On Spatial Joins in MapReduce

Ibrahim Sabek[1], Mohamed F. Mokbel[1,2]
[1]Department of Computer Science and Engineering, University of Minnesota, USA
[2]Qatar Computing Research Institute, HBKU, Qatar
{sabek,mokbel}@cs.umn.edu

ABSTRACT

This paper provides the first attempt for a full-fledged query optimizer for MapReduce-based spatial join algorithms. The optimizer develops its own taxonomy that covers almost all possible ways of doing a spatial join for any two input datasets. The optimizer comes in two flavors; cost-based and rule-based. Given two input data sets, the cost-based query optimizer evaluates the costs of all possible options in the developed taxonomy, and selects the one with the lowest cost. The rule-based query optimizer abstracts the developed cost models of the cost-based optimizer into a set of simple easy-to-check heuristic rules. Then, it applies its rules to select the lowest cost option. Both query optimizers are deployed and experimentally evaluated inside a widely used open-source MapReduce-based big spatial data system. Exhaustive experiments show that both query optimizers are always successful in taking the right decision for spatially joining any two datasets of up to 500GB each.

CCS CONCEPTS

• **Information systems** → **Query optimization**; **Join algorithms**; **MapReduce-based systems**; **Geographic information systems**;

KEYWORDS

Hadoop, MapReduce, Spatial Join, Query Optimization

ACM Reference format:
Ibrahim Sabek, Mohamed F. Mokbel. 2017. On Spatial Joins in MapReduce. In *Proceedings of SIGSPATIAL'17, Los Angeles Area, CA, USA, November 7–10, 2017,* 10 pages.
https://doi.org/10.1145/3139958.3139967

1 INTRODUCTION

Spatial join is a fundamental operation that joins two spatial datasets based on a spatial predicate, e.g., overlap, cover, touch. It is a core operation in many important applications. For example, locating synapses to model human brains [37], determining cells proximity in medical images [49], and identifying intersections in topological features (e.g., cities, roads, rivers) as a means of urban planning [21]. Due to its importance, there have been numerous research efforts in the last three decades to introduce new spatial join algorithms (e.g., see [6, 8–10, 22, 24, 25, 29, 31, 33, 35, 42, 43, 52]). Most recently, and coupled with the recent explosion of big spatial data [17], recent research efforts are dedicated to take advantage of the widely-used MapReduce platform [11] to enable spatial joins for big spatial data [2, 16, 24, 44, 48, 51–53].

Unlike the case of traditional spatial algorithms where there are extensive studies on benchmarking and query optimization issues (e.g., see [4, 18, 20, 23, 28, 30, 41, 45–47]), there are no similar efforts in Map-reduce-based spatial join algorithms. With the wide spectrum of spatial join possibilities in MapReduce environments, it becomes challenging to understand which spatial join technique will be best suited for which input. Unfortunately, query optimization techniques for traditional spatial joins are not applicable to the case of MapReduce spatial joins as they do not take into consideration the specifics of the MapReduce environment.

In this paper, we aim to provide the first attempt of having a comprehensive MapReduce-based spatial join query optimizer. Given two input spatial datasets, our query optimizer would find the best way to execute the spatial join between them. Our query optimizer comes up with a very thorough taxonomy of doing a spatial join in MapReduce. All existing algorithms (e.g., [2, 16, 24, 44, 48, 51–53]) along with non-explored yet algorithms for MapReduce-based spatial join can be mapped to our taxonomy as special cases. Then, given two input spatial datasets, our query optimizer walks through its developed taxonomy to find the lowest cost option. In general, our taxonomy abstracts any MapReduce-based spatial join algorithm to two phases, namely, *partitioning* and *joining* phases, which are executed in the *map*, and *reduce* functions, respectively. The objective of the *partitioning* phase is to ensure that each input data set is *spatially* partitioned into a set of Hadoop File System (HDFS) blocks. And, each partition is annotated with a Minimum Bounding Rectangle (MBR) that encloses all the spatial objects inside it. Then, the *joining* phase produces the output result by joining corresponding partitions from the two input datasets that have overlapping MBRs.

Our query optimizer taxonomy differentiates between three cases of the two input data sets, R and S: (1) Case 1: None of the two input datasets R and S is *spatially* partitioned. In this case, we have to *spatially* partition both R and S. Our query optimizer considers five different options for spatial partitioning, namely, Grid [38], QuadTree [19], KDTree [7], STR [32], and STR+ [32]. Then, a one-to-one join will be applied between corresponding partitions from R and S. (2) Case 2: Only one of the two input datasets, say R, is *spatially* partitioned. In this case, we can either (a) ignore the current partitioning of R and *spatially* partition both R and S on one of the five possible options of Case 1, or (b) *spatially* partition S on the same partitioning scheme of R. In both options, a one-to-one join will be applied between corresponding *spatial* partitions of R and S. (3) Case 3: Both input datasets R and S are

[1]This work is partially supported by the National Science Foundation, USA, under Grants IIS-1525953, CNS-1512877, IIS-1218168.

SIGSPATIAL'17, November 7–10, 2017, Los Angeles Area, CA, USA

ACM ISBN 978-1-4503-5490-5/17/11…$15.00
https://doi.org/10.1145/3139958.3139967

spatially partitioned. In this case, we can either (a) ignore the current partitioning of both R and S and *spatially* repartition them on one of the five possible options of Case 1, (b) ignore the current partitioning of only one of the input datasets, say S, and *spatially* repartition it on the the same partitioning scheme of R, or (c) keep R and S as is and skip the whole partitioning phase. We will then follow by joining corresponding *spatial* partitions of R and S. This spatial join which will end up being one-to-one join in case of the first two options, and a one-to-many join in case we go for the third option, i.e., skipping the partitioning phase.

Our query optimizer comes up in two flavors, *cost-based* and *rule-based*. The *cost-based* query optimizer develops accurate cost models for all the possible options in our taxonomy. These options include: (a) the five options in Case 1 that correspond to the five considered spatial partitioning methods, (b) the six options for Case 2 that correspond to the five options of Case 1 and the option for partitioning one dataset on the other, and (c) the eight options of Case 3 that correspond to the five options of Case 1 in addition to the two options of partitioning one input dataset R or S on the other one, and the last option of skipping the whole partitioning phase. The cost model is built while taking into account three sets of parameters: (1) The characteristics of input data sets, which include the number of partitions in each one of them, the utilization of input partitions, and the ratio of data objects that need to be replicated across multiple spatial partitions. (2) Platform-specific parameters, which include the number of machines in the MapReduce cluster as well as the number of *map* and *reduce* tasks assigned to each machine, and (3) A set of performance statistics that are experimentally evaluated and collected, which include the cost of reading/writing one HDFS partition, the cost of moving one partition from one machine to another, and the cost of in-memory joining two spatial partitions. Our *rule-based* query optimizer abstracts our developed cost model into a set of simple easy-to-check heuristic rules. Then, given two input datasets R and S with their characteristics, we apply our rules to come up with the best spatial join option to join them.

Our query optimizer is deployed inside a widely used open-source MapReduce-based big spatial data system, SpatialHadoop [16]. Within such deployment, we have performed exhaustive experimental evaluation on real big spatial datasets extracted from OpenStreetMap [40] where we join two relations of up to 500GB each. All our experiments show that both our cost-based and rule-based query optimizers are always successful in selecting the right way of doing a spatial join between any two input datasets. There are very few cases where the cost-based query optimizer gets the right choice while the rule-based one did not get it. This is mainly due to the accuracy of the cost model, but that is modulo the overhead in calculating the cost models.

The rest of the paper is organized as follows. Section 2 introduces our taxonomy of MapReduce-based spatial join algorithms. Our cost-based and rule-based query optimizers are presented in Sections 3 and 4, respectively. Section 5 introduces our experimental evaluation. Sections 6 and 7 summarize the related work and conclude the paper.

2 LANDSCAPE OF MAPREDUCE-BASED SPATIAL JOIN ALGORITHMS

Figure 1 gives our proposed landscape for almost all possible ways of executing a spatial join operation in a MapReduce environment.

