MSTSpatial: A Merge Sort Tree Based Index For Spatio-Temporal Range Queries

Rajiv Sangle, rajivsangle@iisc.ac.in
Varad Kulkarni, varadk@iisc.ac.in

***Abstract*—We present MSTSpatial: A Merge Sort Tree based index for answering Spatio-Temporal Range Queries Efficiently for large spatio-temporal data. High quality spatio-temporal data is being generated at a rapid pace with the advent of highly precise smart sensors and the rise of IoT (Internet of Things) devices. These "location-aware" applications and devices characterise moving objects using data ranging from smartphone activity logs to satellite images. These modern datasets derived from these devices, therefore, extensively span the spatio-temporal space of such objects. For our course project, we have analyzed various indexing and querying paradigms for spatio-temporal data analysis that take scalability over distributed systems into account. However, most of the work done for answering Spatio-Temporal Range Queries is based on R-Tree or 3D R-Tree implementations and not much work has been done in exploring segment trees for the same. We have implemented an index for efficient storage of spatio-temporal points in Apache Spark, and compare our results with a baseline naive approach, which stores the points without any pre-processing. We then use this index to answer spatio-temporal range queries efficiently and show that we achieve a 100x speed-up in answering 150 sequential queries, while the baseline approach naively iterates every time through the persisted RDD to answer queries. On achieving good results with this baseline, it compels us to explore and compare our work with existing frameworks, which is a part of our future proposed work. We successfully built a novel index based on Distributed Merge Sort Tree which is a variant of a Segment Tree, and show that at an added cost for pre-computation as compared to the baseline, we can answer large number of range queries rather efficiently. We have tested our index on an OSM dataset with 20 Million spatio-temporal points and achieved a 100x speedup with our baseline code.**

Keywords: Indexing, Querying, Segment Tree, Merge Sort Tree, Spatio-Temporal Range Query.

I. Introduction

Tracking the movement of both physical and digital entities is central to many applications.

At a macro level, tracking physical objects is important to observe and predict the dynamics of moving resources. Whereas at a micro level, even the movement and psychic of an individual can be tracked using the data from smartphones or IoT devices. The challenge however is to analyse this spatio-temporal data at large scales. The amount of spatio-temporal data currently generated and collected exceeds the capabilities of any single machine. Also, analytics optimized for running on a single machine failed to account for both the complexity in distributed large scale data and also the overhead costs associated with it. [1]

Efficient querying of spatio-temporal datasets in both time and space resources is important for improving the accuracy of data processing, and also for data pre-processing for running analytics efficiently. Spatio-temporal query though costly in time, is a basic yet fundamental operation.

A lot of computational resources are invested in spatio-temporal queries due to the sub-components of distance-based queries and data aggregation operations. [2]

Many indexing techniques like R-Tree [3], quad-tree [4], and KD-tree [5] have been designed to optimize spatio-temporal queries. Though they fail to provide high-performance large-scale spatio-temporal data.

This motivates us to look for MapReduce based frameworks to analyze spatio-temporal data effectively and efficiently. SpatialHadoop [2] and HadoopTrajectory [6] are two such functional extensions to the Apache Hadoop MapReduce Framework using which algorithms have been proposed to optimize spatio-temporal query.

However, the MapReduce framework, in general, is very process-centric. The Map and Reduce operations for prediction and analysis of spatio-temporal data become very verbose and inefficient.

Spark on the other hand offers a data-centric approach through lazy-evaluation of Transformations on Resilient Distributed Datasets (RDDs) and a subsequent Action. Spark is not only suitable for large-scale distributed datasets but is also optimized internally for load balancing and code-reusability.

As we have discussed above that Spark has benefits over the traditional MapReduce. Hence, in this project we hope to use Spark as a tool for implementing a large scale analytics framework.

However, we must mention that there are ongoing developments with concepts like Spatial-RDDs (Apache Sedona (formerly GeoSpark)) [7] and space-filling curves (GeoMesa) [8] that address query optimization in distributed setting, and further extend capabilities by integrating with Spark primitives.

