Spatial Indexing and Analytics on Hadoop

Randall T. Whitman, Michael B. Park, Sarah M. Ambrose, and Erik G. Hoel
Environmental Systems Research Institute (Esri)
380 New York Street
Redlands, California 92373
1-909-793-2853
{rwhitman, mpark, sambrose, ehoel}@esri.com

ABSTRACT

Effective processing of extremely large volumes of spatial data has led to many organizations employing distributed processing frameworks. Hadoop is one such open-source framework that is enjoying widespread adoption. In this paper, we detail an approach to indexing and performing key analytics on spatial data that is persisted in HDFS. Our technique differs from other approaches in that it combines spatial indexing, data load balancing, and data clustering in order to optimize performance across the cluster. In addition, our index supports efficient, random-access queries without requiring a MapReduce job; neither a full table scan, nor any MapReduce overhead is incurred when searching. This facilitates large numbers of concurrent query executions. We will also demonstrate how indexing and clustering positively impacts the performance of range and k-NN queries on large real-world datasets. The performance analysis will enable a number of interesting observations to be made on the behavior of spatial indexes and spatial queries in this distributed processing environment.

Categories and Subject Descriptors

H.2.8 [Database Management]: Database Applications – Spatial databases and GIS, C.1.3.4 [Processor Architectures]: Parallel Architectures – Distributed architectures, H.3.1 [Information Storage and Retrieval]: Content Analysis and Indexing – Indexing methods.

General Terms

Algorithms, Management, Performance.

Keywords

Hadoop, MapReduce, HDFS, spatial indexing, analytics, k-NN, quadtree, distributed processing.

SIGSPATIAL '14, November 04 - 07 2014, Dallas/Fort Worth, TX, USA.
Copyright 2014 ACM 978-1-4503-3131-9/14/11…$15.00
http://dx.doi.org/10.1145/2666310.2666387

1. INTRODUCTION

Improved performance of both visualization and spatial statistical analysis on large volumes of data stored in Hadoop were the motivation behind designing a spatial index on Hadoop. When indexing and organizing data on Hadoop, there are two models that must be supported. In the first model, users are willing to trade the overhead of copying and clustering the source data for additional query performance. Along with building the index, copying the data in the same spatial order as the partitioned spatial index allows greater levels of locality of reference. This results in increased query performance, particularly when the query involves proximity analysis (e.g., k-NN).

The second model arises in situations where either the volume of source data is too large (e.g., petabyte scale), or the user is unwilling to allow the source data to be copied and reorganized. In this case, the source data remains unaffected. However, spatial analytical operations will be slower than in the first case due to additional network traffic and lower levels of locality of reference. The observed performance differences on real world data will be highlighted later in this paper.

In addition, the spatial index supports pan and zoom in ArcGIS desktop (for reasonable zoomed-in extents) by performant range query without MapReduce overhead – if every range query required a MapReduce job, frequent pan and zoom operations would overwhelm the Hadoop cluster.

It is important to note that a key aspect of this problem space relates to having more traditional volumes of data (e.g., tens to hundreds of millions of spatial objects), and very large computational problems (big computation). For example, this would include computing an all pairs shortest path or a k-NN query for all objects in the dataset.

The rest of this paper is organized as follows. Section 2 reviews related work in this domain. Section 3 describes our algorithm for construction of a PMR quadtree using the Hadoop MapReduce [6] framework. In addition to constructing a PMR quadtree, we present details of how to further optimize the data and index by reorganizing the source data to correspond to the ordering found within the index. Section 4 details two fundamental spatial operations, a range query and a k-NN query that have been implemented on Hadoop. We focus on how they differ from their traditional sequential counterparts. Section 5 presents some detailed performance analyses of the PMR quadtree build operation, as well as large analytic operations that perform range and k-NN queries in parallel across the cluster. Concluding remarks are contained in Section 6.

2. RELATED WORK

Over the past two years, there has been significant research interest in the topic of indexing and performing analytics on spatial data within the Hadoop framework. At a high level, many approaches seem similar, though closer analysis reveals important differences both algorithmically as well as at a capability level. In addition, each approach differs from a query processing standpoint. Some of the more notable recent examples are discussed below.

Hadoop++, HAIL, and LIAH provide 20X to 50X faster queries on Hadoop, via various indexing techniques [7]. However, none of these projects provide a spatial indexing capability. Cary et al. [4] discuss building an R-Tree index with Hadoop MapReduce, but do not discuss querying.

Hadoop-GIS/RESQUE [1] provides spatially-aware query planning and optimization for Hive (an infrastructure built on top of Hadoop that supports a SQL-like language for querying the data [23]). Hadoop-GIS is designed for efficient spatial joins, motivated by use-cases related to medical pathology imaging. It uses a global index of rectangular-tile partitions and a local R*-tree [3] index within each tile. The on-demand indexing is not persisted, facilitating multiple queries on the same data. Focusing on the familiarity and usability of a query-language interface, it is not clear that Hadoop-GIS would be suitable for custom MapReduce applications. Finally, Hadoop-GIS requires deployment of a customized Hive open-source implementation (it has been offered as a contribution to Apache Hive).

SpatialHadoop [8] provides an open-source spatial extension framework to Hadoop that can be used in custom MapReduce applications (note that SpatialHahoop utilizes the open source GIS Tools for Hadoop geometry API [10]). SpatialHadoop employs a two-level indexing scheme. The global index determines which partitions need be searched for the query – the disjoint spatial partition regions having been chosen by STR bulk-loading a sample of the data into a uniform grid or a single-level R+-tree. A local R+-tree index is used in each of the blocks of each partition. In addition to a range query, a k-NN query, and a spatial join, several computational geometry operations are implemented [9]. The researchers claim up to a two order of magnitude speedup when performing spatial queries over non-indexed data on Hadoop and have published results with 1.7 billion points from OpenStreetMap [20]. SpatialHadoop runs a MapReduce job for every query; it does not appear to be designed to perform multiple spatial searches (each with separate spatial criteria parameters) within the same MapReduce job. SpatialHadoop avoids the cost of random seeks in HDFS by storing a complete copy of the data inline in the index. With SpatialHadoop, the spatial index is not incrementally maintained as an indexed dataset grows in size.

