-k results in each partition. The process is summarized in Alg. 1.

The algorithm starts (Lines 2–3) by constructing a grid with cells of size $w \times h$, i.e., with the same dimensions as the regions to be discovered. For each cell C, an upper bound $UB(C)$ is computed for the score of any region R centered inside C. This is based on the fact that R can enclose at most those points located within C or its neighboring cells. The cells are then inserted into a priority queue Q in decreasing order of their upper bound.

Whenever a cell is extracted from Q (Lines 6–8), it is scanned bottom-up using a horizontal sweep line. This generates a series of *maximal slabs*, each one associated with a respective upper bound, according to which they are inserted into Q. Moreover, the series of slabs leading up to a maximal slab are organized in a *slab tree*, so that these sub-maximal slabs can be visited later if needed. This permits the algorithm to backtrack and explore other results in case the one found is inadmissible due to overlap.

Whenever a maximal slab is extracted from Q (Lines 9–12), it is scanned from left to right using a vertical sweep line. This generates a series of *maximal regions*, each one associated with an upper bound, according to which they are added to Q. In addition, one level of the associated slab tree is traversed, generating one or two new slabs, which now become maximal, and are thus inserted in Q, along with the reduced slab tree, to be processed accordingly. Maximal regions are also associated with a corresponding *region tree*, operating similarly to the slab tree.

Finally, whenever a maximal region is extracted from Q (Lines 13–16), a result is produced, comprising the next region with the highest utility score in the local top-k list. At this point, a check is performed to determine whether this result overlaps with a previously accepted result, and thus determine whether it is admissible or not. New maximal regions are also generated from the region tree and added to Q for future consideration.

Algorithm 1: Local *k-BRS* algorithm.

```
1  Function Local(k)
2      L ← ∅ ; G ← ConstructGrid()
3      Q ← InitializePriorityQueue(G)
4      while |L| < k & |Q| > 0 do
5          E ← Q.nextEntry()
6          if E.type = "Cell" then
7              S ← GenerateSlabs(E)
8              Q.addAll(S)
9          else if E.type = "Slab" then
10             R ← GenerateRegions(E)
11             S ← GetNextSlabs(E.slabTree)
12             Q.addAll(R ∪ S)
13         else if E.type = "Region" then
14             if overlapAcceptable(L, E) then L.add(E)
15             R ← GetNextRegions(E.regionTree)
16             Q.addAll(R)
17     return L
```

4 MULTI-ROUND ALGORITHM

Our first method is an incremental, multi-rounds algorithm (*MR*) that gradually builds the global top-k list of results by retrieving local top-k results from each worker at each round. While aggregating the local top-k results received at the end of each round, the coordinator resolves overlaps, and, if needed, contacts again the affected workers, informing them about the occurred overlaps and asking them for accordingly revised top-k results. This process may take up to k rounds to complete. The steps are outlined in Alg. 2. We start our discussion with data partitioning, and then we explain the querying algorithm.

Data partitioning. Data points are spatially partitioned by a uniform grid with partition width w_p and height h_p (Lines 2–3). Each point with coordinates (x, y) is mapped to partition $P_{i,j}$, with $i = \lceil x/w_p \rceil$ and $j = \lceil y/h_p \rceil$. Data partitioning is performed offline. In Spark, this entails a simple map and groupByKey sequence that parses the original data and generates a new pair RDD with key being the partition id, and value the list of points belonging to this partition. Partitions are held by N nodes (in our case, the Spark workers). Typically, the number of nodes is much less than the number of partitions. It is assumed that $w_p >> w$ and $h_p >> h$, where w and h denote the width and height of the query rectangle.

The node holding each partition is responsible for processing the partition to identify the top-k regions with a top-left corner[1] within the partition. The border cells of the partitions are of particular interest. Notice that a candidate region may intersect two neighboring partitions (e.g., r_8 in Fig. 2). To enable detection of these regions from exactly one partition, and to be able to compute their score, border cells belonging to the top and left borders of each partition are replicated to all partitions they share a border with (for example, the blue-colored cells of $P_{2,3}$, $P_{3,3}$ and $P_{3,2}$ of Fig. 2 are the ones replicated to the node holding $P_{2,2}$). This replication may happen either at query time, when w and h are determined, or may occur offline, assuming that a maximum value for w and h is supported.

Query execution. Upon receiving the query, each partition P is processed in parallel to compute local top-k results (Line 6). The local top-k algorithm (Alg. 1) is used for this purpose, with a

[1] An implementation detail is that, in the *k-BRS* algorithm [19], each input point p is represented by a $w \times h$ rectangle R_p centered at it. The rationale is that any region centered inside R_p encloses p. Now, identifying a result by its top-left corner instead of its center involves a minor adaptation: R_p must have its bottom-right corner (instead of its center) at p. In this way, it still holds that any result R having its top-left corner inside R_p will enclose p.

Algorithm 2: MapReduce implementation of the Multi-round algorithm.

```
1  Function AtCoordinator(k)
       /* Data partitioning                                   */
2      data ← input.map(poi to <partitionIndex, poi>)
3      data ← data.groupByKey(partitionIndex)
4      list globalAns ← ∅
       /* Querying                                            */
5      while |Ans| ≤ k do
6          localAns ← data.map(Local(k, globalAns))
7          roundAns ← localAns.reduce((a, b) ⟹
             AggFunc(a, b, k))
8          globalAns ← globalAns ∪ roundAns
   /* Reduce-based aggregation of results                     */
9  Function AggFunc(res1, res2, k)
10     minAcceptableScore ←
         max(min(res1 scores), min(res2 scores))
11     localAns ← res1 ∪ res2
12     localAns ← SortDesc(localAns)
13     output ← ∅
14     for pos=1 to k do
15         if overlapAcceptable(localAns[pos], roundAns) &
             sc(localAns[pos]) > minAcceptableScore then
16             output ← output ∪ localAns[pos]
17         else
18             break
19     return output
```

minor tweak such that it accepts as input the list of global results computed so far (*globalAns*). This list is taken into consideration in Alg. 1 (Line 14), so that now the overlap condition is checked with respect to $\mathcal{L} \cup globalAns$ instead of $\mathcal{L}$.

The local results per partition are then passed to an aggregation function *AggFunc*, such that a single list of top-k results ends up at the coordinator. The function takes as input two local results, constructs their union, sorts it, and retains the top-k of them, as long as these do not overlap each other. If a non-acceptable overlap is found, or if the minimum score of the two lists surpasses the score of the current region, then the aggregation interrupts and retains only the output produced so far. The logic behind this is that, after any of these cases is observed, any further results returned by the aggregation process are not guaranteed to be correct or complete. This aggregation is associative and commutative. Therefore, it is executed as a reduction in Spark (Line 7), which means that the whole processing (both maps and reductions) is fully parallelized, and the coordinator never constitutes a bottleneck for the algorithm's performance. Finally, after the reduced results are returned to the coordinator, the coordinator merges them with the results of the previous rounds (if any), and loops through the previous map/reduce steps until it collects a total of k results.

