Distributed and Parallel Databases (2019) 37:697–720
https://doi.org/10.1007/s10619-018-7254-0

A framework for parallel map-matching at scale using Spark

Douglas Alves Peixoto[1] · Hung Quoc Viet Nguyen[2] · Bolong Zheng[1] · Xiaofang Zhou[1]

Published online: 10 November 2018

Abstract

Map-matching is a problem of matching recorded GPS trajectories to a digital representation of the road network. GPS data may be inaccurate and heterogeneous, due to limitations or error on electronic sensors, as well as law restrictions. How to accurately match trajectories to the road map is an important preprocessing step for many real-world applications, such as trajectory data mining, traffic analysis, and routes prediction. However, the high availability of GPS trajectories and map data challenges the scalability of current map-matching algorithms, which are limited for small datasets since they focus only on the accuracy of the matching rather than scalability. Therefore, we propose a distributed parallel framework for efficient and scalable offline map-matching on top of the Spark framework. Spark uses distributed in-memory data storage and the MapReduce paradigm to achieve horizontal scaling and fast computation of large datasets. Spark, however, is still limited for dynamic map-matching, and memory consumption in Spark can be an issue for very large datasets. We develop a framework to allow map-matching on top os Spark, while achieving horizontal scalability, memory-wise usage, and maintaining the accuracy of state-of-the-art matching algorithms by: (1) We combine a sampling-based Quadtree spatial partitioning construction and batch-based computation to achieve horizontal scalability of map-matching, as well as reduce cluster memory usage. (2) We employ a safe spatial-boundary approach to preserve matching accuracy of boundary objects. (3) In addition, a cost function for the distributed map-matching workload is provided in order to tune the framework parameters. Our extensive experiments demonstrate that our framework is efficient and scalable to process map-matching on large-scale data, while keeping matching accuracy and low memory usage.

Keywords Map-matching · Spark · Trajectory · Efficiency · Scalability

✉ Douglas Alves Peixoto
d.alvespeixoto@uq.edu.au

Extended author information available on the last page of the article

1 Introduction

Map-matching is the process of matching recorded GPS trajectory observations to road segments on a digital map. This process is useful in applications such as intelligent transportation systems (ITS), traffic analysis, smart cities, and routes recommendation, to name a few. Since GPS records can be incomplete, inaccurate and noisy due to connection problems and signal loss, urban canyons, sparse collection rates, and law restrictions, etc., GPS trajectories may not accurately reflect the location of moving objects. Therefore, map-matching is a process to ease the uncertainty and improve the accuracy of trajectory data analysis by matching the GPS records to the logical model of the real world [33]. The large amount of GPS trajectory data available, however, has introduced a new problem of how to match massive amounts of both map and trajectory data in an efficient manner, since traditional map-matching algorithms focus on the accuracy of the matching rather than performance and scalability. Moreover, map data availability has also increased, for example, the OpenStreetMap (OSM) [18] releases a weekly version of the Map of the World, currently with over 700GB of uncompressed data.

Offline map-matching methods use the knowledge of the complete trajectory geometry and its semantic attributes (e.g. speed, direction) to find its best match on the road network, and commonly needs to be performed only once for the entire dataset [33]; unless a whole new set of trajectory data is acquired for elsewhere; or there is a need for re-processing the original data when more accurate algorithms become available, or most commonly, when a new and updated version of the road map is available. Therefore, offline map-matching plays a key role on trajectory pre-processing by improving data quality and reducing uncertainty.

The overall approach for map-matching is to take recorded serial location points (e.g. GPS coordinates), and relate them to edges in an existing road network graph. However, this approach can quickly became cumbersome for large trajectory and map datasets, since every GPS point record has to be compared with every road edge. Nevertheless, map-matching computation is intrinsically parallelizable. For instance, [12,21,22,26] decompose the data space using spatial data structures (i.e. Grid, Quadtree), then co-group both map and trajectory data by containing spatial partitions, and perform the matching in each spatial partition in a parallel fashion, achieving orders of magnitude speed up. Besides, with the increasing demand for low-latency services over large scale data, a trajectory-based system should provide good scalability and fast response for map-matching. To address this important issue, an alternative is to partition both trajectory and map data into self-contained partitions that can be processed in a fault-tolerant distributed manner, while storing data in main-memory to reduce I/O cost, therefore improving map-matching performance and scalability [23]. In this work, we leverage the parallelizable property of map-matching computation with Spark and Quadtree space partitioning to achieve both scalability and performance speed up. The proposed framework was built to achieve both speed up, by means of parallel computation and in-memory data storage, and scalability using spatial-aware partitioning and distributed data storage and computation. We also focus on memory usage, since the framework is developed on top of an in-memory data structure (i.e. Spark RDD).

Existing works focus either on the matching accuracy or its performance. Accuracy-driven algorithms such as [11,16,17,24], can achieve high accuracy, but are limited to small datasets, since they focus on the accuracy of the matching rather than its performance and scalability, iterating through the entire dataset to find the best match, thus facing performance deterioration as the dataset grows. Performance-based algorithms such as [12,21,22,26], on the other hand, consider spatial partitioning and parallel processing to speed up map matching computation, but do not account for load balancing and memory usage, and are limited for disk-based computation. Furthermore, due to different density of trajectory data distribution in urban areas, we must account for load-balancing when partitioning the data space for parallel processing.

Frameworks like Spark [32] can fill the gap between performance and scalability, since Spark is an in-memory based framework, and supports distributed parallel computation. Spark have been used in a handful number of data-intensive analytics, including large-scale spatial databases [27,29]. We implement our solution on top of Spark's RDD, which provides a robust distributed data structure for MapReduce tasks in main-memory. However, since Spark is a distributed and in-memory storage based framework, we must account for workload balancing and main-memory usage. Existing systems for spatial data using Spark and MapReduce [1,8,27] employ balanced partitioning structures, such as Quadtree, k-d Tree, and STR-Tree, to provide workload balancing. Optimizing load-balancing and memory usage are essential to a good Spark algorithm. The main limitations and challenges of large-scale map-matching using Spark include:

- *Load Balancing and Dynamic Spatial Data Partitioning* Since Spark is designed for parallel distributed computation, we must account for data partitioning and load balancing, which are not directly supported for spatial data in Spark. Existing works for map-matching using MapReduce [12,22] employ uniform Grid space partitioning to organize the dataset into self contained partitions. However, they do not account for load balancing, which is essential in Spark to reduce contention and communication cost. Workload balancing can greatly reduce the cost of map-matching in Spark, as we demonstrate in our experiments.
  However, balanced space partitioning structures should be built in a dynamic fashion as the data is consumed. In a parallel distributed environment, such as Hadoop, the processes need to exchange data through the network after the shuffle phase, to aggregate data in the same partition, which increases network cost. With spark the problem is even more challenging, since the Spark's RDD data structure is read-only, which means that to create a dynamic data structure with spark using the entire dataset would demand to build a new RDD on every iteration of the dynamic process, which is both memory and computationally expensive.
  Thus, related works based on dynamic spatial partitioning such as [26] cannot be applied directly, since we need to find the best partitioning schema beforehand.
- *Memory Consumption and Replication of Boundary Objects* Once Spark is an in-memory storage framework, mainly for commodity hardwares, the amount of memory available in the cluster may be limited, and hence does not comfortably fit the entire map and trajectory datasets. For instance, the fastest storage level of Spark stores the dataset in memory with replication to speed up data recovery in

case of node failure; this can quickly exhaust the cluster memory available. Hence, memory consumption must be taken into account.
Furthermore, both trajectory and road map segments can extend for multiple spatial regions, thus we must account for boundary objects when partitioning the datasets. The easiest solution is to replicate boundary objects to all intersecting partition; for in-memory frameworks like Spark, however, this is undesirable, since replication will increase in-memory storage. To avoid replication, existing work split line segments according to their intersecting partitions, however, due to temporal sparseness of record points and GPS noise, this approach is prone to boundary points mismatch.