GeoMesa [11] offers indexed spatio-temporal range queries, and claims one-second response time for most range queries, on GDELT (an open spatio-temporal dataset of a quarter billion geolocated human events dating back to 1979 [19]). GeoMesa builds upon Accumulo [5] indexing; for the key in Accumulo, GeoMesa interlaces 35-bit geohashes [13] with ISO-8601 date strings. Focused on load-balancing, GeoMesa does not attempt to enhance the spatial locality of data – rather, it distributes the data spatio-temporally (using a random prefix) to ensure load balancing. Data ingest is handled natively by Accumulo. GeoMesa does not support the k-NN query. Finally, as it requires Accumulo, GeoMesa is not applicable for general MapReduce applications on data stored directly as files in HDFS.

The key distinction when querying our spatial index on Hadoop is that the index allows efficient, random-access queries. This does not require a MapReduce job to handle the volume of the data. Neither a table scan, nor any MapReduce overhead, is incurred when searching. Thus, a search can be performed for every record processed in the Map and/or Reduce function of a MapReduce application – in one single MapReduce job. In this manner, the spatial index supports spatial statistical analysis.

3. SPATIAL INDEXING

Our spatial indexing technique relies upon the PMR quadtree spatial index [21]. The PMR quadtree exhibits several important characteristics that make it well suited to parallel or distributed processing environments [18]. Most significantly, it is a regular disjoint decomposition of space. The regularity property facilitates the straightforward mapping of the structure to a collection of processing elements (note: in this work, we employ a linearized quadtree [12] where only the leaf nodes are represented). The disjoint decomposition property is useful in minimizing problems associated with processing spatial data contained in multiple overlapping regions of space (e.g., R-trees).

3.1. Building the Quadtree

The basic process of building the PMR quadtree in parallel across a collection of Hadoop processing elements consists of the following five basic steps:

1. Split the unorganized source data into equal sized collections, physically distributing one collection to each processing element in the cluster.
2. On each processing element, build a partial quadtree incorporating the local data in the collection.
3. Assign and distribute each quadrant in each partial quadtree to a processing element for later assembly of the complete quadtree.
4. On each processing element, combine the received and sorted overlapping quadrants into a locally consistent quadtree.
5. Combine the locally consistent quadtrees into a complete global PMR quadtree, persisting the index as files in HDFS.

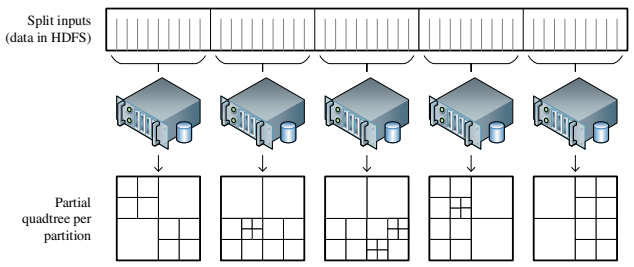


Figure 1: Example highlighting the splitting of the source data into equal sized collections (step 1) and the partial quadtrees being built on each processing element.

The native splitting capability of the Hadoop system is used to split the unordered source data into equal sized collections (step 1). We

are able to accept as input, datasets for which a tabular schema has been defined in the Hive metadata. We support the following formats: delimited text, JSON, and a Hadoop sequence file (a flat file consisting of binary key/value pairs). The number of collections is not constrained to be equal to the number of processing elements in the cluster. Within Hadoop, clients have limited control over the number and size of the collections.

Using the Hadoop MapReduce framework, we chose to build a PMR quadtree during the Map phase on each processing element (step 2). This is done for each collection independently, resulting in a set of partial quadtrees, one for each collection. In contrast to the typical MapReduce pattern where zero or more output records are emitted after receiving and processing a single input record, we instead accumulate multiple input records and build a partial quadtree. If the partial quadtree exceeds memory capacity, it is locally paged to disk. After the Map task inserts all input records in the collection into the partial quadtree, all the entries in the partial quadtree are emitted. The entries emitted by the Map task have a composite key consisting of a Peano key of the quadtree node, and the node depth. Note that input objects whose bounding rectangle intersects more than one quadrant are referenced by multiple entries in the index (similar to the R+-tree). Therefore, the possible presence of duplicate entries in the spatial index must be accounted for in the implementations of the analytic operations.

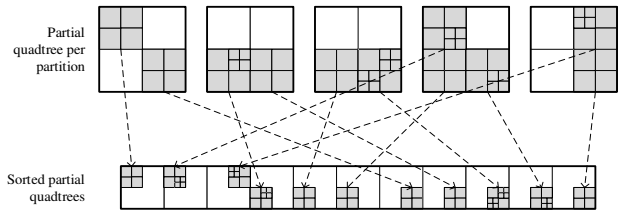


Figure 2: Sorted partial-extent quadtrees resulting from the custom partitioner and the sort, in step 3.

A custom partitioner is defined whose purpose is to assign each entry in a partial quadtree to the processing element that will combine the quadrants of the partial quadtrees into subtrees of a complete quadtree index (step 3). The partitioner assigns the entry to the partition for which the Peano key of the entry lies within the partition boundaries – the partition boundaries having been chosen by a pre-calculation that sampled the input (see Figure 2). A partition corresponds to a subsequence of entries in the linearized PMR quadtree; note that the space corresponding to a partition may contain disjoint parts (see Figure 5).

The MapReduce framework sorts records by key between the Map and Reduce phases. For the key type emitted by the Mapper, used by the Partitioner, and received by the Reducer, we define a composite key consisting of the Peano key and quadrant depth. This results in the MapReduce framework sorting all entries in the quadtree by Peano key and depth.

Entries from all the partial quadtrees are combined into a complete global linear PMR quadtree (step 4), using a quadtree-aware streaming merge sort. The streaming and sorting (at the end of step 3) are done using the built in capabilities of the MapReduce framework. The quadtree-aware merge (i.e., where merging may result in subdividing a quadrant), is accomplished in the Reduce phase of the MapReduce job. In the Reducer, we build a partial quadtree – this time not data-partial with respect to a non-spatial split – but rather the complete data for one quadrant or subtree out of the whole tree (see Figure 3). Because the Reducer receives the entries in sorted order, the subtree for one quadrant is completed and can be emitted before starting the next one. This is conceptually analogous to the flushing step in bulk-loading quadtrees [16].

