In-memory Distributed Spatial Query Processing and Optimization

Mingjie Tang[†], Yongyang Yu[†], Walid G. Aref[†], Ahmed R.Mahmood[†], Qutaibah M. Malluhi[‡],Mourad Ouzzani[⋆]

[†]*Purdue University,* [‡]*Qatar University,* [⋆]*Qatar Computing Research Institute*

{tang49,yu163, aref, amahmoo}@cs.purdue.edu, qmalluhi@qu.edu.qa, mouzzani@qf.org.qa

ABSTRACT

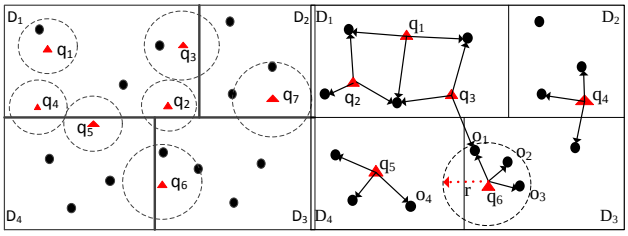
Due to the ubiquitous use of spatial data applications and the large amounts of spatial data that these applications generate and process, the call for scalable spatial query processing is a major challenge. In this paper, we present new techniques for spatial query processing and optimization in an in-memory and distributed setup to address scalability. More specifically, we introduce new techniques for handling query skew and optimize communication cost. We propose a query scheduler and a distributed query optimizer that use a new cost model to optimize the cost of spatial query processing in this in-memory distributed setup. The scheduler and query optimizer generate query execution plans that minimize the effect of query skew. The query scheduler employs new spatial indexing techniques based on Bloom filters to forward queries to the appropriate local sites. Each local computation node is responsible for optimizing and selecting its best local query execution plan based on the indexes and the nature of the spatial queries in that node. All the proposed spatial query processing and optimization techniques are prototyped inside Spark, a distributed main-memory computation system. The experimental study is based on real datasets and demonstrates that distributed spatial query processing can be enhanced by up to an order of magnitude over existing in-memory and distributed spatial systems.

Categories and Subject Descriptors

H.3.4 [**Systems and Software**]: Spatial database

1. INTRODUCTION

Spatial computing is becoming increasingly important with the proliferation of mobile devices. Meanwhile, the growing scale and importance of location data have driven the development of numerous specialized spatial data processing systems, e.g., SpatialHadoop [9], Hadoop-GIS [3] and MD-Hbase [17]. By taking advantage of the power and cost-effectiveness of MapReduce, these systems typically outperform spatial extensions on top of relational database systems by orders of magnitude [3]. MapReduce-based systems allow users to run spatial queries using predefined high-level spatial operators without worrying about fault tolerance or computation distribution. However, these systems have the following two main limitations: (1) They do not leverage the power of distributed memory, and (2) They are unable to reuse intermediate data [26]. Nonetheless, data reuse is very common in spatial data processing. For example, spatial datasets, e.g., Open Street Map (OSM, for short, >60G) and Point of Interest (POI, for short, >20G) [9], are usually large. It is unnecessary to read these datasets continuously from disk (e.g., using HDFS [19]) to respond to user queries. Moreover, intermediate query results have to be written back to HDFS, thus directly impeding the performance of further data analysis steps.



(a) Spatial Range Join (b) kNN Join

Figure 1: Illustration of spatial join operators. Circles centered around the triangle focal points form one dataset, and the black dots form the second dataset. (a) Spatial range join returns the (dot,triangle) pairs when the dot is inside the circle. (b) kNN join returns (triangle,dot) pairs when the dot is among the 3 nearest dots to the triangle.

One way to address the above challenges is to develop an efficient execution engine for large-scale spatial data computation based on an in-memory computation framework (in this case, Spark [26]). Spark is a computation framework that allows users to work on distributed in-memory data without worrying about data distribution or fault-tolerance. Recently, various Spark-based systems have been proposed for spatial data analysis, e.g., SpatialSpark [2], GeoSpark [25], Magellan [1], Simba [24] and Location-Spark [21].

Although addressing several challenges in spatial query processing, none of the existing systems is able to overcome the computation skew introduced by spatial queries. "Spatial query skew" is observed in distributed environments during spatial query processing when certain data partitions get overloaded by spatial queries. Traditionally, distributed spatial computing systems (e.g., [9, 3, 2]) first learn the spatial data distribution by sampling of the input data. Afterwards, spatial data gets evenly partitioned into equi-sized partitions. For example, in Figure 1, the data points with dark dots are evenly distributed into four partitions. Given the partitioned data, consider the spatial range and kNN joins that serve as primitive operations to combine two datasets, say D and Q, with respect to a spatial relation-

ship. Refer to Figure 1a for illustration. For each point $q \in Q$, a spatial range join returns data points in D that are inside the radius of the circle centered at q. In contrast, a kNN join (refer to Figure 1b for illustration) returns the k nearest-neighbors from the dataset D for each query point $q \in Q$. Both spatial operators are expensive and may incur computation skew in certain workers, thus greatly degrading the overall performance of query processing.

For illustration, consider a large spatial dataset, with millions of points of interests (POIs), that is preprocessed and is partitioned into different computation nodes based on the spatial distribution of the data, e.g., one data partition represents data from San Francisco, CA, and another one for Chicago, IL, etc. Assume that we have incoming queries from people looking for different POIs, e.g., restaurants, train or bus stations, and grocery stores, around their locations. These spatial range queries are consolidated into batches to be joined via an index to the POI data (e.g., using indexed nested-loops join). After partitioning the incoming spatial queries based on their locations, we observe the following issues: During rush hours in San Francisco from 4PM to 6PM (PST), San Francisco's corresponding data partition may encounter more queries than the data partition in Chicago, since Chicago is already evening. Without an appropriate optimization technique, the data partition for San Francisco will take much longer time to process its corresponding queries while the workers responsible for the other partitions are lightly loaded. As another example, in Figure 1, the data points (the dark dots) correspond to Uber car's GPS records where multiple users (the triangles) are looking for the Uber carpool service around. Partition D_1 that corresponds to an airport, experiences more queries than other partitions because people may prefer using Uber at this location. Being aware of the spatial query skew provides a new opportunity to optimize queries in distributed spatial data environments. The skew partitions have to be assigned more computation power to reduce the overall processing time.

Furthermore, communication cost, generally a key factor of the overall performance, may become a bottleneck. When a spatial query usually touches more than one data partition, it may be the case that some of these partitions do not contribute to the final query result. For example, in Figure 1a, queries q_2, q_3, q_4, and q_5 overlap more than one data partition (D_1, D_2, and D_4), but these partitions do not contain data points that satisfy the queries. Thus, scheduling queries (e.g., q_4 and q_5) to the overlapping data partition D_4 incurs unnecessary communication cost. More importantly, for the spatial range join or kNN join operators over two large datasets, the cost of network communication may become prohibitive without proper optimization.

In this paper, we introduce LocationSpark, an efficient in-memory distributed spatial query processing system. In particular, it has a query scheduler with an automatic skew analyzer and a plan optimizer to mitigate query skew. The query scheduler uses a cost model to analyze the skew for use by the spatial operators, and a plan generation algorithm to construct a load-balanced query execution plan. After plan generation, local computation nodes select the proper algorithms to improve their local performance based on the available spatial indexes and the registered queries on each node. Finally, to reduce the communication cost when dispatching queries to their overlapping data partitions, LocationSpark adopts a new spatial Bloom filter, termed sFilter, that can speed up query processing by avoiding needless communication with data partitions that do not contribute to the query answer. We implement LocationSpark as a library in Spark that provides and API for spatial query processing and optimization based on Spark's standard dataflow operators.

The main contributions of this paper are as follows:

1. We develop a new spatial computing system for efficient processing and optimization of spatial queries in a distributed in-memory environment.

2. We address data and query skew issues to improve load balancing while executing spatial operators, e.g., spatial range joins and kNN joins, by generating cost-optimized query execution plans over in-memory distributed spatial data.

3. We introduce a new light-weight yet efficient spatial Bloom filter to reduce communication cost.

4. We realize the introduced query processing and optimization techniques inside Spark. We use the developed prototype system, LocationSpark, to conduct a large-scale evaluation on real spatial data and common benchmark algorithms, and compare LocationSpark against state-of-the-art distributed spatial data processing systems. Experimental results illustrate an enhancement in performance by up to an order of magnitude over existing in-memory distributed spatial systems.

The rest of this paper proceeds as follows. Section 2 presents the problem definition and an overview of distributed spatial query processing. Section 3 introduces the cost model and the cost-based query plan scheduler and optimizer and their corresponding algorithms. Section 4 presents an empirical study for local execution plans in local computation nodes. Section 5 introduces the spatial Bloom filter, and explains how it can speedup spatial query processing in a distributed setup. The experimental results are presented in Section 6. Section 7 discusses the related work. Finally, Section 8 concludes the paper.