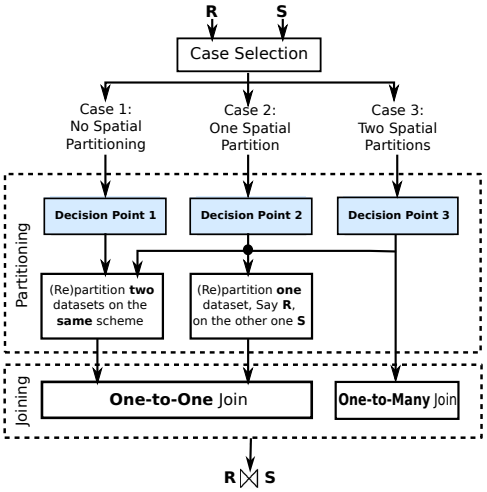


Figure 1: Landscape of Spatial Join Algorithms in MapReduce.

The input to spatial join is two files R and S, stored in the Hadoop Distributed File System (HDFS). The output is a set of objects pairs (r_i,s_i), $r_i \in R$ and $s_i \in S$, where r_i and s_i satisfy a spatial join predicate. Without loss of generality, we assume topological spatial join predicates (e.g., overlap, inside, meet), where the topology of joined object pairs overlap. Our landscape categorizes all MapReduce-based spatial join solutions based on three input cases of R and S: (1) *Case 1*: The two input files R and S are partitioned using the default Hadoop partitioning (Section 2.1). (2) *Case 2*: One input file, say R, is *spatially* partitioned while the other input file, S, has its own default Hadoop partitioning (Section 2.2). (3) *Case 3*: The two input files R and S are *spatially* partitioned (Section 2.3).

In any of these three cases, the spatial join operation is done through two phases, namely *Partitioning* and *Joining*. The main purpose of the *partitioning* phase is to ensure that the HDFS blocks of both input files R and S are partitioned in a way that each block includes data that are spatially related. The *joining* phase is mainly to join corresponding partitions from R and S, i.e., partitions that may have overlapping entries. The key performance of any spatial join operation relies on the ability of minimizing the number of joined partitioned, which is achieved by carefully partitioning each input data set in the *partitioning* phase. Hence, our taxonomy in Figure 1 shows that the *partitioning* phase in each of the three input cases employs a *Decision Point* that decides on how to execute this phase in a way that will be best for the overall spatial join operation. The rest of this section describes the three input cases in details, along with the possible options to the *Decision Point* in each case.

2.1 Case 1: No Spatial Partitioning

In this case, the two input files R and S are partitioned using the default Hadoop HDFS partitioning. Assuming a block size of 128MB, HDFS partitions a file R by storing the first 128MB of R in one partition, then the second 128MB in a second partition, and so on. Since HDFS partitioning has nothing to do with spatial data, nearby objects may go to very different partitions. This means that a straightforward implementation of the spatial join would need to join every single partition from R with every single partition from S, which is prohibitively expensive for large number of partitions. Hence, as far as the spatial join operation is concerned, HDFS partitioning is considered as if the data is not partitioned at all. Figure 2a gives

an example of two non-partitioned input data sets R and S. This is depicted by having one Minimum Bounding Rectangle (MBR) that contains all spatial data in each input file. The details of partitioning and joining phases in Case 1 are as follows:

Partitioning Phase. To avoid a nested loop join of HDFS blocks of R and S, this phase *spatially* partitions the two input files R and S using the same partitioning scheme P. Having the same partitioning scheme P for both R and S enables having one-to-one join between corresponding blocks form R and S. Prior approaches in *spatially* partitioning an HDFS file R on a partitioning scheme P follow a four-steps approach [2, 16, 50]: (1) Load a sample of R in memory, (2) Build an in-memory index P using the sampled data, (3) Allocate a set of HDFS blocks, each annotated with a Minimum Bounding Rectangle (MBR) that corresponds to all lowest level nodes of the index P, (4) Place each record in $r \in R$ in the HDFS block(s) with MBR that overlaps r. However, this approach is geared towards partitioning each file independently, which is not efficient for spatial join, where we should take into consideration the two input data sets together. Hence, we adapt this approach to take the sampled data from both R and S together in a way that is proportional to their sizes. Then, we build one in-memory index P for the two samples. After that, we allocate two independent sets of HDFS blocks for both R and S, yet, with the same exact MBRs. Finally, we place records from each data set into the corresponding allocated HDFS blocks, based on their MBRs. The only decision that needs to be taken in this case is to decide on which partitioning scheme P to be used. Formally, Decision Point 1 is defined as follows:

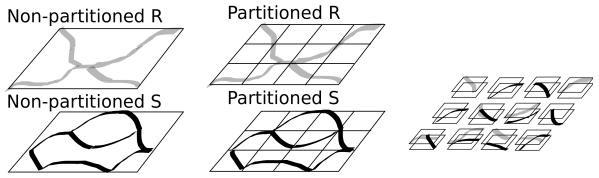
DECISION POINT 1. *Given two input files R and S, find the best partitioning technique that will be used for re-partitioning both of them to minimize the cost of spatial join between R and S.*

Our query optimizer considers the following five partitioning schemes that are mostly used in big spatial data systems [2, 16, 50]

- **Option 1: Grid**. The Grid partitioning technique does not require a random sample as it divides the input space into a set of $\sqrt{M} \times \sqrt{M}$ equal-size grid cells where M is the number of partitions that can accommodate the larger of the two datasets R and S.
- **Option 2: QuadTree**. This partitioning technique constructs an in-memory quad-tree [19] using the sampled data.
- **Option 3: KDTree**. This technique constructs an in-memory KDTree using the sampled data.
- **Option 4: STR**. This technique builds the in-memory index by bulk loading an R-tree [26] using the STR bulk loading algorithm [32]. The leaf nodes of the R-tree are used as boundary partitions. As the partitions may not cover the whole space, partitions may expand to cover all input data.
- **Option 5: STR+**. This technique is similar to STR, except that objects that overlap with multiple cells are replicated.

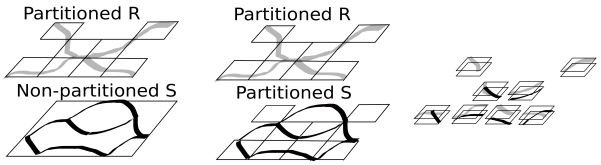
Figure 2b gives an example where both input datasets, R and S, are partitioned using Grid partitioning on a 4×3 grid.

Joining Phase. Since both R and S are partitioned using the same scheme, their HDFS blocks will have the same exact Minimum Bounding Rectangles (MBR). Hence, each partition in R will be joined with only one partition from S that has the same MBR, i.e., one-to-one join. With a default partition size of 128MB, two partitions from R and S can easily fit in memory. Therefore, joining



(a) Input of Case 1 (b) Partition R, S (c) One-to-One Join

Figure 2: Case 1 - Option 1.



(a) Input of Case 2 (b) Partition S (c) One-to-One Join

Figure 3: Case 2 - Option 6.

a pair of partitions is done using any traditional in-memory spatial join algorithm (e.g., see [30] for a comprehensive survey of such algorithms). Since a spatial data object may be stored in more than one overlapping partition, we employ a duplicate avoidance technique [13] within our in-memory spatial join algorithm to ensure that each pair of joined objects will be produced only once. Figure 1c shows the joining phase for Case 1, where each of the 12 R partitions is joined with only one partition from S.

2.2 Case 2: One Spatial Partition

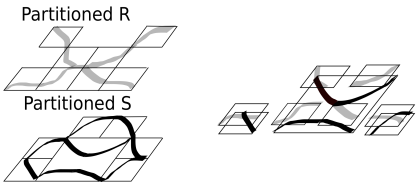
In this case, one input file, say R, is spatially partitioned while the other input file, S, has its own default Hadoop partitioning, i.e., non-partitioned with respect to spatial join. An input file could be spatially partitioned if it is stored in one of the MapReduce-based spatial systems, e.g., ESRI Hadoop [50], Hadoop-GIS [2], SpatialHadoop [16]. Figure 3a gives an example of input files, where R is only spatially partitioned using a QuadTree. The details of partitioning and joining phases are as follows:

Partitioning Phase. As depicted in Figure 1, our query optimizer would go through on of these two ways: (1) Ignore the partitioning of R, and repartition both R and S on a new partitioning scheme as in Case 1, or (2) Spatially partition S on the same scheme of R. Formally, Decision Point 2 is defined as follows:

DECISION POINT 2. *Given two input files R and S, where only R is spatially partitioned, find whether we should ignore R partitioning and repartition it with S on a similar partitioning (Case 1), or we should only spatially partition S on the same scheme of R.*

Given the five partitioning schemes we had for Case 1, this decision point needs to select one of these six options:

- **Options 1 to 5:** Ignore the partitioning of R, and re-partition R along with S using one of the five partitioning techniques described in Case 1 (Section 2.1). To select the best partitioning scheme, we go through Decision Point 1.
- **Option 6:** Keep the partitioning of R, and partition only S based on the boundaries of R. In this case, there is no need to bulk load a sample of S into an in-memory index to get the partition boundaries. Instead, we use the partition boundaries as is from S.