1.1 Contributions and novelty

In this paper we propose a Spark-based framework for large-scale map-matching. We leverage the distributed in-memory nature of Spark for scalable and fast processing of offline map-matching. We provide a sampling-based Quadtree partitioning for load-balancing using a cost-model to allow Spark to use a dynamic spatial data structure. Furthermore, we apply a batch-based data loading and processing to reduce memory consumption. In addition, we employ boundary extension using an empirical evaluation for accuracy maintenance as well as replication reduction. Finally, we provide experimental evaluation and study of parameters of our framework. Our key contributions can be summarize as follows:

1. *Cost-function* We provide an estimative of the distributed map-matching workload cost, in order to tune the system parameters and optimize the sampling-based data partitioning.
2. *Cost-based spatial partitioning* We employ Quadtree partitioning for trajectory and map data. Quadtree has been previously applied for parallel map-matching with good performance outcomes [26]; moreover, our experiments demonstrates that Quadtree provides an efficient and fairly uniform space partitioning when compared with other commonly used dynamic structures, such as k-d Tree and STR-Tree, achieving better performance and scalability. Since building a dynamic spatial index model from a large dataset can be cumbersome, and a data partitioning model must be provided to Spark beforehand, we address this limitation by providing a sampling-based quad-index construction using a cost-based model. Finally we co-partition both map and trajectory data using the quad-index model into the Spark's RDD.
3. *Batch loading and map-matching* Once the map data is loaded, trajectory records can be matched independently, thus we provide a batch-based loading and processing of the input trajectory dataset to reduce distributed memory consumption, specially in situations where the cluster memory size is a constraint.
4. *Empirical boundary replication* In addition, we employ a safe boundary threshold for segmentation and replication as proposed in [26] to reduce uncertainty. However, previous works did not evaluate the choice of the threshold value. Therefore, we conduct a set of experiments to find the appropriate boundary threshold which

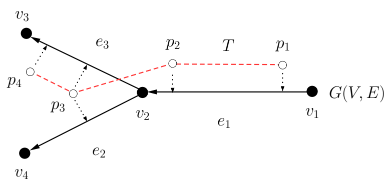


Fig. 1 Example of road network graph $G(V, E)$, with edges $e_{[1..3]}$ and vertexes $v_{[1..4]}$; and a GPS trajectory T (red dotted) with four coordinate points $p_{[1..4]}$ to be matched with the road network (Color figure online)

does not affect map-matching accuracy, yet reduce the number of replication, hence memory consumption.

5. *Experimental evaluation* Finally we provide an evaluation study on the accuracy of our approach, and the performance and scalability of spatial-aware map-matching using different spatial data structures on top of Spark, and comparing our work with a state-of-the-art technique. Our experiments demonstrate that our approach can achieve efficient and scalable map-matching processing.

The remainder of this paper is organized as follows. Section 2 introduces some background knowledge on map-matching and Spark. Section 2.3 introduces the problem statement. In Sect. 3 we discuss the related work. In Sects. 4 and 5 we introduce our proposed cost function and framework respectively. Finally, in Sects. 6 and 7we present out experimental result and conclusions.

2 Preliminaries

2.1 Map-matching algorithms

There are two main algorithmic approaches for map matching in the literature, Local [2,5,14,25], and Global [4,15–17,19]. In short, the three main steps followed by map-matching algorithms are: (1) identifying a set of candidate edges in the road graph within a given radius from the location point, then (2) calculating the weight for each candidate edge (e.g. shortest distance between the point and the edge), and finally (3) retrieving the edges that maximize the weight.

Local (or *incremental*) algorithms only consider the trajectory and the road network geometries to relate a trajectory point to its nearest edge (point-to-edge) in the road map. This method is simpler and faster, and more commonly used in on-line map matching, since they rely on the previous trajectory points observations only, which makes it more difficult to use statistical models on the trajectory topology. However, due to measurement errors and GPS inaccuracy, this approach is prone to error (i.e. point mismatch). Wei et al. [24] provided a comparison between local and global map-matching algorithms, and discovered that local algorithms performed poorly, specially due to Y-splits on road networks. For instance, in Fig. 1 while there are two possible matching candidates, e_2 and e_3, for point p_3, e_3 is the most obvious edge to match, since its next connecting point p_4 is better matched with e_3, and real moving objects are more likely to follow a direct path [16]. Therefore, the best match for trajectory T is the path $P = \{e_1, e_3\}$ connecting v_1 to v_3.

Global algorithms, on the other hand, take into account the geometry and other features of the trajectories and the road network, such as speed, topology, the connectivity between points and edges, and the road network speed limits, in order to find the best match of a trajectory on the road network, thus easing the uncertainty. Global algorithms are mostly used in offline map matching, and use future observations to better match the trajectories correctly. These methods make use of statistical models (e.g. Hidden Markov Model [17], and spatial-temporal analysis [16]), and sacrifice performance to achieve better accuracy. Offline map-matching plays a key role on trajectory pre-processing by improving data quality and reducing uncertainty when whole new trajectory data, or new and more accurate map data, became available.

2.2 Spark framework

Spark [32] framework provides a MapReduce (MR) [7] solution for low-latency data processing using distributed in-memory data storage. Spark is highly scalable and shows faster processing of large-scale data if compared to other MR frameworks like Hadoop [10]; the performance improvements come from avoiding disk I/O and the smaller cost on objects de-serialization by storing the data in main memory, as well as its hash-based aggregation [20,31]. Spark supports data partitioning and parallel computation through its resilient distributed data structure (i.e. RDD), and have been used in a handful number of data-intensive analytics, including spatial databases [27, 29]. However, despite designed for large spatial data, none of the current Spark-based systems support map-matching. Moreover, most of the existing Spark-based systems, such as [27,29], only deal with points and polygons queries; furthermore, those works do not focus on memory usage. In this paper we exploit the in-memory nature and distributed parallel properties of Spark for scalable offline map-matching. In addition, we employ a sampling-based partitioning using on a cost model to cover the Spark limitation on dynamic partitioning.

2.3 Problem statement

In this section we formally describe the problem of Map Matching, firstly introducing some background knowledge.

(**Trajectory**) A trajectory T of a moving object is a sequence of spatial-temporal points, where each point is described as a triple (x, y, t), where (x, y) are the spatial location of the moving object, such as its *latitude* and *longitude* coordinates, at a time t, that is, $T = [(x_1, y_1, t_1), (x_2, y_2, t_2), \ldots, (x_n, y_n, t_n)]$ in a two-dimensional space, where n is the number of sample points, and $t_1 < t_2 < \cdots < t_n$.

A trajectory describes the motion history of any kind of moving object, such as people, animals and natural phenomena. Trajectories of moving objects are continuous in nature, but captured and stored as a collection of spatial-temporal points by GPS devices. A discrete representation of trajectory is shown in Fig. 2.

(**Road network**) A road network is a directed graph $G(V, E)$ representing the digital map of streets and roads of a geographic region, where each edge $e \in E$

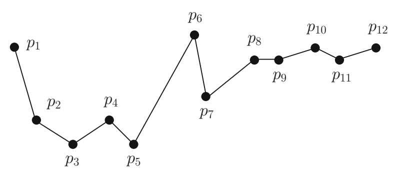


Fig. 2 Example of trajectory as a discrete sequence of spatial-temporal points

represents a road segment in the graph, and each vertex $v \in V$ of the graph represents the intersections and end-points of the road segments.

(**Road edge**) A road segment $e \in E$ is a directed edge from a starting vertex $v_i \in V$ to an ending vertex $v_j \in V$ in a road network graph $G(V, E)$, and associated with a list of intermediate points that describes the road polyline.

In digital map representation, both edges and vertexes in the road network graph are associates with an ID, and a set of semantic attributes, such as *speed* and *length*.

(**Road path**) A road path P is a set of connected road edges, $P = \{e_i, \ldots, e_j\} \in G(V, E)$, connecting two locations v_p to v_q of $G(V, E)$.