2. PRELIMINARIES

2.1 Data Model and Spatial Operators

LocationSpark stores spatial data as key-value pairs. A tuple, say o_i, contains a spatial geometric key k_i and a related value v_i. The spatial data type for key k_i can be a two-dimensional point, e.g., latitude-longitude, a line-segment, a poly-line, a rectangle, or a polygon. The value type v_i is specified by the user, e.g., a text data type if the data tuple is a tweet. In this paper, we assume that queries are issued progressively by users, and are processed by the system in batches (i.e., similar to the DStream model [26]).

LocationSpark supports various types of spatial query predicates including spatial range search, k-NN search, spatial range join, and kNN join. In this paper, we focus our discussion on the spatial range join and the kNN join operators, and two dataset, say Q and D, form the outer and inner tables, respectively, of the join operators.

DEFINITION 1. ***Spatial Range Search*** - *range*(q, D): *Given a spatial range area* q *(e.g., circle or rectangle) and a dataset* D, *range*(q, D) *finds all tuples from* D *that overlap the spatial range defined by* q.

DEFINITION 2. ***Spatial Range Join*** - $Q \bowtie sj\ D$: *Given two dataset* Q *and* D, $Q \bowtie sjD$, *combines each object* $q \in Q$ *with its range search results from* D, $Q \bowtie sjD = \{(q, o)|q \in Q, o \in range(q, D)\}$.

DEFINITION 3. k***NN Search*** - $kNN(q,D)$: *Given a query tuple* q, *a dataset* D, *and an integer* k, $kNN(q,D)$, *returns the output set* $\{o|o \in D$ *and* $\forall s \in D$ *and* $s \neq o, ||o, q|| \leq ||s, q||\}$, *where the number of output objects from* D, $|kNN(q, D)| = k$.

DEFINITION 4. k***NN Join*** - $Q \bowtie knn\ D$: *Given a parameter* k, *kNN join of* Q *and* D *computes each object* $q \in Q$ *with its* k *nearest neighbors from* D. $Q \bowtie knn\ D = \{(q, o)|\forall q \in Q, \forall o \in kNN(q, D)\}$.

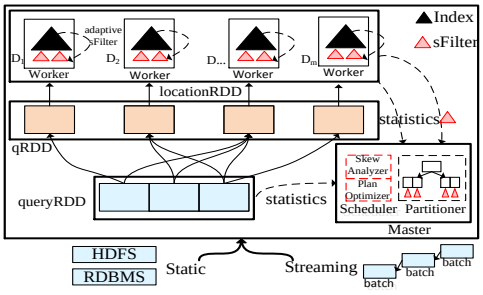


Figure 2: LOCATIONSPARK system architecture

2.2 Overview of In-memory Distributed Spatial Query Processing

To facilitate spatial query processing, we build a distributed spatial index for in-memory spatial data. Given a spatial dataset D, we obtain samples from D and construct a spatial index (e.g., an R-tree) with N leaves over the samples. We refer to this index on the sample data as the *global spatial index*. Next, each worker partitions the dataset D into N partitions according to the built global spatial index via data shuffling. The global spatial index guarantees that each data partition approximately has the same amount of data. Then, each worker i of the N workers has a local data partition D_i that is roughly $1/N$th of the data and builds a local spatial index. Finally, the indexed data (termed the LocationRDD) is cached into memory. Figure 2 gives the architecture of LOCATIONSPARK and the physical representation of the partitioned spatial data based on the procedure outlined above, where the master node stores the global spatial index that indexes the data partitions, while each worker has a local spatial index over the local spatial data within the partition. Notice that the global spatial index partitions the data into LocationRDDs as in Figure 2, and this index can be copied into various workers to help partition the data in parallel. The type of each local index, e.g., a Grid, an R-tree, or an IR-tree, for a data partition can be determined based on the specifics of the application scenarios.

For spatial range join, two strategies are possible; either replicate the outer table and send it to the inner table data or replicate the inner table data and send it to the different processing nodes where the outer table tuples are. In shared execution, the outer table is typically a collection of range query tuples and the inner table is the queried dataset. If this is the case, then it makes sense to send the outer table of queries to the inner data tables as the outer table of queries will be much smaller in size compared to the inner data tables. In this paper, we adopt the first approach because it is impracticable to replicate and forward copies of the large inner data table.

Thus, each tuple $q \in Q$ is replicated and is forwarded to the partitions that spatially overlap q. These overlapping partitions are identified using the global index. Then, a post-processing step merges the local results to produce the final output. For example, outer table tuple q_2 in Figure 1a is replicated and is forwarded to data partitions D_1, D_3, and D_4. Then, we execute a spatial range search on each data partition locally. Next, we merge the local results to form the overall output of tuple q. As illustrated in Figure 2, the outer table that corresponds to a shared execution plan's collection of queries (termed queryRDD) are first partitioned into qRDD based on the overlap between the queries in qRDD and the corresponding data partitions. Then, local search takes place over the local data partitions of LocationRDD.

The kNN join operator is implemented similarly in a simple two-round process. First, each outer focal points $q_i \in Q$ is transferred to the worker that holds the data partition that q_i spatially belongs to. Then, the kNN join is executed locally in each data partition, producing the kNN candidates for each focal point q_i. Afterwards, the maximum distance from q_i to its kNN candidates, say radius r_i, is computed. If the radius r_i overlaps multiple data partitions, point q_i is replicated into these overlapping partitions, and another set of kNN candidates is computed in each of these partitions. Finally, we merge the kNN candidates from the various partitions to get the exact result. For example, in Figure 1b, assume that we want to evaluate a 3NN query for Point q_6. The first step is to find the 3NN candidates for q_6 in data Partition D_3. Next, we find that the radius r for the 3NN candidates from Partition D_3 overlaps Partition D_4. Thus, we need to compute the 3NN of q_6 in Partition D_4 as well. Notice that the radius r can enhance the 3NN search in Partition D_4 because only the data points within Radius r are among the 3NN of q_6. Finally, the 3NN of q_6 are o_1, o_2 and o_3.

2.3 Challenges

The outer and inner tables (or, in shared execution terminology, the queries and the data) are spatially collocated in distributed spatial computing. In the following discussion, we refer to the outer table as being the queries table, e.g., containing the ranges of range operations, or the focal points of kNN operations. We further assume that the outer (or queries) table is the smaller of the two, in contrast to the inner table that we refer to by the data table (in the case of shared execution of multiple queries together). The distribution of the incoming spatial queries (in the outer tables) changes dynamically over time, with bursts in certain spatial regions. Thus, evenly distributing the input data D to the various workers may result in load imbalance at times. LOCATIONSPARK's skew analyzer identifies the skewed data partitions based on a cost model and then repartitions and

redistributes the data accordingly. The plan optimizer selects the optimal repartitioning strategies for both the outer and inner tables, and consequently generates an overall optimized execution plan.

Communication cost is a major factor that affects system performance. LocationSpark adopts a spatial Bloom filter to reduce network communication cost. The spatial Bloom filter's role is to prune the data partitions that overlap the spatial ranges from the outer tables but do not contribute to the final operation's results. This spatial Bloom filter is memory-based and is space- and time-efficient. The spatial Bloom filter adapts its structure as the data and query distributions change.

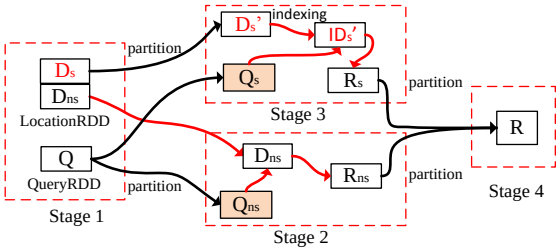


Figure 3: Execution plan for spatial range join. The red lines identify local operations, and black lines show the data partitioning. D_s and D_{ns} are the skew and non-skew partitions, respectively. Queries Q (the outer table) are partitioned into skew Q_s and non-skew Q_{ns} in Stage 1. Stages 2 and 3 execute independently. Stage 4 merges the results.

3. QUERY PLAN SCHEDULER

This section addresses how to dynamically handle query (outer table) skew. First, we present the cost functions for query processing and analyze the bottlenecks. Then, we show how to repartition the skewed data partitions to speedup processing. This is formulated as an optimization problem that we show is NP-complete. Thus, we introduce an approximation algorithm to solve the skew data repartitioning problem. Although presented for spatial range joins, the proposed technique applies to kNN join as well.