We have aimed at a solution that will process high dimensional spatio-temporal data in Spark.

Given, the challenges in the spatio-temporal domain, and existence of a distributed software like Spark, motivates us to study such a problem.

Our contributions can be summarised as follows:

- A novel Merge Sort Tree based index on spatio-temporal points for efficient storage in Apache Spark
- Using the index for answering Spatio-temporal range queries efficiently
- A promising display of results with a naive baseline on a large dataset with 20 Million Points, creating room for further comparisons with existing work in the domain and eventually trying to perform better

The rest of the paper is organised as follows:
Section 2 discusses the Related Work. Section 3 describes our Problem Definition for the project. Section 4 contains the algorithm that we implemented and its design details. Section 5 talks about the experimentation, along with the execution and results. This is followed by the conclusion and future works in Section 5.

II. Related Work

A. Hadoop Frameworks

Frameworks like SpatialHadoop [9], HadoopTrajectory [6], Parallel-Secondo [10], and HadoopGIS [11] have explored the spatio-temporal data analytics domain and implemented various spatial data abstractions in Hadoop.

SpatialHadoop [9] is an extension to Hadoop that injects spatial data awareness in each Hadoop layer, namely, the language, storage, MapReduce, and operations layers. In the language layer, it adds a simple and expressive high level language for spatial data types and operations. It supports geometry types like polygon, point, line string, multi-point. SpatialHadoop provides spatial indexes and spatial data visualization, and a SQL extension called Pigeon [12].

HadoopTrajectory [6] is a hadoop extention for spatio-temporal data processing. This extention adds spatio-temporal types and operators that can be directly used in MapReduce Applications.

Parallel-Secondo [10] integrates Hadoop with SECONDO, which is a database that can handle non-standard data types, like spatial data. Hadoop is employed as the distributed task manager and performs operations on a multi-node spatial DBMS. The common spatial indexes and spatial queries (except KNN) are supported. But it does not handle the spatial data skewness problem, since, it only supports uniform spatial data partitioning techniques. The visualization function in Parallel-Secondo does not scale up to large datasets as it needs to collect the data to the master local machine for plotting.

HadoopGIS [11] makes use of SATO spatial partitioning [13], which is similar to KD-Tree [5], and local spatial indexing for achieving efficient query processing.
HadoopGIS does not offer standard spatial SQL, but can support declarative spatial queries with an integrated architecture with HIVE [14]. It also lacks the support of complex geometry types like convex/concave polygons, line string, multi-point, and multi-polygon. Images can be plotted by the HadoopGIS visualizer on the master local machine.

B. Apache Spark

Apache Spark [15], as we have studied in Module 2 of the course, is one of the big data analytics platform widely used by industries and researchers. It introduces the in-memory Resilient Distributed Datasets (RDDs) on which various transformations and actions can be performed to achieve the dedicated big data task, in a distributed setup that is abstracted from the programmer. Due to the abundant applications of Spark in big data analytics, and the interest it has evoked as a part of this course, we propose to use Apache Spark in our course project.

C. Spark Frameworks

DITA [1], Apache Sedona (formerly GeoSpark) [7], Simba [16] and UlTraMan [17] are frameworks implemented over Apache Spark[15] to support spatio-temporal data analytics.

DITA [1] discusses a trie-like indexing technique and performs various queries on trajectory data.
GeoSpark [18], is an extension of Apache Spark that supports spatial types, indexes, and geometrical operations at scale.

UltraMan [17] proposes a unified platform for big trajectory data management and analytics. It integrates a key-value store in Apache Spark, and enhances the MapReduce paradigm to allow flexible optimizations based on random data access to achieve scalability, efficiency, persistence, and flexibility.

Simba [16] is a system that offers scalable and efficient in-memory spatial query processing and analytics for big spatial data. Simba is also based on Apache Spark and runs in a distributed set-up. Simba extends SparkSQL to support spatial data processing over the DataFrame API. It builds local R-Tree indexes on each DataFrame partition and uses R-Tree grids to perform the spatial partitioning.