As explained previously, we focus on the case that a boolean condition is used to check whether a new result is admissible with respect to existing results based on potential overlap. In the algorithm, this condition is checked by the *overlapAcceptable* function (Line 15). A generalization of this is to penalize the score of overlapping regions using a marginal gain function, as described in [19].

The above algorithm is amenable to configurations and optimizations. First, the size of partitions is a system parameter, which involves the following tradeoff. Very large partitions reduce the area covered by border cells, thereby reducing the probability that overlapping regions of two partitions require more rounds. However, they also lead to scalability problems of the local algorithm (the workers that hold dense partitions run out of memory, swap aggressively, and eventually crash).

Also notice that the algorithm asks for k results per round, per partition. It is however unlikely that the global top-k results come from the same partition, which means that the local nodes (the Spark workers) are likely producing many more results than needed. To reduce this extra effort, the coordinator can instead ask for the top-k' results per partition at each round, with $k' < k$ set either at the beginning, or progressively at each round, in order to fine-tune the tradeoff between the number of rounds and the local effort at the nodes: a high k' value favors towards a lower number of rounds, whereas a lower k' reduces the effort spent by the workers on computing results that are not really useful, because of yet-unknown overlaps. The following lemma formalizes this tradeoff.

Lemma 4.1. *The multi-round algorithm requires at least* $\lceil k/k' \rceil$ *rounds and at most* k *rounds.*

Proof. For the lower bound, consider the case where answers are contained in a single partition. Since each partition generates k' answers at each round, it is not possible to retrieve all results in less than $\lceil k/k' \rceil$ rounds. For the upper bound, consider the top-1 region among all collected local results at a given round. This is guaranteed to be admissible, since it has already been checked locally against the currently existing results (*globalAns*). Hence, at each round, at least one more result is produced. Consequently, the process will terminate after k rounds. □

5 SINGLE-ROUND ALGORITHM AND EXTENSIONS

The multi-round algorithm enables processing of datasets with sizes that could not be practically processed by the centralized algorithm. Still, the iterative nature of the algorithm (and overlapping regions from different partitions) may lead to a potentially high overhead and poor performance. To overcome this drawback, we now introduce a *single-round* algorithm. The algorithm maintains auxiliary information per region that is used during the reduction phase for handing overlapping regions from different partitions. We will then explore the space between the single-round and the multi-round algorithm, proposing a *hybrid* algorithm, and discussing additional optimizations.

5.1 Single-round algorithm

The intuition for the single-round algorithm (*SR*) is the following. When processing each partition, a sufficient number of regions (typically larger than k) will be computed and sent to the coordinator, such that it is guaranteed that the coordinator will have all candidates needed to assemble the global top-k results (discarding non-admissible ones) without further communication to the workers. The challenge is to determine how many, and which, additional regions need to be sent, such that the coordinator is guaranteed to hold sufficient information for extracting the final answer. This challenge arises from the presence of overlaps between candidate regions that are detected in different partitions. For example, in Fig. 2, region r_1 of partition $P_{2,2}$ overlaps with the

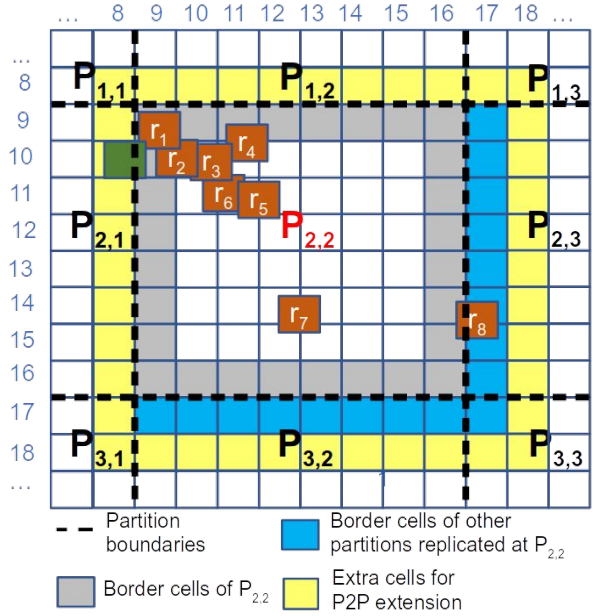

Figure 2: Identified regions within a node's partition. The regions are labeled in decreasing score, i.e., $sc(r_1) > sc(r_2)$.

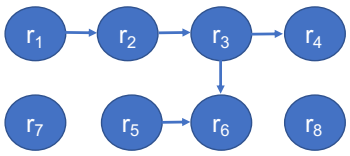

Figure 3: Dependency graph.

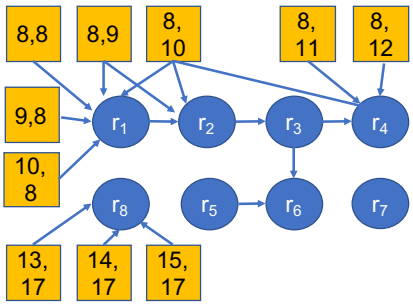

Figure 4: Extended dependency graph.

green region of partition $P_{2,1}$, which has a higher score. Therefore, r_1 will not be in the final results (unless the green region of $P_{2,1}$ intersects with another region of $P_{2,1}$ with higher score), which will make r_2 a possible result. Unfortunately, existence of long chains of such overlaps prohibit local solutions that rely on data replication (e.g., replicating the border cells).

The single-round algorithm addresses this issue by forming a *dependency graph* of the identified candidate regions at each partition. These graphs allow each node to establish a lower bound on the number of regions contained in the local results that will be accepted by the coordinator, if their score is sufficiently high. We start by describing two core components of the algorithm, the *dependency graph* and the *extended dependency graph*.

Definition 5.1 (Dependency graph). Let $\mathcal{R} = r_1, r_2, \ldots$ denote the list of candidate regions detected at a partition P, ordered by their score, i.e., $sc(r_i) \geq sc(r_{i+1})$. Dependency graph $G(\mathcal{R})$ is a directed acyclic graph that contains all candidate regions from $\mathcal{R}$ as vertices, and has an edge between any two regions r_i and r_j if these overlap. The direction of the edge is from the region with the higher score towards the region having the lower score (ties between regions are broken in a consistent manner, by preferring the region with the lowest x coordinate, and then the region with the lowest y coordinate).