3.1 Cost Model

The input dataset D (i.e., inner table of spatial range join) is distributed into N data partitions, and each data partition D_i is indexed and cached in memory. For the query dataset Q (i.e., outer table of spatial range join), each query $q_i \in Q$ is shuffled to the data partitions that spatially overlap with it. The shuffling cost is denoted by $\epsilon(Q, N)$. The execution time of local queries at Partition D_i is $\mathrm{E}(D_i)$. The execution times of local queries depend on the queries and built indexes, and the estimation of $\mathrm{E}(D_i)$ is presented later. After the local results are computed, the post-processing step merges these local results to produce the final output. The corresponding cost is denoted by $\rho(Q)$. Overall, the runtime cost for the spatial range join operation is:

$$C(D, Q) = \epsilon(Q, N) + \max_{i \in [1,N]} \mathrm{E}(D_i) + \rho(Q), \quad (1)$$

where N is the number of data partitions. In reality, the cost of query shuffling is far less than the other costs as the number of queries is much smaller than the number of data items. Thus, the runtime cost can be estimated as follows:

$$C(D, Q) = \max_{i \in [1,N]} \mathrm{E}(D_i) + \rho(Q) \quad (2)$$

In Equation 2, data partitions are categorized into two types: skewed ($\hat{D}$) and non-skewed ($\bar{D}$). The execution time of the local queries in the skewed partitions is the bottleneck. The runtime costs for skewed and non-skewed data partitions are $\max_{i \in [1,\hat{N}]} \mathrm{E}(\hat{D}_i)$ and $\max_{j \in [1,\bar{N}]} \mathrm{E}(\bar{D}_j)$, respectively, where $\hat{N}$ (and $\bar{N}$) is the number of skewed (and non-skewed) data partitions, and $N = \hat{N} + \bar{N}$. Thus, Equation 2 can be rewritten as follows:

$$C(D, Q) = \max\{ \max_{i \in [1,\hat{N}]} \mathrm{E}(\hat{D}_i), \max_{j \in [1,\bar{N}]} \mathrm{E}(\bar{D}_j)\} + \rho(Q) \quad (3)$$

3.2 Execution Plan Generation

The goal of the query optimizer is to minimize the query processing time subject to the following constraints: (1) the limited number of available resources (i.e., the number of partitions) in a cluster, and (2) the overhead of network bandwidth and disk I/O. Given the partitioned and indexed spatial data, the cost estimator for query processing based on sampling that we introduce below, and the available number of data partitions, the optimizer returns an execution plan that minimizes query processing time. First, the optimizer needs to determine if any partitions are skewed. Then, it repartitions them subject to the introduced cluster and networking constraints. Then, the optimizer evaluates the plan on the newly repartitioned data to determine whether it minimizes query execution time or not (Refer to Figure 3).

Estimating the runtime cost of executing the local queries and the cost of merging the final results is not straightforward. The local query processing time $\mathrm{E}(D_i)$ is influenced by various factors including the types of spatial indexes used, the number of data points in D_i, the number of queries directed to D_i, related spatial regions, and the available memory. Similar to [13], we assume that the related cost functions are monotonic, and can be approximated using samples from the outer and inner tables (the queries and the datasets tables). Thus, the local query execution time is formulated as follows: $\mathrm{E}(D_i) = \mathrm{E}_s(\tilde{D}_i, \tilde{Q}_i, \alpha, A)$, where $\tilde{D}_i$ is a sample of the original inner table dataset, $\tilde{Q}_i$ is the sample of the outer table queries, A is the area of the underlying spatial region, and α is the sample ratio to scale up the estimate to the entire dataset. After computing a representative sample of the data points and queries, e.g., using reservoir sampling [22], the cost function $\mathrm{E}(D_i)$ estimates the query processing runtime in data partition D_i. More details on computing $\mathrm{E}(D_i)$, $\rho(Q_i)$, and the sample size can be learned from previous work [13].

Given the estimated runtime cost over skewed and non-skewed partitions, the optimizer splits one skewed data Partition $\hat{D}_i$ into m' data sub-partitions. Assume that $\hat{Q}_i$ is the set of queries originally assigned to Partition $\hat{D}_i$. Let the overheads due to data shuffling, re-indexing, and merging be $\beta(\hat{D}_i)$, $\gamma(\hat{D}_i)$ and $\rho(\hat{Q}_i)$, respectively. Thus, after splitting a skewed Partition $\hat{D}_i$, the new runtime is:

$$\widehat{\mathrm{E}(\hat{D}_i)} = \beta(\hat{D}_i) + \max_{s \in [1,m']} \{\gamma(D_s) + \mathrm{E}(D_s)\}\} + \rho(\hat{Q}_i), \quad (4)$$

Hence, we can split one skewed Partition $\hat{D}_i$ into multiple partitions only if $\widehat{\mathrm{E}(\hat{D}_i)} < \mathrm{E}(\hat{D}_i)$. As a result, the new query execution time, say $\widehat{C(D,Q)}$, is:

$$\widehat{C(D,Q)} = \max\{\max_{i\in[1,\hat{N}]}\{\widehat{\mathrm{E}(\hat{D}_i)}\}, \max_{j\in[1,\bar{N}]}\{\mathrm{E}(\bar{D}_j)\}\} + \rho(\bar{Q}) \quad (5)$$

Thus, we can formulate the query plan generation based on the data repartitioning problem as follows:

DEFINITION 5. *Let D be the set of spatially indexed data partitions, Q be the set of spatial queries, M be the total number of data partitions, and their corresponding cost estimation functions, i.e., query processing $\mathrm{E}(D_i)$, data repartitioning $\beta(D_i)$, and data indexing cost estimates $\gamma(Q_i)$. The query optimization problem is to choose a skewed Partition $\hat{D}$ from D, repartition each $\hat{D}_i \in \hat{D}$ into multiple partitions, and assign spatial queries to the new data partitions. The new data partition set, say D', contains partitions $D'_1, D'_2, \ldots, D'_k$. s.t. (1) the $\widehat{C(D,Q)} < C(D,Q)$ and (2) $|D'| \leq M$.*

Unfortunately, this problem is NP-complete. In the next section, we present a greedy algorithm for this problem.

THEOREM 1. *Optimized query plan generation with data repartitioning for distributed indexed spatial data is NP-complete.*

The proof is given in [16].

3.3 A Greedy Algorithm

The general idea is to search for skewed partitions based on their local query execution time. Then, we split the selected data partitions only if the runtime can be improved. If the runtime cannot be improved, or if all the available data partitions are consumed, the algorithm terminates. While this greedy algorithm cannot guarantee optimal query performance, in the experimental section, it shows significant improvement (by one order of magnitude) over the plan executing on the original partitions. Algorithm 1 gives the pseudocode for the greedy partitioning procedure.

Algorithm 1 includes two functions, namely *numberOfPartitions* and *repartition*. Function *numberOfPartitions* computes the number of partitions m' for splitting one skew partition. Naively, we could split a skew partition into two partitions each time. But this is not necessarily efficient. Given data partitions $D = \{D_1, D_2, \ldots, D_N\}$, let Partition D_1 be the one with the largest local query execution time $\mathrm{E}(D_1)$. From Equation 2, the execution time is approximated by $\mathrm{E}(D_1) + \rho(Q)$. To improve the execution time, Partition D_1 is split into m' partitions, and the query execution time for Partition D_1 is updated to $\widehat{\mathrm{E}(D_1)}$. For all other partitions $D_i \in D$ $(i \neq 1)$, the runtime is the $\max\{\mathrm{E}(D_i)\} + \rho(Q') = \Delta$, where $i = [2, \ldots, N]$ and Q' are the queries related to all data partitions except D_1. Thus, the runtime is $\max\{\Delta, \widehat{\mathrm{E}(D_1)}\}$, and is improved if

$$\max\{\Delta, \widehat{\mathrm{E}(D_1)}\} < \mathrm{E}(D_1) + \rho(Q) \quad (6)$$

As a result, we need to compute the minimum value of m' to satisfy Equation 6, since Δ, $\mathrm{E}(D_1)$, and $\rho(Q)$ are known.

Function *repartition* splits the skewed data partitions and reassigns the spatial queries to the new data partitions using two strategies. The first strategy is to repartition based on the data distribution. Because each data partition D_i is already indexed by a spatial index, the data distribution can be learned directly by recording the number of data points in each branch of the index. Then, we repartition data points in D_i into multiple parts based on the recorded numbers while guaranteeing that each newly generated sub-partition contains an equal amount of data. In the second strategy, we repartition a skewed Partition D_i based on the distribution of the spatial queries. First, we collect a sample Q_s from the queries Q_i that are originally assigned to partition D_i. Then, we compute how Q_s is distributed in Partition D_i by recording the frequencies of the queries as they belong to branches of the index over the data. Thus, we can repartition the indexed data based on the query frequencies. Although the data sizes may not be equal, the execution workload will be balanced. In our experiments, we choose this second approach to overcome query skew. To illustrate how the proposed query-plan optimization algorithm works, consider the following example.

