Spatial Partitioning Techniques in SpatialHadoop *

Ahmed Eldawy Louai Alarabi Mohamed F. Mokbel

Department of Computer Science and Engineering
University of Minnesota
{eldawy,louai,mokbel}@cs.umn.edu

ABSTRACT

SpatialHadoop is an extended MapReduce framework that supports *global indexing* that spatial partitions the data across machines providing orders of magnitude speedup, compared to traditional Hadoop. In this paper, we describe seven alternative partitioning techniques and experimentally study their effect on the quality of the generated index and the performance of range and spatial join queries. We found that using a 1% sample is enough to produce high quality partitions. Also, we found that the total area of partitions is a reasonable measure of the quality of indexes when running spatial join. This study will assist researchers in choosing a good spatial partitioning technique in distributed environments.

1. INDEXING IN SPATIALHADOOP

SpatialHadoop [2, 3] provides a generic indexing algorithm which was used to implement grid, R-tree, and R+-tree based partitioning. This paper extends our previous study by introducing four new partitioning techniques, Z-curve, Hilbert curve, Quad tree, and K-d tree, and experimentally evaluate all of the seven techniques. The partitioning phase of the indexing algorithm runs in three steps, where the first step is fixed and the last two steps are customized for each partitioning technique. The first step computes number of desired partitions n based on file size and HDFS block capacity which are both fixed for all partitioning techniques. The second step reads a random sample, with a sampling ratio ρ, from the input file and uses this sample to partition the space into n cells such that number of sample points in each cell is at most $\lfloor k/n \rfloor$, where k is the sample size. The third step actually partitions the file by assigning each record to one or more cells. Boundary objects are handled using either the *distribution* or *replication* methods. The *distribution* method assigns an object to exactly one overlapping cell and the cell has to be expanded to enclose all contained records. The *replication* method avoids expanding cells by replicating each record to all overlapping cells but the query processor has to employ a duplicate avoidance technique to account for replicated records.

*This work is supported in part by the National Science Foundation, USA, under Grants IIS-0952977 and IIS-1218168 and the University of Minnesota Doctoral Disseration Fellowship.

Proceedings of the VLDB Endowment, Vol. 8, No. 12

2. EXPERIMENTAL SETUP

All experiments run on Amazon EC2 `'m1.large'` instances which have a dual core processor, 7.5 GB RAM and 840 GB disk storage. We use Hadoop 1.2.1 running on Java 1.6 and CentOS 6. Each machine is configured to run three mappers and two reducers. Tables 1 and 2 summarize the datasets and configuration parameters used in our experiments, respectively. Default parameters (in parentheses) are used unless otherwise mentioned. In the following part, we describe the partitioning techniques, the queries we run, and the performance metrics measured in this paper.

2.1 Partitioning Techniques

This paper employs *grid* and *Quad tree* as space partitioning techniques; *STR*, *STR+*, and *K-d tree* as data partitioning techniques; and *Z-curve* and *Hilbert curve* as space filling curve (SFC) partitioning techniques. These techniques can also be grouped, according to boundary object handling, into *replication*-based techniques (i.e., Grid, Quad, STR+, and K-d tree) and *distribution*-based techniques (i.e., STR, Z-Cruve, and Hilbert). Figure 1 illustrates these techniques, where sample points and partition boundaries are shown as dots and rectangles, respectively.

1. Uniform Grid: This technique does not require a random sample as it divides the input MBR using a uniform grid of $\lceil \sqrt{n} \rceil \times \lceil \sqrt{n} \rceil$ grid cells and employs the replication method to handle boundary objects.

2. Quad tree: This technique inserts all sample points into a quad tree [6] with node capacity of $\lfloor k/n \rfloor$, where k is the sample size. The boundaries of all leaf nodes are used as cell boundaries. We use the replication method to assign records to cells.