D. Apache Sedona (formerly GeoSpark)

Apache Sedona [7] extends the concept of in-memory Spark RDDs. A Spatial RDD consists of partitions and within each partition several of spatial objects are contained.

Spatial data is stored in different types of specific file formats (e.g. CSV, GeoJSON [19]) that have interoperability among GIS libraries. Traditional RDDs do not inherently understand the contents of these files or how to efficiently load such data formats.
Spark does not preserve the spatial closeness of spatial objects while portioning the data in the default way. Storing spatially proximal objects in the same partition helps in efficient querying.

These limitations of Spark RDDs with respect to spatial objects are taken care by Spatial RDDs. GeoSpark has a flexible implementation to handle heterogenous spatial objects.

Spatial RDD partitioning is done by grouping spatial objects into the same partition by determining their spatial proximity. An efficient spatial partitioning achieves load balancing (in terms of memory space and spatial computation), and thus avoids communication overheads due to shuffles across partitions.

GeoSpark spark uses global-index based data structures like R-Tree and Quad-Tree to speed up spatial query. But building a spatial index for the entire dataset is not possible because a tree-like spatial index incurs significant additional overheads. Hence, GeoSpark builds a spatial index (R-Tree or Quad-Tree) per RDD partition. These local trees only index spatial objects in their respective partition.

III. Problem Definition

In this section, we will be describing the work done as a part of this course project. We explored the implementation of two parts in geo-spatial data analytics pipeline:
(1) Indexing of Spatio-Temporal data
(2) Using these indexes to answer Spatio-Temporal Range Queries (STRQ) on the Spatio-Temporal data.
The motivation to implement this problem of answering Spatio-Temporal Range Queries (STRQ) in Spark was driven by the MapReduce based approach in SpatialHadoop [20] (STRQ).
We now discuss the STRQ problem definition on the spatio-temporal data.

A. Spatio Temporal Points Definition

First, Let's start by defining what a spatio-temporal(ST) point is. A ST data-point is a 3-dimensional tuple of fields which consists of the x-coordinate (x), the y-coordinate (y), and the timestamp:

$$p_i = (x_i,\ \ y_i,\ \ t_i)$$

and the dataset consists of spatio-temporal datapoints, each comprising two dimensions of space and one dimension of time.
Hence, a dataset is of a collection of such data-points p_is:

$$P = \{p_1, p_2, ..., p_n\}$$

B. Spatio Temporal Range Queries

A query is of the form $Q_i(t_i, x_1, x_2, y_1, y_2)$, s.t $x_1 <= x_2$ and $y_1 <= y_2$, which basically denotes the ith query consisting of a timestamp t_i. The later 4 points denote a rectangle enclosed by the lines, $x = x_1$ and $x = x_2$ as the vertical lines, and $y = y_1$ and $y = y_2$ the horizontal lines.
The figure 1 depicts the Bounding Rectangle (BR) for a rectangular boundary of interest. The BR is defined by the four the data-points at the four corners (red points) that enclose the rectangle.

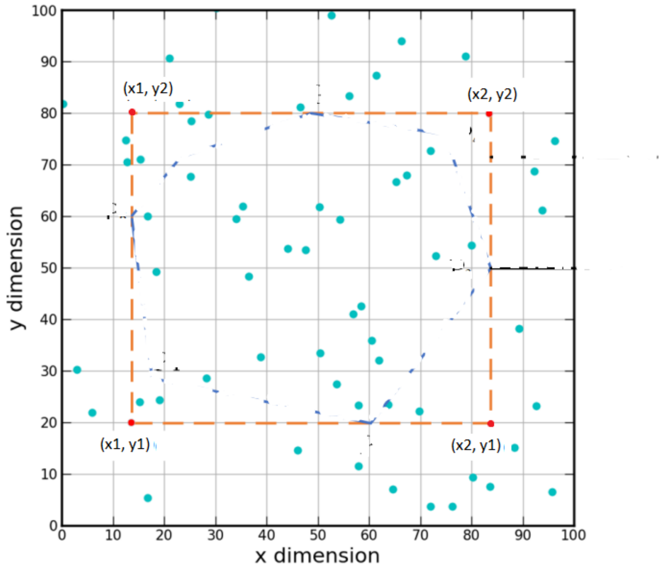

Fig. 1:
BR=((x1, y1), (x1, y2), (x2, y2), (x2, y1))

With reference to figure 1 above, the input to the query will be BR, and the output of the query will be all the data-points (blue points) inside BR at the given time interval t_i.