Running Example. Given data partitions $D = \{D_1, D_2, D_3, D_4, D_5\}$, where the number of data points in each partition is 50, the number of queries in each partition D_i, $1 \leq i \leq 5$ is 30, 20, 10, 10, and 10, respectively, and the available data partitions M is 5. For simplicity, the local query processing cost is $\mathrm{E}(D_i) = |D_i| \times |Q_i| \times p_e$, where $p_e = 0.2$ is a constant. The cost of merging the results is $\rho(Q) = |Q| \times \lambda \times p_m$, where $p_m = 0.05$, and $\lambda = 10$ is the approximate number of retrieved data points per query. The indexing cost after repartitioning is $\beta(D_i, m') = |D_i| \times m' \times p_r$, and $\gamma(D_s) = |D_s| \times p_x$, respectively, where $p_r = 0.01$ and $p_x = 0.02$. Without any optimization, from Equation 2, the estimated runtime cost for this input dataset is 340. LOCATIONSPARK optimizes the query as follows. At first, it chooses data Partition D_1 as the skew partition to be repartitioned because D_1 has the highest local runtime (300), while the second largest cost is D_2's (200). Using Equation 6, we split D_1 into two partitions, i.e., $m' = 2$. Thus, Function `repartition` splits D_1 into the two partitions D'_1 and D'_2 based on the distribution of queries within D_1. The number of data points in D'_1 and D'_2 is 22 and 28, respectively, and the number of queries are 12 and 18, respectively. Therefore, the new runtime is reduced to $\approx 200 + 25$ because D_1's runtime is reduced to ≈ 100 based on Equation 4. Therefore, the two new data partitions D'_1 and D'_2 are introduced in place of D_1. Next, Partition D_2 is chosen to be split into two partitions, and the optimized runtime becomes $\approx 100 + 15$. Finally, the procedure terminates as only one available partition is left.

4. LOCAL EXECUTION

Once the query plan is generated, each computation node chooses a specific local execution plan based on the queries assigned to it and the indexes it has. We implement various centralized algorithms for spatial range join and kNN join operators within each worker and study their performance. The algorithms are implemented in Spark. We use the execution time as the performance measure.

4.1 Spatial Range Join

Two algorithms for spatial range join [20] are implemented. The first is indexed nested-loops join, where we probe the spatial index repeatedly for each outer tuple (or

Algorithm 1: Greedy Partitioning Algorithm

Input: D: Indexed spatial data partitions, Stat: Collected statistics, e.g., the number of data points and queries in data partition D_i, M: number of available data partitions.
Output: *Plan*: Optimized data and query partition plan, C: estimated query cost

```
h: Maximum Heap;
inserts D_i into heap // data partitions are ordered by cost E(D_i) that is computed using Stat
Cost_o ← E(h.top) + ρ(Q) // old execution plan runtime cost
Plan
while M > 0 do
    Var D_x ← h.pop(); //get the partition with maximum runtime cost
    Var m' ← numberOfPartitions(h,D_x, M)
    Var (D_s, PL_s) ← repartition(D_x, m') //split D_x into m' partitions
    Cost_x ← β(D_x) + max_{s∈[1,m']}{γ(D_s) + E(D_s)} + ρ(Q_x)
    //updated runtime cost over selected skew partition
    if Cost_x < Cost_o then
        save Partitions D_s into h
        save Partition plan PL_s into Plan
        Cost_o ← Cost_x
        M ← M-m'
    end
    else
        break;
    end
end
```

range query in the case of shared execution). The tested algorithms are nestRtree, nestGrid and nestQtree, where they use an R-tree, a Grid, and a Quadtree as index for the inner table, respectively.

The second algorithm for spatial range join is based on the dual-tree traversal [6]. It builds two spatial indexes (e.g., an R-tree) over both the input queries (i.e., the outer table) and the data (i.e, the inner table), and performs a depth-first search over the dual trees simultaneously.

Figure 4a gives the performance of nestRtree, nestQtree, and dual-tree, where the number of data points in each worker is 300K. The results for nestGrid are not given as it performs the worst. The dual-tree approach provides a 1.8x speedup over the nestRtree. This conforms with other published results [20]. nestQtree achieves an order of magnitude improvement over the dual-tree approach. The reason is that the minimum bounding rectangles (MBRs) of the spatial queries overlap with multiple MBRs in the data index, and this reduces the pruning power of the underlying R-tree. The same trend is observed in Figure 4b when increasing the number of indexed data points. The dual-tree approach outperforms nestRtree, when the number of data points is smaller than 120k. However, dual-tree slows down afterwards. In this experiment, we only show the results for indexing over two dimensional data points. However, Quadtree performs worst when the indexed data are polygons [18]. Overall, for multidimensional points, the local planner chooses nestQtree as the default approach. For complex geometric types, the local planner uses the dual-tree approach based on an R-tree implementation.

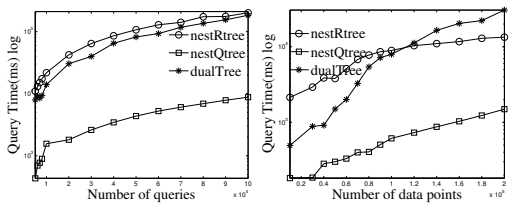
4.2 kNN Join

Similar to the spatial range join, indexed nested-loops can be applied to kNN join, where it computes the set of kNN objects for each query point in the outer table. An index is built on the inner table (the data table). The other kinds of kNN join algorithms are block-based. They partition the queries (the outer table) and the data points (the inner table) into different blocks, and find the kNN candidates for queries in the same block. Then, a post-processing refine step computes kNN for each query point in the same block. Gorder [23] divides query and data points into different rectangles based on the G-order, and utilizes two distance bounds to reduce the processing of unnecessary blocks. For example, the min-distance bound is the minimum distance between the rectangles of the query points and the data points. The max-distance bound is the maximum distance from the queries to their kNN sets. If the max-distance is smaller than the min-distance bound, the related data block is pruned. PGBJ [15] has a similar idea that extends to parallel kNN join using MapReduce. Recently, Spitfire [8] is a parallel kNN self-join algorithm for in-memory data. It replicates the possible kNN candidates into its neighboring data blocks. Both PGBJ and Spitfire are designed for parallel kNN join, but they are not directly applicable to indexed data. The reason is that PGBJ partitions queries and data points based on the selected pivots while Spitfire is specifically optimized for kNN self-join.

LocationSpark enhances the performance of the local kNN join procedure. For the Gorder [23], instead of using the expensive principal component analysis (PCA) in Gorder, we apply the Hilbert curve to partition the query points. We term the modified Gorder method *sfcurve*. We modify PGBJ as follows. First, we compute the pivots of the query points based on a clustering algorithm (e.g., k-means) over sample data, and then partition the query points into different blocks based on the computed pivots. Next, we compute the MBR of each block. Because the data points are already indexed (e.g., using an R-tree), the min-distance from the MBRs of the query points and the index data is computed, and the max-distance bound is calculated based on the kNN results from the pivots. This approach is termed *pgbjk*. In terms of spitfire, we use a spatial index to speedup finding the kNN candidates.

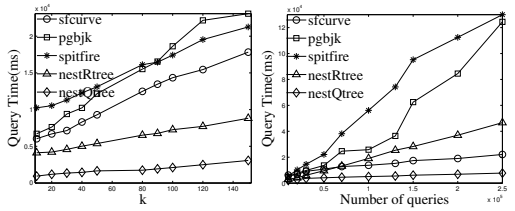
Figure 5a gives the performance of the specialized kNN join approaches within a local computation node when varying k from 10 to 150. The nestQtree approach always performs the best, followed by nestRtree, sfcurve, pgbjk, and spitfire. Notice that block-based approaches induce extensive amounts of kNN candidates for query points in the same block, and it directly degrades the performance of the kNN refine step. More importantly, the min-distance bound between the MBR of the query points and the MBR of the data points is much smaller than the max-distance boundary. Then, most of the data blocks cannot be pruned, and result in redundant computations. In contrast, the nested-loops join algorithms prune some data blocks because the min-distance bound from the query point to the data points of the same block is bigger than the max-distance boundary of this query point. Figure 5b gives the performance of the kNN join algorithms by varying the number of query points. Initially, for less than 70k query points, nestRtree outperforms sfcurve, then nestRtree degrades linearly with more query points. Overall, we adopt nestQtree as the kNN join algorithm for local workers.

5. SPATIAL BLOOM FILTER



(a) Effect of query size (b) Effect of data size

Figure 4: Evaluation of local spatial join algorithms



(a) Effect of k (b) Effect of query size

Figure 5: Evaluation of local kNN join algorithms

In this section, we introduce a new spatial Bloom filter termed *sFilter*. The sFilter can help decide for an outer tuple, say q, of a spatial range join, if there exist tuples in the inner table that actually join with q. This helps reduce the communication overhead. For example, consider an outer tuple q of a spatial range join where q has a range that overlaps multiple data partitions of the inner table. Typically, all the overlapping partitions need to be examined by communicating q's range to them, and searching the data within each partition to test for overlap with q's range. This incurs high communication and search costs. Using the sFilter, given q's range that overlaps multiple partitions of the inner table, the sFilter can decide which overlapping partitions contain data that overlaps q's range without actually communicating with and searching the data in the partitions. Only the partitions that contain data that overlap with q's range are the ones that will be contacted and searched.

5.1 Overview of the sFilter

Figure 6 gives an example of an sFilter. Conceptually, an sFilter is a new in-memory variant of a quadtree that has internal and leaf nodes [18]. Internal nodes are for index navigation, and leaf nodes, each has a marker to indicate whether or not there are data items in the node's corresponding region. We encode the sFilter into two binary codes and execute queries over this encoding.

