GeoMatch: Efficient Large-scale Map Matching on Apache Spark

AYMAN ZEIDAN, Department of Computer Science, CUNY Graduate Center, USA
EEMIL LAGERSPETZ, Department of Computer Science, University of Helsinki, Finland
KAI ZHAO, Robinson College of Business, Georgia State University, USA
PETTERI NURMI and SASU TARKOMA, Department of Computer Science, University of Helsinki, Finland
HUY T. VO, Department of Computer Science, CUNY City College, USA

We develop GeoMatch as a novel, scalable, and efficient big-data pipeline for large-scale map matching on Apache Spark. GeoMatch improves existing spatial big-data solutions by utilizing a novel spatial partitioning scheme inspired by Hilbert space-filling curves. Thanks to its partitioning scheme, GeoMatch can effectively balance operations across different processing units and achieve significant performance gains. GeoMatch also incorporates a dynamically adjustable error-correction technique that provides robustness against positioning errors. We demonstrate the effectiveness of GeoMatch through rigorous and extensive empirical benchmarks that consider large-scale urban spatial datasets ranging from 166,253 to 3.78B location measurements. We separately assess execution performance and accuracy of map matching and develop a benchmark framework for evaluating large-scale map matching. Results of our evaluation show up to 27.25-fold performance improvements compared to previous works while achieving better processing accuracy than current solutions. We also showcase the practical potential of GeoMatch with two urban management applications. GeoMatch and our benchmark framework are open-source.

CCS Concepts: • **Computing methodologies → MapReduce algorithms**;

Additional Key Words and Phrases: Big data, spatial data analysis, spatial partitioning, performance, query processing, apache spark

ACM Reference format:
Ayman Zeidan, Eemil Lagerspetz, Kai Zhao, Petteri Nurmi, Sasu Tarkoma, and Huy T. Vo. 2020. GeoMatch: Efficient Large-scale Map Matching on Apache Spark. *ACM/IMS Trans. Data Sci.* 1, 3, Article 21 (September 2020), 30 pages.
https://doi.org/10.1145/3402904

This work was supported in part by the Alfred P. Sloan Foundation G-2018-11069, Academy of Finland—grant numbers 297741 and 324576, National Science Foundation Award 1827505, and Vingroup Innovation Award VINIF.2019.20.
Authors' addresses: A. Zeidan, Department of Computer Science, CUNY Graduate Center, New York, NY; email: azeidan@gradcenter.cuny.edu; E. Lagerspetz and S. Tarkoma, Department of Computer Science, University of Helsinki, Helsinki, Finland; emails: {lagerspe, starkoma}@cs.helsinki.fi; K. Zhao, Robinson College of Business, Georgia State University, Atlanta Georgia, GA; email: kzhao4@gsu.edu; P. Nurmi, Department of Computer Science, University of Helsinki, Finland, Helsinki, Finland; email: petteri.nurmi@cs.helsinki.fi; H. T. Vo, Department of Computer Science, CUNY City College, New York, NY; email: hvo@cs.ccny.cuny.edu.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2020 Association for Computing Machinery.
2577-3224/2020/09-ART21 $15.00
https://doi.org/10.1145/3402904

1 INTRODUCTION

The availability of urban location data has grown exponentially, thanks to widespread penetration of location technologies to cars, buses, trains, and other means of transportation. Indeed, datasets containing millions or even billions of location measurements collected from taxicabs,[1] buses,[2] and fleet management are becoming available. This growth of spatial data is opening up unprecedented opportunities to analyze and understand mobility and how it relates to urban infrastructure. For example, research has shown how these data can be used to characterize urban mobility patterns, detect transportation bottlenecks, and to optimize transportation infrastructure [13, 18, 35, 39]. Besides academic interest, this increased scale of spatial data has significant commercial potential, with the market value of spatial big-data industry expected to reach $440B by 2020 [27] and double-digit annual revenue through 2022 [15].

Map matching is a central processing task in practically all analyses of urban location data, as otherwise the findings cannot be related to urban infrastructure. Indeed, individual location measurements often are noisy, making it difficult to align them with streets or other infrastructure. Figure 1 illustrates this problem for a busy intersection (49th Street Subway Station) in Midtown Manhattan, New York City. The plot contains location coordinates from taxis that have been overlaid on top of a map of the area. As the figure shows, the measurements are spread around the surrounding area, making it difficult to determine which exact routes were followed. Most map matching techniques operate in two phases [4, 26, 35]. In the first phase, points are matched against road segments and scored according to criteria such as how far the point is from the segment and how popular the road is. In the second phase, the candidate matches are combined to estimate the entire trajectory, taking into account factors such as continuity of measurements.

Given the increased volume of spatial datasets, map matching increasingly needs to be performed on a big-data framework. This is particularly true for the first phase of map matching, where each measurement needs to be compared against each road segment unless effective indexing and parallelization is available. Unfortunately, common big-data frameworks, such as Spark and Hadoop, are ill-suited for spatial processing, as they do not offer native support for spatial structures or operations. While several spatial extensions have been proposed (see Section 2.1), the main focus of these solutions has been on enabling common spatial operations instead of delivering efficient performance. As a result, their performance is unacceptably slow and memory requirements impractical for large datasets. The main reason for the poor performance is the sub-optimal partitioning of measurements, as current solutions mainly rely on random sampling of the data and excessive caching. Partitioning is therefore sensitive to how the measurements are organized into files and to the spatial distribution of measurements. Figure 2 illustrates this problem by showing the distribution of measurements for one of the datasets considered in our experiments. Since the majority of measurements are concentrated along a few small areas, the resulting partitioning is unbalanced, and a small number of cluster nodes perform most of the processing instead of having evenly distributed processing across the nodes. Another challenge related to partitioning is that, as illustrated in Figure 1, positioning errors cause points to fall into neighboring partitions, resulting in either computational overhead when attempting to account for accuracy, or in inaccurate results if these errors are not accounted for (e.g., points are not replicated to multiple partitions).

In this article, we contribute by developing *GeoMatch* as a novel distributed map matching extension for Apache Spark. GeoMatch has been designed to significantly improve large-scale map matching performance and to offer improved support for spatial objects and operations. GeoMatch

[1]NYC Taxi dataset contains over 3.7B data points.
[2]NYC Bus dataset contains over 216M data points.

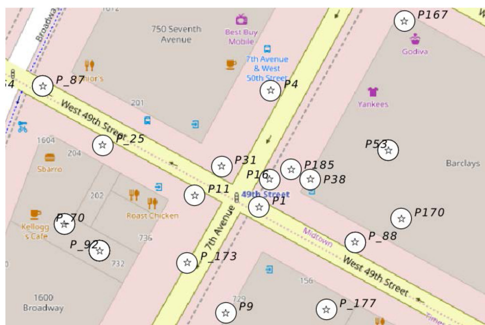


Fig. 1. Taxi pings around NYC's 7th Ave. and West 49th St. in Midtown Manhattan.