(**Map-matching**) Given a trajectory T, and a road network graph $G(V, E)$, map-matching is the problem of how to match T to a path P of $G(V, E)$.

In this work we focus on large scale map-matching.

Large-scale map-matching Given a large set of GPS trajectories $\mathbb{T}$, and large road network graph $G(V, E)$, our goal is to match every trajectory $T_i \in \mathbb{T}$ to a path P of $G(V, E)$ in an efficient, scalable, and memory-wise manner; that is, we want to maximize performance and scalability of large-scale map-matching, and minimize memory consumption at the same time.

We perform large-scale map-matching on top of Spark in order to achieve high performance and scalability. However, since Spark is an in-memory-based framework, performing any operation on top of Spark should be memory-wise, for instance, for map-matching both datasets may be too large to comfortably fit in the cluster memory. Furthermore, we should account for load balancing and communication cost, since a good data partitioning is a key point in distributed computation. Moreover, both trajectory and map data are difficult to fit into the Spark MapReduce computation model, since Spark does not natively support spatial data indexing and partitioning.

3 Related work

Related work on map-matching can be divided into two main categories: (1) Serial map-matching algorithms focusing on accuracy and match trajectories in a serial fashion; and (2) Parallel algorithms use spatial partitioning and parallel computation to speed up map-matching. We also briefly discuss related work on spatial data processing using Spark.

Serial algorithms Lou et al. [16] presented a spatial-temporal algorithm (ST-Matching) for matching trajectories with low sampling rates (e.g. 2 min gap between each point). Firstly, for every trajectory sample point $p_i \in T$ ST-Matching retrieves all candidate points c_j from the road network within a radius r from p_i—any candidate point farther from p_i than r is taken as a impossible match. From the candidates set they compute the projection of p_i on each edge containing c_j; the algorithm chooses the best match first choosing the edges which contain the nearest points from p_i (Spatial Analysis Function), then choosing the edge on which the speed limit is closer to the trajectory speed (Temporal Function). Newson et al. [17] proposed a Hidden Markov Model (HMM) based algorithm to find the most likely road to match a trajectory. The algorithm models the connectivity of the road segments into a HMM where each state is a road segment. The Emission Probability of every state transition is calculated using the Gaussian distribution, where the input is the distance from a trajectory location p_i to a road segment e_j, hence segments farther from p_i are assigned with a lower probability. Finally, the most likely matching road is found using Viterbi algorithm to compute the best path through the HMM lattice. Similar to ST-Matching, the HMM algorithm is robust to noise and temporal sparseness, and only considers matching edges within a 200 m radius from the trajectory point. OHMM [9] extends the HMM algorithm in [17] for incremental on-line map matching. OHMM uses Support Vector Machines (SVM) classifier instead Gaussian model to calculate the transition probabilities of the HMM lattice, thus learning the best match based on the current trajectory state. Similarly, [19] uses a Bayesian classifier, including the topological constraints of the road network, to calculate the transition probabilities in the HMM model for local map-matching. IF-Matching [11] uses information fusion to achieve more accurate map-matching. Along with the geometry and topology of the trajectories IF-Matching also uses speed and direction to better describe the moving object. It also uses the speed constraints of the road to find the best match, however, as the speed of the moving object on the road can be limited at different times of the day due to heavy traffic, the IF-Matching algorithm also applies a function to model the speed on the road network during different times of the day. In [4] the authors present an approach using the Frechet Distance to calculate the most similar road to match the trajectory; however, this method only takes into account the geometry of the trajectories, and it is limited to records with very dense points distribution and low sampling error. [13] and [15] introduce a new multi-track map-matching approach, by matching multiple trajectories to the road graph at same time. The idea is to identify the regular patterns of a group of trajectories in order to find their best match, assuming that all trajectories with same pattern belong to a same path in the road network. Although achieving high accuracy, the main drawback of these serial algorithms is that they focus only on the accuracy of the matching; processing one, or only a few, records at a time, in a single process unit, and do not account for scalability.

Parallel algorithms Huang et al. [12] presented the MR-based algorithm HOM for map-matching focus on performance rather than accuracy. HOM divides the data space into a grid, and assigns each GPS point and road link to its corresponding grid; each grid partition is sent to a computing node. Finally, map-matching is done in each partition in parallel using MR. HOM, however, used an incremental algorithm to match points to its nearest road link, and do not consider boundary objects, thus it is not robust to noise. Tiwari et al. [22] proposed a framework focus on scalability of map-matching. The framework uses MR computation and Hadoop HBase as distributed storage to achieve horizontal scalability. Similar to HOM, it uses grid partitioning and does not account for load balancing, however, it uses ST-Matching [16] on each partition to compute map-matching rather than provide a new algorithm. The main drawbacks of these MR-based approaches is that they do not consider homogeneous distribution during the partitioning, which is essential in MR computation, and do not account for boundary objects. Similarly, [6] propose a system on top of HBase for map-matching, the system combines the rowkey indexing of HBase with Geohash for spatial indexing; however, the systems does not consider load balancing; furthermore, although achieving good scalability by leveraging HBase, map-matching is computed using point-to-curve technique only, and the authors do not consider memory usage and in-memory storage for processing speed up. [23] provides a parallel approach for streaming map-matching in-memory for real-time processing using grid partitioning, however, the authors do not optimize memory consumption and do not handle boundary objects. [26] addresses the problem of load balancing using Quadtree space decomposition, similar to our approach the authors defined a 5km boundary threshold for replication, so that points near partition boundaries are replicated to all nearby partitions; however, it builds the quad-index using the entire dataset dynamically, which is not directly supported by Spark; our work, on the other hand, applies a sampling-based approach to reduce the partitioning cost using Spark. Furthermore, all the works presented to date use disk-based computation, which negatively affects performance for large-scale data due to I/O overhead.

Our work is similar to that in [26], except we use a much smaller space boundary limit, and add one more condition to we make sure our spatial-partitions are no larger than the MR block size in the cluster configuration (i.e. 64MB by default), so that every partition can be processed by a MR task, the MR block size can be adjusted according to the cluster configurations. Moreover, our partitioning is sample-based using a cost-function, we also load the data in batched to main memory using the quadtree partitions to reduce memory usage, which was not used in previous works since they do not focus on memory consumption. We also build our quad-index using a sampling-based approach which can be combined with Spark without the need to repartition each batch. Finally, we also investigate the impact of the boundary threshold parameter on both performance and correctness of existing map-matching algorithms, which was not covered by the related work.

Spatial data on Spark A number of existing works provide unified systems for spatial queries using Spark, aiming to achieve better performance and scalability for spatial data. Simba [27] uses Spark's RDD to support spatial indexing and spatial operations natively. Simba adds spatial keywords to SparkSQL grammar, so that users can express spatial operations in a SQL-like fashion. Similarly, GeoSpark [29], SpatialSpark [28], and SparkGIS [3] have been proposed to process spatial data on top of Spark. However, although achieving high performance and scalability for spatial queries, none of these systems fully support trajectory data processing nor map-matching. OceanST [30] does provide support for trajectory data on Spark, however, only for selection queries using uniform static data partitioning. Our proposed framework, on the other hand, provides support for map-matching using balanced space partitioning.

4 Map-matching workload

We provide an estimative of the workload for the distributed map-matching problem based on the following observations:

Execution time and partitioning In a nutshell, the baseline cost C_M of map-matching for a given matching algorithm of function f, can be estimated as the number of trajectory points m against the number of road edges n to match, i.e. $C_M = f(m*n)$, since for every trajectory point one must find the best match node. Notice that the function f refers to the map-matching algorithm employed; our framework was built to use state-of-the-art map-matching algorithms as "blackbox", however, every map-matching algorithm has it own computational cost/complexity based on the number of data records to process, which is expressed by the function f.