As an example, Fig. 3 depicts the dependency graph for the detected regions (r_1 to r_8) in Fig. 2.

Constructing the dependency graph for each partition $P_{x,y}$ is a local process, since it utilizes only the partition's data and the data in the border cells of the neighboring partitions $P_{x+1,y}$, $P_{x,y+1}$, and $P_{x+1,y+1}$ which is replicated in the node holding the partition. Recall, however, that the regions that overlap the border cells of each partition (gray-shaded cells in Fig. 2) may overlap with candidate regions detected in adjacent partitions (e.g., the green region from $P_{2,1}$ which overlaps with r_1 of $P_{2,2}$). The dependencies induced by these regions cannot be depicted in the dependency graph. Thus, to represent these dependencies, the dependency graph is extended by adding *artificial* dependencies for all (locally unknown) candidate regions that potentially overlap with regions detected at neighboring partitions.

Definition 5.2 (Extended dependency graph). The extended dependency graph $\mathcal{X}(\mathcal{R})$ of a partition is a DAG containing: (a) $G(\mathcal{R})$, and (b) for each vertex $v \in G(\mathcal{R})$ with an upper-left corner contained in a border cell (i, j), one vertex v' for each one of the cells adjacent to (i, j) that belongs to a different partition, and an edge pointing from v' to v.

Intuitively, the extended dependency graph encodes the possible dependencies from regions detected at other partitions. Since these regions are unknown at the node constructing the graph, they are represented with the coordinates of the cell that would contain their upper-left corner. For example, region r_1 of partition $P_{2,2}$ has an upper-left corner at cell $(9, 9)$. The adjacent cells of $(9, 9)$ belonging in other partitions are the cells $(8, 8)$, $(8, 9)$, $(8, 10)$, $(9, 8)$, and $(10, 8)$ of partitions $P_{1,1}$, $P_{1,2}$, and $P_{2,1}$. Any region from another partition that has a non-empty overlap with r_1 will have its upper-left corner at one of these cells. Figure 4 depicts the extended dependency graph of partition $P_{2,2}$.

The significance of the extended dependency graph is that it allows to establish an upper bound on the number of regions in the partition that may be excluded from the results due to (chains of) overlapping regions contained in other partitions. Conversely, it allows each partition to derive a lower bound on the number of *safe regions*, i.e., the regions that will not be invalidated from the contents of other partitions, even by long chains of overlaps.

The progressive construction of the extended dependency graph takes place alongside the local algorithm introduced in Alg. 1, and is summarized in Alg. 3. Precisely, we slightly modify the local algorithm in two ways: (a) for every identified region that passes the filter of acceptable overlap (Line 14), the modified algorithm invokes a function *addRegion*, to add the region and its dependencies in the extended dependency graph, (b) it terminates when *addRegion* has detected sufficient safe regions.

In more detail, before processing a partition, the node initializes an empty extended dependency graph $\mathcal{G}$, three sets *Safe*, *Unsafe*, and *detectedRegions* for keeping the regions, a set $\mathcal{M}$ for keeping the identified dependencies, and a counter for the number of identified safe regions *safeCnt*. Upon invocation, *addRegion*

checks the new region to decide whether it is safe, unsafe, or inadmissible, and to maintain a lower bound for the number of safe regions (variable *safeCnt*).[2] This process is governed by a set of rules. Precisely, for a candidate region r_x, the node checks for three different cases:

Case 1: r_x does not overlap with other identified regions (either safe or unsafe), or with the border cells. r_x is a safe region, i.e., it will be included in the final answer if its score is sufficiently high. We increase *safeCnt* by one, and add it in the *Safe* set and in the extended dependency graph with no dependencies. r_7 is one such example from Fig. 2.

Case 2: r_x overlaps with a safe region r_y, which is already included in the Safe *set.* Since r_y is already identified, its score is higher than the score of r_x. Therefore, node discards r_x and continues. As an example, consider r_5 and r_6 from Fig. 2 – since r_5 is safe and has a higher score than r_6, the latter will be discarded.

Case 3: r_x overlaps with a border cell, or with an unsafe region r_y included in the Unsafe *set.* In this case, r_x is also unsafe. r_x is added in the *Unsafe* set, and in the extended dependency graph, with dependency from r_y and/or the adjacent cells of the other partitions. Let $\mathcal{D}(r_x)$ denote the set of all dependencies for r_x contained in the dependency graph after this process. We have two further sub-cases:

Case 3a: If none of the regions contained in $\mathcal{D}(r_x)$ is included in $\mathcal{M}$, none of these has been considered before. Therefore, either r_x or a region from $\mathcal{D}(r_x)$ will be part of the top-k regions, assuming that their score is sufficiently high. To capture this, r_x and the regions in $\mathcal{D}(r_x)$ are added to $\mathcal{M}$, and counter *safeCnt* is increased by one. From Fig. 2, r_1 and r_8 will belong in this category. Out of these, the coordinator will later detect that r_1 has an overlap with the green-marked region detected at partition $P_{2,1}$ (i.e., the green region will be included in the results instead), whereas r_8 will be included in the results if is score is sufficiently high (hence the increase on *safeCnt* for both of them). Another example is region r_3, which depends on r_2: since r_2 overlaps with r_1 which will be included in $\mathcal{M}$, r_2 will not be included in $\mathcal{M}$. This means that r_3 and its dependencies will be included in $\mathcal{M}$, and *safeCnt* will be increased by one.

Case 3b: If any of the regions contained in $\mathcal{D}(r_x)$ is already included in set $\mathcal{M}$, then *safeCnt* is not increased and r_x is not added in $\mathcal{M}$. As an example, consider regions r_2 and r_1 from Fig. 2. Region r_1 will be identified first, and already contained in $\mathcal{M}$ when r_2 is identified. Therefore, identification of r_2 will not lead to an increase of *safeCnt*.

After completion of *addRegion*, the control returns to the modified local algorithm, which breaks the loop when *safeCnt* reaches k. The local results for the partition are contained in *detectedRegions*, and the extended dependency graph is saved in $\mathcal{G}$.

Similar to the case of the multi-round algorithm, implementation of the single-round algorithm over Spark requires a way to hierarchically merge/reduce the local results (the dependency graphs) in order to get a final dependency graph with no further artificial dependencies in the coordinator, for extracting the final answer. Conceptually, the required merging involves the following steps: (1) we form the union of the two graphs, (2) if the two graphs share a border, we identify the artificial dependencies in the union graph that can now be eliminated, or replaced by dependencies on real regions, (3) we propagate the effect of each dependency elimination or replacement (e.g., switching a region from unsafe to safe), by performing a depth-first traversal of the graph starting from the affected region, and, (4) we reduce the graph by finding the score of the top-k safe region, and removing all regions with a lower score.