5.1.1 Binary Encoding of the sFilter

The sFilter is encoded into two long sequences of bits. The first bit-sequence corresponds to internal nodes while the second bit-sequence corresponds to leaf nodes. Notice that in these two binary sequences, no pointers are needed. Each internal node of the sFilter takes four bits, where each bit represents one of the internal node's children. These children are encoded in clock-wise order starting from the upper-left corner. Each bit value of an internal node determines the type of its corresponding child, i.e., whether the child is internal (a 1 bit) or leaf (a 0 bit). In Figure 6, the root (internal) node A has binary code 1011, i.e., it has three of its children being internal nodes, and its second node is leaf. The four-bit encodings of all the internal nodes are concatenated to form the internal-node bit-sequence of the sFilter. The ordering of the internal nodes in this sequence is based on a breadth-first search (BFS) traversal of the quadtree. In contrast, a leaf node only takes one bit, and its bit value indicates whether or not data points exist inside the spatial quadrant corresponding to the leaf. In Figure 6, internal node B has four children, and the bit values for B's leaf nodes are 1010, i.e., the first and third leaf nodes of B contain data items. To encode the bit-sequence for all the leaf nodes in an sFilter, during the same BFS on the underlying quad-tree of the sFilter to produce the bit-sequence for the internal nodes, we simultaneously construct the bit-sequence for all the leaf nodes. The sFilter is encoded into the two binary sequences in Figure 6. The space usage of an sFilter is $O(((4^{d-1}-1)/3)\times 4 + 4^{d-1})$ Bits, where $O((4^{d-1}-1)/3)$ and $O(4^{d-1})$ are the numbers of internal nodes and leaf nodes, respectively, and $d = o(\log(L))$ is the depth of quadtree and L is the length of the space.

5.1.2 Query Processing Using the sFilter

Consider internal node D in Figure 6. D's binary code is 0010, and the third bit has a value of 1 at memory address a_x of the internal nodes bit sequence. Thus, this bit refers to D's child F that is also an internal node at address a_j. Because the sFilter has no pointers, we need to compute F's address a_j from a_x. Observe that the number of bits with value 1 from the start address a_0 of the binary code to a_x can be used to compute the address.

DEFINITION 6. *Let a be the bit sequence that starts at address a_0. $\chi(a_0, a_x)$ and $\tau(a_0, a_x)$ are the number of bits with value 1 and 0, respectively, from addresses a_0 to a_x inclusive.*

$\chi(a_0, a_x)$ is the number of internal nodes up to a_x. Thus, the address a_j of F is $(a_0 + 5 \times 4)$ because there are 5 bits with value 1 from a_0 to a_x. Similarly, if one child node is a leaf node, its address is inferred from $\tau(a_0, a_x)$ as follows:

PROPOSITION 1. *Let a and b be the sFilter's bit sequences for internal and leaf nodes in memory addresses a_0 and b_0, respectively. To access a node's child in memory, we need to compute its address. The address, say a_j, of the xth child of an internal node at address a_x is computed as follows. If the bit value of a_x is 1, then $a_j = a_0 + 4 \times \chi(a_0, a_x)$. If the bit value of a_x is 0, $a_j = b_0 + \tau(a_0, a_x)$.*

We adopt the following two optimizations to speedup the computation of $\chi(a_0, a_x)$ and $\tau(a_0, a_x)$: (1) Precomputation, and (2) Set counting. Let d_i be the memory address of the first internal node at height (or depth) i of the underlying quadtree when traversed in BFS order. For example, in Figure 6, nodes B and E are the first internal nodes in BFS order at depths 1 and 2 of the quadtree, respectively. For all $i \leq$ depth of the underlying quadtree, we precompute $\chi(a_0, d_i)$, e.g., $\chi(a_0, d_1)$ and $\chi(a_0, d_2)$ in Figure 6. Notice that $d_0 = a_0$ and $\chi(a_0, d_0) = 0$. Then, address a_j that corresponds to the memory address of the xth child of an internal node at address a_x can be computed as follows. $a_j = a_0 + (\chi(a_0, d_1) + \chi(d_1, a_x)) \times 4$. $\chi(a_0, d_1)$ is precomputed. Thus, we only need to compute on the fly $\chi(d_1, a_x)$. Furthermore, evaluating χ can be optimized by a bit set counting approach, i.e, a lookup table or a sideways addition [1] that can achieve constant time complexity.

[1] `https://graphics.stanford.edu/~seander/bithacks.html`

Algorithm 2: Update sFilter in LocationSpark

Input: *LocationRDD*: Distributed/indexed spatial data, Q: Input set of spatial range queries
Output: R: Results of the spatial queries

```
1  Var index ← LocationRDD.index //get global index with embedded sFilters
2  Var qRDD ← partition(Q,index) // Distribute in parallel the input spatial queries using the global index
3  Var update_sFilter ← //function for updating the sFilter in each worker
4  {
5  for each query q_i in this worker do
6  |  if query q_i's return result is empty then
7  |  |  insert(q_i, sFilter) // adapt sFilter given q_i
8  |  end
9  end
10 if sFilter.space > α then
11 |  shrink(sFilter) // shrink the sFilter to save space
12 end
13 }
14 R ← LocationRDD.sjoin(qRDD)(update_sFilter) //execute spatial join and update sFilter in workers
15 Var sFilters ← LocationRDD.collect_sFilter() //collect sFilter from workers
16 mergesFilters(sFilters, index) // update sfilter in global index
17 return R
```

After getting one node's children via Proposition 1, we apply Depth-First Search (DFS) over the binary codes of the internal nodes to answer a spatial range query. The procedure starts from the first four bits of bit sequence a, since these four bits are the root node of the sFilter. Then, we check the four quadrants, say r_s, of the children of the root node, and iterate over r_s to find the quadrants, say r'_s, overlapping the input query range q_i. Next, we continue searching the children of r'_s based on the addresses computed from Proposition 1. This recursive procedure stops if a leaf node is found with value 1, or if all internal nodes are visited. For example, Consider range query q_2 in Figure 6. We start at the root node A (with bit value 1011). Query q_2 is located inside the northwestern (NW) quadrant of A. Because the related bit value for this quadrant is 1, it indicates an internal node type and it refers to child node B. Node B's memory address is computed by $a_0 + 1 \times 4$ because only one non-leaf node (A) is before B. B's related bit value is 0000, i.e., B contains four leaf nodes. The procedure continues until finding one leaf node of B, mainly the southeastern child leaf node, with value 1 that overlaps the query, and thus returns true.

5.2 sFilter in LocationSpark

The depth of the sFilter affects query performance. It is impractical to use only one sFilter in a distributed setting. We embed multiple sFilters into the global and local spatial indexes in LocationSpark. In the master node, separate sFilters are placed into the different branches of the global index, where the role of each sFilter is to locally answer the query for the specific branch it is in. In the local computation nodes, an sFilter is built and it adapts its structure based on data updates and changes in query patterns.

5.2.1 Spatial Query Processing Using the sFilter

Algorithm 2 gives the procedure for performing the spatial range join using the sFilter. Initially, the outer (queries)

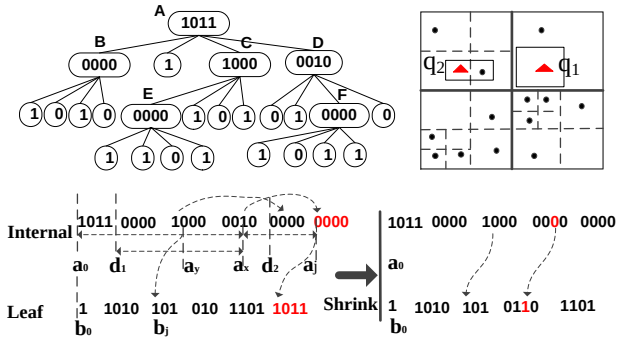


Figure 6: sFilter structure (up left), the related data (up right) and the two bit sequences of the sFilter (down).

table is partitioned according to the global index. The global index identifies the overlapping data partitions for each query q. Then, the sFilter tells which partitions contain data that overlap the query range (Line 2 of the algorithm). After performing the spatial range join (Line 14), the master node fetches the updated sFilter from each data worker, and refreshes the existing sFilters in the master node (Lines 15-16). Lines 2-13 update the sFilter of each worker (as in Figure 2).

The sFilter can improve the kNN search and kNN join because they also depend on spatial range search. Moreover, their query results may enhance the sFilter by lowering the false positive errors as illustrated below.

5.2.2 Query-aware Adaptivity of the sFilter

The build and update operations of the sFilter are first executed at the local workers in parallel. Then, the updated sFilters are propagated to the master node.

The initial sFilter is built from a temporary local quadtree [18] in each partition. Then, the sFilter is adapted based on the query results. For example, consider Query q_1 in Figure 6. Initially, the sFilter reports that there is data for q_1 in the partitions. When q_1 visits the related data partitions, it finds that there are actually no data points overlapping with q_1 in the partitions, i.e., a false-positive (+ve) error. Thus, we mark the quadrants precisely covered by q_1 in the sFilter as empty, and hence reduce the false positive errors if queries visit the marked quadrants again. Function `insert` in Algorithm 2 recursively splits the quadrants covered by the empty query, and marks these generated quadrants as empty. After each local sFilter is updated in each worker, these updates are reflected into the master node. The compact encoding of the sFilter saves the communication cost between the workers and the master.