Nevertheless, since map-matching computation is intrinsically paralellizable, we can greatly decrease the computational cost C_M and improve scalability by co-partitioning the input map and trajectory dataset using some spatial partitioning method, and perform map-matching in each partition in parallel. Equation (1) depicts the estimate cost for spatial-aware map-matching C_M^P in parallel,

$$C_M^P = C_{index} + \frac{f(m*n)}{(B*U)} + C_{pos} \tag{1}$$

where B is the number of data partitions/blocks, and U is the number of processing units (supposing all units with same computational power).

C_{index} is the cost of building the spatial index model, and partitioning the input datasets. The cost of building the spatial index depends on both the index strategy employed (e.g. balanced or static index), and the number of data records used to build the index model. The data partitioning accounts for the cost to partition the entire datasets, both map and trajectory, using the spatial index model. For instance, in the case of our Spark framework, a quad-index is constructed in the master node from a sample S of the input trajectory dataset; the data partitioning, on the other hand, is done for the entire datasets in the Spark cluster. Therefore, in this scenario C_{index} is the cost to build a quad-index with B spatial partitions from S in the master node, plus the cost to partition the map and trajectory datasets into B spatial partitions in the Spark cluster.

Furthermore, there is a post-processing step to merge records from trajectories that have been split across multiple partitions; this step also adds a cost C_{pos} to the final workload. Notice that the cost C_{pos} depends on how we handle boundary records, as well as the partitioning granularity. For instance, Fig. 3 shows an example of space decomposition using Quadtree. If we decide to assign boundary crossing trajectories to all intersecting partitions (i.e. multiple assignments), then C_{pos} is simply the cost of choosing the best match from the result copies. If we decide, however, to split boundary trajectories into sub-trajectories according to their containing spatial partitions (i.e. single assignment), then C_{pos} is the cost of merging the resultant sub-paths at the end of the processing. For both strategies C_{pos} also depends on the partitioning granularity, for instance, in Fig. 3 increasing the partitioning granularity would either increase the number of replications for multiple assignments, or increase the number of splits for

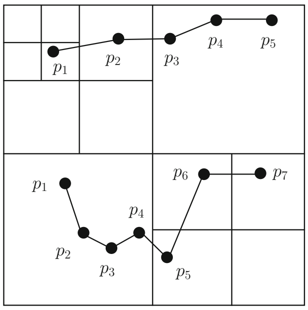


Fig. 3 Example of Quadtree space partitioning for trajectories

single assignment, thus increasing the post-processing cost. Overall, we can estimate C_{pos} on either the number of replicated trajectories on multiple assignment policy, or the number of sub-trajectories to merge in single assignment.

Load balancing and communication cost In Eq. (1) we suppose B as a set of disjoint and homogeneous spatial partitions. However, real life spatial datasets are not uniformly distributed, for instance, the density of data records in a city center is much larger than in the suburbs. In distributed parallel applications, a poorly partitioned dataset can lead to contention, and increase communication and data transfer between the computing nodes. Therefore, we must make sure we employ a partitioning strategy that takes the data distribution into account for better load balancing. Although dynamic partitioning structures, such as Quadtree and k-d Tree, have a higher partitioning cost C_{index} compared to static structures such as Grids, this cost is small compared to the gains in load balancing (see Sect. 6).

Furthermore, global map-matching algorithms use a distance threshold to select candidate edges/nodes for matching. Therefore we must ensure that records within the candidates threshold are assigned to the correct partition. Given that a matching algorithm uses a candidate's distance threshold of size β, if the distance from a record to the partition boundary is smaller than β this will cause the matching algorithm miss some candidates in adjacent partitions. The simplest solution is to replicate points and edges within a distance β from the partitions boundary; however, for regions with high density of boundary records, replication will negatively affect the computation cost by increasing the number of data records [(i.e. m and n in Eq. (1)] (see Sect. 6); moreover, for in-memory storage frameworks, such as Spark, replication will also increase memory usage. In Eq. (2) we estimate C_M^D the cost of distributed map-matching,

$$C_M^D = C_{index} + \frac{f((m + r_m) * (n + r_n))}{(B * U)} + C_{net} + C_{pos} \tag{2}$$

where r_m and r_n are respectively the total number of replication of trajectory points and road nodes, and C_{net} is the network cost of communications and data transfers between the nodes. In MR-based systems, such as Spark, C_{net} is basically the cost of the distributed shuffle operation to send data from *mappers* to *reducers* [7], and it is dependent of the network configurations, such as bandwidth, and both data locality and load balancing. For instance, in the MR model the slave nodes redistribute data based on the output keys (e.g. partition ID), such that all data belonging to one key (partition) is located on the same slave node, furthermore, MR always try to assign work to idle notes to best use the cluster resources, which means that a poorly distributed dataset would cause the slave nodes to shuffle more data, increasing networking cost. Therefore, for Spark map-matching C_{net} is highly dependent on the space partitioning strategy.

Bearing that in mind, our goal is to propose a framework for efficient and scalable map-matching in a distributed fashion, by reducing the cost C_M^D according to Eq. (2).

5 Map-matching framework

Our goal is to reduce the overall cost of distributed map-matching according to Eq. 2. In addition, we aim to reduce cluster memory consumption with Spark. The following steps summarize our proposed framework in Spark, as shown in Fig. 4. In the next sections we provide a detailed description of our framework.

1. *Sampling-based index construction* We select a small sample of the input trajectory dataset to build a quad-index in the master node. After that, the index is broadcast to the memory of all slave nodes.
2. *Data reading and partitioning* We read both map and trajectory datasets as a Spark's RDD and assign every trajectory segment and map edge to its intersecting partition using the quad-index, and accounting for boundary objects.
3. *Co-grouping* We co-group partitions containing both map edges and trajectory segment by spatial index into a single co-partition RDD.
4. *Map-matching computation* We perform map-matching in each RDD co-partition in a parallel fashion using Spark.
5. *Post-processing* A final post-processing step is performed to group the match results by trajectory key.

By building our quad-index from a sample of the input data we aim to reduce the C_{index} in Eq. 2, since it accounts for the cost of building the spatial index and partitioning the dataset afterward, and also allow this model to be used in Spark. By co-grouping both map and trajectory data into balanced partitions, we aim to reduce the cost of map-matching processing by increasing parallelization and reducing the communication cost C_{net}. By wisely replicating boundary segments, we aim to reduce both the number of replications r_m and r_n, and the post-processing cost C_{pos}, without affecting accuracy.

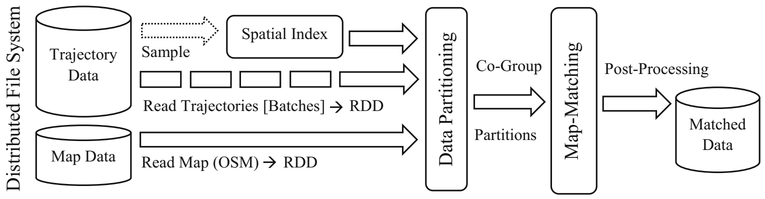


Fig. 4 Spark map-matching framework overview

5.1 Sampling-based space decomposition

Since we need to provide Spark with a data partitioning model, before it can load and partition both map and trajectory data, we select a sample of the input trajectory dataset to build a quad-index in the master node. We employ a Quadtree space decomposition for both map and trajectory records, for it provides a fairly uniform partitioning of spatial records.

Firstly, we must estimate the best number of spatial partitions in the quad-model, this can be calculated by taking the maximum between the number of processing units available in the cluster, and the dataset size to load over the Spark's RDD block size, as in the following Eq. (3):

$$N = \arg\max \left(\frac{|T| + |M|}{|RDD_{Block}|}, U \right) \tag{3}$$

where N is number of partitions, $|T|$ and $|M|$ are the trajectory and map datasets size respectively in bytes, and $|RDD_{Block}|$ is the RDD data block size (64MB by default), and U is the number of processing units (e.g. CPU cores).