IV. Methodology

1) Preliminary Details and Assumptions: Given below are a few assumptions we made for simplifying the problem statement.

- Firstly, for the time instance, we process every point on a per day basis at the granularity of 1 second, i.e every timestamp in some given time zone is subtracted from a reference time '00:00:00'. This is a valid assumption, as in real worlds such analytics could just be done on a daily basis, without having much preference over a few set of days. By doing this, we restrict our distinct timestamps to 86400.
- All the distinct bounding rectangles(BR) for queries should be fed to the system before building an index. Once initialized with initial set of BRs, Adding new BR in the system is a very easy operation
- The total number of distinct bounding rectangles(BR) in the query is a moderately small number, this is a rather fair assumption as in real scenarios, let's say we are getting data from a moving drone, that only captures images of specific areas everyday, or a static surveillance camera only has a constant vision, hence total distinct rectangles could be rather small in the real world, small typically means in the ranges of ten thousands as compared to the large number of queries. Below are some calculations for the permissible number.
It typically is in the order of thousands or ten-thousands, depending on the system. By small, we mean with respect

to the number of queries which can be in order of 100 million. The exact number permissible would be bounded by the memory of the executor. As our storage is O(TimeStamps*DistinctBoundedRectangles) (which we will see in the next section), and distinct time stamps can be in the orders of 80k, so Number of DistinctBoundedRectangles shall be bounded by executor_memory/(80K*(sizeOf(int))*(sizeOf(One BR))). Size of 1 BR is 4 integer values, hence 16B, so the final value is close to executor_memory/(5MB). Typically 3K BR on a system with 16GB memory, and up-to 26K in more powerful systems with around 128GB memory. Basically linearly scales with the increase in memory.

2) Building The Index: In our index there are 2 parts, namely the global index, and the local index. The global index is used to decide which partition the data point would go to, and the local index will be used to perform the actual search in for our given query.

Global Partitioning and Global Index

For every point in the input dataset (x_i, y_i, t_i), we obtain its partition number by time modulo number of partitions, i.e $t_i \% numPartitions$. We used number of partitions equal to 16. Now, in a given partition, for every time instant t_i, we shall have a separate 2-D space, for which we introduce our novel MST based index below.

Merge Sort Tree Based Local Index

Once, we have received the partition, we first read the query input, and extract all the distinct bounding rectangles possible in the input. Then, for a given timestamp, for every such bounding rectangle, we compute the answer using the Merge Sort Tree. Below are the details.

For a given distinct bounding box, for the given time stamp, we need to calculate the total number of points lying inside. For this, first we sort the entire 2-D space(per time instance), with respect to the x-axis. Now as for the (x_1, x_2) is concerned, we can find the possible range for our points w.r.t x axis by using a binary search. Now, let's say we achieve a range (L,R) on the basis of the x-axis. The y-points are still unsorted. We need to count in the unsorted range, how many of the points in the dataset lie in between y_1 and y_2. Now, this would be answered by a Merge Sort Tree. We will build a Merge Sort Tree(MST) for the y-axis of points(in the order of sorted x) in the dataset for each partition. Then, for all such (L,R), we will use the MST to answer queries of the form:

What is the count of numbers in a range [L,R] in an unsorted array(here: the array of y-axis points), that lie in between a given range(here in between y_1 and y_2).

This problem is a classic range query problem for which we can use an MST. Hence, essentially every partition will be building such MSTs to answer such queries.