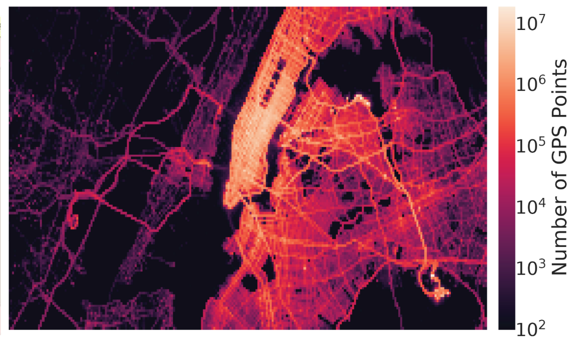


Fig. 2. Sample distribution for a dataset containing taxi trajectories collected from NYC (logarithmic scale).

achieves its performance improvements through an efficient and intuitive load-balancing scheme that evenly distributes parts of the spatial index between available computing cores. At the core of the load-balancing scheme is a locality preserving partitioning that has been inspired by Hilbert space-filling curves and their use for spatial indexing [16]. Unlike existing frameworks that use a small sample of the overall dataset for determining how to partition measurements, GeoMatch creates a locality preserving partitioning that allows more balanced and intuitive distribution across cluster nodes while also performing more efficiently than existing frameworks. To cope with inaccuracies resulting from positioning errors, GeoMatch integrates an error-correction mechanism that allows dynamically adjusting the size of the region that is used in map matching. This step helps to ensure each partition covers all spatial objects located within proximity of the partition's points. For example, areas around the start and endpoints of a street (or line segment) would be extended so they can be considered when matching points within neighboring partitions against the road network. Finally, GeoMatch offers improved support for spatial objects and file formats, natively supporting map matching from mixed-type datasets. As we experimentally demonstrate, these steps allow GeoMatch to achieve significant performance, robustness, and accuracy improvements compared to existing spatial big-data frameworks.

We demonstrate effectiveness of GeoMatch through rigorous and extensive map matching benchmarks using three datasets that range from 166,253 to 3.78B measurements. We develop a comprehensive evaluation benchmark setup for large-scale map matching and compare GeoMatch against five popular Spark spatial extensions (GeoSpark, LocationSpark, STARK, Magellan, and Simba). The results of our experiments demonstrate that GeoMatch is capable of achieving up to 27.25 folds faster runtime performance, more stable scalability, and better spatial object precise support. Moreover, for our largest dataset, the other studied frameworks struggle significantly due to the size of the dataset achieving slow performance and even running out of memory. Finally, we demonstrate that the indexing scheme used by GeoMatch results in over 4-fold improvements in map matching performance and overall accuracy of 99.66% as compared to the baseline method.

To demonstrate the practical benefits of GeoMatch, we present two real-world urban management problems, Unmet Taxi Demand Inference, and Bus Service Reliability Analysis, which require efficient large-scale map matching that GeoMatch is currently addressing. These applications were selected from ongoing collaborations with New York City agencies: Taxi and Limousine Commission (TLC), Department of Transportation (DOT), and TransitCenter (a non-profit foundation),[3]

[3]https://transitcenter.org.

which involved interdisciplinary efforts from computer scientists, urban planners, transportation engineers, and data scientists. Unmet taxi demand inference problem can be described as the instance when a passenger requires a taxi but is unable to find one within five minutes. Bus service reliability analysis problem, however, relates to the excess waiting time for passengers due to delays. In both applications, differences between suburban and downtown areas are pronounced, requiring mechanisms that can efficiently process measurements from areas with varying densities.

Summary of Contributions

(1) **Efficient Large-Scale Map Matching Framework:** We develop GeoMatch as an Apache Spark-based map-matching pipeline for map matching large-scale spatial datasets. GeoMatch is scalable due to its efficient indexing and partitioning schemes. It integrates a load-balancing mechanism that improves the robustness of map matching operations and enables it to achieve highly stable runtimes across experiments. Moreover, GeoMatch can operate independent of input formats, making it easy to integrate into real-world analysis pipelines. We present two case studies of such applications, *Unmet Taxi Demand* and *Bus Service Reliability Analysis*.
(2) **Benchmark Evaluations:** We conduct an extensive benchmark of the speed and accuracy of existing large-scale spatial data processing frameworks. Our experiments, carried out on three large-scale datasets, demonstrate that GeoMatch is 2.41×–27.25× faster than current works in large-scale map matching and achieves 99.99% accuracy and over 4-fold improvement in processing speed compared to exhaustive search.
(3) **Novel Benchmarking Framework:** For conducting our evaluation, we develop a principled framework for assessing the performance of large-scale map matching systems. We have open-sourced our framework as well as our benchmarking setup on GitHub: https://github.com/bdilab/GeoMatch.
(4) **Improved Robustness and Accuracy:** GeoMatch integrates a novel error-correction technique that enables extending matching regions around an object's minimum bounding rectangle (MBR). Our experiments demonstrate that this extension allows us to mitigate GPS reflections and other fluctuations, yielding more accurate and robust results than existing competing frameworks.

The present article extends our earlier work [37]. We have extended our work by (i) further improving the performance of GeoMatch; (ii) developing an error-correction technique for handling points along the edges of partitions; (iii) developing a performance and accuracy benchmark framework; (iv) extending our benchmark evaluations by incorporating more error criteria and further frameworks; (v) releasing both our framework and benchmark framework as open-source; and (vi) applying GeoMatch to two practical examples of real-world urban management problems.

2 SPATIAL BIG-DATA PROCESSING

Spatial big-data processing frameworks extend generic frameworks like Apache Hadoop or Apache Spark by including support for spatial data structures and operations. We next briefly survey existing spatial big-data processing frameworks and compare their features against GeoMatch. We also survey existing benchmark methods for large-scale map matching.

2.1 Existing Spatial Big-Data Processing Frameworks

Hadoop-based frameworks focus on making MapReduce tasks spatially aware. However, they inherit Hadoop's fault-tolerance limitation and must write intermediate results to disk, which

results in significantly slower performance than Spark-based solutions. Esri GIS Tools for Hadoop [14] is a set of Hive User Defined Functions that are mainly released as a utility to extend the functionality of Esri's ArcGIS mapping software to include support of Well-Known Text (WKT) files stored on Hadoop. Hadoop-GIS [3] is built on top of Hadoop and adopts a streaming approach. It extends Hive to offer support for spatial objects and operations and translate queries into spatially capable MapReduce tasks. SpatialHadoop [9] shares some similarities with Hadoop-GIS but offers tighter integration with Hadoop via the use of lower-level APIs, has fewer interactions with HDFS than Hadoop-GIS, and its MapReduce tasks are more spatially aware, since they operate on data as spatial objects from the start. SATO [31] is a generic solution for optimal spatial partitioning on MapReduce systems with the main objective of targeting the spatial partitioning problem, which causes query skews. It can be used as a standalone program but it has been integrated into Hadoop-GIS. Unlike GeoMatch, SATO relies on a subsample of the original dataset for determining the partitioning, which makes its performance sensitive to skews in the data.