Given the number of partitions N, we build our Quadtree model using Spark as follows: We select a sample of the dataset S of size $|S|$ to build the quad-index I by the driver program (i.e. master node). We decide to split a partition N_i in the Quadtree when the number of records r_i in the partition is $r_i \geq 4 * (|S|/N)$, this is to ensure that each partition will have roughly the same quantity of data record, since $|S|$ is the size of the input sample dataset, and N is the number of desired partitions, we want each partition to have $(|S|/N)$, since in quadtree partitioning a spatial partition is divided into 4 once it reaches a certain limit, we set this limit to $4 * (|S|/N)$. After its construction, the index model I is broadcast to the memory of all slave nodes.

5.2 Data partitioning

Given the quad-index model I, we partition the input datasets using MapReduce [32] on top of Spark's RDD, so that the number of data records in each partition is roughly uniform for load balancing. Furthermore, to allow parallel map-matching we must ensure that both map and trajectory records in a same spatial region are assigned to the same partition, so that there is no need to look for matching paths in other partitions. Therefore, we load both map and trajectory datasets into Spark's RDD in main-memory, and assign every trajectory segment and map edge to its intersecting spatial partition using I. Finally we co-group all records mapped to the same spatial partition into the same data block to be processed in parallel, as shown in Fig. 4.

Batch processing In order to reduce memory usage, we partition and match trajectories in batches. Once building the quad-index and partitioning the map data, the map-matching process is independent for every trajectory, that is, once a trajectory T_i is assigned to its intersecting partitions, we can match T_i independently of the remainder records. Therefore, to reduce cluster memory consumption, we split and load the trajectory dataset into RDD in smaller chunks. We load trajectories into our application in batches of roughly ($U * |RDD_{Block}| * \alpha$) in size. Where U is the number of processing units, so that we use all available cores, and α is the Spark's RDD in-memory storage fraction (0.8 by default). In this scenario we assume the cluster memory can comfortably fit the map data and at least one trajectory data batch in-memory.

For the best of our knowledge, no other related work has applied batch processing for map-matching. Furthermore, as in Spark batch processing is not spatially-aware, the data batches are partitions of the input dataset read from disk or memory. Using the proposed framework, however, we can control the partitions load to memory in each batch, and make sure that each data batch (both map and trajectory) contains the data in a same spatial region, hence the distributed processing is fully decentralized, thus no other regions need to be loaded/searched for candidate matches.

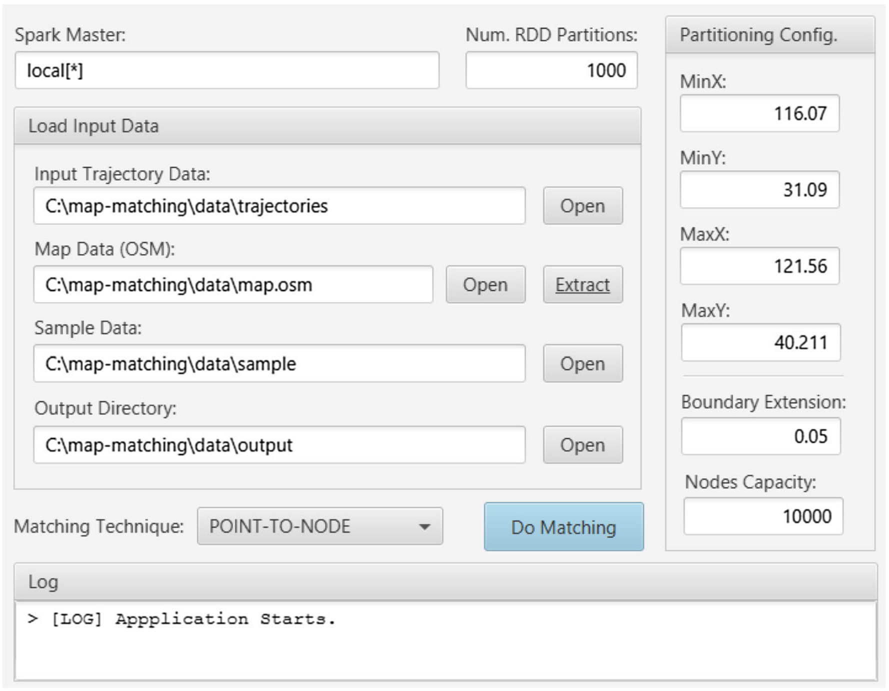
Boundary objects and replication When assigning data records to partitions, however, we expect some trajectories and road segments to overlap with more than one partition, in this case we split the segments according with its intersecting partitions. However, during the map matching computation some boundary points can be mismatched, thus we replicate both road segments and trajectory segments within a certain distance threshold from the partition boundary in order to reduce both replication and uncertainty. Our empirical study performed in Sect. 6 demonstrated that boundary extensions β greater than 500m did not affect the map-matching accuracy, therefore we employ a $\beta = 500$ m threshold in our spatial partitions.

In addition, the metadata about the whole trajectory (e.g. speed, length, sampling rate, etc.) is stored along with its sub-trajectories during the partitioning, to be used by global map matching algorithms when necessary.

5.3 Map-matching computation

Parallel matching using MapReduce Given a map-matching algorithm $M()$, each data partition N_i, containing both map and trajectory data, is sent to be processed in parallel in the Spark cluster using MapReduce with $M()$ as follows: For every sub-trajectory sub_j^T in N_i—where sub_j^T refers to the *j-th* sub-trajectory of a parent trajectory T—the $map()$ function outputs a $\langle key, value\rangle$ pair containing the parent-trajectory identifier T as *key*, and the path P_j^T that best matches sub_j^T with regards to $M()$ as *value*, that is, the mapper outputs $\langle T, P_j^T\rangle$. We employ the a nearest-neighbor and HMM map-matching algorithm in our framework; however, any map matching algorithm from the state-of-the-art can be used in our framework.

Post-processing The $reduce()$ function groups sub-paths by parent-trajectory key T (i.e. the post-processing C_{pos} step) and outputs the final results. The purpose of the post-processing step if to merge sub-paths of trajectories which might have been split and sent to separated spatial partitions due to their large extension. This post-processing step is not covered by related work which use spatial partitioning, they either assume the entire trajectory fits in the partition, or dont merge the sub-trajectories in the final step.



Spark Master: local[*]
Num. RDD Partitions: 1000

Input Trajectory Data: C:\map-matching\data\trajectories
Map Data (OSM): C:\map-matching\data\map.osm
Sample Data: C:\map-matching\data\sample
Output Directory: C:\map-matching\data\output

Matching Technique: POINT-TO-NODE

MinX: 116.07
MinY: 31.09
MaxX: 121.56
MaxY: 40.211
Boundary Extension: 0.05
Nodes Capacity: 10000

```
> [LOG] Appplication Starts.
```

Fig. 5 Application user interface

5.4 User interface

Additionally, we provide an easy-to-use user interface with our application, shown in Fig. 5. Users are able to setup the Spark and partitioning parameters, load data, extract OSM map data from the Internet, as well as choose the matching technique to apply. The application was built using a component-based design, thus it allows easy plug in of additional map-matching algorithms inside the framework. The application is available to download in the project repository.[1]

6 Experiments

We present a set of highlighted experiments on a real trajectory dataset to evaluate the performance, accuracy, memory usage, and scalability of our approach.

6.1 Experimental setup

We provide two different implementations to perform the experiment as follows:

[1] https://github.com/douglasapeixoto/map-matching-framework.

1. Firstly we implemented our proposed batch-based framework, where after loading and partitioning the entire map data, we load and partition the input trajectory dataset in batches to reduce distributed memory consumption.
2. In the second implementation, we bulk load and co-partition the entire datasets (map and trajectory) into main-memory to speed up map-matching at the cost of cluster memory usage.

We use a 54GB trajectory dataset collected throughout China, and a 6GB OSM map from the Chinese road network. The trajectory dataset contains around 22 million heterogeneous trajectories from taxis and personal vehicles in a period of 5 days; while the OSM data contains around 1.5 million nodes. The data is initially stored in HDFS, we use the default MR block size of 64MB.