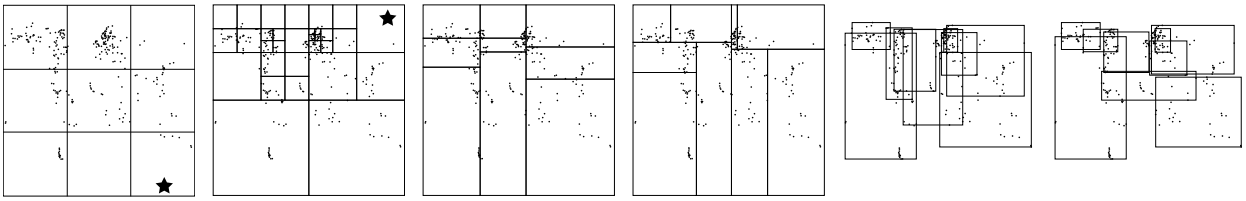
3. STR: This technique bulk loads the random sample into an R-tree using the STR algorithm [8] and the capacity of each node is set to $\lfloor k/n \rfloor$. The MBRs of leaf nodes are used as cell boundaries. Boundary objects are handled using the distribution method where it assigns a record r to the cell with maximal overlap.

4. STR+: This technique is similar to the STR technique but it uses the replication method to handle boundary objects.

5. K-d tree: This technique uses the K-d tree [1] partitioning method to partition the space into n cells. It starts with the input MBR as one cell and partitions it $n - 1$ times to produce n cells. Records are assigned to cells using the replication method.

6. Z-curve: This technique sorts the sample points by their order on the Z-curve and partitions the curve into n splits, each containing roughly $\lfloor k/n \rfloor$ points. It uses the distribution method to assign a record r to one cell by mapping the center point of its MBR to one of the n splits.

7. Hilbert curve: This technique is exactly the same as the Z-curve technique but it uses Hilbert space filling curve which has better spatial properties.



(a) Grid (b) Quad Tree (c) STR and STR+ (d) K-d Tree (e) Z-curve (f) Hilbert Curve

Figure 1: Partitioning Techniques. Examples of nearly-empty partitions are starred.

Name	Size	Records	Average Record Size
`All Objects`	88GB	250M	378 bytes
`Buildings`	25GB	109M	234 bytes
`Roads`	23GB	70M	337 bytes
`Lakes`	8.6GB	9M	1KB
`Cities`	1.4GB	170K	8.4KB

Table 1: Datasets

Parameter	Values (default)
HDFS block capacity (B)	32, (64), 128, 256 MB
Cluster size (N)	5, 10, 15, (20), 25, 35
Sample ratio (ρ)	(0.01), 0.02, 0.05, 0.1, 0.2, 0.5, 1.0
Selection ratio (σ)	0.0001%, 0.01%, (1%)

Table 2: Parameters

2.2 Queries

To test the performance of the partitions, we perform range and spatial join queries as shown in [3]. For a range query, the rectangular query range A is centered around a random record drawn from the input file. The size of A is adjusted such that the $Area(A) = \sigma.Area(InMBR)$, where the selection ratio $\sigma \in [0, 1]$ is a parameter we change in our experiments and $Area(InMBR)$ is the area of the MBR of the input. In the spatial join query, we use `overlap` as the join predicate as it is widely used.

2.3 Performance Metrics

To measure and compare the performance of the different partitioning techniques, we use two categories of performance metrics, namely, *quality measures* and *performance measures*.

The **quality measures** are five metrics computed on the partitioned data to assess its quality. Four of them, Q1-Q4, are derived from the R*-tree optimization criteria, where they were shown to correlate with the performance of the range query [4]. Q1 is the total area occupied by all partitions and is used as an indicator of the *dead space* covered by partitions without containing any actual records. Q2 is the total overlap between pairs of partitions. Q3 is the total *margin* of all partitions where the margin of a rectangle is the sum of width and height. For a fixed area, minimizing the margin favors squares over rectangles. Q4 is the disk utilization measured as the ratio between actual data in partitions and the total capacity of all occupied file system blocks. Finally, Q5 is the standard deviation of partition sizes and is used to measure the load balance or skewness across partitions.