However, the query performance of the sFilter degrades as the size of the index increases. Function `shrink` in Algorithm 2 merges some branches of the sFilter at the price of increasing false +ve errors. For example, one can shrink internal node F in Figure 6 into a leaf node, and updating its bit value to 1, although one quadrant of F does not contain data. Therefore, we might track the visit frequencies of the internal nodes, and merge internal nodes with low visiting frequency. Then, some well-known data caching policies, e.g., LRU or MRU, can be used. However, the overhead to track the visit frequencies is expensive. In our implementation, we adopt a simple bottom-up approach. We start merging the nodes from the lower levels of the index to the

higher levels until the space constraint is met. In Figure 6, we shrink the sFilter from internal node F, and replace it by a leaf node, and update its binary code to 1. F's leaf children are removed. The experimental results show that this approach increases the false +ve errors, but enhances the overall query performance.

6. PERFORMANCE STUDY

LocationSpark is implemented on RDDs, which are the distributed memory abstraction in Spark. LocationSpark is a library of Spark and provides Class LocationRDD to conduct spatial operations [16]. Statistics are maintained at the driver program of Spark, and the execution plans are generated at the driver. Local spatial indexes are persisted in the RDD data partitions, while the global index is realized by extending the interface of the RDD data partitioner. The data tuples and related spatial indexes are encapsulated into the RDD data partitions. Thus, Spark's fault tolerance naturally applies in LocationSpark. The spatial indexes are immutable and are implemented based on the path copy approaches. Thus, each updated version of the spatial index can be persisted into disk for fault tolerance. This enables the recovery of a local index from disk in case of failure in a worker. The Spark cluster is managed by YARN, and a failure in the master nodes is detected and managed by ZooKeeper. In case of master node failure, the lost master node is evicted and a standby node is chosen to recover the master. As a result, the global index and the sFilter in the master node are recoverable. Finally, the built spatial index data can be stored into disk, and enable further data analysis without additional data repartitioning or indexing. LocationSpark is open-source, and can be downloaded from `https://github.com/merlintang/SpatialSpark`.

6.1 Experimental Setup

Experiments are conducted on two datasets. **Twitter**: 1.5 Billion Tweets (around 250GB) are collected over a period of nearly 20 months (from January 2013 to July 2014) and is restricted to the USA spatial region. The format of a tweet is: identifier, timestamp, longitude-latitude coordinates, and text. **OSMP**: is shared by the authors of SpatialHadoop [9]. OSMP represents the map features of the whole world, where each spatial object is identified by its coordinates (longitude, latitude) and an object ID. It contains 1.7 Billion points with a total size of 62.3GB. We generate two types of queries. (1) Uniformly distributed (USA, for short): We uniformly sample data points from the corresponding dataset and generate spatial queries from the samples. These are the default queries in our experiments. (2) Skewed spatial queries: These are synthesized around specific spatial areas, e.g., Chicago, San Francisco, New York (CHI, SF, NY, correspondingly, for short). The spatial queries and data points are the outer table Q and the inner table D for the experimental studies of the spatial range and kNN joins presented below.

Our study compares LocationSpark with the following: (1) **GeoSpark** [25] uses ideas from SpatialHadoop but is implemented over Spark. (2) **SpatialSpark** [2] performs partition-based spatial joins. (3) **Magellan** [1] is developed based on dataframe of Spark to benefit from Spark SQL's plan optimizer. However, Magellan does not have spatial indexing. (4) **State-of-art kNN-join**: Since none of the three systems support kNN join, we compare LocationSpark with a state-of-art kNN-join approach (PGBJ [15]) that is provided by PGBJ's authors. LocationSpark(opt) refers to the optimized query scheduler and sFilter.

We use a cluster of six physical nodes that consists of Dell compute nodes with two 8-core Intel E5-2650v2 CPUs, 32 GB of memory, and 48TB of local storage per node. Spark 1.5.0 is used with YARN cluster resource management. Performance is measured by the average query execution time.

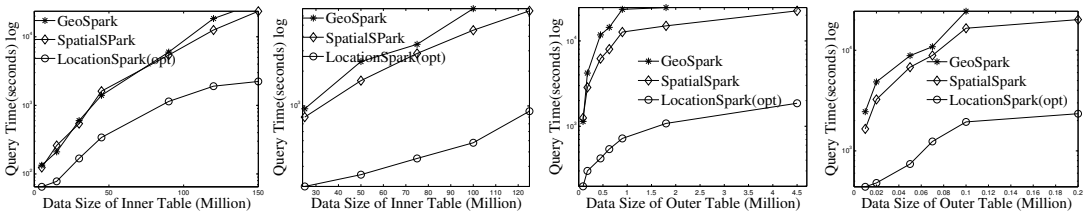
Dataset	System	Query time(ms)	Index build time(s)
Twitter	LocationSpark(R-tree)	390	32
	LocationSpark(Qtree)	**301**	16
	Magellan	15093	/
	SpatialSpark	16874	35
	SpatialSpark(no-index)	14741	/
	GeoSpark	4321	45
OSMP	LocationSpark(R-tree)	1212	67
	LocationSpark(Qtree)	**734**	18
	Magellan	41291	/
	SpatialSpark	24189	64
	SpatialSpark(no-index)	17210	/
	GeoSpark	4781	87

Table 1: Performance of the spatial range search

6.2 Spatial Range Search and Join

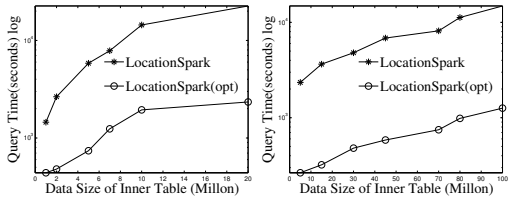
Table 1 summarizes the spatial range search and spatial index build time by the various approaches. For a fair comparison, we cache the indexed data into memory, and record the spatial range query processing time. From Table 1, observe the following: (1) LocationSpark is 50 times better than Magellan on query execution time for the two datasets, mainly because the spatial index (e.g., Global and Local index) of LocationSpark can avoid visiting unnecessary data partitions. (2) LocationSpark with different local indexes, e.g., the R-tree and Quadtree, outperforms SpatialSpark. The speedup is around 50 times, since SpatialSpark (without index) has to scan all the data partitions. SpatialSpark(with index) stores the global indexes into disk, and finds data partitions by scanning the global index in disk. This incurs extra I/O overhead. Also, the local index is not utilized during local searching. (3) LocationSpark is around 10 times faster than GeoSpark in spatial range search execution time because GeoSpark does not utilize the built global indexes and scans all data partitions. (4) The local index with Quadtree for LocationSpark achieves superior performance over the R-tree one in term of index construction and query execution time as discussed in Section 4. 5) The index build time among the three systems is comparable because they all scan the data points, which is the dominant factor, and then build the index in memory.

Performance results (mainly, the execution times of the spatial range join) are listed in Figure 7. For fair comparison, the runtime is counted as end to end, which includes the time to initiate the job, build indexes, execute the join query, and save results into HDFS. Performance results for Magellan are not shown because it performs Cartesian product and hence has the worst execution time. Figures 7a and 7b present the results by varying the data sizes of D (the inner table) from 25 million to 150 million, while keeping the size of Q (the outer table) to 0.5 million. The execution time of GeoSpark shows quadratic increase as the



(a) Twitter (b) OSMP (c) Twitter (d) OSMP

Figure 7: The performance of spatial range join



(a) Twitter (b) OSMP

Figure 8: Performance of kNN join by increasing the number of data points

data size increases. GeoSpark's running time is almost 3 hrs when the data size is 150 million, which is extremely slow. SpatialSpark shows similar trends. The reason is that both GeoSpark and SpatialSpark suffer from (1) the spatial skew issue where some workers process more data and take longer time to finish. (2) the local execution plan based on the R-tree and the Grid is slow. (3) processing of queries go to data partitions that do not contribute to the final results. LocationSpark with the optimized query plans and the sFilter outperforms the two other systems by an order of magnitude. A detailed analysis for this speedup is presented below. Also, we study the effect of the outer table size on performance. Figures 7c and 7d give the run time, and demonstrate that LocationSpark is 10 times faster than the other two systems.

6.3 Performance of kNN Search and Join

Performance of kNN search is listed in Table 2. LocationSpark outperforms GeoSpark by an order of magnitude. GeoSpark broadcasts the query points to each data partition, and accesses each data partition to get the kNN set for the query. Then, GeoSpark collects the local results from each partition, then sorts the tuples based on the distance to query point of kNN. This is prohibitively expensive, and results in large execution time. LocationSpark only searches for data partitions that contribute to the kNN query point based on the global and local spatial indexes and the sFilter. It avoids redundant computations and unnecessary network communication for irrelevant data partitions.