All algorithms are implemented in the Spark Java library version 2.0.1. Experiments are conducted on a cluster with 16 physical nodes (1 master and 15 slaves). Each node has eight cores and 64GB of memory—in our experiments we configured Spark to use 7 cores and 60GB of memory in each slave node.

We evaluate our framework's performance and scalability by varying both the dataset size and the number of nodes in the cluster. We evaluate our work using three different adaptive spatial partitioning structures, i.e. Quatree, k-d Tree, and STR-Tree, and the HOM method proposed in [12] for MapReduce using grid partitioning. We adapt our framework for all aforementioned spatial structures using Spark. Adaptive spatial indexes were built using one million sample trajectories. Figure 6 illustrates the spatial structures used in our evaluation, including the spatial boundary extension β as described in Sect. 5.2. We evaluate our framework by employing the incremental nearest-neighbor matching algorithm, and the global HMM algorithm [17], which demonstrated better accuracy over sparse and noisy trajectory datasets.

6.2 Study of parameters

In this section we study the effect of the number of partitions, as well as the boundary threshold size β, on the number of data replications, and in the overall performance. In this experiment we selected 1000 trajectories with high density of sample points (i.e. average sampling rate of one second, and a minimum of 100 points per trajectory), the total number of sample points is 320,000. Table 1 demonstrates the number of records and the map-matching accuracy as β grows. We fixed the number of spatial partitions

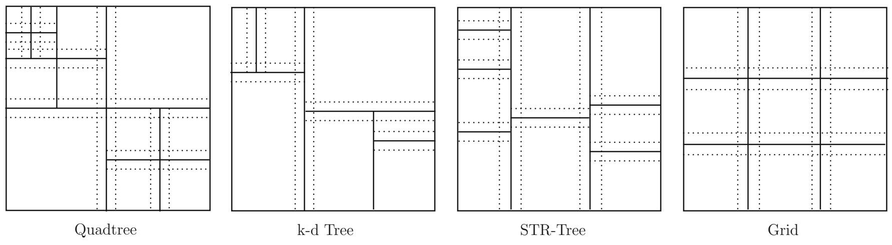


Fig. 6 Spatial partitioning structures used in the comparative study, with their respective boundary extensions (dotted lines)

Table 1 Effect of the boundary extension threshold (in meters) on data replication and map-matching accuracy, with $N = 1000$

Boundary (β) (m)	# of records	Iterative (%)	HMM (%)
0	320,000	94.5	86.9
100	326,785	97.4	92.5
200	334,596	98.8	96.3
300	335,598	99.7	98.4
500	336,714	100.0	100.0

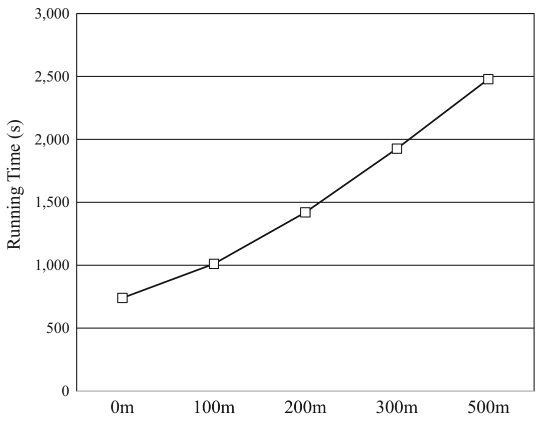


Fig. 7 Running time varying the boundary extension threshold. Using 54GB trajectory data and 6GB OSM data

N to 1000 in this experiment. We evaluate the accuracy by comparing the results of our distributed framework with the original implementation of two map-matching algorithms, iterative and HMM, running in a single machine in a greedy manner (i.e. without parallelization or partitioning). From the results in Table 1 we use a boundary threshold of $\beta = 500$ m in our framework.

Boundary replication is necessary to reduce uncertainty, as demonstrated in Table 1, however, replication will also increase memory usage and the computation cost C_M^D, by increasing the number of data records to match, as given in Eq. 2. Figure 7 show the overall running time of the framework varying the boundary threshold, this experiment was performed using the entire map and trajectory datasets. As expected, the overall running time is proportional to the number of records in the partitions, which increases with higher boundary values due to replication, and also increased the cost of the post-processing phase to find the best match in the duplicated records.

Similarly, the spatial partitioning granularity (i.e. number of spatial partitions N) will affect the number of replications, as shown in Table 2 (with boundary extension fixed in $\beta = 500$ m). Even though a large number of partitions N can reduce the overall cost of a distributed map-matching C_M^D, as given in Eq. 2, it will also increase the number of replications r_m and r_n. This is due to the increasing number of boundary point in partitions of small granularity. Therefore, in our framework we choose the

Table 2 Effect of the number of partitions on data replication

# of partitions (N)	# of records
1000	336,714
2000	401,430
5000	581,491
10,000	914,012

Partitions with a boundary threshold of 500 m

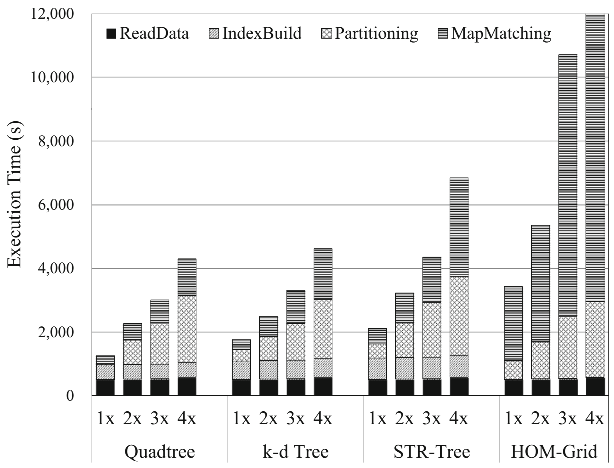


Fig. 8 Spatial-aware map-matching execution time comparison (in seconds) on Spark, using multiple spatial partitioning methods as the dataset grows

number of spatial partitions N according to Eq. 3 based on the Spark data block size and the available cluster resources.

6.3 Performance and scalability study

Dataset size Figure 8 compares the running time for each phase of the map-matching computation as we increase the dataset size. We use one to four times the input trajectory dataset to evaluate scalability. The partitioning phase accounts for the C_{index} cost of the quad-index model construction, and Spark data partitioning with all the dataset stored in-memory.

The balanced spatial structures (i.e. Quadtree, k-d Tree, and STR-Tree) had a better overall performance due to their more homogeneous distribution of the spatial data across the partitions, thus providing better load-balancing, which plays a key role in Spark performance. Uniform Grid, on the other hand, performed poorly executing over 17k seconds for 4x the dataset, this is mainly due to high density of data in some spatial partitions as the dataset grows, this lead to contention in some processing units, as well as an increase in cluster communication and shuffle, i.e. data transfer cost C_{net}.

In Fig. 8, *ReadData* accounts for the time to read both trajectory and map data from HDFS into Spark. We noticed that, even though the size of the map dataset is smaller than the trajectory, the time to read and parse the OSM data to our application was considerably higher using the Spark XML library, thus, the time taken to read additional copies of the input trajectory dataset was not significant to increase the overall read time.

The *IndexBuild* accounts for the time taken to read a sample of the trajectory dataset from the HDFS, and build the dynamic spatial index in the master node. Indexes were built with the same number of sample trajectories in all experiments. Even though a static grid can be constructed in basically zero time, our experiments demonstrated that using sample-based index construction was sufficient to improve performance compared to static Grid with no sampling.

The *Partitioning* accounts for the time taken to co-partition both map and trajectory data with regards to the spatial index. Boundary objects handling is also performed in this phase. As observed, the co-partitioning was the most demanding phase for balanced spatial structures, however, this cost was compensated by the gains in map-matching performance. The poor data distribution on static Grid, however, resulted in a much higher map-matching cost.