The **performance measures** assess the running time of the partitioning process and queries running on the partitioned data. The *partitioning time* is the total time spent by the cluster to partition the data. For range queries, we measure both the time spent to answer a single query and the throughput of the cluster to answer a batch of queries in terms of jobs per minute. For spatial join, we measure the total running time for a single query.

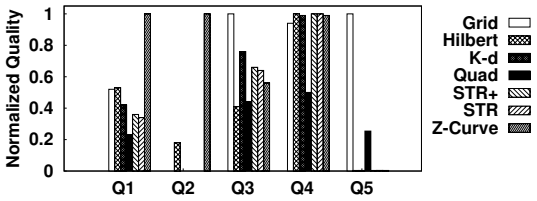


Figure 2: Five Quality Measures for `Buildings`

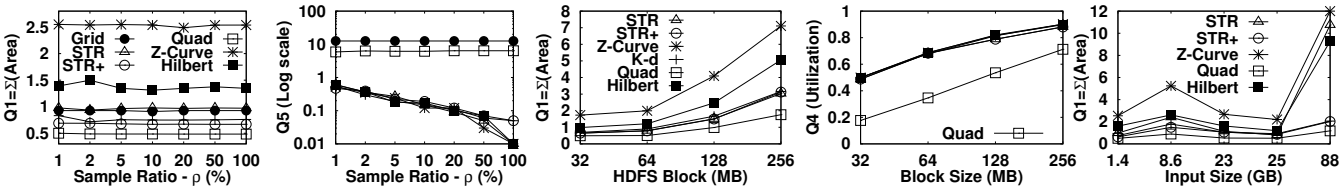
3. EXPERIMENTAL RESULTS

This section shows the results of the experiments carried out on the cluster. In each experiment, we use all default parameters mentioned in Table 2 unless otherwise mentioned. For clarity, all figures use the same set of symbols, and we omit the *key* in some figures to keep them readable.

3.1 Quality Measures

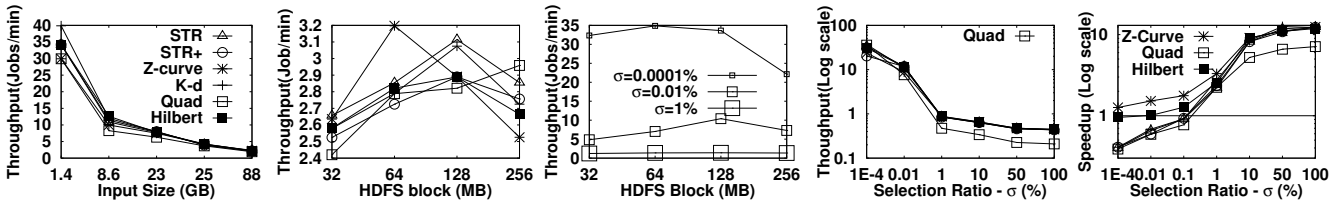
Figure 2 shows the five quality measures for the `Buildings` dataset. All values are normalized to the range $[0, 1]$ to keep the figure concise. According to Q1, Quad tree partitioning gives the best performance as it keeps the overall area limited by regular space partitioning and pruning empty partitions. On the other hand, Z-curve partitioning is the worst as it generates a lot of overlap between partitions due to the huge jumps in the curve. Q2 does not seem to be very helpful as all *replication*-based techniques generate disjoint partitions, which always produce Q2=0. Unlike Q2, the variance in Q3 is high and, surprisingly, Hilbert curve provided the best value with the Quad tree being very close. This shows that there is a room for improvement if we apply more sophisticated partitioning techniques. It is interesting that both *space partitioning* techniques perform relatively poor in Q4 and Q5. The reason is that they employ regular space partitioning, which is susceptible to producing nearly-empty partitions (examples starred in Figure 1 a&b) which is adversary to both disk utilization (Q4) and skewness (Q5). In fact, the value of Q5 is too small to notice in other techniques as they have the freedom to accurately adjust partition boundaries to maximize utilization and minimize skewness. In a future work, we can try to combine several small partitions into one bigger partition to improve Q5, and study its effect on other quality measures.