Spark-based frameworks rely on Spark's Resilient Distributed dataset (RDD), which is the core technology of Spark that solved two major Hadoop drawbacks—the limited in-memory processing and the need to write intermediate results to disk to achieve fault-tolerance. These advantages enable Spark-based solutions to execute faster than MapReduce-based solutions, making Spark the preferred option for large-scale map matching. As an example of Spark-based frameworks, SpatialSpark [33] offers two modes of operation: broadcast spatial join, which is ideal for use with one small dataset and one large dataset, and partitioned spatial join, which is ideal for two large datasets. Simba [32] allows spatial operations using Spark SQL or DataFrames and represents its datasets as tables. SQL queries are optimized using a Cost-Based Optimizer (CBO) to produce an optimal parallel execution plan. Simba, however, focuses on multidimensional queries by indexing each dimension separately, ultimately increasing query's complexity. STARK [11] is a spatio-temporal framework that aims to optimize queries for datasets with spatial and temporal components. The temporal component is currently not taken into consideration during partitioning, and STARK can only build spatial global indexes. LocationSpark's [29] main objective is targeting the query skew problem. It offers its own Spark integration through spatially aware RDDs. Magellan [28] extends Spark's DataFrame API to allow users to write spatial queries using standard SQL or DataFrame. Magellan examines the user's query and object types to build and optimize the query execution plan. GeoSpark [34] introduces SRDD (Spatial RDD), an extension of Spark's RDD that allows users to execute spatial operations. We compare Spark-based frameworks against our GeoMatch framework in the following section.

Other frameworks. Besides these frameworks, there have been other efforts at developing support for map matching on top of big-data processing infrastructure [5, 24, 26]. As an example, CloudTP is a cloud-based system for trajectory data preprocessing and exploration that leverages Azure cloud technologies for performance [5, 26]. Unlike GeoMatch, CloudTP is not available open-source, making it difficult to benchmark its performance. Additionally, CloudTP does not address scalability, relying instead on the cloud provider's capability to offer suitable computing resources. Other related systems include DMM [4] and the Spark-based system of Peixoto et al. [24]. The former is not available as open source and both systems rely on sampling when constructing the index used for data partitioning.

2.2 Comparison of Spark Based Spatial Big-Data Frameworks

To further motivate GeoMatch, we next compare five popular Spark-based state-of-the-art spatial big-data frameworks and highlight how GeoMatch extends on them. Table 1 shows a summary

Table 1. Comparison of In-memory Map Matching Methods

Feature	GeoSpark	Location-Spark	Magellan	STARK	Simba	**GeoMatch**
Implemented with	Java	Scala	Scala	Scala	Scala	Scala
Modify Spark's Core	**No**	**No**	Yes	Yes	Yes	**No**
Spatial Library	**JTS**	Custom	Custom	**JTS**	**JTS**	**JTS**
Random Sampling	One set	One set	**None**	User set	User set	**None**
Sample Processing	Master Node	Master Node	**No sampling**	Master Node	Master Node	**No sampling**
Allow Non-spatial Data	**Yes**	Programming required	No	**Yes**	Programming required	**Yes**
Street Map Matching	**Find Nearest**	Point in MBR	Point in MBR	**Find Nearest**	Not Supported	**Find Nearest**
Deterministic Results	**Yes**	No	**Yes**	**Yes**	N/A	**Yes**
Relative Performance	10.93-16.73	N/A	N/A	1.0-1.0	N/A	**27.25-39.03**
Memory Requirements	Expo-nential	Expo-nential	Expo-nential	Expo-nential	N/A	**Linear**
Accurate Matching	Programming required	Programming required	Programming required	**Supported**	Programming required	**Supported**
Allow mixed objects	No	No	No	**Yes**	No	**Yes**
Dataset Configuration	**In-dependent**	**In-dependent**	Dependent	Dependent	Dependent	**In-dependent**

comparison between all frameworks including GeoMatch. As part of our experiments (see Section 6), we compare these frameworks in terms of their support for different operations and data structures required in map matching. Moreover, we examine the accuracy of their results by comparing their individual outputs to a high-precision baseline derived through exhaustive search.

Compared Frameworks. We compare the five most cited and recent frameworks: GeoSpark, LocationSpark, Magellan, STARK, and Simba. GeoSpark offers several partitioning techniques including Hilbert curve. However, it only supports datasets of a single type, and only the join query operation allows for operations with two datasets. LocationSpark offers a number of indexing options and uses a Hilbert curve to enhance its *k*NN query and it has been shown to outperform other frameworks [29], such as Simba. However, LocationSpark's techniques are limited to Point and Box spatial objects. Magellan was the first framework to extend Spark SQL to offer geospatial analytics. It allows users to index data during loading, but we have found its scalability to be very limited and only support small dataset sizes. STARK supports Grid and cost-based Binary Space Partitioning [12] and works best when its users have good knowledge of their datasets to customize their spatial query operations. Simba extends Spark SQL, limits its support to Point spatial objects, and implements a Sort-Tile-Recursive (STR) [17] algorithm as its main partitioner.

Support for Spatial Objects. Support for spatial objects in existing spatial extensions varies considerably with few offering support of operations and objects required in map matching (e.g., joining multiple LineStrings and Points). Even fewer frameworks allow for operations on mixed-type dataset objects. GeoSpark supports non-spatial data through the JTS *userData* field [19]. LocationSpark, Magellan, and Simba lack support for LineString objects. Consequently, streets must be represented as Rectangles via their minimum bounding rectangle (MBR). LocationSpark and Magellan do not allow mixing or carrying non-spatial data. Therefore, their source code must be modified; corresponding geometry objects are extended to add fields that allow for non-spatial

data. This, subsequently, increases memory requirements, but preserves original trip records and produces more accurate results. STARK allows for non-spatial data through either JTS or as part of its operational tuples. GeoMatch supports both necessary spatial data structures and non-spatial data. Specifically, GeoMatch supports four types of spatial objects and uses custom objects that encapsulate spatial and non-spatial data together as one unit (see Section 3).

Spatial Indexing and Partitioning. In spatial big-data frameworks, indexing plays a critical role in determining how tasks are partitioned across cluster nodes. Due to performance reasons, existing frameworks construct the index from a subsample of all data objects. The general idea is to group objects based on the intersection of their MBRs, and a global join of the index is then used to group data from non-empty intersecting MBRs into partitions for distributed processing. As shown in Figure 2, urban datasets are often skewed, which means the initial index is constructed from an unbalanced sample. This easily results in a heavily skewed index where some partitions are (nearly) empty, making effective parallelization difficult. Of the existing frameworks, GeoSpark builds a global index by sampling its input datasets. MBRs of the samples are sent back to the Master node that then builds a global index and partitions the datasets accordingly. LocationSpark has a dedicated layer (query scheduler) to address query skews. This layer samples statistics from each partition to index the datasets and to create a more balanced partitioning scheme. Statistical collection in this manner may produce different results in subsequent runs with the same input conditions, resulting in runtime irregularities similar to those detailed in Section 6.3. Magellan does not sample either of its datasets, but can be instructed to index one or both sets while they are being loaded. However, this significantly increases the spatial query execution time. STARK allows the user to decide when and if to use spatial indexing. Based on their datasets, the user must decide on the best type of partitioner, sample rate, and number of dimensions. The default partitioner in Simba (STRPartitioner) samples each of its input datasets to determine its entire MBR. The number of partitions is determined through a cost-based optimizer that depends on the query type. GeoMatch achieves a balanced index using a Hilbert space-filling curve (HSFC). While other frameworks also support HSFC indexing, they do not use it for partitioning and construct the index only from a small sample of measurements. For example, GeoSpark offers a HSFC partitioning scheme that decides on the best way to partition its datasets, and LocationSpark uses HSFC to enhance its *k*NN join queries. GeoMatch builds the HSFC for the entire dataset without using any sampling and uses the resulting index for partitioning tasks across different nodes.