[2]The problem of counting the number of safe regions is not equivalent to the problem of actually detecting the safe regions. Addressing the latter problem typically leads to less regions marked as safe, which translates to larger dependency graphs and degradation of the algorithm's performance.

Algorithm 3: Single-round algorithm – progressive construction of extended dependency graph.

```
1  G ← ∅ , Safe ← ∅ , Unsafe ← ∅ , detectedRegions ← ∅ ,
     M ← ∅ , safeCnt ← 0
2  Function addRegion(region)
3      if region does not overlap with any region in Safe,
         Unsafe, and border cells then
4          G.addNode(region)
5          Safe ← Safe ∪ region
6          safeCnt ← safeCnt + 1
7      else if region overlaps with a region in Safe then
8          continue
9      else if region overlaps with a region in Unsafe or
         border cells then
10         G.addNode(region)
11         G.addDependencies(region)
12         Unsafe ← Unsafe ∪ region
13         if region dependencies is included in M then
14             continue
15         else
16             safeCnt ← safeCnt + 1
17             M ← M∪( region dependencies)
18     detectedRegions ← Safe ∪ Unsafe
```

Notice that the effect of the propagation step (step 3) relies on the same rules that we introduced earlier to determine whether a region is safe, unsafe, or can be safely removed. Therefore, a simple way to implement the above process is to initialize an empty graph, and keep adding the regions of the two partitions in descending order of score by invoking *addRegion* function of Alg. 3. The adding process can stop when *safeCnt* exceeds k.

Theorem 5.3. *The final dependency graph produced by the algorithm only contains safe regions, which are the answer to the user's query.*

Proof sketch: Since the final dependency graph contains the input from all partitions, it will no longer contain artificial dependencies, which cause the unsafe regions (notice that inadmissible regions are not included in the graph – see lines 7-8 of Alg. 3). Also, regions are added in this graph in order of descending score, and the graph is completed as soon as the graph contains k nodes. The proof can be formalized with induction. □

The discussed algorithm is amenable to several optimizations and extensions for reducing network load and/or wallclock time. We present these in the following.

5.2 Spatial-aware tree-based aggregation

The aggregation function at the single-round algorithm is commutative and associative, similar to the multi-round algorithm. Therefore, it can be expressed in Spark as a reduction, enabling the Spark engine to fully distribute this part of the algorithm as well. Notice however that Spark does not take spatial proximity of partitions into account when executing the reducers. As

Algorithm 4: MapReduce implementation of Single-round algorithm. Parameters res_1, res_2, res_3, ... represent the results grouped by the same key.

```
1  Function AtCoordinator(k)
2      data ← input.map(poi to <partitionIndex, poi>)
3      data ← data.groupByKey(partitionIndex)
       /* first compute the local results per
          partition                                  */
4      localAns ← data.map(modified Alg1.Local(k))
       /* recursive hierarchical aggr.               */
5      for i=1 to treeHeight do
6          localAns ←
             localAns.map(recomputePartitionIndexes(b))
             .groupByKey(partitionIndex).map(mergeRegions(k))
7      Ans ← localAns
8  Function mergeRegions(<res_1, res_2, res_3, ...>, k)
9      G ← ∅
10     pos ← 0
11     unionRes ← sortDesc(res_1 ∪ res_2 ∪ res_3 ...)
12     while (G.safeCnt < k) do
13         G.addRegion(unionRes.get(pos))
14         pos ← pos + 1
15     return G
```

such, for a 16-partitions example of Fig. 5, one possible reduction order could be the following (where $\oplus$ denotes the reduction/aggregation function of the results): $(((1 \oplus 16) \oplus (5 \oplus 11)) \oplus (9 \oplus 7)) \oplus (4 \oplus 13) \ldots$. This random-order reduction precludes a core optimization, since merging distant partitions (e.g., partitions 1 and 16) does not help the reduction function to increase the per-partition number of safe regions. Ideally, we could reduce partition results in a proximity-aware hierarchical approach (i.e., order $((1 \oplus 2) \oplus (3 \oplus 4)) \ldots$). Then, the merging process would, for example, exploit the partial results of partition 1 to mark the partial results of partition 2 that border with partition 1 as safe, or to exclude them, depending on their overlap and scores.

To exploit this observation, we enforce an explicit reduction order to Spark by introducing a hierarchical structure of aggregations. Fig. 5 depicts a small example with a hierarchical aggregation of 16 partitions. First, we apply a Z-order curve to assign an id to all partitions. Notice that close-by ids now mostly have spatial proximity. Then, each partition with id *id* assumes place in the hierarchy as the child of parent with id $\lceil id/b^2 \rceil$, where b^2 is the desired fan-out of our hierarchy. In our example, partitions with id 1 to 4 become children of a parent with id 1, i.e., their results will be merged together with a reduction in order to receive their parent node 1. The aggregation process continues recursively, until we reach to a single parent.

In concrete terms, let P denote the total number of partitions, and $p = b^i$ be the smallest power of $b \in \mathbb{N}^+$ that is greater than or equal to P, for a user-configured value of b. The space is recursively partitioned to p tiles of equal size ($b \times b$ tiles at each recursion), giving rise to a (b^2)-ary aggregation tree. This aggregation tree is represented in Spark with a chain of *map* and *groupByKey* sequences (Alg. 4, lines 5-7). The *map* functions determine the place of the tile in the hierarchy (essentially the id of the parent reducer), and the *groupByKey* functions bring together the results of the neighboring partitions, for the merging algorithm to run (function *mergeRegions*). The process continues until the root of the hierarchy, and the results are finally collected by the coordinator. Again, this whole process is executed in a decentralized fashion, and the coordinator only receives the final results. The value b that determines the number of levels in the tree hierarchy is important. A very small value of b, e.g., 2, leads to many levels, introducing significant synchronization overhead in Spark. At the other extreme, very large b values lead to low parallelism and large memory requirements at the nodes.

5.3 Peer-to-peer communication

Up to now, construction of the extended dependency graph did not exploit information regarding the neighboring partitions. As such, all artificial dependencies from neighboring partitions were set in a pessimistic way to dominate the local regions, leading to potentially long dependency chains (e.g., Fig. 6 contains a dependency chain that includes r_1 to r_4, because r_1 is unsafe). To alleviate this issue, we introduce moderate communication between the nodes, for exchanging key statistics regarding their neighboring partitions (e.g., the exact highest score of any region within all border cells, the maximum score of all regions in the border cells, or even the individual regions identified in the border cells and their scores). These statistics or results can then be used during the local algorithm to replace or tighten the artificial dependencies, i.e., to upper-bound the score of the artificial vertices in the extended dependency graph. The amount and type of data to exchange between nodes can lead to a spectrum of configurations that enable a tradeoff between the number of safe regions and the computational overhead and network volume.