In Figure 3(a), we measure Q1 while varying the sampling ratio ρ from 1% to 100% for the `Cities` dataset. Surprisingly, the quality measure is hardly affected by the sampling ratio. We observe a similar behavior across different datasets, partitioning techniques, and quality measures (except Q5). This finding might seem counter-intuitive because drawing a larger sample should reduce the error caused by sampling and increase the quality of the partitioning. However, there are two reasons that make this finding accountable. First, each partitioning technique has its own inherent limitations which impose some upper bound on its quality. For example, Z-curve partitioning would generally produce overlapping partitions due to the loss of locality in the Z-curve even if it operates on the whole file. From this experiment, it looks like that this



(a) Q1 for `Cities` (b) Q5 for `Cities` (c) Block Size - Q1 (d) Block Size - Q4 (e) Q1 - All Datasets

Figure 3: Quality Measures



(a) Input Size (I) (b) Block Size (B) (c) Block Size (B) (d) Selection Ratio (σ) (e) Speedup

Figure 4: Range Query

upper bound is easily reached for the evaluated partitioning techniques even with a 1% sample. The second reason is that records are converted to points as they are sampled to make the in-memory bulk-loading step simpler and more efficient. This means that even when ρ =100%, there is still some approximation of shapes into points. We believe that this is a very important finding because it tells that the limitations of these partitioning techniques are not caused by the random sampling.

The only exception to the previous finding is with Q5 (skewness) which actually improves for most partitioning techniques as the sample ratio increases, as shown in Figure 3(b). There are two interesting findings in this figure. First, *space partitioning* techniques are hardly affected by the sample ratio. Grid partitioning does not use the sample at all while the Quad tree uses the sample only to decide which tree nodes to split, which is not much affected by the sample size. Second, as the sample size approaches 100%, the *distribution*-based techniques achieve a near-perfect load balancing, as opposed to *replication*-based techniques. The reason is that the effect of replication is not taken into account while subdividing the space using the sample of points.

Figures 3(c) and 3(d) show the tradeoff between partitioning quality and disk utilization as we increase the partition size for `Buildings`. This tradeoff is well known in the literature and this experiment confirms that it still holds in MapReduce environments.

Finally, Figure 3(e) shows the value of the quality measure Q1 for all datasets. Although we cannot really compare the quality of different datasets as they have different characteristics, we can still observe that the relative quality between techniques is consistent across datasets. In addition, we see a huge drop for the *distribution*-based techniques with `All Objects` (largest dataset) as they have to greatly expand partitions to enclose very large objects (e.g., countries borders). This does not happen with *replication*-based techniques which do not expand partitions boundaries. This is an important finding which helps users in choosing a partitioning technique for a dataset depending on the shape of the objects.

3.2 Range Query Performance

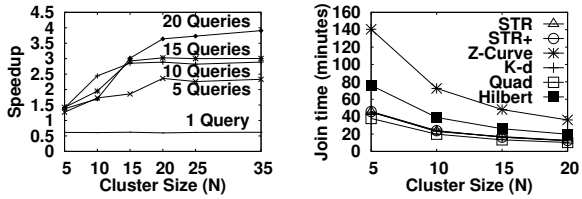
Figure 4 shows the performance of range queries over the constructed indexes. In each experiment, we submit a batch of queries, and measure the throughput of the cluster in terms of queries/minute. In general, we found that all partitioning techniques that have been experimented behave roughly the same due to the simplicity of the query in the MapReduce environment. It has been shown in [4] that the performance of the range query reflects the four quality measures Q1-Q4. Therefore, we will focus in measuring the effect of query selectivity and number of machines in the MapReduce environment. In Figure 4(a), the input size is increased and the throughput is measured for each partitioning technique. As expected, the performance degrades as the input size increases as more partitions need to be processed with larger files.