Accurate Map Matching. Implementing map matching while relying on the streets' MBRs produces inaccurate results when the street is not aligned with its MBR. Consider, e.g., West 49th Street in Figure 1. Very few points intersect the street and hence the MBR of the street will encompass many unnecessary points. However, many of these are far from the road and hence should not be matched against the road. Additionally, it is not easy to discern the best match when a single point falls within multiple streets' MBRs as is the case with points P_1, P_{16}, and P_{31} in the same figure. Out of the five frameworks, only STARK supports accurate matching, and only GeoSpark and STARK allowed for nearest-match search, while others rely on basic point-in-MBR matching. As we experimentally demonstrate in Section 6, this greatly affects the accuracy when matching GPS points to street segments.

2.3 Benchmarks for Spatial Big-data Frameworks

Large-scale map matching benchmarks are usually offered as part of the framework's publication and performed using metrics that highlight the features of frameworks. None of the frameworks surveyed in this work consider accuracy of the results of their spatial operations, and to the best of our knowledge, no attempts have been made elsewhere. GeoSpark compares

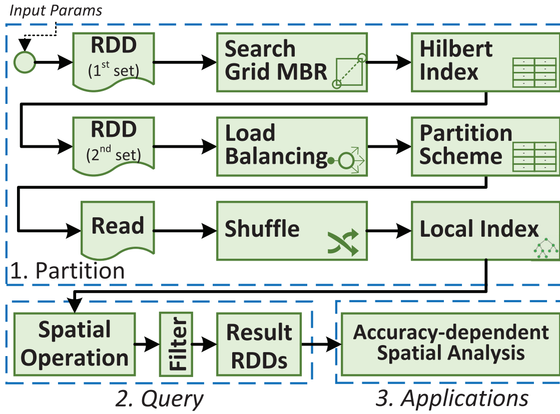


Fig. 3. GeoMatch pipeline.

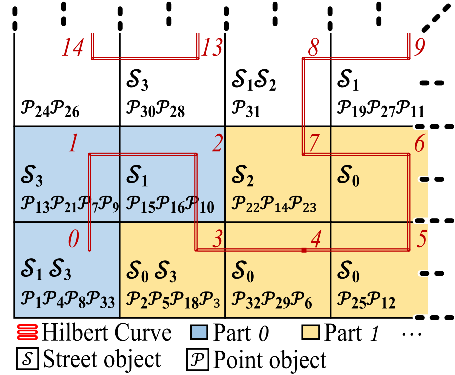


Fig. 4. A partial $8X8$ Hilbert curve clustering.

its results against SpatialHadoop using Spatial co-location pattern recognition. The results are limited to showing the runtimes performance using two different datasets and three indexing techniques. LocationSpark introduces a spatial bitmap filter that aims at reducing data shuffling and thus improving query processing times. The authors experimentally compare query and index building time performances to Magellan, SpatialSpark, GeoSpark, and Simba. Moreover, they examine Spatial and *k*NN search and join queries using two different datasets without examining the accuracy of their results. Simba used a synthetic dataset as part of its experiments and focused on the cost of indexing, throughput, latency, and runtime of execution.

Other benchmarks and surveys. Eldawy et al. [10] survey spatial systems based on their approach, architecture, and supported spatial components. The comparison mainly focused on MapReduce and Spatial Database systems without performing empirical comparisons. Pandey et al. [23] evaluated five frameworks based on available features and supported queries. The performed tests included Range, *k*NN, Spatial Join, and *k*NN Join queries using LineString, Point, Polygon, and Rectangle objects. Experimental results reported on the throughput, memory usage, and indexing and shuffling costs. In contrast, we also examine the effects of using datasets with varying densities, the accuracy of the produced results, and the ability of the framework to account for GPS inaccuracies.

3 THE GEOMATCH PIPELINE

GeoMatch is an extendable, scalable, and accurate map matching pipeline that overcomes limitations of current spatial Spark extensions for large-scale map matching described in the previous section. GeoMatch is implemented in Scala and adds fast spatial processing capabilities to Apache Spark through spatial partitioning, object recognition, and query processing without modifying the core of Spark. Figure 3 shows the high-level workflow of GeoMatch and Table 1 compares its features with existing frameworks. GeoMatch is currently designed to match two mixed-object spatial datasets (i.e., road network and position data) and supports four object types (Point, LineString, Polygon, and Rectangle) and two operations ($spatialJoinKNN$ and $spatialJoinDistance$). New objects and operations can be easily added without modification to the core of GeoMatch. To reduce processing overhead, optimize memory usage, and produce fast and accurate results, GeoMatch goes through several simple but highly efficient steps that operate on

Table 2. GeoMatch Inputs

Parameter	Default	Value Type	Description
Dataset1	N/A	Spark RDD	The first input dataset
Dataset2	N/A	Spark RDD	The second input dataset
HilbertN	256 (2^8)	Integer	The size of the Hilbert space-filling curve
k	3	Integer	The maximum number of matches to choose from the first dataset.
matchDist	150	Integer	The maximum distance for accepted matches. After this distance, the match is rejected.[6]
searchMBR	(-1,-1,-1,-1)	Integer Tuple	The search Grid MBR. If not provided, GeoMatch will compute it from Dataset1.

relevant data[4] only. The partitioning scheme is based on one of the input datasets and is used to spatially group objects from both datasets on the processing nodes. The final output of the match process is a tuple consisting of (1) an object from the second dataset and (2) a list of the best k matches from the first dataset. The list of best matches can be used to account for miss-alignments (e.g., GPS pings) for tasks such as path reconstruction or service demand analysis.

3.1 Design Features

Inputs: Many frameworks require extensive knowledge about the dataset distribution to achieve higher spatial query performance. This introduces an additional usability restriction that could affect the performance and accuracy of the query. In GeoMatch users are not required to have such knowledge, since the datasets are examined internally. Users are only required to provide the input datasets as Spark RDDs and can override any of the default values for one or more of the parameters shown in Table 2. Note that the default value for the match distance has been specified based on characteristics of downtown New York City, since all datasets considered in our experiments originated from there. In particular, the average block's width in NYC is about 264 ft, and hence a match distance threshold of 150 ft ensures that at least half of the block's width is covered.