(a) Input of Case 3 (b) One-to-Many Join

Figure 4: Case 3 - Option 8.

Figure 3b shows using Option 6 for the input of Figure 3 where, S is partitioned using the same QuadTree of R.

Joining Phase. All the six options of the partitioning phase end up in having both R and S partitioned on the same partitioning scheme. Then, the joining phase ends up to be a one-to-one join between corresponding partitions, similar to the joining phase in Case 1. Figure 3c shows the one-to-one join process in this case.

2.3 Case 3: Two Spatial Partitions

In this case, the two input files R and S are already spatially partitioned, though it is not on the same partitioning scheme. Figure 4a gives an example where R is spatially partitioned using a QuadTree, while S is spatially partitioned using a KDTree. The details of partitioning and joining phases in Case 3 are as follows:

Partitioning Phase. As depicted in Figure 1, our query optimizer would go through on of these three ways: (1) Ignore the partitioning of R and S, and repartition both of them on a new partitioning scheme as in Case 1, (2) Ignore the partitioning of one of the two input files, say S, and repartition it according to the other input file R, same as in Case 2, or (3) Skip the partitioning phase, and proceed with R and S as is to the joining phase. Formally, Decision Point 3 is defined as follows:

DECISION POINT 3. *Given two spatially partitioned input files, R and S, find whether we should re-partition both R and S on the same partitioning scheme as in Case 1, only re-partition R (S) on the partitioning scheme of S (R) as in Case 2, or proceed with R and S as is to the joining phase.*

Given the five partitioning schemes we had for Case 1, this decision point needs to select one of these eight options:

- **Options 1 to 5:** Ignore the partitioning of R and S, and repartition them using one of the five partitioning techniques described in Case 1 (Section 2.1). To select the best partitioning scheme, we go through Decision Point 1.
- **Options 6 and (7):** Same as Option 6 in Case 2, ignore the partitioning of one of the two input files, say R (S), and re-partition it on the same partitioning scheme of the other file S (R).
- **Option 8:** Skip the partitioning phase, and go directly to the joining phase.

Joining Phase. In case we go with any of the first seven options in the partitioning phase, the joining phase will be simply a one-to-one join between corresponding R and S partitions, as in Cases 1 and 2. When going for Option 8, in the rare case that both input files are partitioned using the same exact partitioning scheme, we go for a one-to-one join as in the first seven options. Otherwise, we go for a one-to-many join where one partition from R can be joined with multiple partitions from S. Figure 4b shows the one-to-many join where one of S partitions is joined with four partitions from R.

Algorithm 1 Function SPATIALJOIN(Dataset R, Dataset S)

```
1: P ← { GRID, QUAD, KDTREE, STR, STRPLUS }
2: PartR ← NULL, PartS ← NULL, JoinMethod ← OnetoOne
       /* Evaluating Options 1 to 5 (Decisions 1, 2 and 3) */
3: MinC ← Min. cost of five partitioning options P (Sec. 3.1)
4: PartR, PartS ← Partitioning p with MinC, where p ∈ P
       /* Evaluating Options 6 and 7 (Decisions 2 and 3) */
5: if R is spatially partitioned then
6:     if MinC ≥ Cost of re-partitioning S on R (Sec. 3.2) then
7:         MinC ← Cost of re-partitioning S on R
8:         PartR ← NULL, PartS ← Partitioning of R
9:     end if
10: end if
11: if S is spatially partitioned then
12:     if MinC ≥ Cost of re-partitioning R on S (Sec. 3.2) then
13:         MinC ← Cost of re-partitioning R on S
14:         PartR ← Partitioning of R, PartS ← NULL
15:     end if
16: end if
       /* Evaluating Option 8 (Decision 3) */
17: if Both R and S are spatially partitioned then
18:     if MinC ≥ Cost of joining R and S (Sec. 3.3) then
19:         PartR, PartS ← NULL, JoinMethod ← OnetoMany
20:     end if
21: end if
22: if PartR then Re-partition R using PartR end if
23: if PartS then Re-partition S using PartS end if
24: return JOIN (R, S, JoinMethod)
```

3 COST-BASED QUERY OPTIMIZER

This section discusses our spatial join query optimizer that aims to find out the lowest cost way to do a spatial join for two input datasets R and S. The query optimizer evaluates the cost of all possible options for the three decision points discussed in Section 2, and depicted in Figure 1, to come up with the lowest cost option.

Algorithm. Algorithm 1 depicts the pseudo code for our query optimizer that takes two input datasets R and S and returns the spatial join result between them. The algorithm keeps track of the current best (i.e., lowest cost) option through three variables: *PartR* and *PartS*, initialized by *Null*, that indicate the way to (re)partition R and S, respectively, and *JoinMethod*, initialized by *OnetoOne* that indicates how to do the joining phase. The algorithm then starts by computing the cost for the five possible options of partitioning both R and S, namely, Grid, QuadTree, KDTree, STR, and STR+. The details of cost computations for each partitioning method are described later in Section 3.1. Note that we do these computations regardless of whether R and S are spatially partitioned or not, as these five options will be taken into consideration in each of the three decision points, described in Section 2. Based on these computations, we update the values of *PartR* and *PartS* to include the partitioning scheme that had the lowest computed cost (Line 4 in Algorithm 1). If neither R nor S is spatially partitioned (i.e., Case 1), we skip the rest of the algorithm till Line 22, where we partition R and S on *PartR* and *PartS*, and return the final result using *OnetoOne* join.

If any of R and S is spatially partitioned, we would need to evaluate the cost of the option of partitioning R using S partitioning scheme and vice versa (Lines 5 to 16 in Algorithm 1). The details of the cost computations of partitioning one input file on the other are described later in Section 3.2. Note that these computations is done even if both R and S are spatially partitioned as this will be needed for Decision Points 2 and 3. In case, any of these computations ends up to be of a lower cost than the best of the five options computed earlier, we set a new minimum computation cost, and update the values of *PartR* and *PartS*, accordingly. If only one of R and S is spatially partitioned (i.e., Case 2, Section 2.2), we skip the rest of the algorithm till Line 22. In this case, one of *PartR* and *PartS*

was set to *Null*, which means that one of R and S would be repartitioned, while the other one will be left intact. Then, we end up doing *OnetoOne* join between R and S.

Finally, if both R and S are spatially partitioned (i.e., Case 3, Section 2.3), we would need to compute the cost of Option 8 in Decision Point 3, which skips the partitioning phase, and go directly to one-to-many join. The details of such cost computations are described later in Section 3.3. If this cost ends up to be lower than any of the earlier options, we set both *PartR* and *PartS* to *Null*, and the join method to be *OnetoMany*. Then, we directly proceed to one-to-many join of R and S without doing any repartitioning.

Spatial join cost. In all cases, the cost of the spatial join operation is the sum of the costs of both the *partitioning* and *joining* phases. The cost of the *partitioning* phase is composed of two components: (1) The cost of estimating the partitioning boundaries of the two input datasets, C_{est}, and (2) The cost of actually partitioning the objects in the two input datasets, C_{part}. Meanwhile, the cost of the *joining* phase is composed of two components: (1) The cost of moving the pairs of overlapping partitions into the same node, C_{prep}, and (2) The cost of actually joining the contents of these overlapping partitions C_{join}. Thus, the total cost is computed as:

$$C_{tot} = C_{est} + C_{part} + C_{prep} + C_{join} \quad (1)$$

The rest of this section calculates these costs for all the options that are mentioned in Algorithm 1.

Used terms. Throughout this section, we will use the following terms: N_R and N_S are the number of partitions, i.e., HDFS blocks, in input files R and S, respectively. H is the number of machines in the MapReduce cluster. K and L are the number of *map* and *reduce* tasks in each machine, respectively. Without loss of generality, we assume that input file R is larger than S. In case that only one of the two input files is spatially partitioned, we assume that it is R.