For kNN join, Table 3 presents the performance results when varying k on the Twitter and OSMP datasets. In terms of runtime, LocationSpark with optimized query plans and with the sFilter always performs the best. LocationSpark without any optimizations gives better performance than that of PGBJ. The reason is due to having in-memory computations and avoiding expensive disk I/O when compared to MapReduce jobs. Furthermore, LocationSpark with optimization shows around 10 times speedup over PGBJ, because the optimized plan migrates and splits the skewed query regions.

We test the performance of the kNN join operator when increasing the number of data points while having the number of queries fixed to 1 million around the Chicago area. The results are illustrated in Figure 8. Observe that LocationSpark with optimizations performs an order of magnitude better than the basic approach. The reason is that the optimized query plan identifies and repartitions the skew partitions. In this experiment, the top five slowest tasks in LocationSpark without optimization take around 33 minutes, while more than 75% tasks only take less than 30 seconds. On the other hand, with an optimized query plan, the top five slowest tasks take less than 4 minutes. This directly enhances the execution time.

Dataset	System	k=10	k=20	k=30
Twitter	LocationSpark(R-tree)	81	82	83
	LocationSpark(Q-tree)	74	75	74
	GeoSpark	1334	1813	1821
OSMP	LocationSpark(R-tree)	183	184	184
	LocationSpark(Q-tree)	73	73	74
	GeoSpark	4781	4947	4984

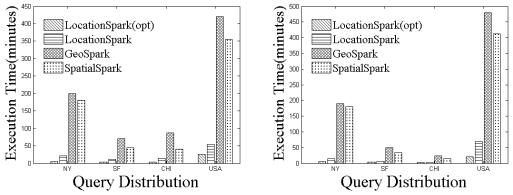
Table 2: Runtime (in microseconds) of kNN search

Dataset	System	k=50	k=100	k=150
Twitter	LocationSpark(Q-tree)	340	745	1231
	LocationSpark(Opt)	165	220	230
	PGBJ	3422	3549	3544
OSMP	LocationSpark(Q-tree)	547	1241	1544
	LocationSpark(Opt)	260	300	340
	PGBJ	5588	5612	5668

Table 3: Runtime (in seconds) of kNN join

6.4 Effect of Query Distribution

We study the performance under various query distributions. As illustrated before, the query execution plan is the most effective factor in distributed spatial computing. From the experimental results for spatial range join and kNN join above, we already observe that the system with optimization achieves much better performance over the unoptimized versions. In this experiment, we study the performance of the optimized query scheduling plan in LocationSpark under various query distributions. The performance of the spatial range join operator over query set Q (the outer table) and dataset D (the inner table) is used as the benchmark. The number of tuples for D is fixed as 15 million and 50 million for Twitter and OSMP, respectively, while the size of Q is 0.5 million, and each query in Q is generated from different spatial regions, e.g., CHI, SF, NY and USA. We do not plot



(a) Twitter (b) OSMP

Figure 9: Performance of spatial range join on various query distributions

the runtime of Magellan on spatial join, as it uses Cartesian join, and hence has the worst performance. Figure 9 gives the execution runtimes for the spatial range join operators in different spatial regions. From Figure 9, GeoSpark performs the worst, followed by SpatialSpark and then LocationSpark. LocationSpark with the optimized query plan achieves an order of magnitude speedup over GeoSpark and SpatialSpark in terms of execution time. LocationSpark with optimized plans achieves more than 10 times speedup over LocationSpark without the optimized plans for the skewed spatial queries.

6.5 Effect of the sFilter

In this experiment, we measure the query processing time, the index construction time, the false positive ratio and the space usage of the sFilter. Table 4 gives the performance of various indexes in a local computation node. The Bloom filter is tested using breeze [2]. The sFilter(ad) represents the sFilter with adaptation of its structure given changes in the queries, and with the merging to reduce its size as introduced in Section 5.2.2. From Table 4, observe that sFilter achieves one and two orders of magnitude speedup over the Quadtree- and R-tree-based approaches in terms of spatial range search execution time, respectively. The sFilter(ad) improves the query processing time over the approach without optimization, but the sFilter(ad) has the overhead to merge branches of the index to control its size, and increases the false positive ratio. The Bloom filter does not support spatial range queries. Table 4 also gives the space usage overhead for various local indexes in each worker. The sFilter is 5-6 orders of magnitude less than the other types of indexes, e.g., the R-tree and the Quadtree. This is due to the bit encoding of the sFilter that eliminates the need for pointers. Moreover, the sFilter reduces the unnecessary network communication. We study the shuffle cost for redistributing the queries. The results are given in Figure 10. The sFilter reduces the shuffling cost for both spatial range and kNN join operations. The shuffle cost reduction depends on the data and query distribution. Thus, the more unbalanced the distribution of queries and data in the various computation nodes, the more shuffle cost is reduced. For example, for kNN join, the shuffle cost is improved from 1114575 to 928933 when k is 30, achieving 18% reduction in network communication cost.

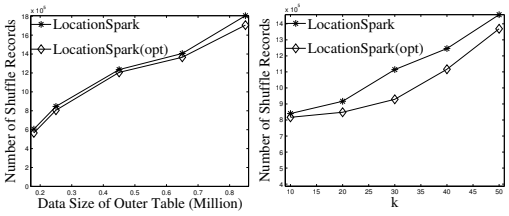
6.6 Effect of the Number of Workers

Spark's parallel computation ability depends on the number of executors and number of CPU cores assigned to each executor, that is, the number of executors times the number

[2] https://github.com/scalanlp/breeze

Dataset	Index	Query time(ms)	Index build(s)	False positive	Memory usage(MB)
Twitter	R-tree	19	17	/	112
	Q-tree	0.4	1.8	/	37
	sFilter	0.022	2	0.07	0.006
	sFilter(ad)	0.018	2.3	0.09	0.003
	Bloom filter	0.004	1.54	0.01	140
OSMP	R-tree	4	32	/	170
	Q-tree	0.5	1.2	/	63
	sFilter	0.008	2.4	0.06	0.008
	sFilter(ad)	0.006	6	0.10	0.006
	Bloom filter	0.002	2.7	0.01	180

Table 4: Performance of the sFilter



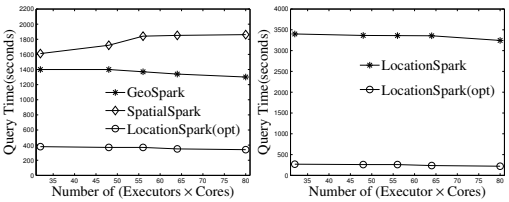
(a) Spatial range join (b) kNN join

Figure 10: The effect of the sFilter on reducing the shuffle cost

of CPU cores per executor. Therefore, to demonstrate the scalability of the proposed approach, we change the number of executors from 4 to 10, and fix the number of CPU cores assigned to each executor. We study the runtime performance for spatial range join and kNN join operations using the Twitter and OSMP datasets. Because the corresponding performance on the OSMP dataset gives similar trends as the Twitter dataset, we only present the performance for the Twitter dataset in Figure 11, where the outer table size is fixed to 1 million around Chicago area, and the inner table size is 15 million. We observe that the performance of LocationSpark for the spatial range join and the kNN join improves gradually with the increase in the number of executors. In contrast, GeoSpark and SpatialSpark do not scale well in comparison to LocationSpark for spatial range join. The performance of Magellan for spatial join is not shown because it is based on Cartesian product and shows the worst performance.

7. RELATED WORK

Spatial data management has been extensively studied for decades and several surveys provide good overviews. Gaede and Günther [10] provide a good summary of spatial data indexing. Sowell et al. give a survey and experimental study



(a) Spatial range join (b) kNN join

Figure 11: Performance of spatial range join and kNN join when varying the number of executors

of iterative spatial-join in memory [20]. Recently, there has been considerable interest in supporting spatial data management over Hadoop MapReduce. Hadoop-GIS [3] supports spatial queries in Hadoop by using a uniform grid index. SpatialHadoop [9] builds global and local spatial indexes, and modifies the HDFS record reader to read data more efficiently. MD-Hbase [17] extends HBase to support spatial data update and queries. Hadoop MapReduce is good at data processing for high throughput and fault-tolerance.

Taking advantage of the very large memory pools available in modern machines, Spark and Spark-related systems (e.g., Graphx, Spark-SQL, and DStream) [12, 26] are developed to overcome the drawbacks of MapReduce in specific application domains. In order to process big spatial data more efficiently, it is natural to develop an efficient spatial data management systems based on Spark. Several prototypes have been proposed to support spatial operations over Spark, e.g., GeoSpark [25], SpatialSpark [2], Magellan [1], Simba [24]. However, some important factors impede the performance of these systems, mainly, query skew, lack of adaptivity, and excessive and unoptimized network and I/O communication overheads. For existing spatial join [20, 6] and kNN join approaches [23, 8, 15], we conduct experiments to study their performance in Section 4. The reader is referred to Section 4.