Data Format: GeoMatch operates on data as lightweight spatial objects from the start without restrictions on the original data format. This increases efficiency and results in better usability, flexibility, and eliminates the need to make assumptions that may slow development or execution. Users have complete control over parsing their data and decide how to represent data using GeoMatch's lightweight objects. Each object is limited to two fields: a *Payload* field—a string value that is carried with the object through the computation steps and to the output; and a *Coordinates* field—a list of one or more coordinate pairs representing the geometry object.

Hilbert Space-filling Curve: Partitioning aims at grouping objects by spatial proximity and affects the speed of execution and accuracy of output. If the data are not partitioned properly, shuffling overhead increases and could require additional and expensive large-scale operations such as *join* and *sort*. GeoMatch integrates a novel approach and does not rely on sampling to build its partitioning scheme. Instead, it examines the larger of the two datasets before it finally spatially partitions both datasets (e.g., location measurements and road network). The partitioning scheme of GeoMatch builds on top of the well-known clustering properties of a Hilbert space-filling curve.

[4]Relevant data are the subset of the data that contribute to the output on each processing node.

Each object in the second input dataset is assigned to exactly one Hilbert index. Objects from the first dataset may be duplicated if GeoMatch decides they may contribute to matches on other partitions. The Hilbert indexes are divided between the available partitions such that the load is fairly distributed (i.e., query skew mitigation). Indexes are assigned to the same partition in order of their spatial proximity. The partition assignment acts as a *global index* that groups objects prior to executing the spatial query. Data shuffling is minimized by eliminating the need for large-scale sorting and by ensuring that objects from the second dataset are never duplicated.

To further illustrate the efficiency of GeoMatch's clustering process, consider the partial $8X8$ Hilbert space-filling curve shown in Figure 4 that is built around the search region. In this example, we assume that the first dataset consists of four streets (S_0–S_3), and the second dataset consists of 36 points (P_0–P_{35}). The figure shows the streets and points after being partitioned based on their spatial proximity. Streets and points that fall outside the search region are naturally excluded from the matching process. Note that the figure only shows part of the grid and therefore does not show all objects. Assuming that there 4 available partitions, we set $partLoad = \frac{36}{4} = 9$. Subsequently, Hilbert indexes 0, 1, and 2 are assigned to $part_0$ regardless of the fact that their point count is over 9. This is allowed to increase accuracy and to prevent an index from spanning multiple partitions. Similarly, indexes 3–7 are assigned to partition ($part_1$), and so forth.

Data Partitioning: A naive approach to spatial data partitioning using a Hilbert space-filling curve is to simply compute the index of every record in the datasets followed by a *union* transformation. However, doing so will require additional expensive transformations to group and sort the objects (i.e., $repartitionAndSortWithinPartitions$). In GeoMatch, we gain higher performance by taking advantage of Spark's internal *PartitionerAwareUnionRDD* transformation. This transformation is highly efficient, migrates the smaller partition towards the larger one, and places the left operand data before the right one. In our experiments, this ensured that street objects appeared before point objects in every partition and eliminated any need for large-scale data sorting.

Querying: To achieve higher accuracy, the actual distance between the two geometries must be computed to determine if the match is kept or discarded. In GeoMatch, we perform this calculation locally after objects from both datasets are grouped on their partitions. The aim is to achieve results that are as close to those achieved via exhaustive map search (baseline). GeoMatch achieves higher accuracy by ensuring that objects with a potential match are present in all partitions. This is done by selectively duplicating objects from the *first* dataset to different partitions if objects from the second dataset span these partitions.

3.2 Implementation

In this section, we detail how GeoMatch works to solve the map matching problem on Apache Spark using two datasets in the form of LineString (e.g., Streets) and Point (e.g., GPS points from taxi/bus trip records). The output is a tuple consisting of the original trip record and a list of up to three closest streets. For this task, we set the first dataset to the NYC LION Streets[5] and the second dataset to the SMALL taxi dataset. The Hilbert's curve is set to 256.[6] k is set to 3 and the search grid MBR is set to -1, which instructs GeoMatch to compute it internally from the first dataset.

Initial Setup: The first step in GeoMatch is to compute the MBR of the first dataset. This is done in parallel with the final results merged on the master node. Next, the Hilbert grid bounds are built using the computed MBR and its region is divided into $2^n \times 2^n$ boxes of equal heights and widths.

[5]For performance gains, the smaller dataset should be set as the first input set.
[6]The average block in NYC is 264 ft×900 ft with an average length of 582ft. Dividing the MBR into 256 yields a box size that is close to that average. The corresponding Hilbert's curve order is 8.

Table 3. Sample Index-point Counts

Hilbert Index	Count
0	88
1	48
2	29
...	...

Table 4. Sample Index-partition Map

From	To	Assigned Partition
1	25	0
26	30	1
50	72	2
...	...	...

Hilbert Curve Indexing: Using the grid generated from the first step, the Hilbert index for each object in both datasets is computed. The process is object-dependent and, therefore, GeoMatch is able to accommodate datasets with mixed type objects. Point objects are the simplest, as their single index is computed based on their coordinates. LineStreet objects utilize a modified Digital Differential Analyzer (DDA) algorithm [7] to compute the indexes that each of the streets' segments passes through. Polygon and Rectangle objects utilize a modified Scan-Line Algorithm [6] to compute the indexes enclosed within their areas. Due to the nature of DDA and Scan-Line and the structure of the corresponding spatial objects, some boundary, or near to the boundary, indexes may be excluded. To remedy this, both algorithms internally double the resolution of the Hilbert region to increase accuracy.

Load Balancing: A performance gain in GeoMatch comes from the near-fair distribution of the load based on the distribution of data such that no partition is more overloaded than others. To balance the load, we first compute a summary of the index distribution of the second dataset. The summary shows the number of spatial objects that fall within each index. Table 3 shows an example of an index-to-point counts. Using this list, the optimal partition load is computed by dividing the sum of all points in all indexes by the total number of partitions used by the second dataset: $partLoad = \frac{pointCount}{partCount}$. The value of *partLoad* indicates how many geometry points each available partition should process. GeoMatch may decide to exceed this limit only in favor of keeping geometries with the *same index* together to increase accuracy.

Partitioning Scheme: Using *partLoad* and the list of index counts from the previous steps, a partitioning scheme is built to spatially cluster the datasets. The scheme simply indicates which indexes are assigned to which partition such that the computation load is fairly distributed across *all* partitions while keeping spatially close indexes in the same partition. Table 4 shows an example of an index-to-partition distribution.

Shuffle: Once the partitioning scheme is built, it is used to *independently* partition both RDDs. The two RDDs are then joined using Spark's *union* transformation, which internally invokes the *PartitionerAwareUnionRDD* transformation described earlier. This results in major performance gains, as nearby objects now reside on the same partition, eliminating further data shuffling.