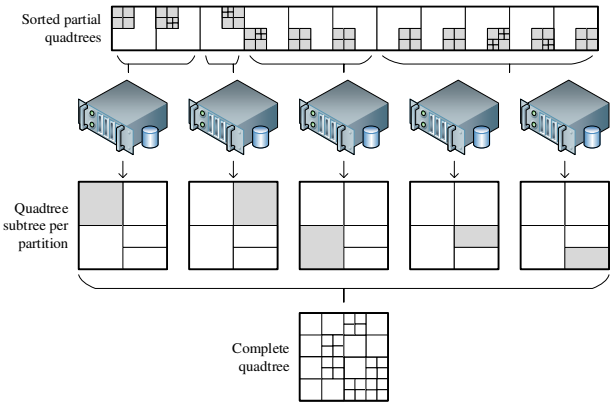


Figure 3: Example showing subsets of the partial quadtrees being distributed and merged on the collection of processing elements. The final combined result (the complete quadtree) is also shown.

Persisting the index in index-portion files in HDFS (step 5), is an outcome of Reducer output getting written to files in the MapReduce framework. Due to the quadtree-compatible partitioning by Peano key (step 3), the logical concatenation of the index-portion files constitutes the complete global quadtree index. Each segment of the index is stored in Hadoop map file format – an indexed key-value store. The key type is the composite key of Peano key and quadrant depth, packed into 58 bits and 5 bits respectively. The value type holds the bounding rectangle of the geometry, as four doubles – except that for point datasets the point is stored as two doubles. The value also contains a row locator that specifies which file, and the position within the file. The row locator facilitates the lookup of the full object/row during query processing. The record format is shown in Figure 4. A Hadoop map file is chosen for the following properties: index-segment files can be built in isolation in separate reduce tasks; files are written append-only; capability to seek a random key for ensuing queries; and splittable as the input to a MapReduce job.

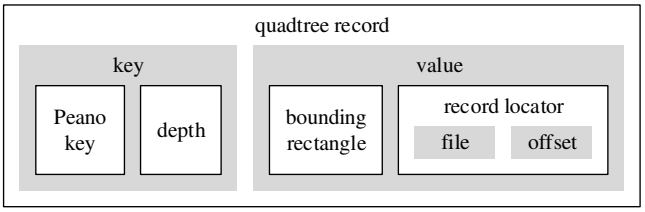


Figure 4: Quadtree index record format.

It is important to note that the PMR quadtree can be readily optimized when one realizes that much of the data in the "big data" space is generated by sensor, moving objects, etc. (polygonal data in this space is rare, other than in the filtering role). Essentially, these produce objects with point geometry. Because of this, it is possible to further optimize the quadtree index record to utilize a point geometry rather than a bounding rectangle as part of the value. Not only does this save space, but it also removes the necessity to perform secondary filtering when resolving spatial

queries as the index contains the full geometry of the indexed objects.

Strictly speaking, it would be possible for the Reduce task (step 3), to build a quadtree portion without the interim quadtree (step 1) having been built beforehand, but in the interest of performance, we obviate much of the potential paging to disk in the Reducer, by this interim quadtree which provides a first-pass approximation to the depth at the entry will lie in the final, complete quadtree.

3.2. Ordering and Clustering

In the case of the first model, besides construction of a PMR quadtree, it is possible to further optimize the data and index by reorganizing the source data to correspond to the ordering found within the index. The process of building a spatially-ordered copy of the data and indexing it can be completed by a sequence of five MapReduce jobs. These five jobs correspond to the following:

1. Choose spatial partitions based on sampling the input.
2. Make the spatially-ordered and partitioned copy.
3. Build buffer regions.
4. Build a quadtree index on the main, complete data.
5. Build quadtree index on buffer region data.

As in the description of the quadtree build (see Section 3.1), the preliminary MapReduce job to choose partitions samples the data in order to balance the size of data in partitions. This is done for load balancing of both the build as well as subsequent querying. For each sample object, we calculate the Peano-key address, consistent with the quadtree to be built. We bucket the samples into quadtree quadrants of a certain, generally shallow, partitioning depth. Then we essentially distribute 2^{d_p} uniform ordered rectangular blocks (in subsequences) among the desired number of partitions (where d_p is the partitioning depth). In contrast to the rectangular partitioning of SpatialHadoop [8] and Hadoop-GIS [2], we allow the partition regions to have jagged shapes and to contain multiple disjoint pieces. An example of a disjoint partition region is depicted in Figure 5, labeled region 7. The list of partitions is stored, by starting Peano-key address, in the table metadata (by augmenting the table properties in the Hive metastore).

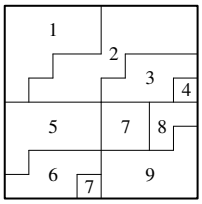


Figure 5: Quadtree containing nine partition regions.

The optional spatially ordered copy trades disk space and pre-computation in order to optimize searches by reducing the number of blocks that need to be read from disk in complete-row lookup. In step 3, a MapReduce job loads and transforms the original data into the spatially ordered copy. The Map function calculates the Peano key for the geometry of each row. For depth, it uses the maximum depth of the quadtree to be built, so the ordering of the complete ordered data is compatible with the ordering of its index. The custom partitioner for building the ordered copy, partitions based on Peano key using the pre-calculated partition boundaries; it is the same partitioner used by the MapReduce application that builds the quadtree index. Each partition is stored in a key-value sequence file in HDFS, where the key is null (as typical when a tabular schema is defined in Hive) and the value is the complete row. Hadoop sequence files are chosen for the following properties: they are splittable, for use as input to MapReduce jobs; it is possible to look up rows one-by-one after index search; they are compact and performant; and finally, they handle various data formats including binary formats.