3.1 Evaluating Costs of Options 1 to 5

This section computes the spatial join cost for Options 1 to 5, which will feed into Line 4 in Algorithm 1. Per Equation 1, this cost is composed of the following four terms:

Estimation cost, C_{est_1}. In case of Grid partitioning, C_{est_1} is set to 0 as the partition boundaries are preset regardless of input data. For other partitioning schemes, bulk loading a sampled data would require reading all the input partitions form R and S. Assuming a *single* machine, this would cost $\left[N_R + N_S\right]C_e$, where C_e is the cost of reading and sampling one partition. C_e is similar for all five options, where it is experimentally evaluated and stored as a constant during the sampling phase. However, since we have a total of $K \times H$ map tasks running in *parallel*, C_{est_1} can be expressed as:

$$C_{est_1} = \left[\frac{N_R + N_S}{KH}\right]C_e$$

Partitioning cost, C_{part_1}. This cost depends on the number of partitions that will be allocated for spatially repartitioning R and S. Since both R and S will be spatially partitioned on the same scheme, they would have the same number of spatial partitions, regardless of their input size. That number will be dependent on the number of input partitions of the larger input, N_R. However, the total number of allocated spatial partitions for both R and S may be more or less than the $2 \times N_R$, based on two main factors:

(1) *Block Utilization (U_R).* This is the utilization value between 0 and 1 that represents how much the HDFS blocks are utilized, with 1 means fully utilized. In case R has default Hadoop partitioning, U_R is set to 1 as an inherent property of HDFS blocks. In case R has spatial partitioning, U_R ranges between 0 and 1. U_R is computed for the larger input R while estimating its boundaries.

(2) *Data Replication (D_p).* This is the average number of partitions that an input $r \in R$ would overlap with their boundaries, as r would be replicated in each overlapping object. In case of STR partitioning, D_p is set to 1 as it has the inherent property that partitions can expand to contain any object as a means of avoiding replication. For other options, D_p is experimentally computed in the sampling phase for each scheme. Hence, our query optimizer stores four constant values for D_p as one for each scheme.

Hence, the number of allocated partitions would be $2N_RU_RD_p$. In case both U_R and D_p are one, i.e., 100% utilization and no replication, the number of output partitions will be double the number of input partitions of the largest input. Assuming a *single* machine, the cost of writing the output partitions would be $2N_RU_RD_pC_b$, where C_b is the cost of writing one partition. C_b is the same value for all options, where it is experimentally evaluated in the sampling phase and stored as a constant. However, since we have a total of $K \times H$ map tasks running in *parallel*, C_{part_1} can be expressed as:

$$C_{part_1} = \left[\frac{2N_RU_RD_p}{KH}\right]C_b$$

Preparation cost, C_{prep_1}. This is the cost of shuffling the new spatial partitions of R and S such that all pairs of joined partitions are in the same machine. Since our join is one-to-one, and the number of spatial partitions for R and S is the same, we would need to shuffle only the partitions of one of these relations, say R. As discussed in the partitioning cost above, the number of spatial partitions for R would be $N_RU_RD_p$. Assuming a *single* machine, the cost of shuffling the spatial partitions of one input would be $N_RU_RD_pC_p$, where C_p is the cost of moving one partition to another machine. C_p is the same value for all options, where it is experimentally evaluated and stored as a constant during the sampling phase. However, the preparation cost is part of the joining phase, which is all performed through the reducers. Since we have a total of $L \times H$ reducers running in *parallel*, C_{prep_1} can be expressed as:

$$C_{prep_1} = \left[\frac{N_RU_RD_p}{LH}\right]C_p$$

Joining cost, C_{join_1}. We would need to do an in-memory one-to-one join for all the $N_RU_RD_p$ spatial partitions in each input dataset. Given that the cost of in-memory joining one pair of partitions is C_j (experimentally evaluated and stored as a constant during the sampling phase), and that this will be done in parallel, using $L \times H$ reducers, C_{join_1} can be expressed as:

$$C_{join_1} = \left[\frac{N_RU_RD_p}{LH}\right]C_j$$

TOTAL COST, C_{tot_1}. The total cost of spatial join using any of the first five options is the sum of *estimation*, *partitioning*, *preparation*, and *joining* costs described above.

$$C_{tot_1} = \left[\frac{N_R + N_S}{KH}\right]C_e + \frac{N_RU_RD_p}{H}\left[\frac{2C_b}{K} + \frac{C_p + C_j}{L}\right] \quad (2)$$

Note that K, L, and H are given environment parameters, N_R and N_S are input parameters, U_R is computed during the estimation process, D_p is an experimentally estimated number for each

partitioning method, while C_e, C_b, C_p, and C_j are experimentally estimated during the sampling phase, regardless of the partitioning method, and stored as constants in our query optimizer.

3.2 Evaluating Costs of Options 6 and 7

This section computes the spatial join cost for Options 6 and 7, which will feed into Lines 6 and 12 in Algorithm 1. Without loss of generality, we would assume that R is spatially partitioned, while S is not. So, we compute the cost of partitioning S on R scheme. Per Equation 1, this cost is composed of:

Estimation cost, C_{est_2}. This cost is set to zero as we will the use partition boundaries of R as is, without doing any estimation.

Partitioning cost, C_{part_2}. This cost is computed in a similar way to the partitioning cost in Section 3.1, except for two differences: (1) Since we only need to partition one dataset, then the term $2N_R$ will be replaced by only N_R, and (2) Since we know the exact number of partitions N_R that we will generate for S, there is no need to use the utilization and replication factors to estimate it.

$$C_{part_2} = \lceil \frac{N_R}{KH} \rceil C_b$$

Preparation (Joining) cost, C_{prep_2} (C_{join_2}). This is also similar to the preparation (joining) cost computed in Section 3.1, except that there is no need to use the utilization and replication factors as we already know that we need to exactly move N_R partitions.

$$C_{prep_2} = \lceil \frac{N_R}{LH} \rceil C_p,\ C_{join_2} = \lceil \frac{N_R}{LH} \rceil C_j$$

TOTAL COST, C_{tot_2}. The total cost of spatial join when partitioning input S on the partitioning scheme of R, is the sum of *partitioning*, *preparation*, and *joining* costs described above.

$$C_{tot_2} = \frac{N_R}{H} \lceil \frac{C_b}{K} + \frac{C_p + C_j}{L} \rceil \quad (3)$$

3.3 Evaluating Cost of Option 8

This section computes the spatial join cost for Option 8, which will feed into Line 18 in Algorithm 1. In particular, we compute the cost of joining R and S directly using a one-to-many join, without any partitioning. Per Equation 1, this cost is composed of:

Estimation (Partitioning) cost, C_{est_3} (C_{part_3}). As this option does not encounter any partitioning, both these costs are set to 0.

Preparation (Joining) cost, C_{prep_3} (C_{join_3}). This cost is based on the number of overlapping partitions between R and S, as this will present the number of partitions that we need shuffle (join) between nodes. The number of overlapping partitions is estimated as $\beta_{RS} N_R N_S$, where β_{RS} is a value between 0 and 1, computed as the selectivity estimation of joining R and S, per the formula in [5]. Hence, similar to the preparation (joining) cost computed in Section 3.2, this cost will be

$$C_{prep_3} = \lceil \frac{\beta_{RS} N_R N_S}{LH} \rceil C_p,\ C_{join_3} = \lceil \frac{\beta_{RS} N_R N_S}{LH} \rceil C_j$$

TOTAL COST, C_{tot_2}. The total cost of spatial join when skipping the partitioning phase and directly doing one-to-many join between R nd S is the sum of *preparation* and *joining* costs described above.

$$C_{tot_3} = \lceil \frac{\beta_{RS} N_R N_S}{LH} \rceil \lceil C_p + C_j \rceil \quad (4)$$

4 RULE-BASED QUERY OPTIMIZER

Typically, a cost-based optimizer is accurate, yet, expensive as it requires collecting and maintaining statistics as well as estimating parameters to calculate its cost equations. Hence, in this section, we introduce our *rule-based* query optimizer that abstracts our developed cost model in Section 3 to a set of simple easy-to-check three heuristic rules. Then, given two input datasets R and S, with N_R and N_S input partitions (i.e., HDFS blocks), respectively, we apply the following three rules to come up with the best possible option to execute a spatial join between R and S.