Querying: Map matching in GeoMatch starts after the partitions are joined. On each partition, a *local* R-Tree of the first dataset (i.e., NYC LION Streets) is built to speed up the query process. As described earlier, our approach naturally ensures that the street objects appear before the point objects; therefore, it is easy to determine the tree's first and last items. Moreover, to reduce the number of false-positive matches caused by large R-Tree MBRs, we break each street into its individual segments and insert them into the R-Tree. Furthermore, the segment's MBR is expanded by 150ft to account for the inaccuracies in the initial GPS reporting. Finally, the local R-Tree is queried and a list of candidate streets is selected for each point. Next, the distance between the point and each of its street matches is calculated; if the distance is larger than 150ft, the match is rejected. The closest three streets (if any) are kept.

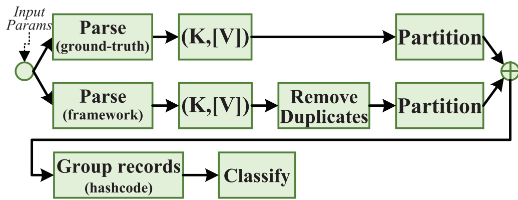


Fig. 5. Accuracy benchmark flow.

Parameter	Description
baseFile	The base comparison file (e.g., map-matching ground-truth)
resultFile	Classification file (e.g., results of a map-matching technique)
parserBase	The base file's line parser.
parserResult	The classification file's line parser.
n	Number of matches to classify as accurate.

Fig. 6. Accuracy benchmark input parameters.

4 ACCURACY BENCHMARK FRAMEWORK

In map matching, speed of execution is important but not at the expense of accuracy. Existing evaluations of large-scale map matching have predominantly focused on execution performance without assessing how the underlying optimizations affect the accuracy of map matching. To provide a comprehensive assessment of map matching frameworks, as our second contribution, we present a benchmark framework that assesses both accuracy and execution performance. For our analysis, we define accuracy relative to a baseline and define it as the number of records with matches that agree with the selected baseline method (see Section 5).

Our benchmark framework has been designed to be easy-to-use and not require any knowledge of the dataset being analyzed. The overall workflow of our benchmark framework is shown in Figure 5 and can be configured using the parameters shown in Table 6. We use the benchmark framework in our experiments to classify the accuracy of the output of each studied framework and present our findings in Section 6.6.

4.1 Design and Implementation

The framework has been implemented using Scala and utilizes Spark's RDDs for distributed processing. Each input file is independently parsed and partitioned before finally joining them using the *PartitionerAwareUnionRDD* transformation. For the purposes of benchmarking the results of the studied map matching techniques (Section 6), we set the first input file to the one produced through the baseline process described in Section 5. The second input file is set to the result obtained from GeoMatch, GeoSpark, LocationSpark, then STARK.

Initial Step: Five input parameters, provided as command-line arguments (e.g., $baseFile$, $resultFile$, $parserBase$, $parserResult$, and n), are required to start the process. The first is the input file that contains matches generated by the framework being evaluated. The second is the ground-truth file that contains correct matches for each input record. The third and fourth inputs specify functions that transform these input files into key value pairs, separating input records (key) and the list of the matches for that record (value). A parser is the full name of a Scala class that implements the provided *BenchmarkInputFileParser* interface and is dynamically instantiated. Each file is parsed using the provided parser that in turn produces an RDD of key-value pairs. The framework does not make assumptions as to how the parsing algorithm works, since we recognize that each file's structure is different. Finally, the second file is filtered for duplicates, since some techniques tend to produce results such that the matched record is repeated multiple times but with a different list of matches. To remove these duplicates, a *reduceByKey* transformation merges the lists of matches of similar records, ensuring that unique matches are left in the final list.

Partitioning and Grouping: Before records can be correctly classified, similar records must be present on the same processing nodes. In the partitioning and grouping step, records with similar

keys from both RDDs are migrated to the same partition. We accomplish this using a custom *KeyPartitioner* that assigns each record to a partition number based on the hash code of the record's key. Next, the partitioned RDDs are joined. At this point, matches of similar keys are not yet grouped. To compare the key's list of matches, a *reduceByKey* transformation groups the records and produces a three-part tuple consisting of the key, first file matches, and second file matches $(key, list_1, list_2)$.

4.2 Record Classification

To evaluate a framework, we compare its output against those in the ground truth and classify the records into different categories. In the process, the position of each match in the second file (evaluated benchmark) is looked up in the list of matches in the first file (ground truth). For example, if $match_1$, $match_2$, and $match_3$ are the first three matches for $record_1$ in the second file, we look up their positions in the list of matches of $record_1$ in the first file. Based on their order, the record is classified into one or more of the following categories:

- **Correct Matches:** the number of records correctly matched in the second file. A record is considered correctly matched if at least one of the n matches appears in the first n matches of the baseline file.
- **Neither Matched:** the number of records that received no matches in either input file. This usually indicates an error in the matching technique or that the record is located far away and outside the match area.
- **FW Only Matches:** the number of records that received at least one match in the second input file but received no matches in the first. This can be the result of a poor matching technique or due to an incorrect geometry representation (e.g., LineString vs. Rectangle).
- **FW Failed Matches:** the number of records that received no matches in the second input file but received at least one match in the first. This can be a result of a poor matching technique or due to an incorrect geometry representation (e.g., LineString vs. Rectangle).
- **FW Overmatch:** the number of records that received matches in the second input file that is greater than the number of matches received in the first input file. This can be a result of a poor matching technique or due to an incorrect geometry representation (e.g., LineString vs. Rectangle).
- **FW Undermatch:** the number of records that received matches in the second input file that is smaller than the number of matches received in the first input file. This can be a result of a poor matching technique or due to an incorrect geometry representation (e.g., LineString vs. Rectangle).
- **Excluded Records:** the number of records that are missing from the second input file but are present in the first input file. As mentioned earlier, this happens when the matching technique excludes records that it cannot match due to an incorrect matching technique of improper geometry representation (e.g., LineString vs. Rectangle).

The final classification report shows the aggregate results for all records across all nodes.

5 DATASETS

In our experiments, we consider three real-world trip-record datasets (Table 5) of varying source, size, and GPS densities.[7,8,9] All three datasets were collected in NYC with two datasets

[7] www.nyc.gov/html/tlc/html/industry/taxicab_serv_enh.shtml.
[8] www.nyc.gov/html/tlc/html/about/trip_record_data.shtml.
[9] web.mta.info/developers/MTA-Bus-Time-historical-data.html.

