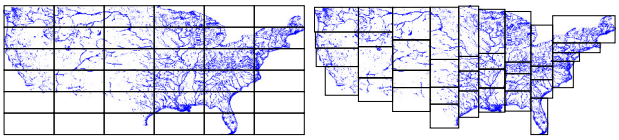

(a) Grid Partitioning (b) R-tree Partitioning

Fig. 4. The Partitioning Phase

task. (2) It ensures that the local index is treated by Hadoop *load balancer* as one unit when it relocates blocks across machines. According to the partitioning done in the first phase, it is expected that each partition fits in one HDFS block. In case a partition is too large to fit in one block, we break it into smaller chunks of 64 MB each, which can be written as single blocks. To ensure that local indexes remain aligned to blocks after concatenation, each file is appended with dummy data (zeros) to make it exactly 64 MB.

3) Phase III: Global Indexing: The purpose of this phase is to build the requested index structure (e.g., Grid or R-tree) as a *global* index that indexes all partitions. Once the MapReduce partition job is done, we initiate an HDFS `concat` command which concatenates all local index files into one file that represents the final indexed file. Then, the master node builds an in-memory global index which indexes all file blocks using their rectangular boundaries as the index key. The global index is constructed using bulk loading and is kept in the main memory of the master node all the time. In case the master node fails and restarts, the global index is lazily reconstructed from the rectangular boundaries of the file blocks, only when required.

C. Grid file

This section describes how the general index building algorithm outlined in Section V-B is used to build a grid index. The grid file [32] is a simple flat index that partitions the data according to a grid such that records overlapping each grid cell are stored in one file block as a single partition. For simplicity, we use a uniform grid assuming that data is uniformly distributed. In the *partitioning* phase, after the number of partitions n is calculated, partition boundaries are computed by creating a uniform grid of size $\lceil\sqrt{n}\rceil \times \lceil\sqrt{n}\rceil$ in the space domain and taking the boundaries of grid cells as partition boundaries as depicted in Fig. 4(a). This might produce more than n partitions, but it ensures that the average partition size remains less than the HDFS block size. When physically partitioning the data, a record r with a spatial extent, is replicated to every grid cell it overlaps. In the *local indexing* phase, the records of each grid cell are just written to a heap file without building any local indexes because the grid index is a one-level flat index where contents of each grid cell are stored in no particular order. Finally, the *global indexing* phase concatenates all these files and builds the global index, which is a two dimensional directory table pointing to the corresponding blocks in the concatenated file.

D. R-tree

This section describes how the general index building algorithm outlined in Section V-B is used to partition spatial data over computing nodes based on R-tree indexing, as in Fig. 4(b), followed by an R-tree local index in each partition. In the partitioning phase to compute partition boundaries, we bulk load a random sample from the input file to an in-memory R-tree using the Sort-Tile-Recursive (STR) algorithm [30]. The size of the random sample is set to a default ratio of 1% of the input file, with a maximum size of 100MB to ensure it fits in memory. Both the ratio and maximum limit can be set in configuration files. If the file contains shapes rather than points, the center point of the shape's MBR is used in the bulk loading process. To read the sample efficiently when input file is very large, we run a MapReduce job that scans all records and outputs each one with a probability of 1%. This job also keeps track of the total size of sampled points, S, in bytes. If S is less than 100MB, the sample is used to construct the R-tree. Otherwise, a second sample operation is executed on the output of the first one with a ratio of $\frac{100^{MB}}{S}$, which produces a sub-sample with an expected size of 100MB.

Once the sample is read, the master node runs the STR algorithm with the parameter d (R-tree degree) set to $\lceil\sqrt{n}\rceil$ to ensure the second level of the tree contains at least n nodes. Once the tree is constructed, we take the boundaries of the nodes in the second level and use them in the physical partitioning step. We choose the STR algorithm as it creates a balanced tree with roughly the same number of points in each leaf node. Fig. 4(b) shows an example of R-tree partitioning with 36 blocks ($d = 6$). Similar to a traditional R-tree, the physical partitioning step does not replicate records, but it assigns a record r to the partition that needs the least enlargement to cover r and resolves ties by selecting the partition with smallest area.

In the *local indexing* phase, records of each partition are bulk loaded into an R-tree using the STR algorithm [30], which is then dumped to a file. The block in a local index file is annotated with its minimum bounding rectangle (MBR) of its contents, which is calculated while building the local index. As records are overlapping, the partitions might end up being overlapped, similar to traditional R-tree nodes. The *global indexing* phase concatenates all local index files and creates the global index by bulk loading all blocks into an R-tree using their MBRs as the index key.

E. R+-tree

This section describes how the general index building algorithm outlined in Section V-B is used to partition spatial data over computing nodes based on R+-tree with an R+-tree local index in each partition. R+-tree [34] is a variation of the R-tree where nodes at each level are kept disjoint while records overlapping multiple nodes are replicated to each node to ensure efficient query answering. The algorithm for building an R+-tree in SpatialHadoop is very similar to that of the R-tree except for three changes. (1) In the R+-tree physical partitioning step, each record is replicated to each partition it overlaps with. (2) In the *local indexing* phase, the records of each partition are inserted into an R+-tree which is then dumped to a local index file. (3) Unlike the case of