RULE 1. *In case that neither R nor S is spatially partitioned: If both N_R and N_S are less than a given threshold t, then we partition both R and S on Grid (Option 1), otherwise, we partition both R and S on STR (Option 4).*

This rule is mainly for Case 1 (Section 2.1), where we need to select among five possible partitioning options, all with the cost of Equation 2 in Section 3.1. The main two factors in this equation are the first part of it, which presents the estimation cost C_{est_1} and then $2N_R U_R D_p$. In case both N_R and N_S are considered of small size (i.e., less than a system threshold t), the number of partitions $2N_R U_R D_p$ becomes very small, and hence the effect of partitioning and joining costs will decrease. This makes the effect of estimation cost C_{est_1} a dominating factor, and hence we will select the partitioning option with lowest estimation cost, which is Grid as it has zero cost for boundary estimation. In case either N_R or N_S is considered of large value (i.e., more than a system threshold t), the number of partitions $2N_R U_R D_p$ becomes the dominating factor. However, N_R and U_R would have the same value regardless of all partitioning schemes, as these are properties of the input data. So, the value D_R becomes the decisive factor, and we go with the STR option as it has the lowest possible value for D_P, which is one.

RULE 2. *In case only one input dataset, say R, is spatially partitioned: If both N_R and N_S are less than a given threshold t, then we partition both R and S on Grid (Option 1), otherwise, we partition S on the same partitioning scheme of R (Option 6).*

This rule is mainly for Case 2 (Section 2.1), where we need to select among six possible options; five of them are calculated from Equation 2, while the sixth option is calculated from Equation 3. In case both N_R and N_S are small (i.e., less than a system threshold t), per Rule 1, the best option in Equation 2 would be Grid partitioning, where the first part of the equation is set to zero. Comparing this with Equation 3 will make the utilization factor U_R a dominating factor. We roughly assume that for small data sizes, U_R would be of a small value, which will make the term $N_R U_R D_R$ much less than N_R, and hence Equation 2 will be less than Equation 3, and we will go with Grid partitioning for both R and S (Option 1). In case either R or S is large, per Rule 1, we will use STR partitioning of Equation 2 where D_p is set to 1. Considering that U_R would be closer to 1 and that we do have an additional term in Equation 2 for the boundary estimation, we will consider that Equation 3 would have less value, hence we will go with Option 6, i.e., partitioning S on the same partitioning scheme of R. Note that we have made some approximation decisions here. We could have make our rules more complicated by considering the actual value of U_R, and decide accordingly. Yet, that will defeat the purpose of having simple set of rules that approximate our actual cost model.

RULE 3. *In case both R and S are spatially partitioned: Assuming that R is larger than S, if $\beta_{RS}N_S \leq 1$, we skip the whole partitioning phase (Option 8), otherwise, we partition both R and S on Grid (Option 1)* only if *both N_R and N_S are less than a given threshold t, otherwise we repartition S, on the same partitioning scheme of R (Option 6).*

This rule is mainly for Case 3 (Section 2.3), where we need to select among eight possible options; five of them are calculated from Equation 2, two are calculated from Equation 3, and one is calculated from Equation 4. We assume that one of the relations, say R, is larger than the other one S. In all cases, the main decisive factor here is the selectivity estimation β_{RS} of joining both R and S. Comparing Equations 2, 3, and 4, if $\beta_{RS}N_S \leq 1$, then definitely Equation 4 would have less cost than Equation 3 as $\beta_{RS}N_RN_S$ would be less than N_R. Also, if $\beta_{RS}N_S < 1$, it is highly likely that Equation 4 would have less cost than Equation 2 as $\beta_{RS}N_RN_S$ is likely to be less than $2N_RU_RD_P$. This means that we can set a rule that if $\beta_{RS}N_S \leq 1$, we just ignore the whole partitioning phase and go directly to one-to-many join (Option 8). Note that this is kind of conservative assumption, which is acceptable for a rule-based query optimizer. In case that $\beta_{RS}N_S > 1$, we would operate in a similar way as in Rule 2, where we will partition both R and S on a grid if both of them are considered small, otherwise, we will repartition the smaller file S into the partitioning scheme of R.

Algorithm. Algorithm 2 gives the pseudo code for our *rule-based* optimizer that applies the above three rules to find the best way to do a spatial join between two input datasets R and S. The algorithm keeps track of the current best options through three variables: *PartR* and *PartS*, initialized by *Null* and *JoinMethod*, initialized by *OnetoOne*. The algorithm starts by checking the condition $\beta_{RS}N_S \leq 1$ that is needed in Rule 3. If the condition is true, we set the join method to *OnetoMany*, skip the partitioning phase, and jump to the end of algorithm to join R and S on a *one-to-many* join. If the condition is false, we check if both R and S are small, i.e., the number of blocks N_R and N_S is less than a certain system threshold t. If this is the case, we set both *PartR* and *PartS* to Grid (Line 6 in Algorithm 2) and proceed towards the end of the algorithm to partition both R and S on Grid and do a one-to-one join. Note that we do so regardless of whether R and S are spatially partitioned or not, as this is a common action in the three rules above. In case either R or S is large, we check if the larger dataset R is already spatially partitioned. If this is the case, we set *PartS* to the partitioning scheme of R, which is a common case in Rules 2 and 3 (Line 9 in Algorithm 2). Then, we proceed towards the end of the algorithm to partition S and do a one-to-one join. In contrast, if the smaller dataset S is the one that is partitioned, we will set *PartR* to the partitioning scheme of S, which takes place only in Rule 2. Finally, in case neither R nor S is spatially partitioned, we apply Rule 1 and set both *PartR* and *PartS* to STR (Line 14 in Algorithm 2), followed by one-to-one join.

5 EXPERIMENTS

In this section, we experimentally evaluate the performance of our query optimizer with its two variants; Cost-based, described in Section 3, and Rule-based, described in Section 4. To the best of our knowledge, our query optimizer is the first optimizer specifically designed for spatial join algorithms in the MapReduce environment. The closest competitor to us is AsterixDB [3], a big data management system, that supports spatial joins and contains a general rule-based query optimizer. Therefore, we first compare the performance of our query optimizer with AsterixDB in Section 5.1. Then, in Section 5.2, we extensively investigate the performance of the two variants of our query optimizer according to different datasets characteristics. Finally, in Section 5.3, we show the effect of collected statistics and estimated parameters on the accuracy of our query optimizer.

Algorithm 2 Function SPATIALJOIN(Dataset R, Dataset S)

```
1: PartR ← NULL, PartS ← NULL, JoinMethod ← OnetoOne
2: if Both R and S spatially partitioned and β_RS N_S ≤ 1 then
3:     JoinMethod ← OnetoMany
4: else
5:     if N_R ≤ t and N_S ≤ t then
6:         PartR, PartS ← GRID
7:     else
8:         if R is spatially partitioned then
9:             PartS ← Partitioning of R
10:        else
11:            if S is spatially partitioned then
12:                PartR ← Partitioning of S
13:            else
14:                PartR, PartS ← STR
15:            end if
16:        end if
17:    end if
18: end if
19: if PartR then Re-partition R using PartR end if
20: if PartS then Re-partition S using PartS end if
21: return JOIN (R, S, JoinMethod)
```

Parameter	Estimated Value
Reading and sampling one block, C_e	14 MS
Writing one block, C_b	46 MS
Moving one block to remote machine, C_p	80 MS
Joining two blocks, C_j	140 MS
Data replication of Grid, D_{grid}	3.1
Data replication of QuadTree, D_{quad}	4.6
Data replication of KDTree, D_{kdtree}	6.1
Data replication of STR+, D_{str+}	2.2

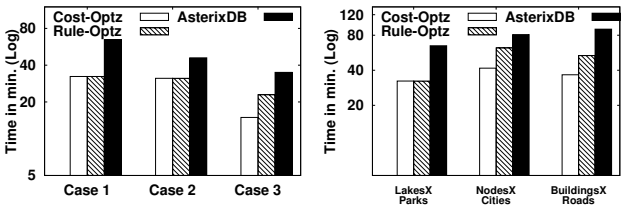
Table 1: Estimated parameters and statistics for cost model.

Parameter	Default Value
HDFS Block Capacity	128 MB
Cluster Size (H)	10
Num. Mappers / Machine (K)	14
Num. Reducers / Machine (L)	8
Map Task Memory Size	6 GB
Reduce Task Memory Size	8 GB

Table 2: Different MapReduce parameters.

Datasets. All experiments are based on collected data from OpenStreetMap [40]. Unless mentioned otherwise, our datasets contain the boundaries representation of two geometric features; Lakes and Parks around the whole world, where these datasets consist of polygon objects. Each object record consists of an identifier, spatial boundaries information, and a sequence of tags, each of which contains a set of strings. These two datasets are synthesized to cover different data sizes and skewness as shown later.