Summarising, for each bounding rectangle, the answer would be a range query from range (L,R) in the array of y-axis of points, that lie in between y_1 and y_2.

At the end, we shall have a structure as below at every partition for all distinct timestamps:

t_i = map('DISTINCT_BOUNDING_RECTANGLE: COUNT_OF_POINTS_IN_BOUNDING_RECTANGLE'))

For example, for time stamp 3, let us say there are 3 distinct bounding rectangles in the dataset: (1,2,3,4),(2,5,6,8) and (4,7,6,10). Our index for timestamp 3 will look like: 3:{(1,2,3,4):(.),(2,5,6,7):(.),(4,7,6,10):(.)}, where (.) denotes the count of points in the dataset, at that time instance, lying in that bounding rectangle.

Now, let us see, how to use this index to answer queries.

3) Answering STRQ Using MSTSpatial: Once, we have the local index built for every partition as described above, when we get an input query, $Q_i(t_i, x_1, x_2, y_1, y_2)$, from t_i, and our global partitioning function, we will know which partition to look into. This way we have pruned our search from the entire search space, just down to one partition.

Once, we load the partition data, all we need to first extract from the local index the entry for t_i. We shall have a map associated with every t_i that we built using the Merge Sort Tree, as discussed above. For obtaining the result, we simply need to look for the key (x_1, x_2, y_1, y_2) in the map, and that will contain our required answer.

A. Experimental Setup and Results Obtained

1) Setup: In this section we will discuss the experiments performed and the results obtained. Let's start with the experimental setup. We performed our experiments on the Turing cluster. Turing is a cluster having 24 compute nodes each with 8 cores and 32GB Memory, and 1 Head node with 6 cores and 48GB Memory. We ran our spark jobs with 2GB driver memory, 4 executors and 4 cores per executor, and 16GB executor memory.

Our entire code base[21] is in python and we have implemented all the mentioned algorithms in PySpark with compatibility for python versions ($>=$ 2.7). We used OSM datasets directly downloadable from geofabrik [22]. Table I shows the dataset specifications. We generated 1 set of queries with 150 queries and 10 distinct bounding rectangles(BR)(As mentioned in the Methodology Section). In the baseline code, we are inserting all the points in the rdd, without any partitioning logic. And while answering queries we persist the rdds, hence the transformations are not performed multiple times.

2) Evaluations: Initially, the correctness of our algorithm was confirmed by using the output of the naive approach as the baseline, and we achieved a completely correct output, hence our algorithm stands correct. Now, moving on-to the

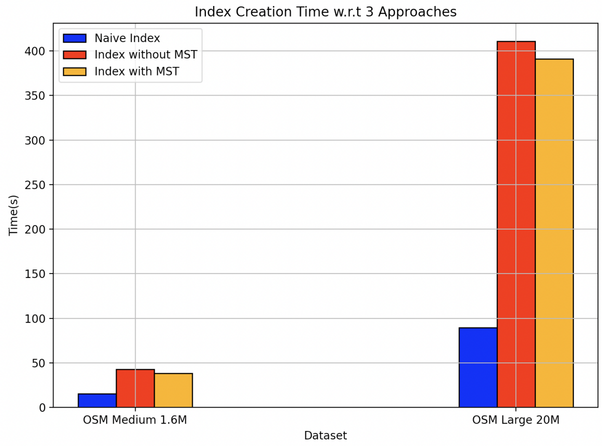

Fig. 2: Index Creation Time with all 3 strategies: Naive, Index without MST, Index with MST