This P2P-style communication, though, is not natural in Spark's MapReduce paradigm. One way to simulate it within Spark is by introducing a preliminary round, where each node runs a fast preparation step on each of its partitions and computes the desired statistics. However, this extra round introduces a high synchronization overhead for Spark. Instead, we choose to extend the idea of replication of border cells we described earlier, and assign to each node the additional task of first extracting coarse-grained statistics for the border cells of its eight neighboring partitions, prior to analyzing its own partition. In this way, the code for statistics extraction can be fully integrated in the map process of the single-round algorithm, without requiring an additional round of synchronization.

Particularly, we extend the data held for each partition by one row/column in each direction. In the example of Fig. 2, the yellow-colored cells of the neighboring partitions will also be copied to the node holding $P_{2,2}$, in addition to the blue-colored cells.[3]. When the processing for $P_{2,2}$ is initiated, the first task of the node is to process these replicated border cells and extract these statistics that will be subsequently used for constructing the extended dependency graph. We examined two levels of granularity for these statistics (in decreasing granularity):

- Executing the local algorithm on these cells to extract all contained regions – possibly overlapping each other. The scores of these regions will be an upper bound of the true scores. Therefore, these scores can be used to remove some of the dependency relations of the artificial dependencies in the extended dependency graph, i.e., if the region of the local partition has a higher score than the region in the replicated area (Fig. 6 (left)).

[3]Notice the asymmetry between the replicated cells from the left and the replicated cells from the right of $P_{2,2}$ (similarly for the cells above and below) This happens because the node holding $P_{2,2}$ is also responsible for finding the regions that overlap $P_{2,3}$, but not the ones overlapping $P_{2,1}$

Algorithm 5: Hybrid algorithm – region merging. Parameters $res_1, res_2, res_3, \ldots$ represent the results grouped by the same key.

```
1 Function mergeRegions(<res_1, res_2, res_3, ...>, k)
2     minAcceptableScore ←
        max(min(sc(res_1)), min(sc(res_2)), min(sc(res_3)), ...)
3     G ← ∅ , pos ← 0
4     unionRes ← sortDesc(res_1 ∪ res_2 ∪ res_3 ...)
5     while G.safeCnt < k & sc(unionRes.get(pos)) >
        minAcceptableScore do
6         G.addRegion(unionRes.get(pos))
7         pos ← pos + 1
8     return G
```

- For each border cell of a neighboring partition with coordinates (i, j), compute the score of the $2\epsilon \times 2\epsilon$ region consisting of cells $(i, j), (i+1, j), (i, j+1), (i+1, j+1)$. This does not require executing the local algorithm, since the exact region boundaries are known. We only need to compute the score function for the region, and use it as an upper bound for the cell. (Fig. 6 (right)).

The first approach produces the tightest upper bounds, but preludes execution of the full local algorithm on the border cells, thereby adding non-negligible time overhead. The second approach produces weaker upper bounds for the scores, but it is much more efficient. In our experiments, the second approach provided the best tradeoff in terms of overall execution time (and, thus, is the only one reported).

5.4 Hybrid algorithm

The single-round algorithm is very conservative, requiring k safe regions to be obtained from every partition. In practice, this leads to additional load for extending the local graphs, that could otherwise be avoided. We next describe a hybrid algorithm (*HY*), which covers the space between the single-round and multi-round algorithms, balancing the number of rounds and the number of results expected from each round. The intuition is that we can execute the single-round algorithm, but now requesting a smaller number of safe regions $k' << k$ per partition, aggregate the partial results, and then progressively ask for more results only from the partitions from which we already consumed at least one safe region. The partial results collected per round are a sorted subset of the final results (at least the next k' answers, but typically much more), and can be presented progressively to the user.

Similar to the case of the multi-round algorithm, the aggregation function (see Alg. 5) needs to stop accepting more data when it can no longer guarantee correctness of results. Correctness of results is guaranteed by establishing a bound for the region scores, above which all regions are guaranteed to be complete. When aggregating partial results of two partitions, say, res_1 and res_2, this score bound is simply the maximum of the two minimum scores in each of the partial results (Line 2). The intuition behind this bound is that the reducer does not have sufficient information about regions with lower score for one of the two, but all possible solutions (safe and unsafe regions) for higher scores are already included in the individual dependency graphs. Notice that this bound is naturally moved up in the hierarchy, as the aggregate results are pushed to parent reducers, and regions with lower scores are filtered out.

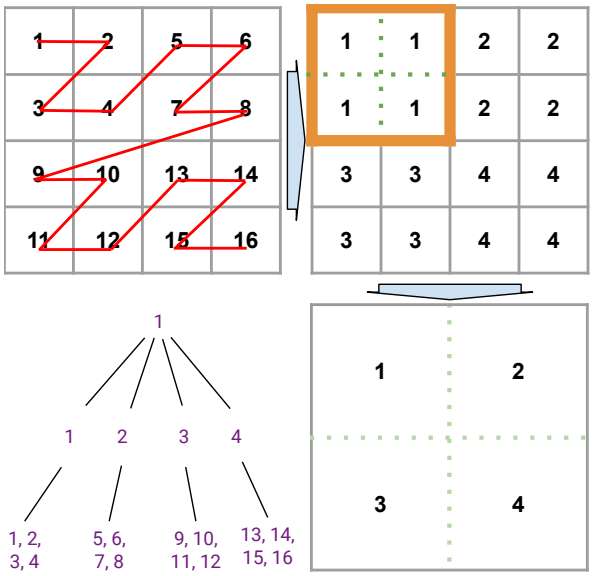

Figure 5: Hierarchical aggregation in 4×4 partitions with $b = 2$ by applying Z-order. The tree depicts the aggregation hierarchy, from 16 partitions to a single result.

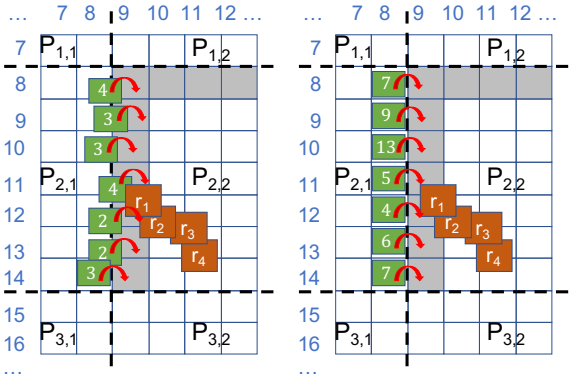

Figure 6: Exchanging region scores (left) or upper bounds for border cells (right).