For spatially-ordered tables, a MapReduce job is necessary to build a buffer region around each partition. The buffer region contains spatial objects outside the partition region that are within a user-specified buffer radius, which is to be chosen based on the expected size of proximity queries. The buffer region is designed to avoid reading from multiple partitions for proximity queries near partition boundaries. This applies to a range query with a query range overlapping a second partition by less than the buffer radius. It also applies to a k-NN query where some of the k objects nearest to the query location are in a partition neighboring the partition containing the query center, but distant from the partition boundary by at most the buffer radius. For each input row, the Map function determines whether the spatial object needs to be in any buffer regions - by inflating the bounding rectangle of the spatial object. If the spatial object will be in any buffer regions, the Mapper then determines the distance to the neighboring partitions. For each partition at a distance of at most the buffer radius, it emits a record of which the key is the partition number of the partition that needs to include the spatial object in its buffer region, and the value is the row annotated with a row identifier. The row identifier is needed for de-duplication during queries. The partitioner simply returns the partition number from the key, and the reducer is trivial. The storage is a sequence file whose value is the complete row annotated with the row identifier information.

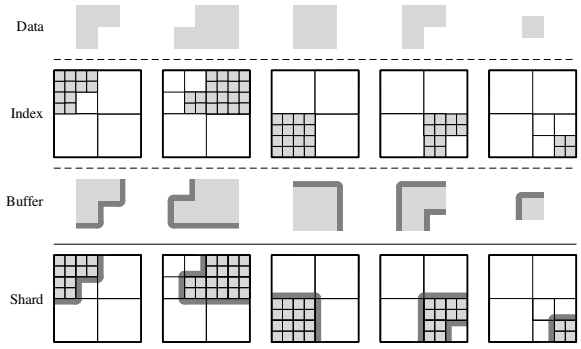


Figure 6: Example highlighting the relationship between data, index, buffer, and the shard.

The same MapReduce application that builds the quadtree index on unordered data is used to build the quadtree index on the spatially ordered derived table. The same partitioner, based on Peano keys, is used for building both the spatially ordered copy and the quadtree index. This results in the entries of one index partition corresponding to the rows of one partition of the complete ordered data.

Finally (step 5), a quadtree index is built on the buffer region data in each partition. The one difference, when building the quadtree index on the buffer region data – rather than on the main data – is that the partitioning phase is done by already-determined target partition number, rather than by Peano key. This also implies that

the buffer region indices are a separate index per partition, and do not constitute a global index of buffer regions across partitions.

4. SPATIAL ANALYTICS

Spatially indexing datasets, while an interesting topic, is a means to an end - most notably, providing the ability to improve the performance of spatial queries against the indexed dataset. Users typically do not care whether a dataset is indexed – they instead focus on what can be done with the data; what queries can be answered, what they can learn from their data.

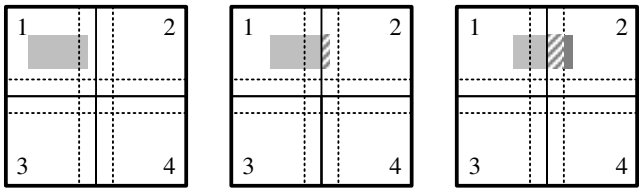
Utilizing our PMR quadtree implementation on Hadoop, we have implemented the traditional range and k-NN queries – without requiring a MapReduce job for every search parameter. These can be considered building blocks in the implementation of more sophisticated and useful queries in the future.

4.1. Range Query

The design of the spatial index allows searching with near-standard random-access range query techniques. The key differences between standard implementations of range queries on a PMR quadtree and ours are the following:

1. The index is partitioned into multiple files.
2. The buffer-region index is used along with the main index, in the case of spatially-ordered derived files.
3. Optimizations are made for the common case of point data.

As the quadtree index consists of multiple segments (i.e., separate files in HDFS for each partition), a query region may intersect with the multiple partitions of the index. The partition boundary list from the table metadata is used – as it is in the Partitioner phase of the MapReduce job that built the index – to determine the range of partitions in which to search. The lowest partition is the one containing the low-coordinate corner of the search MBR, and the highest partition is the one containing the high-coordinate corner of the search range. Only those partitions of the index that are within this range are opened during the search.



(a) Ignore buffer, one partition (b) Buffer advantageous, one partition (c) Ignore buffer, two partitions

Figure 7: Example highlighting the interplay between the query region and partition buffers. The optimizations shown in (a) and (c) are only applicable to point data.

The buffer region may be used to avoid searching in multiple partitions of the index when the query region is near the boundary of a partition region. If the query region overlaps a second partition region by a distance less than the buffer radius (see Figure 7b), the buffer region index is utilized in order to avoid reading from the second partition. However, when the query region overlaps a second partition region by more than the buffer radius (see Figure 7c), the buffer region provides no such advantage – the buffer region is read only for the purpose of including a line or polygon that was assigned to a non-participating partition, but overlaps one (or more) of the participating partitions.

When the spatially ordered copy is used, we expect rows that are close to each other spatially will also usually be stored near each other in the data file. Thus, we minimize the number of blocks that need to be read from disk when looking up the complete rows.

Point data is, in our experience, by far the most common geometry type of spatial data in the Big Data domain. In the case of point geometry, two numbers (floating point doubles) suffice to store the full geometry of the spatial object as an entry in the quadtree, rather than the more common four doubles to represent the bounding rectangle or higher dimensional line or polygon data. Storing two doubles for a point, rather than four doubles for bounding rectangle, allows the size of the index to be reduced by about 30%. Not only is it desirable to reduce storage space, but also we expect less frequent occurrence of the need to read across disk blocks when reading the index, for same-size query regions, when the index records are shorter.

Unlike lines and polygons (where the full geometry must be read out of the complete data file in order to refine the intersection test), with point data, the full geometry is obtained directly from the entry in the quadtree index. Thus, for queries that require only the geometry (without attributes – e.g., when drawing to the screen of the client application), we can read from the index without having to open the file of complete data. However, when symbolizing or analyzing on any attribute, it is necessary to read the full data (the complete feature is not being stored inline in the index).