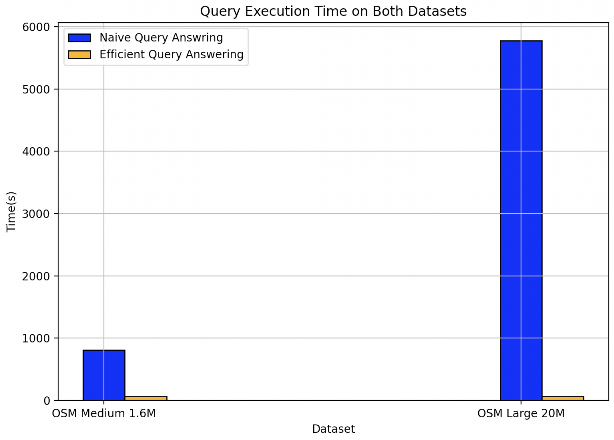

Fig. 3: Query Execution Times For each of the Datasets

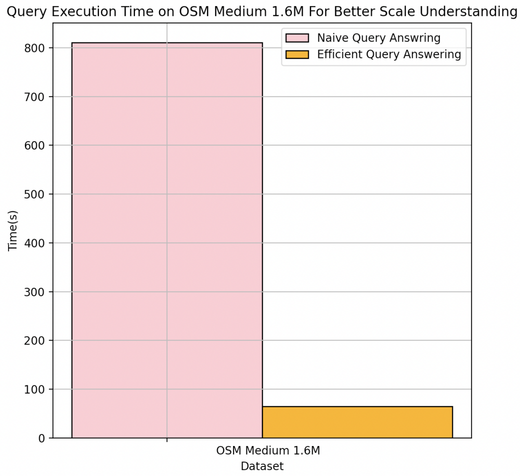

Fig. 4: Query Execution Times For OSM Medium For better Scale Understanding

DataSet	Num of Points
Osm Medium	1.6M
Osm Large	20M

TABLE I: OSM Datasets For Spatio-Temporal Range Queries

experiments.

Firstly, we report the index creation times for 6 distinct configurations. Refer to figure 2. The bars in the plot denote the profiled time required to generate the indexes. As expected, as there is no pre-processing done in the naive approach, the index creation time is minium for the same. When we create the index without using a Merge Sort Tree based approach, we can see that the time required is more. Although the difference is not much significantly high here, we can note that this is due to the moderate number (10) of distinct BRs used. As we would increase the number of BRs and as the width keeps$(x_2 - x_1)$ growing, we will start obtaining better gains, as the size of the bounding boxes would be large, hence increasing the lengths of the queried ranges, in turn forcing one to go for the Merge Sort Tree implementation. As future work we would like to confirm this hypothesis as we could not get enough time to collect evidence for the stated claim.

Secondly, we ran two experiments on each of the datasets for answering the queries. We generated 150 queries and ran it on both, OSM Medium and OSM Large and here are our findings. In figure 3(refer figure 4 for a better scale), we can observe that the query execution times are constant for our apporach irrespective of the dataset size. This is because we are only referring to a single partition for every query, hence the time did not change with respect to data size. Interestingly, in the naive approach we see the difference in execution times for both the data sets, which denotes that PySpark reads the data from all persisted partitions for each query, and with increase in number of data-points the look up time also increases. Lastly, to report from the plots we can see that we achieved close to a 100x speed-up as compared to the baseline and as a future scope, we hope to out-perform the existing platforms.

V. Conclusion and Future Outlook

In future outlook, we really want to see how we are performing with respect to existing platforms, having out performed the naive approach by a drastic margin. Also, given that we are using Segment Trees, allows us to look into direction of incremental spatio-temporal range querying. Segment trees allows us to perform point updates as well as range updates in O(LogN) time, given we perform Segment Tree with Lazy Propagation for range updates. Looking into this direction is rather interesting as, we have seen less literature on incremental spatio-temporal data. We also have underway a naive implementation of Trajectory Similarity Search problem using the Dynamic Time Warp Operator, which we would like to strengthen and give an end-to-end user intuitive programming model for performing the two operations, namely: (1.) Spatio-temporal range queries, (2.) Trajectory Similarity Search problem.

In conclusion, we have successfully built a novel index for answering STRQ which outperforms the baseline by 100x, which leaves us to explore and hopefully achieve a promising performance with respect to the existing baselines and look forward to out-performing them.

References

[1] Zeyuan Shang, Guoliang Li, and Zhifeng Bao. Dita: distributed in-memory trajectory analytics. In *Proceedings of the 2018 International Conference on Management of Data*, pages 725–740, 2018.