Finally, the *MapMatching* phase accounts for the execution time to process map-matching algorithm in the co-partitions, as well as perform the post-processing phase to merge resulting sub-paths by trajectory. Overall, our Quadtree based approach demonstrated better tread-off between the partitioning and the map-matching phases.

Batch processing Figure 9 shows the average execution time of map-matching in batches as the dataset grows. Based on our cluster configurations, i.e. $U = 105$ cores, $|RDD_{Block}| = 64$MB, we split our trajectory dataset into ten batches of roughly 5.4GB. Each batch is loaded into our framework and processed by Spark in FIFO mode. We could optimize the process by overlapping the batches computation to a full use of the cluster resources, however, our goal with this approach is to reduce memory storage usage, therefore we do not overlap neither the batches loading nor the batches computation. Again the dynamic spatial models demonstrated better performance over uniform Grid partitioning. In this scenario, our Quadtree method outperformed the remainder methods showing a slight linear increase as the dataset grows.

Figure 10 compares the execution time between the two approaches to process the entire dataset. Although storing all the data in-memory had a better performance gain for most approaches—due to a better process and resources allocation by Spark—batch loading has a better trade-off between performance and memory usage.

Figure 11 shows the memory consumption using batch loading for each individual batch, against storing the entire trajectory dataset in-memory. As in the previous experiment, the trajectory dataset was split into ten batches of roughly 5.4GB each. The batch approach has an overall gain of 5.2× in memory consumption against storing the entire dataset in memory; then umber of batches, however, can be adjusted to fit the available resources. One would expect a gain close to number of batches (i.e. 10×), however, this was not achieved mainly due to the map data being entirely stored in-memory in our framework.

Number of nodes Similarly, Figures 12 and 13 depict the results when we vary the number of slaves nodes in the cluster. We compare the previous methods using one copy of the dataset. We use 5, 10, and 15 slave nodes respectively to evaluate the effect of the number of nodes on each method's execution time.

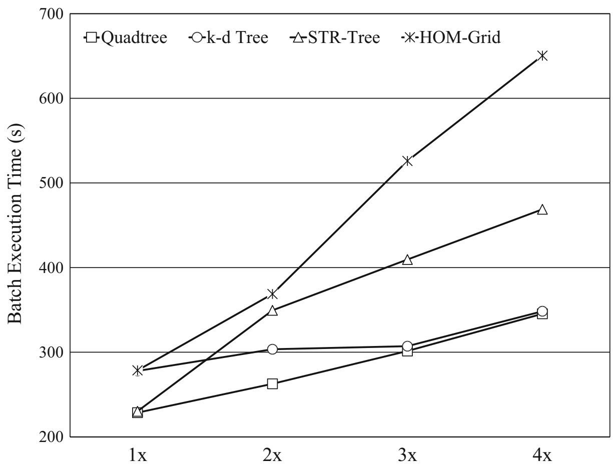


Fig. 9 Average batch processing time comparison (in seconds) by multiple spatial partitioning methods as the dataset grows. Execution time accounts for the average time to read, partition, and process map-matching on each data batch using Spark

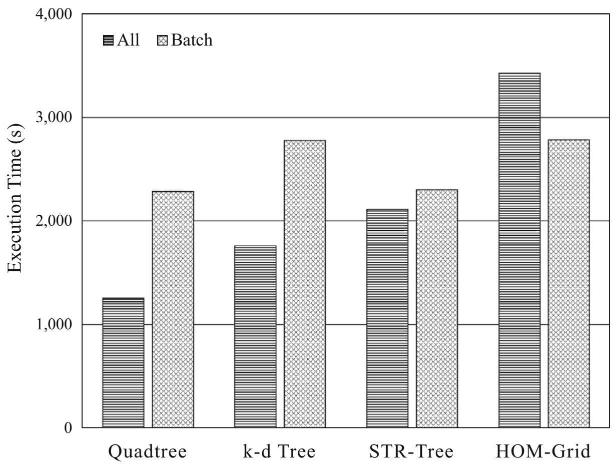


Fig. 10 Batch loading versus all dataset loading, performance comparison

As in the previous experiments, the balanced spatial structures performed better in all scenarios, with our Quadtree-based method performing better in all phases as show in Fig. 12. Also for batch processing, Fig. 13, our Quadtree method had the best performance gains and scalability, demonstrating near linear performance improvement as the number of nodes increases.

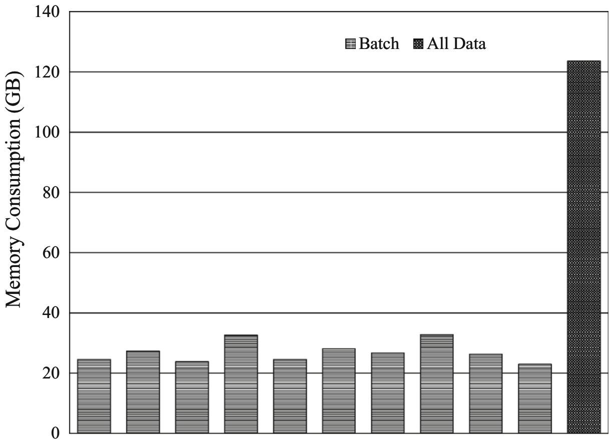


Fig. 11 Memory consumption comparison (in GB), using batch loading (individual batches) and all-data loading

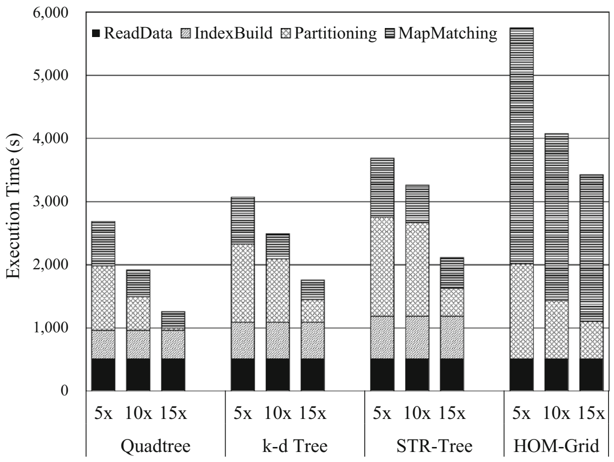


Fig. 12 Spatial-aware map-matching execution time comparison (in seconds) on Spark, using multiple spatial partitioning methods by increasing the number of computing nodes

7 Conclusions

Map-matching is an important pre-processing step to improve trajectory data quality and reduce uncertainty, due to inaccuracy of raw GPS data. The large amount of digital data available, however, has introduced a new problem of how to match massive amounts of both map and trajectory data in an efficient manner. In this work we introduced a Spark-based framework for the problem of large-scale offline map-matching. We introduced new features on top of Spark to allow efficient, scalable, and memory-wise processing of large-scale map-matching. First, we introduced a cost

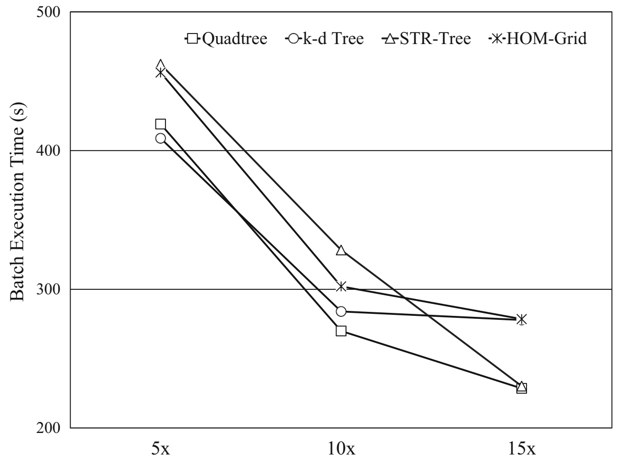


Fig. 13 Average batch processing time comparison (in seconds) by number of nodes. Execution time accounts for the average time to read, partition, and process map-matching on each data batch using Spark

function for the distributed map-matching problem. Secondly, we use a sample-based quad-index construction, and Quadtree co-partition of map and trajectory data to allow parallel and load-balanced map-matching. We build our partitions on top of Spark's RDD to achieve efficiency and scalability. We employ a safe boundary threshold, and wise split strategy to reduce replication. Finally we proposed a batch-based method for large-scale map-matching, using data loading and processing in smaller batches to reduce memory usage. A comparative study and experiments demonstrate that our framework achieved good efficiency and scalability on map-matching processing with lower memory consumption. As a future work, we aim to extend our framework for on-line streams map-matching using Spark streaming.