6 EXPERIMENTAL EVALUATION

We have conducted an experimental evaluation to compare the performance of our proposed algorithms, investigate their scalability, and explore the impact of the various parameters.

6.1 Experimental setting

Datasets. We conducted our experiments using a real-world dataset comprising 64 million records representing Points of Interest from OpenStreetMap[4] and geolocated photos from Flickr[5] worldwide. We have mapped these to 26 million distinct locations (points). We have assigned a weight (score) to each point denoting the number of records (POIs or photos) mapped to it.

Experimental setup. The experiments are executed on a cluster of 11 nodes, each with 30 GB of RAM. Ten of the nodes are configured as workers, and the eleventh is indicated as the master/coordinator. The cluster runs Spark 2.4.3, and Hadoop HDFS

[4]https://www.openstreetmap.org
[5]https://www.flickr.com

Parameters	Values
number of points $\|\mathcal{D}\|$	5, 10, 15, 20, **26** million
number of nodes $\|\mathcal{N}\|$	1, 2, 4, 6, 8, **10**
top-k regions	50, 100, 200, **300**, 400, 500
region width and height ϵ	0.00025, 0.0005, 0.00075, **0.001**, 0.00125, 0.0015 (from approximately $27x27m^2$ to $162x162m^2$)
k' – Multi-round	3, 5, 10, 20, 30, 40, 50, 60, **300**
k' – Hybrid	3, 4, 5, 6, **10**, 15

Table 1: Experimental parameters (default value is bold).

2.9.1. The input file is distributed across the 10 workers, with replication factor set to one.

Implementation and configuration. All algorithms are implemented in Scala. Our implementations include by default the hierarchical aggregation for the hybrid algorithm and the corresponding commutative and associative reduction function for the single-round algorithm, since the coordinator becomes a bottleneck (and crashes for some parameters) otherwise.

We partitioned the input points into a uniform grid with dimensions 20,000 × 20,000, leading to partitions of size approximately $2km \times 2km$. The most densely populated partition contains approximately 18,000 points. The value of b in the tree-based aggregation was set to 8. We found these values to provide consistently good performance without creating bottlenecks. Using larger partitions creates bottlenecks during the local execution of the algorithm to the workers, whereas a significantly larger base leads to bottlenecks during the aggregation step (the reducer needs to aggregate too many partial results, which may lead to crashes around dense areas). With respect to all other parameters, unless otherwise mentioned, we use the default values shown in Table 1. The region width and height ϵ is measured in degrees.

6.2 Multi-round vs Single-round

We start by comparing the performance of the multi-round (*MR*) and the single-round (*SR*) algorithms.

We first vary the number of requested top-k regions from 50 to 500. Fig. 7 shows the results – wallclock time for both algorithms on the left Y axis and number of rounds for *MR* on the right Y axis. We see that the value of k has only a minor influence on the performance of *SR*. Conversely, the execution time of *MR* increases as k increases, eventually becoming around 40 times higher than that of *SR*. This stark difference is attributed to the number of rounds required by *MR*. Indicatively, for $k = 500$, *MR* requires 58 rounds, while *SR* only requires one. Of course, one round of execution for *SR* is more time consuming than for *MR* due to the extra time spent on creating and maintaining the dependency graph and continuing the computation of results until k safe ones are found. Indeed, we can observe that the one round of *SR* takes around 700 seconds to complete, whereas each round of *MR* takes on average 400 seconds. Yet, this difference is very small to compensate the extra overhead incurred by the high number of rounds required by *MR*.

In our second experiment (Fig. 8), we fix k to 300 and vary the width and height ϵ of the regions to be discovered. We see that the performance of both algorithms is affected by ϵ. These results are consistent to [19], which shows that the centralized algorithm (being used to retrieve the local top-k results in each partition) becomes slower as ϵ increases. Thus, they also exemplify the importance of having a parallel and distributed solution to achieve scalability. Still, since *SR* only needs to run the local algorithm once, it scales much better than *MR*, which is again affected by

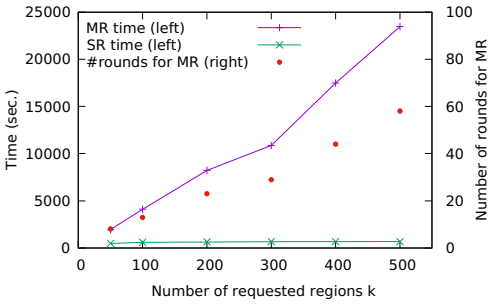

Figure 7: Performance of *MR* and *SR* for varying k.

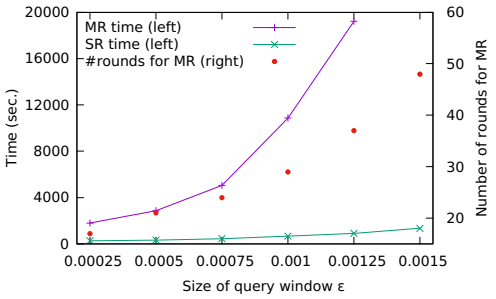

Figure 8: Performance of *MR* and *SR* for varying ϵ (region width and height).

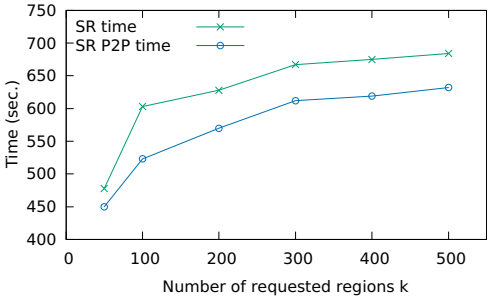

Figure 9: Effect of P2P extension on *SR* for varying k.

the need to execute multiple rounds, eventually requiring one order of magnitude more time compared to *SR*. Indicatively, for $\epsilon = 0.0015°$, *SR* takes 1352 seconds, while *MR* requires 44383 (for illustration reasons, the latter point is omitted in the plot).

We also evaluate the impact of the P2P extension on the performance of *SR*. Figure 9 plots the execution time of the algorithm with and without this extension. As expected, the extension always saves time, since it enables each node to create smaller dependency graphs per partition, and it has very low overhead. Indicatively, the average size of the dependency graph per partition with the P2P extension was reduced by approximate 40% for $k = 50$ and by 55% for $k = 500$. Since P2P consistently improves the performance of *SR*, it is applied in all remaining results.

6.3 Limiting the number of local results

We now examine how limiting the number of local results affects the performance of the algorithms. Recall that both *MR* and the Hybrid algorithm (*HY*) support requesting $k' < k$ results or safe regions from each partition at each round.