In Figure 4(b), the block size of the partitioned data is increased from 32MB to 256MB and the performance of range query is measured for `All Objects`. Despite the slight variance in times across the different techniques, we can notice a common *bitonic* trend of the performance where it rises up and then goes down again. This interesting behavior happens due to the trade-off between number of matched partitions and amount of processing per partition. A smaller block size reduces the amount of work per partition but increases number of matched partitions per query, while a larger block size has an opposite effect. As shown in figure, the sweet spot that balances this trade-off varies from one partitioning technique to another but most techniques are balanced at the 128MB block which is the default value in recent Hadoop releases.

To further study the effect of block size, Figure 4(c) shows the range query performance on `Buildings` when it is indexed using a Quad tree of different block sizes. We run three sets of range queries with $\sigma \in \{0.0001\%, 0.01\%, 1\%\}$. We see the same bitonic trend in all cases but the peak is different where it is 64MB when $\sigma = 0.0001\%$ and 128MB for the other two values. The peak at 128MB when $\sigma = 1\%$ does exist but is hard to notice due to the scale of the figure. This experiments indicates that the workload should be accounted while tuning the index.

In Figure 4(d), we increase the selection ratio (σ) and evaluate the range query performance for all partitioning techniques. As the selection ratio increases, the performance degrades as more partitions are processed. Eventually, the performance of all of them converge as they end up scanning large portions (or all) of the input file. Unlike centralized systems, Quad tree takes more time scanning the file as the MapReduce job needs to create one task per partition while our Quad tree technique produces much more partitions than other techniques.

In Figure 4(e) we show the speedup of running range query using MapReduce compared to a centralized techniques. In this figure, values below one, illustrated by a horizontal line, indicate that the centralized query is faster. This experiment shows that a centralized system outperforms MapReduce when the query area is small as it



(a) Range Query (b) Spatial Join

Figure 5: Scale-out with cluster size

Roads (23GB)	Q1	Lakes (8.6 GB)						Buildings (25 GB)					
		0.75	0.89	0.89	1.47	1.47	2.71	0.50	0.71	0.78	0.71	1.07	1.81
Q1		Quad	STR+	K-d	STR	Hilbert	Zcurve	Quad	STR+	K-d	STR	Hilbert	Zcurve
0.52	Quad	276	440	463	675	749	1446	628	919	888	895	1054	1633
0.66	STR+	419	381	439	577	613	1048	801	740	834	763	960	1346
0.71	K-d	465	404	416	560	664	1058	787	836	735	782	935	1321
0.71	STR	409	401	437	570	599	1031	834	765	795	733	928	1321
1.03	Hilbert	512	464	491	612	630	1100	1248	1085	1026	1030	1184	1646
1.88	Zcurve	740	589	608	837	780	1246	1979	1587	1519	1519	1644	2223

Figure 6: Spatial join performance (best viewed in color)

avoids the overhead of MapReduce when only a little work needs to be done. The results of this experiment can be incorporated into a query optimizer to choose the best approach based on the size of the query area.

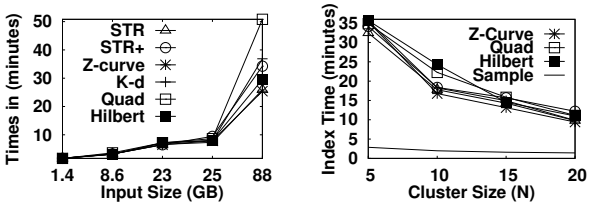
Figure 5(a) shows how the range query scales out when the cluster size changes from 5 to 35 nodes. On each cluster, we submit five batches of sizes 1, 5, 10, 15, and 20 range queries, and measure the speedup of MapReduce over centralized processing. Unlike centralized systems, which better utilize all its processing capabilities, MapReduce has an upper bound on the speedup regardless of number of machines in the cluster. The reason is that MapReduce breaks a job into coarse-grained tasks, as one per partition, which limits the level of parallelism it can achieve, while centralized systems break jobs into finer-grained partitions to achieve a higher level of parallelism as its resources allow. As the batch size increases, MapReduce can achieve a higher speedup but it again stabilizes as soon as the cluster becomes underutilized.