Acknowledgements This research is partially supported by the Brazilian National Council for Scientific and Technological Development (CNPq).

References

1. Aji, A., Wang, F., Vo, H., Lee, R., Liu, Q., Zhang, X., Saltz, J.: Hadoop-gis: a high performance spatial data warehousing system over mapreduce. VLDB **6**, 1009–1020 (2013)
2. Alt, H., Efrat, A., Rote, G., Wenk, C.: Matching planar maps. In: ACM-SIAM Symposium on Discrete Algorithms, pp. 589–598. Society for Industrial and Applied Mathematics (2003)
3. Baig, F., Mehrotra, M., Vo, H., Wang, F., Saltz, J., Kurc, T.: Sparkgis: Efficient comparison and evaluation of algorithm results in tissue image analysis studies. In: VLDB Workshop on Big Graphs Online Querying, pp. 134–146. Springer, New York (2016)
4. Brakatsoulas, S., Pfoser, D., Salas, R., Wenk, C.: On map-matching vehicle tracking data. In: VLDB, pp. 853–864. VLDB Endowment (2005)
5. Chawathe, S.S.: Segment-based map matching. In: IEEE Intelligent Vehicles Symposium, pp. 1190–1197. IEEE (2007)
6. Cho, W., Choi, E.: A GPS trajectory map-matching mechanism with DTG big data on the hbase system. In: Proceedings of the 2015 International Conference on Big Data Applications and Services, pp. 22–29. ACM (2015)

7. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. Commun. ACM **51**(1), 107–113 (2008)
8. Eldawy, A., Mokbel, M.F.: Spatialhadoop: a mapreduce framework for spatial data. In: ICDE, pp. 1352–1363 (2015)
9. Goh, C.Y., Dauwels, J., Mitrovic, N., Asif, M., Oran, A., Jaillet, P.: Online map-matching based on hidden markov model for real-time traffic sensing applications. In: International Conference on Intelligent Transportation Systems (ITSC), pp. 776–781. IEEE (2012)
10. Hadoop: https://hadoop.apache.org/
11. Hu, G., Shao, J., Liu, F., Wang, Y., Shen, H.T.: If-matching: towards accurate map-matching with information fusion. TKDE **29**(1), 114–127 (2017)
12. Huang, J., Qiao, S., Yu, H., Qie, J., Liu, C.: Parallel map matching on massive vehicle GPS data using mapreduce. In: International Conference on Embedded and Ubiquitous Computing, & International Conference on High Performance Computing and Communications, pp. 1498–1503. IEEE (2013)
13. Javanmard, A., Haridasan, M., Zhang, L.: Multi-track map matching. In: SIGSPATIAL, pp. 394–397. ACM (2012)
14. Kim, S., Kim, J.H.: Adaptive fuzzy-network-based c-measure map-matching algorithm for car navigation system. IEEE Trans. Ind. Electron. **48**(2), 432–441 (2001)
15. Li, Y., Huang, Q., Kerber, M., Zhang, L., Guibas, L.: Large-scale joint map matching of GPS traces. In: SIGSPATIAL, pp. 214–223. ACM (2013)
16. Lou, Y., Zhang, C., Zheng, Y., Xie, X., Wang, W., Huang, Y.: Map-matching for low-sampling-rate GPS trajectories. In: SIGSPATIAL, pp. 352–361. ACM (2009)
17. Newson, P., Krumm, J.: Hidden markov map matching through noise and sparseness. In: SIGSPATIAL, pp. 336–343. ACM (2009)
18. OpenStreetMap: https://www.openstreetmap.org/
19. Pink, O., Hummel, B.: A statistical approach to map matching using road network geometry, topology and vehicular motion constraints. In: International Conference on Intelligent Transportation Systems (ITSC), pp. 862–867. IEEE (2008)
20. Shi, J., Qiu, Y., Minhas, U.F., Jiao, L., Wang, C., Reinwald, B., Özcan, F.: Clash of the titans: Mapreduce vs. spark for large scale data analytics. In: VLDB, pp. 2110–2121 (2015)
21. Tang, Y., Zhu, A.D., Xiao, X.: An efficient algorithm for mapping vehicle trajectories onto road networks. In: SIGSPATIAL, pp. 601–604. ACM (2012)
22. Tiwari, V.S., Arya, A., Chaturvedi, S.: Framework for horizontal scaling of map matching: using map-reduce. In: International Conference on Information Technology, pp. 30–34. IEEE (2014)
23. Wang, H., Li, J., Hou, Z., Fang, R., Mei, W., Huang, J.: Research on parallelized real-time map matching algorithm for massive GPS data. Clust. Comput. **20**(2), 1123–1134 (2017)
24. Wei, H., Wang, Y., Forman, G., Zhu, Y., Guan, H.: Fast Viterbi map matching with tunable weight functions. In: SIGSPATIAL, pp. 613–616. ACM (2012)
25. Wenk, C., Salas, R., Pfoser, D.: Addressing the need for map-matching speed: Localizing global curve-matching algorithms. In: International Conference on Scientific and Statistical Database Management (SSDBM), pp. 379–388. IEEE (2006)
26. Xia, Y., Liu, Y., Ye, Z., Wu, W., Zhu, M.: Quadtree-based domain decomposition for parallel map-matching on gps data. In: International Conference on Intelligent Transportation Systems (ITSC), pp. 808–813. IEEE (2012)
27. Xie, D., Li, F., Yao, B., Li, G., Zhou, L., Guo, M.: Simba: Efficient in-memory spatial analytics. In: SIGMOD. ACM (2016)
28. You, S., Zhang, J., Gruenwald, L.: Large-scale spatial join query processing in cloud. In: ICDE Workshops, pp. 34–41. IEEE (2015)
29. Yu, J., Wu, J., Sarwat, M.: Geospark: A cluster computing framework for processing large-scale spatial data. In: SIGSPATIAL, p. 70. ACM (2015)
30. Yuan, M., Deng, K., Zeng, J., Li, Y., Ni, B., He, X., Wang, F., Dai, W., Yang, Q.: Oceanst: a distributed analytic system for large-scale spatiotemporal mobile broadband data. VLDB **7**(13), 1561–1564 (2014)
31. Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M.J., Shenker, S., Stoica, I.: Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In: USENIX Conference on Networked Systems Design and Implementation, p. 2 (2012)
32. Zaharia, M., Chowdhury, M., Franklin, M.J., Shenker, S., Stoica, I.: Spark: cluster computing with working sets. In: USENIX Conference on Hot Topics in Cloud Computing, p. 10 (2010)

33. Zheng, K., Zheng, Y., Xie, X., Zhou, X.: Reducing uncertainty of low-sampling-rate trajectories. In: ICDE, pp. 1144–1155. IEEE (2012)

Affiliations

Douglas Alves Peixoto[1] · Hung Quoc Viet Nguyen[2] · Bolong Zheng[1] · Xiaofang Zhou[1]

Hung Quoc Viet Nguyen
quocviethung1@gmail.com

Bolong Zheng
b.zheng@uq.edu.au

Xiaofang Zhou
zxf@uq.edu.au

1 School of Information Technology and Electrical Engineering, The University of Queensland, Brisbane, Australia

2 School of Information and Communication Technology, Griffith University, Gold Coast, Australia