Figure 10 plots the execution time (left Y axis) and number of rounds (right axis) for *MR* for various k' values. Clearly, the

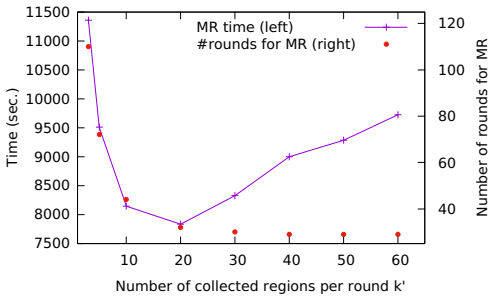

Figure 10: Execution time of *MR* for varying k'.

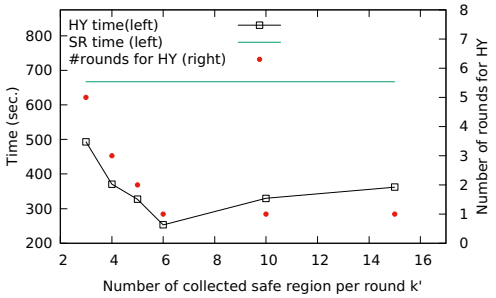

Figure 11: Execution time of *HY* for varying k' (with *SR* included for reference).

value of k' has significant influence on performance. As expected, setting a very low k' leads to a high number of rounds, which translates to excessive synchronization overhead and to longer execution times. Increasing k' until 20 reduces total time by limiting the time spent by the workers on computing the local results (which, in most cases, are anyway not needed by the coordinator). Since this time reduction is per round, and many rounds are required, the overall performance increase is significant. On the other hand, increasing k' beyond a certain point no longer reduces the number of rounds, but increases the average execution time per round, and consequently, the total execution time of the algorithm. Interestingly, already for $k' > 20$, execution time increases with the value of k', i.e., the performance gain because of reduction of the number of rounds is overshadowed by the additional time required for computing more local results, and merging them in the reduction phase.

Figure 11 depicts the influence of the respective parameter k' for *HY*. In this case, k' denotes the number of safe regions requested per partition. We see that a very low value of k' raises the need for additional rounds, since the appearance of non-admissible results prevents the coordinator from obtaining a valid top-k at the first place. However, with k' equal to 6, *HY* already completes in a single round and achieves the optimal performance, which is around one third of the baseline performance of *SR* (this would correspond to the worst performance of *HY*). It is also worth noticing that, in absolute time difference, *HY* is not as sensitive as *MR* with respect to a higher-than-optimal number of local results (cf., a high k' value in Fig 10). For example, even when collecting 15 safe regions per round (2.5 times more than the optimal), the difference in time is only around 100 seconds, compared to around 2000 seconds for *MR*. This is attributed to the fact that a sub-optimal value of k' affects much fewer rounds in *HY*, compared to a sub-optimal value of k' for *MR*. Thus, when tuning *HY*, it is relatively easier to find a good value for k'.

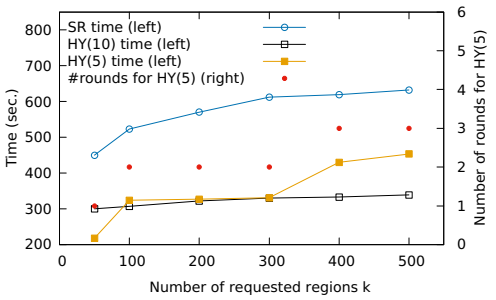

Figure 12: Comparison of *SR* and *HY* for varying k. *HY*(5) and *HY*(10) correspond to *HY* with $k' = 5$ and $k' = 10$, respectively. *HY*(10) always completes in a single round.

6.4 Comparing *SR* and *HY*

We now evaluate the performance of *SR* and *HY* for varying k and ϵ. We omit *MR* in these experiments since, as already indicated in previous results, it is always outperformed significantly by *SR*.

Figure12 presents the execution time of both algorithms, for different values of k, and for two hybrid executions with $k' = 5$ and $k' = 10$, noted as *HY*(5) and *HY*(10), respectively. The plot also includes the number of rounds for *HY*(5) (right Y axis). The number of rounds for *HY*(10) (and for *SR*) is always 1, and therefore it is omitted. As expected, the value of k brings a noticeable increase on the cost of *SR*, since it leads to larger dependency graphs. For *HY*(5), a higher k leads to more rounds, which also causes an increase in execution time. Nevertheless, *HY*(5) still outperforms *SR*, because due to the low value of k' the additional rounds are much faster compared to one round of *SR*. Also, *HY*(10) exhibits a very mild increase of the execution time (300 seconds for $k = 50$, compared to 339 seconds for $k = 500$). Since Hybrid(10) always takes 1 round, even for k=500, this increase is solely attributed to the extra cost during the hierarchical reduction: reducers need to maintain longer lists, and pass these lists to their parent nodes in the tree hierarchy. Still, this increase is negligible. Therefore, it is better to opt for a slightly higher value of k', in order to avoid the risk of running multiple rounds.

Figure 13 shows the execution time of *SR* and *HY*(10) for varying ϵ. Both algorithms exhibit a similar trend, but with *HY* consistently outperforming *SR*, and with the absolute difference between the execution time growing as ϵ increases. Detailed profiling on this result reveals that the additional time is exclusively spent on the local algorithm. As ϵ increases, the local algorithm spends more time for generating *each* result. Since *SR* has to compute 300 results in each partition, whereas *HY*(10) only 10, the total difference between the execution time of the two algorithms increases when the time spent per result is higher.

6.5 Scalability

Our last set of experiments focuses on investigating the scalability of the three algorithms, when varying the size of the dataset or number of executor nodes.

Given the original dataset, containing 26 million distinct points, we derive datasets of size from 5 to 20 million points by applying uniform sampling. Fig. 14 plots the execution time of the three algorithms. We observe that *MR* is the slowest algorithm in all cases, while *HY* performs better than *SR*. Moreover, *MR* demonstrates poor scalability compared to the other two, which exhibit similar performance, with *HY* scaling slightly better.

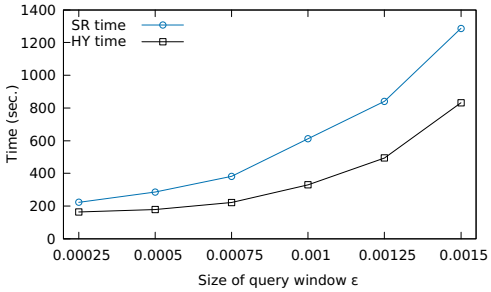

Figure 13: Comparison of *SR* and *HY* for varying ϵ. The number of rounds for *HY* is always one in this set of experiments (for k' set to 10).