Kwon *et al.* [14, 13] propose a skew handler to address the computation skew in a MapReduce platform. AQWA [5] is a disk-based approach that handles spatial computation skew in MapReduce. In LocationSpark, we overcome the spatial query skew for spatial range join and kNN join operators, and provide an optimized query execution plan. These operators are not addressed in AQWA. The query planner in LocationSpark is different from relational query planners, i.e., join order and selection estimation. ARF [4] supports one dimensional range query filter for data in disk. Calderoni *et al.* [7] study spatial Bloom filter for private data. Yet, it does not support spatial range querying.

8. CONCLUSIONS

We presented a query executor and optimizer to improve the query execution plan generated for spatial queries. We conduct an extensive experimental study for local execution plan generation. We introduce a new spatial bloom filter to reduce the redundant network communication cost. Empirical studies on various real datasets demonstrate the superiority of our approaches compared with existing systems.

9. REFERENCES

[1] Magellan. `https://github.com/harsha2010/magellan`.

[2] Spatialspark. `http://simin.me/projects/spatialspark/`.

[3] A. Aji, F. Wang, H. Vo, R. Lee, Q. Liu, X. Zhang, and J. Saltz. Hadoop gis: A high performance spatial data warehousing system over mapreduce. *VLDB*, 2013.

[4] K. Alexiou, D. Kossmann, and P.-A. Larson. Adaptive range filters for cold data: Avoiding trips to siberia. In *VLDB*, 2013.

[5] A. M. Aly, A. R. Mahmood, M. S. Hassan, W. G. Aref, M. Ouzzani, H. Elmeleegy, and T. Qadah. AQWA: adaptive query-workload-aware partitioning of big spatial data. In *VLDB*, 2015.

[6] T. Brinkhoff, H.-P. Kriegel, and B. Seeger. Efficient processing of spatial joins using r-trees. *SIGMOD Rec.*, 1993.

[7] L. Calderoni, P. Palmieri, and D. Maio. Location privacy without mutual trust. *Comput. Commun.*

[8] G. Chatzimilioudis, C. Costa, D. Zeinalipour-Yazti, W. Lee, and E. Pitoura. Distributed in-memory processing of all k nearest neighbor queries. *TKDE*, 2016.

[9] A. Eldawy and M. Mokbel. Spatialhadoop: A mapreduce framework for spatial data. In *ICDE*, 2015.

[10] V. Gaede and O. Günther. Multidimensional access methods. *ACM Comput. Surv.*, 30:170–231, 1998.

[11] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

[12] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. Graphx: Graph processing in a distributed dataflow framework. In *OSDI*, 2014.

[13] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia. Skew-resistant parallel processing of feature-extracting scientific user-defined functions. In *SoCC*, 2010.

[14] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia. Skewtune: Mitigating skew in mapreduce applications. In *SIGMOD*, 2012.

[15] W. Lu, Y. Shen, S. Chen, and B. C. Ooi. Efficient processing of k nearest neighbor joins using mapreduce. *Proc. VLDB Endow.*, 2012.

[16] T. Mingjie, Y. Yongyang, G. A. Walid, M. M. Qutaibah, O. Mourad, and R. Ahmed. Locationspark: A distributed in-memory data management system for big spatial data. *Purdue technical report*, 2016.

[17] S. Nishimura, S. Das, D. Agrawal, and A. Abbadi. Md-hbase: A scalable multi-dimensional data infrastructure for location aware services. In *MDM 11*.

[18] H. Samet. *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann Publishers Inc., 2005.

[19] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *MSST*, 2010.

[20] B. Sowell, M. V. Salles, T. Cao, A. Demers, and J. Gehrke. An experimental analysis of iterated spatial joins in main memory. *Proc. VLDB Endow.*, 2013.

[21] M. Tang, Y. Yu, W. G. Aref, Q. Malluhi, and M. Ouzzani. Locationspark: A distributed in-memory data management system for big spatial data. *VLDB*, 2016.

[22] J. S. Vitter. Random sampling with a reservoir. *ACM Trans. Math. Softw.*, 1985.

[23] C. Xia, H. Lu, B. Chin, and O. J. Hu. Gorder: An efficient method for knn join processing. In *VLDB*, 2004.

[24] D. Xie, F. Li, B. Yao, G. Li, L. Zhou, and M. Guo. Simba: Efficient in-memory spatial analytics. In *SIGMOD*, 2016.

[25] J. Yu, J. Wu, and M. Sarwat. Geospark: A cluster computing framework for processing large-scale spatial

data. In *SIGSPATIAL*, 2015.

[26] M. Zaharia. *An Architecture for Fast and General Data Processing on Large Clusters*. Association for Computing Machinery and Morgan, 2016.

APPENDIX

.1 Program of Spatial Operators

```
val datardd=spark.textFile(datafile) // Load spatial data
val LocationRDD = SpatialRDD(datardd).cache() // Generate LocationRDD
val box = Box(23.10094f,-86.8612f, 32.41f, -85.222f)
LocationRDD.rangeFilter(box, udf) // Spatial range query with udf
val k=100
val Q=Point(21.10334f,-86.3332f)
LocationRDD.knnFilter(Q,k,udf)// kNN Query
val rectangles=spark.textFile(sjoinqueryfiles)
LocationRDD.sjoin(rectangles) // Spatial-join
val knnqueries=spark.textFile(knnqueryfiles)
LocationRDD.knnjoin(knnqueries, k) // kNN-join
```

List 1: Spatial Operators in the LocationSpark

.2 Proof of data repartition and query plan optimization problem

Proof. We prove it by reduction from the Bin-Packing problem [11]. Bin-Packing is defined as giving the finite set U of items, the size of each item, says $s(u)$, is a positive real number, while $u \in U$, and a positive integer bin capacity B, and a positive Integer M, we want to find a partition of U into disjoint set $U_1, U_2, ..., U_m$ such that the sum of the size of items in each U_i is B or less. Let $D'_1, D'_2, \ldots, D'_k$ be the solution to the reduced optimal data repartition problem. Next we show that based on $D'_1, D'_2, \ldots, D'_k$, we can obtain the solution to the original Bin-Packing problem in polynomial time.

To reduce this Bin-Packing problem to optimal data repartition problem, we first assume the skew and non-skew set is known. In practical, we can sort data partitions based on their approximate runtime $\mathrm{E}(D_i)$, and then find skew partitions based on certain threshold. This takes polynomial time. Then the runtime over non-skew partition is $\max_{j\in[1,\bar{N}]}\{\mathrm{E}(\bar{D}_j)\} = \Delta$, and is a constant value. Meanwhile, we disregard the minimization condition over $\rho(\bar{Q})$. Therefore, from Equation 5, we can simplify this data repartition problem as

$$\text{minimize}\{\max\{\max_{i\in[1,\hat{N}]}\{\widehat{\mathrm{E}(\hat{D}_i)}\}, \Delta\}\} \tag{7}$$

From this formula, we can find Δ is a constant value, thus, we have to minimize our cost function to be smaller than Δ, that is,

$$\text{minimize}\{\max_{i\in[1,\hat{N}]}\{\widehat{\mathrm{E}(\hat{D}_i)}\}\} \leq \Delta \tag{8}$$

From Equation 4, we can update Equation 8 to be

$$\text{minimize}\{\max_{i\in[1,\hat{N}]}\{\beta(D_i)+\max_{s\in[1,m']}\{\gamma(D_s)+\mathrm{E}(D_s)\}\}+\mathrm{M}(Q_i)\}\} \leq \Delta \tag{9}$$

To simply this function further, we disregard the optimization over $\beta(D_i), \gamma(D_s), textM(Q_i)$, we can simply our problem as following

$$\text{minimize}\{\max_{i\in[1,\hat{N}]}\{\max_{s\in[1,m']}\{\mathrm{E}(D_s)\}\}\} \leq \Delta \tag{10}$$

Suppose the total number of new generated data partition is k', this function is written as this way.

$$\text{minimize}\{\max_{s\in[1,k']}\{\mathrm{E}(D_s)\}\} \leq \Delta \tag{11}$$

Therefore, if $\mathrm{E}(D_s) \leq \Delta$, when $s \in [1, k']$, we can make sure our optimal function is at least less than its upper bound.

In addition, we assume that optimizer could split the skew data partitions into m' partitions and $m' > M$, and each new computed data partition $\hat{D}_i$ is associated with the cost $\widehat{\mathrm{E}(\hat{D}_i)}$. Now, optimizer can pack the computed partitions $\hat{D}_i$ into a disjoint set, s.t., (1) the size of disjoint set is M, because M is maximum number of data partitions. (2) the sum of cost in each bin would be smaller than Δ based on Equation 11. From this way, we can find this is same as the Bin-Packing problem. Thus, if $D'_1, D'_2, \ldots, D'_k$ is the solution of the optimal data repartition problem, then, we can use $D'_1, D'_2, \ldots, D'_k$ to compute the solution for the Bin-Packing problem. Since Bin-Packing is known to be NP-complete, and Bin-Packing is easier than optimal data repartition problem, we can deduce that optimal data repartition problem is also NP-complete. □