Statistics and Parameters. We estimate the statistics of our cost model, described in Section 3, as in Table 1 and tune the MapReduce parameters that will be used in the remaining experiments as in Table 2. To obtain unbaised HDFS block-based statistics, we join two different datasets; Buildings and Cities from OpenStreetMap [40], rather than Lakes and Parks. Each of these datasets has four versions with sizes 10GB, 50GB, 100GB and 500GB. The value of each parameter is estimated for each combination of these versions, and then such values are averaged. To estimate the data replication statistics, we apply the four partitioning techniques, corresponding to Options 1, 2, 3 and 5, on the Buildings, and calculate the replication factor average of each technique. To tune the MapReduce



(a) Different Input Cases (b) Different Data Sizes

Figure 5: Comparison with AsterixDB Query Optimizer.

parameters, we run the spatial join in Case 1 with Option 4, which uses the STR partitioning scheme to partition both input datasets and then applies a one-to-one join between the generated partitions. In this experiment, we join two pairs of uniform datasets from Lakes and Parks; 500Gx3G and 500Gx500G. Finally, we set the system threshold t of our rule-based optimizer to the number of partitions in a dataset of $60GB$, which can fit in the memory of one machine.

Environment. We run all experiments on a cluster of 10 machines (one master and 9 slaves), which is running Hadoop 2.6.0, SpatialHadoop 2.4.2, and Ubuntu Linux 14.04. Each machine has 8 quad-core 3.00 GHz processors, 64GB RAM, and 4TB hard disk.

Metrics. In all experiments, we use the total running time of the spatial join operation as an evaluation metric. We focus on joining two spatial datasets with an overlap predicate as it is widely used. However, the performance conclusions can be generalized to the remaining topological spatial join predicates, e.g., inside, meet.

5.1 Comparison with AsterixDB Query Optimizer

In this section, we compare the performance of our query optimizer with its two variants; Cost-based (termed *Cost-Optz*) and Rule-based (termed *Rule-Optz*), and the AsterixDB query optimizer while having the three input cases and using three different input datasets. To have a fair comparison, we deployed both SpatialHadoop (the host of our query optimizer) and AsterixDB on the same cluster of machines, and using the same parameters shown in Table 2.

Figure 5(a) shows the running time for each query optimizer in the three input cases. In this experiment, we join Lakes and Parks datasets, each of size 500G. In Case 2, we only index the Lakes dataset with STR, and in Case 3, we index both datasets with STR as well. For the three input cases, both *Cost-Optz* and *Rule-Optz* were able to significantly reduce the running time compared to the AsterixDB optimizer. Specifically, both our query optimizer variants and AsterixDB optimizer have an average running time of 27.47 and 48.57 minutes, respectively. The poor performance of AsterixDB comes from using generic optimization hints that are not designed for performing spatial joins, for example, using an index whenever it exists. However, it could be better to ignore existing indexes and spatially re-partition data from scratch before applying the join operation as suggested by our query optimizer. Note that *Cost-Optz* and *Rule-Optz* have the same running time in Case 1 and Case 2 because they both select Option 1. However, in Case 3, *Cost-Optz* and *Rule-Optz* select Option 8 and Option 6, respectively.

Figure 5(b) shows the running time of each query optimizer while joining three different pairs of input datasets in Case 1. These datasets include different geometric features from OpenStreetMap; (1) Lakes and Parks, which contain polygon objects, (2) Nodes and Cities, which contain point and polygon objects, respectively, (3) Buildings and Roads, which contain polygon and line objects, respectively. Each of these datasets has a size of 500GB. As can be seen in the figure, both *Cost-Optz* and *Rule-Optz* significantly outperform the AsterixDB optimizer. In the second dataset, both *Cost-Optz* and *Rule-Optz* are 2 and 1.3 times faster than the AsterixDB optimizer, respectively. The reason for that is AsterixDB builds an STR-like [3] index for non-indexed datasets as an initial optimization step, which is not the optimal option for the nearly-uniform Nodes (points) dataset. In contrast, *Cost-Optz* selects the Grid option, i.e. Option 1, which is the best solution for uniform distributions. Note that *Rule-Optz* will choose Option 4, which is similar to AsterixDB, however, it still outperforms the AsterixDB performance because of the implementation efficiency of spatial join in SpatialHadoop. The same performance conclusions apply to the third dataset where both variants are still able to be 2.5 and 1.7 times faster than the AsterixDB optimizer, respectively.

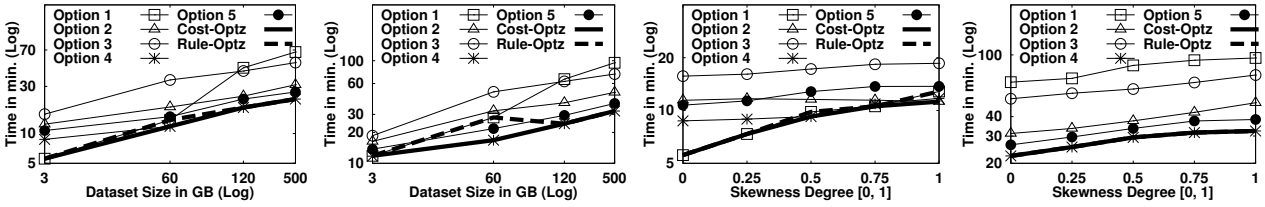
5.2 Effect of Input Data Characteristics

This section evaluates the performance of our query optimizer according to different input data characteristics. Given an input case, we compare the running time of the spatial join options decided by *Cost-Optz* and *Rule-Optz* with the running time of all options in that input case. In our experiments, we focus only on studying input Case 1, and omit the results of Case 2 and Case 3 because our query optimizer exhibits a similar performance behavior.

To generate a uniform variation of any dataset, e.g. Lakes, we select a fixed uniformly-chosen fraction of objects from dense area. Then, we replicate each object many times at random positions in its neighborhood to obtain uniform datasets with different sizes. To generate a skewed dataset, we replicate objects similar to the uniform dataset, however, with much higher densities at some areas compared to others. We assume that each dataset is mapped to a fixed grid, where each cell could have dense area or not. Then, we use the ratio of dense areas to the total number of cells in this grid as skewness degree, where 0 means most uniform and 1 means most skewed. We assume that a 1% sample of data is enough to estimate the partitioning boundaries in all options, as shown in [14]. We also assume that the best number of grid cells is predefined for each dataset size, so that each cell contains 128MB of data at maximum.

Varying Dataset Size. Figure 6(a) provides the overall performance of *Cost-Optz*, *Rule-Optz* and the five spatial join options in Case 1 while varying the Parks dataset size from 3GB to 500GB and fixing the Lakes dataset at 500GB. The two datasets are completely uniform (Skewness degree is 0). *Cost-Optz* is able to cover the best spatial join options at all different sizes. However, *Rule-Optz* misses the optimal decision at dataset size 60GB, where it selects Option 1, i.e. using Grid, instead of Option 4, i.e. using STR. This confirms the accuracy of our cost model in *Cost-Optz*, compared to the *Rule-Optz*, however, that comes with the overhead of maintaining statistics.

In Figure 6(b), the experiment is run with the same five options and dataset size settings in Figure 6(a), however, the objects of the two datasets are completely skewed (skewness degree is 1). Compared to the uniform case, there is no difference in the performance behavior of the two variants. We can see from the figure that *Cost-Optz* is still able to predict the best options at the different sizes, while *Rule-Optz* misses at most one case, specifically at size of 60GB. However, the *Rule-Optz*'s decision is 66% slower than *Cost-Optz*'s



(a) Uniform Datasets (b) Skewed Datasets (c) Small Datasets (d) Large Datasets

Figure 6: Effect of data size and skewness on the overall performance.

one in case of skewed datasets, while the *Rule-Optz*'s decision is 16% slower than *Cost-Optz*'one in case of uniform datasets.

Varying Dataset Skewness. Figure 6(c) gives the overall performance of *Cost-Optz*, *Rule-Optz* and the five options in Case 1 while varying the skewness degree. We fix the Parks and Lakes datasets at sizes of 3GB and 3GB, while changing the skewness degree from 0 to 1. Clearly, options 1 and 4 give the best performance in all skewness cases, and *Cost-Optz* still selects them as the best decision. However, *Rule-Optz* misses the best options at degrees of 0.5 and 1, yet, with only 10% delay from the optimal running time. The reason for that is, though the input datasets are small, it could happen that the QuadTree prunes some empty partitions, and hence the number of allocated partitions $2N_R U_R D_p$ becomes smaller than the ones generated by other options. Considering such corner cases in the *Rule-Optz* will complicate the decision making process, yet, it will increase its accuracy. This leaves a room for future research to come up with better rules using our cost model. In our case, the main objective of our rule-based query optimizer is to prune obvious expensive options, while trying to estimate the best solution.