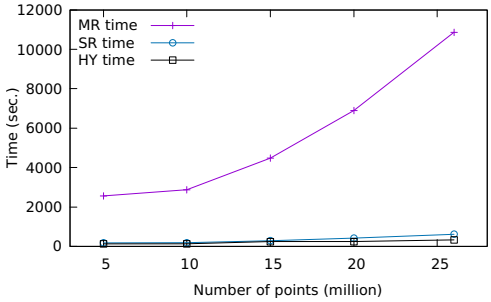

Figure 14: Execution time of *MR*, *SR*, *HY* for varying the number of points.

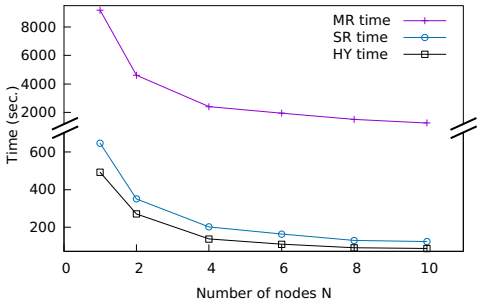

Figure 15: Execution time of *MR*, *SR*, *HY* for varying the number of nodes.

In our second experiment (Fig. 15), we vary the number of nodes from 1 to 10. Since it is not possible to process the whole dataset (26 million points) in a single node due to memory limitations, we sample 5 million points as input (i.e., the largest dataset size executable in a single node). As shown, addition of extra nodes decreases overall execution time for all algorithms, with *SR* and *HY* exhibiting linear speedup.

6.6 Summary

The experiments show that *SR* substantially outperforms *MR* in all cases, indicating that the extra cost incurred by multiple rounds dominates that for retrieving a sufficiently larger number of local results to ensure the construction of the global top-k in a single round. Overall, *HY* is the most efficient algorithm, indicating that in practice it suffices to compute just a few more than k local results to ensure that the coordinator can assemble the correct top-k list, despite any inadmissible local results.

Moreover, both algorithms *MR* and *HY* can benefit from setting the parameter k' to a much lower value than k (e.g., around 0.1 $\times$ k). This significantly increases the efficiency of *HY* due to the drastic reduction of the cost of local processing, while still obtaining the global top-k results in one or two rounds.

7 CONCLUSIONS

We have presented the first scalable algorithms for addressing the *top-k Best Region Search* problem. Our approach relies on distributing the dataset and the expensive computational part over cluster resources, thereby allowing the processing of large datasets in parallel. Starting from a multi-round algorithm, we proceed to devise one that requires a single round and is more efficient by one order of magnitude. Then, we also propose a hybrid algorithm, which further reduces computational time by a factor of two. Our future work focuses on continuous algorithms, for maintaining the answer in dynamic datasets, as well as approximation techniques, for further reducing the computational cost when small, bounded errors can be tolerated.

ACKNOWLEDGMENTS

This work was partially funded by the EU H2020 project SmartDataLake (825041).

REFERENCES

[1] D. Amagata and T. Hara. A general framework for MaxRS and MaxCRS monitoring in spatial data streams. *ACM TSAS*, 3(1):1:1–1:34, 2017.
[2] L. Anselin. Local indicators of spatial association—lisa. *Geographical analysis*, 27(2):93–115, 1995.
[3] X. Cao, G. Cong, T. Guo, C. S. Jensen, and B. C. Ooi. Efficient processing of spatial group keyword queries. *ACM Trans. Database Syst.*, 40(2):13:1–13:48, 2015.
[4] X. Cao, G. Cong, C. S. Jensen, and M. L. Yiu. Retrieving regions of interest for user exploration. *PVLDB*, 7(9):733–744, 2014.
[5] Z. Chen, Q. Yuan, and W. Liu. Monitoring best region in spatial data streams in road networks. *Data Knowl. Eng.*, 120:100–118, 2019.
[6] D. Choi, C. Chung, and Y. Tao. A scalable algorithm for maximizing range sum in spatial databases. *PVLDB*, 5(11):1088–1099, 2012.
[7] D. Choi, J. Pei, and X. Lin. Finding the minimum spatial keyword cover. In *ICDE*, pages 685–696, 2016.
[8] K. Deng, X. Li, J. Lu, and X. Zhou. Best keyword cover search. *IEEE Trans. Knowl. Data Eng.*, 27(1):61–73, 2015.
[9] A. Eldawy and M. F. Mokbel. The era of big spatial data: A survey. *Foundations and Trends in Databases*, 6(3-4):163–273, 2016.
[10] M. Ester, H. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *KDD*, pages 226–231, 1996.
[11] K. Feng, G. Cong, S. S. Bhowmick, W. Peng, and C. Miao. Towards best region search for data exploration. In *SIGMOD*, pages 1055–1070, 2016.
[12] K. Feng, T. Guo, G. Cong, S. S. Bhowmick, and S. Ma. SURGE: continuous detection of bursty regions over a stream of spatial objects. In *ICDE*, pages 1292–1295, 2018.
[13] M. T. Goodrich, J.-J. Tsay, D. E. Vengroff, and J. S. Vitter. External-memory computational geometry. In *FOCS*, pages 714–723, 1993.
[14] H. Imai and T. Asano. Finding the connected components and a maximum clique of an intersection graph of rectangles in the plane. *J. Algorithms*, 4(4):310–323, 1983.
[15] S. Luo, Y. Luo, S. Zhou, G. Cong, and J. Guan. Distributed spatial keyword querying on road networks. In *EDBT*, pages 235–246, 2014.
[16] S. C. Nandy and B. B. Bhattacharya. A unified algorithm for finding maximum and minimum object enclosing rectangles and cuboids. *Computers & Mathematics with Applications*, 29(8):45–61, 1995.
[17] J. Qi, V. Kumar, R. Zhang, E. Tanin, G. Trajcevski, and P. Scheuermann. Continuous maintenance of range sum heat maps. In *ICDE*, pages 1625–1628, 2018.
[18] C. Sheng and Y. Tao. New results on two-dimensional orthogonal range aggregation in external memory. In *PODS*, pages 129–139, 2011.
[19] D. Skoutas, D. Sacharidis, and K. Patroumpas. Efficient progressive and diversified top-k best region search. In *SIGSPATIAL*, pages 299–308, 2018.
[20] Y. Tao, X. Hu, D. Choi, and C. Chung. Approximate MaxRS in spatial databases. *PVLDB*, 6(13):1546–1557, 2013.
[21] R. C. Wong, M. T. Özsu, P. S. Yu, A. W. Fu, and L. Liu. Efficient method for maximizing bichromatic reverse nearest neighbor. *PVLDB*, 2(1):1126–1137, 2009.
[22] X. Xiao, B. Yao, and F. Li. Optimal location queries in road network databases. In *ICDE*, pages 804–815, 2011.