[2] Xin Li, Huayan Yu, Ligang Yuan, and Xiaolin Qin. Query optimization for distributed spatio-temporal sensing data processing. *Sensors*, 22(5):1748, 2022.

[3] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, pages 47–57, 1984.

[4] Raphael A Finkel and Jon Louis Bentley. Quad trees a data structure for retrieval on composite keys. *Acta informatica*, 4(1):1–9, 1974.

[5] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[6] Mohamed Bakli, Mahmoud Sakr, and Taysir Hassan A Soliman. Hadooptrajectory: a hadoop spatiotemporal data processing extension. *Journal of Geographical Systems*, 21(2):211–235, 2019.

[7] Jia Yu, Zongsi Zhang, and Mohamed Sarwat. Spatial data management in apache spark: the geospark perspective and beyond. *GeoInformatica*, 23(1):37–78, 2019.

[8] James N Hughes, Andrew Annex, Christopher N Eichelberger, Anthony Fox, Andrew Hulbert, and Michael Ronquest. Geomesa: a distributed architecture for spatio-temporal fusion. In *Geospatial informatics, fusion, and motion video analytics V*, volume 9473, pages 128–140. SPIE, 2015.

[9] Ahmed Eldawy and Mohamed F Mokbel. Spatialhadoop: A mapreduce framework for spatial data. In *2015 IEEE 31st international conference on Data Engineering*, pages 1352–1363. IEEE, 2015.

[10] Jiamin Lu and Ralf Hartmut Güting. Parallel secondo: boosting database engines with hadoop. In *2012 IEEE 18th International Conference on Parallel and Distributed Systems*, pages 738–743. IEEE, 2012.

[11] 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. In *Proceedings of the VLDB Endowment International Conference on Very Large Data Bases*, volume 6. NIH Public Access, 2013.

[12] Ahmed Eldawy and Mohamed F Mokbel. Pigeon: A spatial mapreduce language. In *2014 IEEE 30th International Conference on Data Engineering*, pages 1242–1245. IEEE, 2014.

[13] Hoang Vo, Ablimit Aji, and Fusheng Wang. Sato: a spatial data partitioning framework for scalable query processing. In *Proceedings of the 22nd ACM SIGSPATIAL international conference on advances in geographic information systems*, pages 545–548, 2014.

[14] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: a warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment*, 2(2):1626–1629, 2009.

[15] Eman Shaikh, Iman Mohiuddin, Yasmeen Alufaisan, and Irum Nahvi. Apache spark: A big data processing engine. In *2019 2nd IEEE Middle East and North Africa COMMunications Conference (MENACOMM)*, pages 1–6. IEEE, 2019.

[16] Dong Xie, Feifei Li, Bin Yao, Gefei Li, Liang Zhou, and Minyi Guo. Simba: Efficient in-memory spatial analytics. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1071–1085, 2016.

[17] Xin Ding, Lu Chen, Yunjun Gao, Christian S Jensen, and Hujun Bao. Ultraman: A unified platform for big trajectory data management and analytics. *Proceedings of the VLDB Endowment*, 11(7):787–799, 2018.

[18] Jia Yu, Jinxuan Wu, and Mohamed Sarwat. Geospark: A cluster computing framework for processing large-scale spatial data. In *Proceedings of the 23rd SIGSPATIAL international conference on advances in geographic information systems*, pages 1–4, 2015.

[19] Howard Butler, Martin Daly, Allan Doyle, Sean Gillies, Tim Schaub, and Christopher Schmidt. Geojson. *Electronic. URL: http://geojson.org*, 2014.

[20] Bernd-Uwe Pagel, Hans-Werner Six, Heinrich Toben, and Peter Widmayer. Towards an analysis of range query performance in spatial data structures. In *Proceedings of the twelfth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 214–221, 1993.

[21] https://github.com/varadkulkarni11/mstspatial.

[22] https://download.geofabrik.de/asia/india.html.