Another optimization for point geometry is applicable when buffer regions are present, i.e., in the case of a spatially-ordered derived table. With line and polygon data, it is necessary to include the buffer-region index in all queries because we assign each row to only one partition, even if it overlaps multiple partition regions. In the case of point data, in the worst case, the point can lie on the partition boundary. Thus, if the query region is fully contained within a partition region, we do not need to open the buffer index file (nor the buffer-region complete data file – see Figure 7a). Furthermore, in the case of a range query that overlaps a second partition region by more than the buffer radius, once it is already necessary to search in multiple partitions, we do not need to search in the buffer region when it offers no advantage (see Figure 7c).

4.2. k-NN Query

The design of the spatial index facilitates implementation of a k-NN query using the standard incremental k-NN algorithm [15]. This is due in part to being able to support random reads into a spatial index file – without requiring a MapReduce job for every search parameter. The key differences between our implementation on Hadoop, and the standard algorithm, are the following:

1. The index is partitioned into multiple files, and we search only one partition.
2. The buffer region index is used along with the main index.

For k-NN queries, we search only in the partition containing the query center. The partition containing the query center is determined by the Peano key of the query center and the partition boundary list from the table metadata. When a quadrant is dequeued from the incremental k-NN priority queue and the quadrant overlaps more than one partition, we know that the quadrant corresponds to an implicit internal node in the quadtree of actual

data. This is the case because the quadtree index was constructed such that every leaf quadrant is fully contained within a partition of the index. The determination that a quadrant is an implicit internal node allows us to enqueue its subquadrants, without having to read from disk. As a consequence, we avoid reading from any other partition, despite top-down tree traversal for enqueueing.

To support k-NN query around an arbitrary query center – including a query center that may happen to lie near a partition boundary – requires using the spatially-ordered copy of the data, which provides the buffer region. With the buffer region, k-NN queries are supported where the distance to the kth-nearest object is at most the buffer radius.

The buffer region of spatially-ordered derived tables allows performing a k-NN query while reading from only one partition, even when the query center is near a partition boundary. When a quadrant is dequeued from the incremental k-NN priority queue, one of three conditions holds: (1) the quadrant overlaps multiple partition regions; (2) the quadrant is within the query partition; or (3) the quadrant is outside the query partition. In case (1), as explained above, the quadrant is an internal node. For case (2), the quadrant inside the partition, we search in the main index and not the buffer region index. Symmetrically, for case (3), the quadrant outside the query partition, we search in the buffer region index but not the main index. Furthermore, a quadrant outside the query partition, is discarded before reading from disk, if the distance from the quadrant to the query region is greater than the buffer radius.

As for point geometry type, the benefits of smaller storage size of the index, apply to k-NN as well as to range query. Likewise, queries that request the geometry only, can read from the index only, and not from the complete data, for Point data. However, no additional algorithmic optimization is made for Point data with k-NN query.

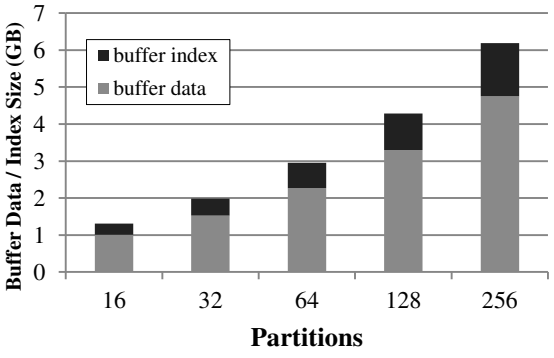


Figure 8: Buffer data and index size (in GB) for various partition counts.

5. PERFORMANCE COMPARISON

In order to test the performance (wall clock and disk I/O) as well as the storage requirements, we selected a recently available public dataset – the New York City Taxi and Limousine Commission's 2013 taxi tripsheet dataset [25]. It consists of over 170 million records, detailing both trip and fare data for every taxi trip recorded in the seven boroughs of New York City in 2013. The data is available for download as a collection of 24 CSV files (~50GB uncompressed). The trip data contains information such as hack license, pickup date/time, dropoff date/time, passenger count, trip duration, trip distance, and pickup and dropoff latitude/longitude. The fare data contains hack license, pickup date/time, fare, tip amount, tolls, and total trip cost.

Performance tests were run on a 20 node Hadoop cluster. Each node in the cluster was a garden variety desktop PC containing either 4 or 8 cores. Each desktop was outfitted with 16GB RAM, and 2.5TB of reasonably fast disk. Each machine was running CentOS 6.5 Linux and Hadoop 2.2.

5.1. Indexing

The data footprint was measured for a collection of different partition sizes, with the count of partitions ranging between 16 and 256. The source data required 28.7 GB of storage; the PMR quadtree occupied an additional 7.2 GB. Both were constant across all partition counts. What did vary by partition count was the amount of storage required for the buffer regions and their indexes. Not surprisingly, the more partitions, the more storage overhead was required. These values are shown in Figure 8.

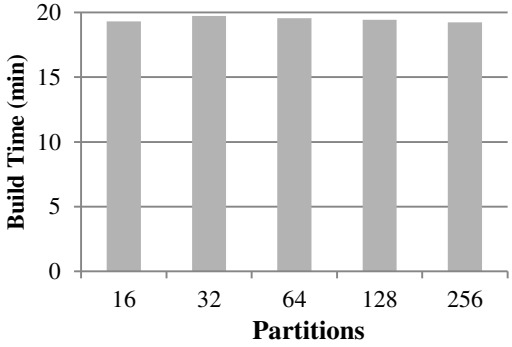


Figure 9: Build times (in minutes) for various partition counts.

Build times for the PMR quadtrees were relatively unaffected by partition count (i.e., 19.2 – 19.7 minutes for partition counts between 16 and 256 as shown in Figure 9). This is somewhat unexpected given that the size of the buffer data and index storage grows from 1.3 GB (partition count 16) to 6.2 GB (partition count 256). Persisting the larger volume of buffer data and index to HDFS will take more time; however, this is offset by having more Mappers (and opportunities for parallelism) participating in the process.

For comparison purposes, we observed that on a single-node computer, indexing the data using a regular grid (a far simpler spatial index), took approximately two hours.