Table 5. Experimental Dataset Summary

Dataset Name and Abbreviation	Size	Num. Records and Type	Remarks	Baseline Time
TLC TPEP and LPEP (LARGE)[7]	141.99 GB	3.78B Point (99.71% valid records)	• Non-uniform distribution (Figure 2). • 10.9M duplicate records. • 158.9M unmatchable records.	71.5 mins.
TLC Trip Record (SMALL)[8]	7.06 GB	165.1M Point (98.01% valid records)	• 12 files (Jan.–Dec.) • Ideal for testing frameworks that cannot handle the LARGE dataset.	3.31 mins.
NYC Bus Trip Record (BUS)[9]	7.16 GB	216M Point (64.18% valid records)	• Similar format to the LARGE dataset. • Covers half of the NYC LION streets. • Good for testing the technique's behavior when some streets are significantly overloaded than others.	5.85 mins.
NYC LION streets (road network)[11]	17.7 MB	166,253 LineStrings (100% valid records)	• Single line base map of streets in the greater NYC region.	N/A

containing measurements from NYC taxis and one from NYC buses. Each record in these sets contains information about a single trip including GPS locations. Due to inaccuracies in the GPS reporting mechanism,[10] we aim to match the GPS locations in these datasets to the nearest city street in the NYC road network dataset[11] released by NYC Department of City Planning.

To gauge the accuracy of the map matching techniques studied in this article, we compare the matching results of each technique against a baseline obtained through an exhaustive search process and a standard map matching algorithm for sparse data, i.e., we identify most likely street segments based on their offset from the observed position and topological constraints [20, 36]. Note that our goal is not to evaluate the underlying map matching algorithm itself, but to assess how the distribution of operations affects its performance. Indeed, estimating the accuracy of the map matching algorithm would require precise ground truth, which is rarely available for large-scale spatial data. Most large-scale spatial datasets, including ours, only contain the position and state of the vehicle without information about the road the vehicle is currently on [21, 25, 38]. In some cases it may be possible to separate trips from other data, e.g., using passenger state information, but even then route information contains uncertainty, as drivers may follow different routes and roadworks, or other events may force the driver to take detours.

The process of establishing a baseline, as shown in Figure 7, starts by parsing the first input dataset (e.g., entire NYC street network) and building the proper spatial objects. Next, an entry-optimized R-Tree is constructed, containing all objects in the first dataset. By optimizing the R-Tree, we can minimize the number of false-positive matches and account for GPS inaccuracies. This step is object-dependent; LineStrings are subdivided into segments and each segment's MBR is expanded by a predefined amount (Figure 8). Finally, the step ends with broadcasting the R-Tree to all processing nodes. After the R-Tree is broadcast, the second dataset is parsed and proper spatial objects are built. For each object, the R-Tree is queried and the distance between the objects is calculated; if the distance is within a predefined radius, the match is accepted. The best k matches (if available) are kept in order of proximity. The final output consists of the original trip records

[10] GPS inaccuracies are caused by a number of reasons, including signal delays, interference, and signal blocking.
[11] www1.nyc.gov/site/planning/data-maps/open-data/dwn-lion.page.

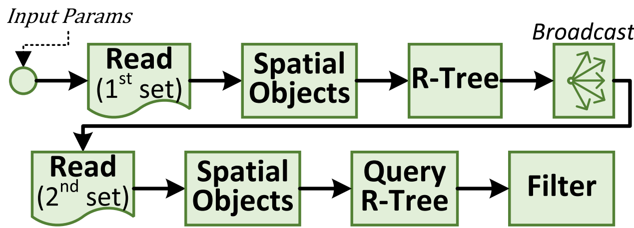


Fig. 7. Work flow of ground-truth process.

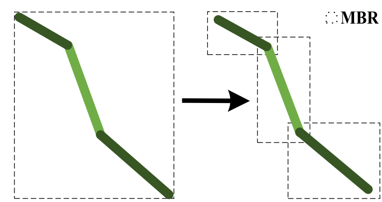


Fig. 8. Example object segmentation with MBR expansion.

with valid GPS coordinates followed by up to k matches. Duplicate records and records with invalid GPS coordinates are omitted from the output.

6 EXPERIMENTS

We perform extensive map matching benchmarks using the datasets detailed in Table 5. We consider the same frameworks compared in Table 1. However, of the six frameworks, we were forced to exclude Simba and Magellan. The former does not support the necessary spatial structures, whereas the latter timed out in all benchmark experiments. In each of our experiments, the goal is to match the points with their respective streets using proper spatial objects. Tests and analyses were performed using the latest source code obtained from the respective GIT repositories for each studied framework. All experiments were conducted at the operational data facility of our research center. Our cluster consists of 20 high-end nodes each with 24 TB of disk space, 256 GB of RAM, and 64 AMD cores (total 1,200+ cores) running Cloudera Data Hub 6.1.0 with Apache Spark 2.4.0. Note that none of the studied frameworks, including GeoMatch, take into consideration the trajectory path, as the focus of our work is on developing support for large-scale map matching. The output of GeoMatch can be used to further infer the entire trajectory path if needed.

6.1 Experimental Setup and Measures

In our experiments, we measure the scalability, execution times and accuracy of frameworks under operational constraints. Each of our experiments was repeated three times to accurately measure the behavior and to observe runtime stability. We consider two types of scalability in our experiments. In *weak scalability*, the size of the input dataset is gradually increased along with the number of processing cores, as shown in Table 6. The aim of these experiments is to observe if the framework utilizes the additional processing power under the constraint of additional inputs. In *strong scalability*, the entire dataset is processed while gradually increasing the number of processing cores. The aim of these experiments is to observe the scalability of the framework and how efficiently it utilizes the additional processing power.

To evaluate accuracy, we need to perform separate processing steps on the output of the frameworks. First, we ensure that the resulting RDD is in the form (*Point*, *List of Polygon*). Second, the distance between the point and the matched street segment is calculated to gauge the accuracy of the match. If the distance is greater than a predefined limit (e.g., 150 *ft*), then the street selection is rejected. The final result consists of the original trip record followed by a list of up to three street IDs ordered by proximity. Finally, since all of the frameworks studied here exclude points that could not be matched with at least one street, an additional resource-expensive step is required to reintroduce the missing objects. The exclusion is done to reduce the memory and computing resources. However, this produces incomplete results with respect to the input and can effect subsequent tasks that require a complete output. Finally, a full-join operation on the results and the

Table 6. Weak Scalability Experiment Configurations

		Dataset Points (Million)		
Test	Cores	SMALL	LARGE	BUS
1	50	26.85 (Jan.-Feb.)	28.33	27.1
2	100	56.89 (Jan.-Apr.)	57.32	56.31
3	150	85.48 (Jan.-Jun.)	86.83	85.63
4	200	111.28 (Jan.-Aug.)	114.63	111.69
5	250	138.88 (Jan.-Oct.)	142.09	142.29

original dataset ensured that unmatched points were all included in the final output. The inclusion of unmatched points is necessary for future tasks like error analysis without incurring additional steps. The final format of the result was the original trip record, with a list of up to three street IDs appended to each data line. GeoMatch follows a technique that naturally passes objects from input to output without caching them. Therefore, it does not exclude points that it cannot match.