Figure 6(d) shows the performance of *Cost-Optz* and *Rule-Optz* while fixing datasets of sizes of 500GB and 500GB and varying the skewness degree. Both variants are able to obtain the best spatial join running times with an accuracy of 100%. This confirms the efficiency of our query optimizer in the large skewed input cases, which are typical input scenarios. The efficiency comes from considering the block utilization, which is a skewness-affected factor, in the decisions and cost modeling of our optimizer.

5.3 Effect of Statistics and Parameters Estimation

This section presents the effect of parameters and statistics estimates on the performance of the query optimizer with its two variants. Using four different experiments, we study the effect of the parameter estimates in the three input cases and with both skewed and uniform datasets. We simulate Case 3 by indexing both Lakes and Parks datasets, each of size 500GB, with QuadTree. For Case 2, we only keep the Lakes index and ignore the Parks index.

Figures 7(a) and 7(b) show the performance of three variants of *Cost-Optz* while using uniform and skewed datasets. These variants are (1) typical *Cost-Optz*, (2) *Cost-Optz* without data replication D_p (i.e. $D_p = 1$), and (3) *Cost-Optz* without utilization U_R ($U_R = 1$). Such experiment shows the effect of ignoring either the replication or the utilization estimate on the *Cost-Optz* performance. We average the running times of each *Cost-Optz* variant across the three input cases. In case of uniform datasets, we find that typical *Cost-Optz* is at least 15% and 180% faster than *Cost-Optz-no-*D_p and *Cost-Optz-no-*U_R, respectively. This performance gap becomes higher in case of skewed datasets, where the typical *Cost-Optz* is at least 1.25 and 2 times faster than the other two variants. We can see that the effect of ignoring the utilization estimate is much larger than the replication one. This is because the block utilization directly affects the number of generated partitions and their skewness, unlike, data replication which captures the additional overhead of small ratio of objects.

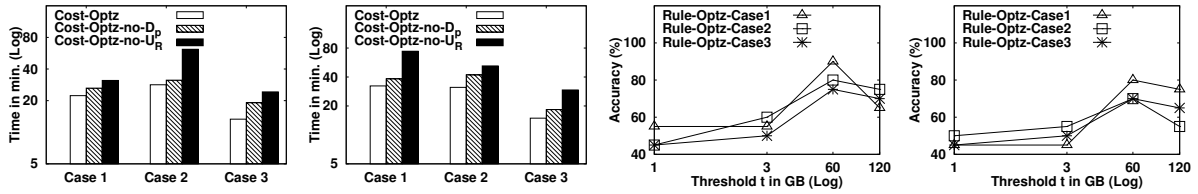
Figures 7(c) and 7(d) show the accuracy of *Rule-Optz* while varying the value of system threshold t, described in Section 4. The accuracy of *Rule-Optz* is defined as the ratio of decisions that match the optimal decisions taken by *Cost-Optz*. We can see that tuning the value of t is crucial for obtaining heuristic decisions with high accuracy. In this experiment, given t of 60GB, the maximum accuracy achieved in uniform and skewed cases are 90% and 80%, respectively. We find that a good estimation for the value of t is the maximum dataset size that can fit in a single machine memory.

6 RELATED WORK

Traditional spatial join algorithms. There have been numerous efforts to spatially join two input relations, e.g., see [30] for a comprehensive survey. In general, these algorithms are categorized into three categories based on whether the two input relations are indexed or not, as follows: (a) None of the input relations is indexed [29, 35, 42, 43, 54], (b) Only one of the input relations is indexed [6, 12, 25, 33, 36], and (c) Both input relations are indexed [8–10, 27, 31]. With such numerous algorithms, various benchmarking studies were performed to show the strengths and weaknesses of each algorithm with respect to various query workloads [23, 28, 41, 45, 46]. This helps in fueling query optimization ideas that find the best way of doing a spatial join between two relations in various settings.

Big Spatial Data. There is a recent explosion in the amount of spatial data produced by various devices such as smart phones, satellites and medical devices. This motivates several research efforts to support spatial operations for big spatial data, e.g. range queries [51], KNN [15], spatial join [42, 52] and computational geometry [15]. In addition, many full-fledged systems have been proposed to provide an end-to-end big spatial data solution, e.g. SECONDO [34], HadoopGIS [2], ESRI Hadoop [1, 50], and SpatialHadoop [16].

MapReduce-based spatial join algorithms. Research efforts in MapReduce-based spatial join algorithms come in two flavors, either as stand alone algorithms (e.g., [24, 44, 48, 52, 53]), or as part of big spatial data engine (e.g., HadoopGIS [2], SpatialHadoop [16]). In either case, the proposed algorithms are usually tailored towards one specific scenario. For example, spatial join in SpatialHadoop assumes that both input data sets are spatially partitioned, while most of the algorithms assume that none of the two input data sets is spatially partitioned [2, 24, 44, 48, 52, 53]. The common theme of all these approaches is that they start by ensuring that the two



(a) Replic. D_p and Util. U_R (Uniform) (b) Replic. D_p and Util. U_R (Skewed) (c) Threshold t (Uniform) (d) Threshold t (Skewed)

Figure 7: Effect of query optimizer parameters on the overall performance.

input data sets are spatially partitioned. Then, they join overlapping partitions from the spatially partitioned input data sets. Contents of overlapping partitions are joined using any in-memory spatial join algorithm, e.g., [39]. Unfortunately, and up to our knowledge, there are no prior efforts in benchmarking or in developing query optimizations efforts in MapReduce-based spatial join algorithms.

7 CONCLUSION

This paper has delivered a full-fledged query optimizer for MapReduce-based spatial join algorithms. The optimizer is equipped by a very thorough taxonomy of doing a spatial join in the MapReduce environment. The taxonomy is developed for all cases of input datasets. Then, a detailed cost model is developed for each possible case in this taxonomy. The cost models are fed into a cost-based query optimizer that finds the lowest cost option of doing a spatial join between two inputs. The paper also provided a rule-based query optimizer that abstracts the developed cost model into a set of simple heuristic rules. Exhaustive experiments based on a real deployment inside SpatialHadoop, and based on real datasets of up to 500GB show that both optimizers are always successful in selecting the right spatial join decision. There are very few cases where the cost-based query optimizer gets the right choice while the rule-based one did not get it. This is mainly due to the accuracy of the cost model, but that is modulo the overhead in calculating the cost models.