5.2. Analytics

When analyzing the performance of both range queries and k-NN queries on Hadoop, there are a number of interesting dimensions to the problem. We first considered the impact of sequential and non-sequential reads of the index and data when resolving the queries.

As shown in Figure 10, the range query performance times varied by PMR quadtree splitting threshold. The performance curve is reminiscent to observed performance curves in the sequential environment where optimal performance does not correspond to the minimal splitting threshold [17]. In our test cases, the threshold of 1024 was the best performing. By superimposing the number of random reads, it is clear that performance is governed in large part by I/O.

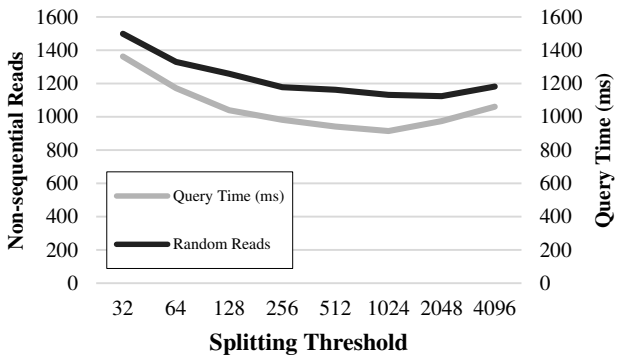


Figure 10: Range query times (in milliseconds) and non-sequential reads for different splitting thresholds.

We also examined the impact of ordering the source data following construction of the spatial index on query performance. We observed that ordered data has a very significant impact upon the range query performance, and less of an impact on k-NN query performance.

In Figure 11, we highlight the impact of unordered data and the accompanying non-sequential reads for the range query. The range query on unordered data took more than twenty times as much time (55.9 seconds versus 2.4 seconds). In addition, the unordered data resulted in roughly eight times more non-sequential reads.

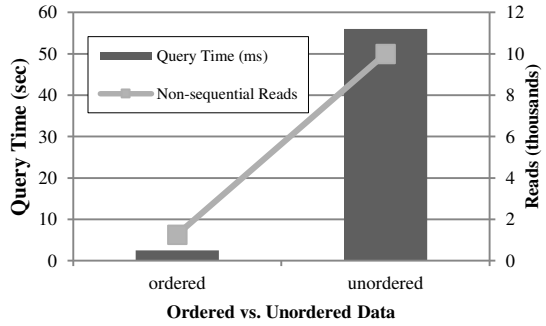


Figure 11: Range query times (in milliseconds) and non-sequential reads, comparing ordered and unordered data.

The k-NN query implementation also demonstrated the impact of ordered data, though to a far smaller extent than was observed with the range query. Using our New York City taxi data, we saw that a k-NN query (where k was large – e.g., 1000), the query on unordered data took roughly twice as much time as the same collection of query points on the ordered data (16.2 seconds versus 8.4 seconds). However, the difference in non-sequential reads was not as great – 1000 versus 878 with the ordered data. This accounts for much of the more similar query performance.

Another interesting question that warranted experimentation involved the MapReduce programming model and the utility of spatially indexing the data. As is often the case with MapReduce jobs, full scans of the data are sometimes performed. This is common with certain classes of analytic functions that rely upon global statistics gathered on the source data – e.g., hot spot analysis. We tested two implementations of a range query, one based upon a MapReduce-based full scan of the non-indexed data, and the second our implementation as described previously.

In Figure 12, three performance curves are plotted. The nearly horizontal line represents the query time required for the naïve MapReduce implementation (triangle data points). Given that the implementation requires examination of all data across the Hadoop cluster, it is not surprising that the query times vary a small amount, regardless of the number of records returned by the range query.

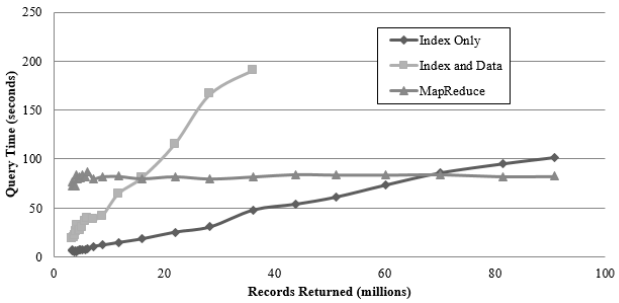


Figure 12: Range query times (in seconds) using both naïve MapReduce full scan implementation and the PMR quadtree-based algorithm.

The second line depicted is the dark gray line (diamond data points); it corresponds to the time required to return the point locations of all features intersecting the various query regions. As was discussed in Section 3, the PMR quadtree implementation is optimized for point data. If only the points are required (e.g., for rendering to a display) without the associated attributes, high levels of performance may be obtained. The third line depicted (light gray, square data points) corresponds to the time required to return the completed features (geometry plus all attribution).

It is interesting and significant to note the break even points depicted in Figure 12. If the client requires only the point geometries of the features when answering the range queries, it appears that when the volume of features returned exceeds roughly 65 million (~40% of our test dataset), the naïve MapReduce range query implementation will outperform the PMR quadtree based implementation. This reflects the increased complexity and computational costs of the quadtree implementation on a per feature basis. However, the quadtree implementation offers better selectivity and avoids the need to examine all features in the dataset.

Finally, when the client requires both the point geometries and the attributes of the features, when the volume of returned features exceeds approximately 15 million (~10% of the dataset), the MapReduce implementation offers superior performance. Figure 12 shows that the index performs better when the range is small enough to return a small subset of the data, but that when a substantial portion of the data is returned, a full scan with MapReduce completes faster. Another significance of this test is to show that for certain classes of problems, utilizing a spatial index is not necessarily optimal from a performance standpoint.

Another important consideration when designing a spatial index and query algorithms for use with a real-time user interface is to minimize the amount of time necessary between query invocation and the first result being returned to the client application for rendering. Figure 13 depicts the time between query invocation and first results for various range query sizes (note that range queries are used when rendering features to the user interface). In the figure, times are shown for range query result sizes varying from 3 to nearly 90 million features. At the lower end (which would

correspond to even a prohibitively large volume of features being displayed in a user interface – e.g., more than 3 million points), the observed times between query invocation and the first feature being returned were less than 400 milliseconds.