3.3 Spatial Join Performance

Figure 5(b) shows the performance of joining `Roads` and `Buildings` when both are partitioned using the same technique. In general, all the techniques scale well with the cluster size. Unlike range query, one spatial join query is usually enough to utilize all cluster resources and achieve a high degree of parallelism. The two SFC-based techniques perform relatively worse than other techniques as they produce a lot of overlapping partitions (e.g., see Figure 1 e&f).

Figure 6 further studies the performance of spatial join when the two input files are partitioned using different techniques. In this experiment, we run two spatial join queries, `Roads`×`Lakes` and `Roads`×`Buildings`, for every possible combination of partitioning techniques on inputs. The figure shows the values of Q1 for input files, and the total running times which are color-coded from red (slowest) to green (fastest). The figure shows a direct correspondence between the values of Q1 and the overall performance. Statistically, we found a strong linear correlation with a coefficient of 89% and 92%, for the two experimented queries.

In addition to its good quality measures, the Quad tree outperforms all other techniques as it minimizes the number of overlapping partitions between the two files by employing a regular space partitioning which is perfectly aligned across different files. This finding conforms with earlier work in the literature which showed that Quad tree partitioning outperforms both R-tree and



(a) Partitioning time (b) Scale out

Figure 7: Partitioning Time

R+-tree partitioning when running a spatial join query in a traditional DBMS environment [7]. Furthermore, the state-of-the-art work in spatial join in main memory is based on Quad-tree-like partitioning [5]. This paper is the first to extend those earlier findings to MapReduce as well.

3.4 Partitioning Time

Figure 7(a) shows the overall partitioning time for different input datasets. An interesting finding is that the performance of all techniques is very similar as the partitioning is dominated by the MapReduce job that scans the whole file. The main difference between all techniques is in the in-memory step which operates on a small sample. This opens the space to incorporate more complicated techniques. Figure 7(b) shows the partitioning time, for `Buildings`, as the number of machines in the cluster increases from 5 to 20. Although the sampling step takes less than two minutes, it does not scale as good as the partitioning step due to the fixed overhead associated with each map task in the MapReduce job. This calls for more efficient sampling techniques in terms of performance. We also found that the partitioning time is not greatly affected by the block size, hence, we omitted that experiment for limited space.

4. CONCLUSION

In this paper, we experimentally evaluated seven spatial partitioning techniques, all employed inside SpatialHadoop. We showed that SpatialHadoop is scalable when indexing a file using any of these techniques. It was shown that even with a 1% sample of the file, we can partition a file with a very high quality. While range query performed similarly on all of them, we showed that they can be tuned with system parameters such as block size according to the query work load. We also showed the the performance of spatial join is strongly correlated with the value of Q1 (total area of partitions) and found that Quad tree outperformed other techniques being experimented.

5. REFERENCES

[1] J. L. Bentley. Multidimensional Binary Search Trees Used for Associative Searching. *Commun. ACM*, 18(9):509–517, 1975.

[2] A. Eldawy and M. F. Mokbel. A Demonstration of SpatialHadoop: An Efficient MapReduce Framework for Spatial Data. In *VLDB*, 2013.

[3] A. Eldawy and M. F. Mokbel. SpatialHadoop: A MapReduce Framework for Spatial Data. In *ICDE*, 2015.

[4] N. B. *et al*. The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles. In *SIGMOD*, pages 322–331, 1990.

[5] S. N. *et al*. Touch: In-memory Spatial Join by Hierarchical Data-oriented Partitioning. In *SIGMOD*, pages 701–712, 2013.

[6] R. A. Finkel and J. L. Bentley. Quad Trees: A Data Structure for Retrieval on Composite Keys. *Acta Inf.*, 4:1–9, 1974.

[7] E. G. Hoel and H. Samet. Performance of Data-Parallel Spatial Operations. In *VLDB*, pages 156–167, 1994.

[8] S. Leutenegger, M. Lopez, and J. Edgington. STR: A Simple and Efficient Algorithm for R-Tree Packing. In *ICDE*, pages 497–506, 1997.