REFERENCES

[1] ESRI Tools on Hadoop. http://esri.github.io/gis-tools-for-hadoop/.
[2] Ablimit Aji, Fusheng Wang, Hoang Vo, Rubao Lee, Qiaoling Liu, Xiaodong Zhang, and Joel Saltz. Hadoop GIS: A High Performance Spatial Data Warehousing System over Mapreduce. *PVLDB*, 6(11), 2013.
[3] Sattam Alsubaiee, Yasser Altowim, Hotham Altwaijry, Alexander Behm, Vinayak Borkar, Yingyi Bu, Michael Carey, Inci Cetindil, Madhusudan Cheelangi, Khurram Faraaz, Eugenia Gabrielova, Raman Grover, Zachary Heilbron, Young-Seok Kim, Chen Li, Guangqiang Li, Ji Mahn Ok, Nicola Onose, Pouria Pirzadeh, Vassilis Tsotras, Rares Vernica, Jian Wen, and Till Westmann. AsterixDB: A Scalable, Open Source BDMS. *PVLDB*, 2014.
[4] Ning An, Zhen-Yu Yang, and Anand Sivasubramaniam. Selectivity Estimation for Spatial Joins. In *ICDE*, 2001.
[5] Walid G. Aref and Hanan Samet. A Cost Model for Query Optimization Using R-Trees. In *SIGSPATIAL*, 1994.
[6] Lars Arge, Octavian Procopiuc, Sridhar Ramaswamy, Torsten Suel, Jan Vahrenhold, and Jeffrey Vitter. A Unified Approach for Indexed and Non-indexed Spatial Joins. In *EDBT*, 2000.
[7] Jon Louis Bentley. Multidimensional Binary Search Trees Used for Associative Searching. *CACM*, 1975.
[8] Thomas Brinkhoff, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. Multi-step Processing of Spatial Joins. *SIGMOD Record*, 23(2), 1994.
[9] Thomas Brinkhoff, Hans-Peter Kriegel, and Bernhard Seeger. Efficient Processing of Spatial Joins Using R-trees. In *SIGMOD*, 1993.
[10] Thomas Brinkhoff, Hans-Peter Kriegel, and Bernhard Seeger. Parallel Processing of Spatial Joins using R-trees. In *ICDE*, 1996.
[11] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *CACM*, 51(1), 2008.
[12] Jochen Van den Bercken, Bernhard Seeger, and Peter Widmayer. The Bulk Index Join: A Generic Approach to Processing Non-Equijoins. In *ICDE*, 1999.
[13] Jens-Peter Dittrich and Bernhard Seeger. Data Redundancy and Duplicate Detection in Spatial Join Processing. In *ICDE*, 2000.
[14] Ahmed Eldawy, Louai Alarabi, and Mohamed F. Mokbel. Spatial Partitioning Techniques in SpatialHadoop. *PVLDB*, 8(12), 2015.
[15] Ahmed Eldawy, Yuan Li, Mohamed F. Mokbel, and Ravi Janardan. CGHadoop: Computational Geometry in MapReduce. In *SIGSPATIAL*, 2013.
[16] Ahmed Eldawy and Mohamed F. Mokbel. SpatialHadoop: A MapReduce Framework for Spatial Data. In *ICDE*, 2015.
[17] Ahmed Eldawy and Mohamed F. Mokbel. The Era of Big Spatial Data . In *ICDE*, 2016.
[18] Christos Faloutsos, Bernhard Seeger, Agma Traina, and Caetano Traina Jr. Spatial Join Selectivity Using Power Laws. In *SIGMOD*, 2000.
[19] R.A. Finkel and J.L. Bentley. Quad Trees a Data Structure for Retrieval on Composite Keys. *Acta Informatica*, 1974.
[20] Miguel R. Fornari, Joao Luiz D. Comba, and Cirano Iochpe. Query Optimizer for Spatial Join Operations. In *GIS*, 2006.
[21] Huijun Gao, Hao Zhang, Daosheng Hu, Ran Tian, and Dazhi Guo. Multi-scale Features of Urban Planning Spatial Data. In *Geoinformatics*, 2010.
[22] Oliver GuĬnther. Efficient Computation of Spatial Joins. In *ICDE*, 1993.
[23] Oliver Gunther, Vincent Oria, Philippe Picouet, Jean-Marc Saglio, and Michel Scholl. Benchmarking Spatial Joins A La Carte. In *SSDM*, 1998.
[24] Himanshu Gupta, Bhupesh Chawda, Sumit Negi, Tanveer A. Faruquie, L. V. Subramaniam, and Mukesh Mohania. Processing Multi-way Spatial Joins on Map-reduce. In *EDBT*, 2013.
[25] Christophe Gurret and Philippe Rigaux. The Sort/Sweep Algorithm: A New Method for R-tree based Spatial Joins. In *SSDM*, 2000.
[26] A. Guttman. R-trees: A Dynamic Index Structure for Spatial Searching. *SIGMOD Rec.*, 1984.
[27] Lilian Harada, Miyuki Nakano, Masaru Kitsuregawa, and Mikio Takagi. Query Processing for Multi-Attribute Clustered Records. In *VLDB*, 1990.
[28] Erik G. Hoel and Hanan Samet. Benchmarking Spatial Join Operations with Spatial Output. In *VLDB*, 1995.
[29] Edwin H. Jacox and Hanan Samet. Iterative Spatial Join. *TODS*, 28(3), 2003.
[30] Edwin H. Jacox and Hanan Samet. Spatial Join Techniques. *TODS*, 32(1), 2007.
[31] Jin-Deog Kim and Bong-Hee Hong. Parallel Spatial Join Algorithms using Grid Files. In *DANTE*, 1999.
[32] Scott T. Leutenegger, Mario A. Lopez, and Jeffrey Edgington. STR: A Simple and Efficient Algorithm for R-tree Packing. In *ICDE*, 1997.
[33] Ming-Ling Lo and Chinya Ravishankar. Spatial Joins Using Seeded Trees. In *SIGMOD*, 1994.
[34] Jiamin Lu and Ralf Hartmut Guting. Parallel Secondo: Boosting Database Engines with Hadoop. In *ICPADS*, 2012.
[35] Gang Luo, Jeffrey F. Naughton, and Curt J. Ellmann. A Non-blocking Parallel Spatial Join Algorithm. In *ICDE*, 2002.
[36] Nikos Mamoulis, Panos Kalnis, Spiridon Bakiras, and Xiaochen Li. Optimization of Spatial Joins on Mobile Devices. In *SSTD*, 2003.
[37] Henry Markram, Karlheinz Meier, Thomas Lippert, Sten Grillner, Richard Frackowiak, Stanislas Dehaene, Alois Knoll, Haim Sompolinsky, Kris Verstreken, Javier DeFelipe, Seth Grant, Jean-Pierre Changeux, and Alois Saria. Introducing the human brain project. *Procedia Computer Science*, 2011.
[38] J. Nievergelt, Hans Hinterberger, and Kenneth C. Sevcik. The Grid File: An Adaptable, Symmetric Multikey File Structure. *TODS*, 9(1), 1984.
[39] J. Nievergelt and F. P. Preparata. Plane-sweep Algorithms for Intersecting Geometric Figures. *CACM*, 1982.
[40] OpenStreetMap. https://www.openstreetmap.org/.
[41] Apostolos Papadopoulos, Philippe Rigaux, and Michel Scholl. A Performance Evaluation of Spatial Join Processing Strategies. *Adv. in Spatial Databases*, 1999.
[42] Jignesh M. Patel and David J. DeWitt. Partition Based Spatial-merge Join. In *SIGMOD*, 1996.
[43] Jignesh M. Patel and David J. DeWitt. Clone Join and Shadow Join: Two Parallel Spatial Join Algorithms. In *GIS*, 2000.
[44] Satish Puri, Dinesh Agarwal, Xi He, and Sushil K. Prasad. MapReduce Algorithms for GIS Polygonal Overlay Processing. In *IPDPSW*, 2013.
[45] Darius Sidlauskas and Christian S. Jensen. Spatial Joins in Main Memory: Implementation Matters! *PVLDB*, 8(1), 2014.
[46] Benjamin Sowell, Marcos Vaz Salles, Tuan Cao, Alan Demers, and Johannes Gehrke. An Experimental Analysis of Iterated Spatial Joins in Main Memory. *PVLDB*, 6(14), 2013.
[47] Chengyu Sun, Divyakant Agrawal, and Amr El Abbadi. Selectivity Estimation for Spatial Joins with Geometric Selections. In *EDBT*, 2002.
[48] Kai Wang, Jizhong Han, Bibo Tu, Jiao Dai, Wei Zhou, and Xuan Song. Accelerating Spatial Data Processing with MapReduce. In *ICPADS*, 2010.
[49] Kaibo Wang, Yin Huai, Rubao Lee, Fusheng Wang, Xiaodong Zhang, and Joel Saltz. Accelerating Pathology Image Data Cross-comparison on CPU-GPU Hybrid Systems. *PVLDB*, 2012.
[50] Randall T. Whitman, Michael B. Park, Sarah M. Ambrose, and Erik G. Hoel. Spatial Indexing and Analytics on Hadoop. In *SIGSPATIAL*, 2014.
[51] Shubin Zhang, Jizhong Han, Zhiyong Liu, Kai Wang, and Shengzhong Feng. Spatial Queries Evaluation with MapReduce. In *GCC*, 2009.
[52] Shubin Zhang, Jizhong Han, Zhiyong Liu, Kai Wang, and Zhiyong Xu. SJMR: Parallelizing spatial join with MapReduce on clusters. In *CLUSTER*, 2009.
[53] Yunqin Zhong, Jizhong Han, Tieying Zhang, Zhenhua Li, Jinyun Fang, and Guihai Chen. Towards Parallel Spatial Query Processing for Big Spatial Data. In *IPDPSW*, 2012.
[54] Xiaofang Zhou, David J. Abel, and David Truffet. Data Partitioning for Parallel Spatial Join Processing. *Geoinformatica*, 1998.