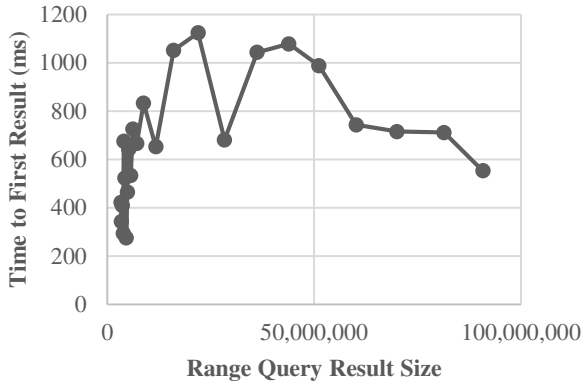


Figure 13: Time (in milliseconds) to return the first feature (i.e., latency) for various range query result sizes.

A complementary metric to the latency between query invocation and the first returned result record is the rate by which results are returned to the client. Figure 14 depicts the observed rates for the same range queries as depicted in Figure 13. When returning only the point geometries (which are contained in the spatial index), we are observing upwards of 900,000 records being returned for larger result sizes (depicted as the dark line in the figure). When measuring the number of geometries and associated attributes, the rate is considerably smaller – nearing 200,000 records per second. This performance difference is attributable to the need to process both the spatial index as well as the sorted attribute records.

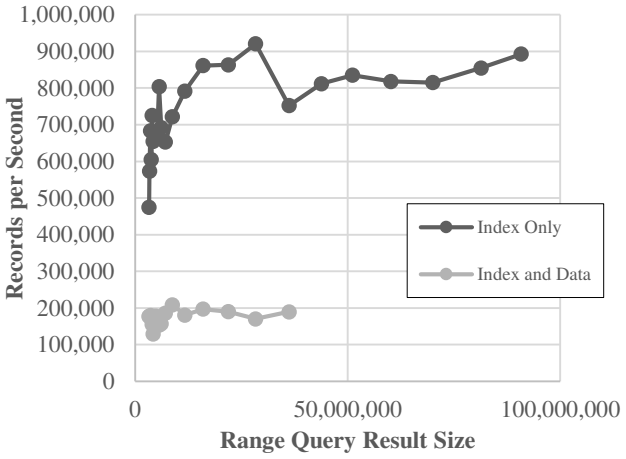


Figure 14: Number of records returned per second for various range query result sizes.

6. CONCLUSION AND FUTURE WORK

In this paper we have detailed a MapReduce-based implementation of a PMR quadtree that runs in a distributed manner across a Hadoop cluster. In addition, we described two key spatial queries (range and k-NN) that were implemented which consume the PMR quadtree. Differing from the quadtree build algorithm, these two queries were not implemented using a MapReduce programming model. This provides the benefit of allow simultaneous execution of large numbers of these queries against a dataset.

Performance analysis was performed against the quadtree index and query implementations using several "big data" scale datasets on a 20 node Hadoop cluster. We measured performance relative to various parameters or metrics. These included:

1. Index built time by partition count,
2. Buffer index and data overhead by partition count,
3. Range query performance by splitting threshold,
4. Range query times on ordered and unordered data,
5. Range query times using both naïve MapReduce and spatial index based algorithms,
6. Latency between invocation of a range query and the first result record being returned, and
7. The rate at which records are returned by range query size.

These performance tests led to a collection of observations regarding the performance of spatial indexes, range, and k-NN queries in a MapReduce/Hadoop environment:

1. Build times are relatively unaffected by partition count,
2. Buffer overhead increases with partition count,
3. PMR quadtree splitting thresholds impact query performance, much as they do in the traditional sequential environment,
4. Ordered data can lead to significant increases in spatial query performance as compared with unordered data,
5. As range query result sizes grow to a significant fraction of the dataset size, a simple MapReduce full table scan implementation can outperform one based upon a spatial index and ordered data,
6. Small latencies (e.g., < 0.4 seconds) between query invocation and the first range query result can be obtained in Hadoop, and
7. Result rows may be returned at high rates with properly architected systems.

It is important to note that with the MapReduce programming model, it is not clear that a spatial index is always warranted (as noted in observation 5 above). When you are performing either a range query against the data (e.g., when rendering a reasonable portion of the data in the client application), or when you are doing an analysis that requires significant proximity awareness, spatially indexing the data is warranted. Researchers have shown that a spatial index provides performance advantages for spatial joins [1], [9]. However, when performing range queries where a significant fraction of the dataset is being returned, or when spatially joining two very large datasets, it is not clear than a persisted spatial index offers much of a performance advantage over on-the-fly spatial indexing [22].

We have identified two primary topics for future work that extend our Hadoop-based PMR quadtree. The first topic involves extending the 2D PMR quadtree to a 3D PMR octree. This will facilitate the indexing of spatio-temporal data – organizations commonly temporally partition their data in HDFS. The bias toward temporal partitioning is a reflection of how the data is collected, as well as how it is most often consumed (e.g., temporal moment or range queries in conjunction with spatial queries). The

second topic involves extending the PMR octree to support the incremental maintenance of the index. A common use case is for organizations to pour data into existing datasets on their Hadoop cluster on a regular periodic basis. This requires a spatial index that can be incrementally maintained. The architecture of our spatial index is well suited to supporting incremental maintenance as additional data is added to a spatially indexed dataset; the incremental indexes on the periodically ingested data will be searched in tandem with the original index when resolving queries.

In the analytic space, there is a collection of aggregate and spatial statistics tools that will be of high utility when identifying significant data on large volume Hadoop clusters. These tools will include kernel density (magnitude per unit area from point features using a kernel function to fit a smoothly tapered surface to each point), and hot spot analysis (taking a set of weighted features, identifies statistically significant hot spots and cold spots using the Getis-Ord Gi* statistic [14]). It is important to note that most data stored in this space is relatively uninteresting. The key is finding the interesting data (the proverbial needle in the haystack). These analytic functions will be implemented to run on Hadoop using MapReduce or on Spark [26] using the resilient distributed dataset (RDD) programming model.

7. ACKNOWLEDGMENTS

We wish to thank other key development staff within Esri that have contributed to this work through discussion and critical feedback. These people include Bill Moreland, Mark Janikas, Mansour Raad, Lauren Bennett, Sud Menon, Scott Morehouse, and David Kaiser.

8. REFERENCES

[1] Aji, A., Wang, F., Vo, H., Lee, R., Liu, Q., Zhang, X., and Saltz, J. 2013. Hadoop-GIS: A High Performance Spatial Data Warehousing System over MapReduce, Proc. VLDB Endow. 6, 11 (August 2013), 1009-1020. DOI=http://dx.doi.org/10.14778/2536222.2536227.

[2] Aji A., Vo H., and Wang, F. 2014. Effective Spatial Data Partitioning for Scalable Query Processing in MapReduce, In Proceedings of the 40th International Conference on Very Large Data Bases (VLDB '14), Hangshou, China.

[3] Beckmann, N., Kriegel, H.-P., Schneider, R., and Seeger, B. 1990. The R*-tree: an efficient and robust access method for points and rectangles. In Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data (SIGMOD '90). ACM, New York, NY, USA, 322-331. DOI=http://doi.acm.org/10.1145/93597.98741.

[4] Cary, A., Sun, Z., Hristidis, V., and Rishe, N. 2009. Experiences on Processing Spatial Data with MapReduce. Scientific and Statistical Database Management, 302-319.

[5] Cordova, A., Rinaldi, B., Wall, M. 2014. Accumulo. O'Reilly Media, Inc., Sebastopol, CA, USA.

[6] Dean, J., and Ghemawat, S. 2008. MapReduce: simplified data processing on large clusters. Commun. ACM, 51, 1 (January 2008), 107-113. DOI= http://dx.doi.org/10.1145/1327452.1327492.

[7] Dittrich, J., Richter, S., Schuh, S., and Quiané-Ruiz, J.-A. 2013. Efficient OR Hadoop: Why not both? IEEE Data Engineering Bulletin. 36, 1, 15-23.

[8] Eldawy, A., and Mokbel, M. 2013. A Demonstration of SpatialHadoop: An Efficient MapReduce Framework for Spatial Data, Proc. VLDB Endow. 6, 12 (August 2013), 1230-1233. DOI=http://dx.doi.org/10.14778/2536274.2536283.

[9] Eldawy, A., Li, Y., Mokbel, M., and Janardan, R. 2013. CG_Hadoop: Computational Geometry in MapReduce, In Proceedings of the 21st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (SIGSPATIAL'13). ACM, New York, NY, USA, 294-303. DOI=http://doi.acm.org/10.1145/2525314.2525349.

[10] Esri. 2013. GIS Tools for Hadoop. https://github.com/Esri/gis-tools-for-hadoop.

[11] Fox, A., Eichelberger, C., Hughes, J., and Lyon, S. 2013. Spatio-temporal Indexing in Non-relational Distributed Databases, In Proceedings of the IEEE International Conference on Big Data, October 2013.

[12] Gargantini, I. 1982. An effective way to represent quadtrees. Commun. ACM 25, 12 (December 1982), 905-910. DOI= http://doi.acm.org/10.1145/358728.358741.

[13] Geohash.org. Tips & Tricks. http://geohash.org/site/tips.html.

[14] Getis, A., and Ord, K. 1992. The Analysis of Spatial Association by Use of Distance Statistics, Geographical Analysis, 24, 3 (July 1992), 189–206. DOI= http://dx.doi.org/10.1111/j.1538-4632.1992.tb00261.x

[15] Hjaltason, G., and Samet, H. 1999. Distance Browsing in Spatial Databases, ACM Trans. Database Syst. 24, 2 (June 1999), 265-318. DOI=http://doi.acm.org/10.1145/320248.320255.

[16] Hjaltason, G., and Samet, H. 2002. Speeding up construction of PMR quadtree-based spatial indexes. VLDB Journal, 11, 2 (October 2002), 109-137. DOI=http://dx.doi.org/10.1007/s00778-002-0067-8.

[17] Hoel, E, and Samet, H. 1991. Efficient Processing of Spatial Queries in Line Segment Databases, In Proceedings of the 2nd International Symposium on Advances in Spatial Databases (SSD'91), Zurich, August 1991.

[18] Hoel, E., and Samet, H. 2003. Data-parallel Polygonization, Parallel Computing, 29, 10 (October 2003), 1381-1401. DOI=http://dx.doi.org/10.1016/j.parco.2003.05.001.

[19] Leetaru, K., and Schrodt, P. 2013. GDELT: Global Database of Events, Language, and Tone, 1979-2012. International Studies Association Annual Conference, San Francisco, (April 2013).

[20] Neis, P., Zipf, A. 2012. Analyzing the Contributor Activity of a Volunteered Geographic Information Project — The Case of OpenStreetMap. ISPRS International Journal of Geo-Information, 1, 2, 146-165.

[21] Nelson, R., and Samet, H. 1986. A consistent hierarchical representation for vector data. In Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '86), Evans, D., and Athay, R., (Eds.). ACM, New York, NY, USA, 197-206. DOI=http://doi.acm.org/10.1145/15922.15908.

[22] Raad, M. 2013. BigData Spatial Joins, Blog post. http://thunderheadxpler.blogspot.com/2013/10/bigdata-spatial-joins.html.

[23] Thusoo, A., Sarma, J., Jain, N., Shao, Z., Chakka, P., Anthony, S., Liu, H., Wyckoff, P., and Murthy, A. 2009. Hive – A Warehousing Solution Over a Map-Reduce Framework. Proc. VLDB Endow. 2, 2 (August 2009), 1626-1629.

[24] White, T. 2009. Hadoop: The Definitive Guide (1st ed.). O'Reilly Media, Inc., Sebastopol, CA, USA.

[25] Whong, C. 2014. FOILing NYC's Taxi Trip Data, Blog post, http://chriswhong.com/open-data/foil_nyc_taxi.

[26] Zaharia, M., Chowdhury, M., Franklin, M., Shenker, S., and Stoica, I. 2010. Spark: cluster computing with working sets. In Proceedings of the 2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud '10), Boston, June 2010.