6.2 Constraints

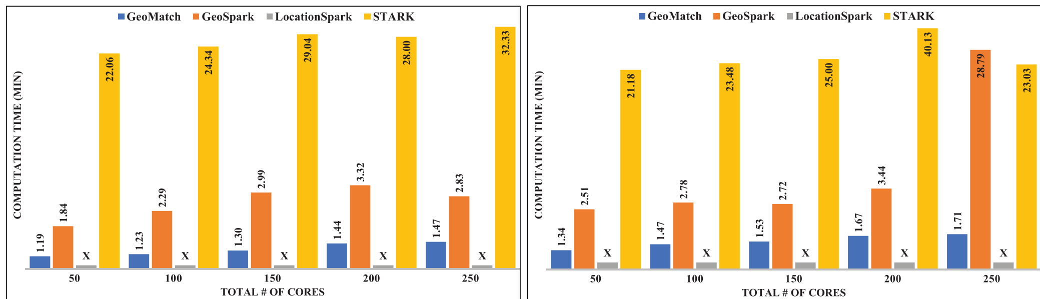
To emulate real-world analyses that are subject to operational constraints, we additionally enforce the following restrictions on the operations of the systems:

Time: Data analyses often operate under time and budget constraints. Therefore, we only consider completed jobs within an upper limit of 180 minutes and stop any experiment if it exceeds this limit. Hence, we exclude results for Magellan, as its tasks consistently timed out, requiring more than 180 minutes.

Error-correction: To account for inaccuracies in the initial GPS reporting, results obtained from each framework are filtered and sorted by proximity. The distance between each point and its street matches is calculated; if the distance is within $150ft$, then the match is kept, otherwise, it is removed. For each record, the best three matches for each point were kept in order of proximity.

Completeness: Map matching is usually the first step in spatial-data analysis. Therefore, the input dataset must be complete with respect to the input. For example, if the input dataset consists of M records, the output dataset should, also, consist of M records. LocationSpark, GeoSpark, and STARK do not include unmatched records and produce outputs that are different in size compared to the original dataset. We opted to leave these outputs without further processing, since including missing points would require an additional *join* operation that would further increase their runtimes. This does not apply to records filtered after the spatial operation due to distances larger than $150ft$. These records are redirected to the output file and do not require additional memory for processing. GeoMatch does not suffer from this problem, since it naturally passes unmatched points through its pipeline and produces matches in order of their proximity.

Memory Requirements: Spark applications benefit from its in-memory processing; however, they are still limited by the amount of memory available to the drivers and executors. The map matching process greatly benefits from efficiently using the available memory. In the case of



(a) LineString (Error Correction with GeoMatch only) (b) Rectangle (With Error Correction)

Fig. 9. SMALL dataset – weak scalability test (logarithmic scale). Runtime for LocationSpark (grey bar X) not provided due to (a) the lack of support of LineString objects, (b) consistent fails or over 180-minute limit.

Table 7. SMALL Dataset – Standard Deviation

			Weak Scalability					Strong Scalability				
Framework			50	100	150	200	250	50	100	150	200	250
LineString		GeoMatch	0.01	0.01	0.00	0.01	0.03	0.02	0.03	0.02	0.03	0.01
	No Error Corr.	GeoSpark	0.26	0.02	0.29	0.44	0.33	0.85	0.24	0.62	0.25	0.57
		LocationSpark	-	-	-	-	-	-	-	-	-	-
		STARK	0.19	0.14	7.95	0.35	9.39	2.55	9.20	5.07	17.12	0.30
Rectangle		GeoMatch	0.01	0.01	0.01	0.00	0.01	0.05	0.06	0.11	0.02	0.02
	With Err. Corr.	GeoSpark	0.25	0.17	0.08	0.41	19.01	1.34	0.62	0.83	0.72	0.50
		LocationSpark	-	-	-	-	-	-	-	-	-	-
		STARK	0.37	0.09	0.15	9.86	1.20	4.88	2.21	8.49	2.48	11.03

A value of "-" indicates that either the spatial object is not supported or the task failed to complete.

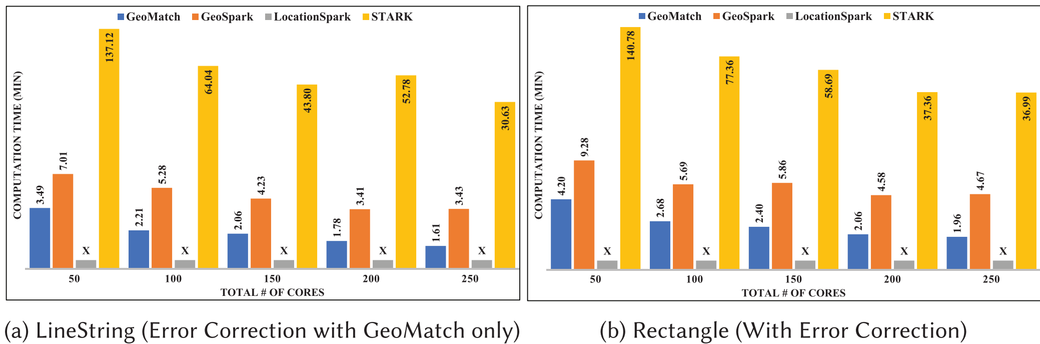
LocationSpark, GeoSpark, and STARK, we set the memory allowance to the maximum allowed by our cluster for them to complete their tasks—8GB for the driver and 32GB for each executor. GeoMatch, by design, requires less memory, and we set its jobs to 6GB for the driver and only 8GB for the executor.

6.3 Small Taxi dataset (SMALL)

In this experiment, we match points from the SMALL dataset with streets from the NYC LION street dataset and report the average runtimes. Depending on the framework's supported objects, we first experiment with streets represented as LineStrings then repeat the experiments using Rectangle representation. Matches with distances larger than $150ft$ are removed before the final results are produced.

Weak scalability test: The input size and the number of processing cores are gradually increased, as described earlier. Each experiment was repeated three times first using LineStrings street representation, and then using Rectangles street representations. Figure 9 shows the average runtimes in minutes of these repeated experiments and Table 7 shows their standard deviation.

When representing streets as LineString objects, GeoMatch completed all tests first while producing a complete output dataset and accounting for errors in GPS reporting. Its runtimes remained relatively close across all five experiments with a standard deviation of 0.11 minute. This indicates that GeoMatch can properly utilize the additional processing power and distribute the workload across the processing nodes. GeoSpark finished second with runtimes slightly less close



(a) LineString (Error Correction with GeoMatch only) (b) Rectangle (With Error Correction)

Fig. 10. SMALL dataset – strong scalability test (logarithmic scale).

to each other and a standard deviation of 0.52 minute. STARK finished last with runtimes varying greatly across the experiments and scored a standard deviation of 3.60 minutes. Runtimes for LocationSpark are not provided, since the framework lacks support for LineString objects.

When representing streets as Rectangle objects, the MBR is expanded by 150 ft before the start of the spatial query. GeoMatch completed all tests first while, again, producing a complete output dataset. Its runtimes remained relatively close across the experiments with a standard deviation of 0.14 minute. GeoSpark finished second across all experiments except for the last one with 250 cores. The standard deviation for its runtimes was 10.38 minutes. STARK finished last with varying runtimes and standard deviation of 6.89 minutes. Runtimes for LocationSpark are not provided, because its tests either failed or were stopped for exceeding the 180-minute limit.

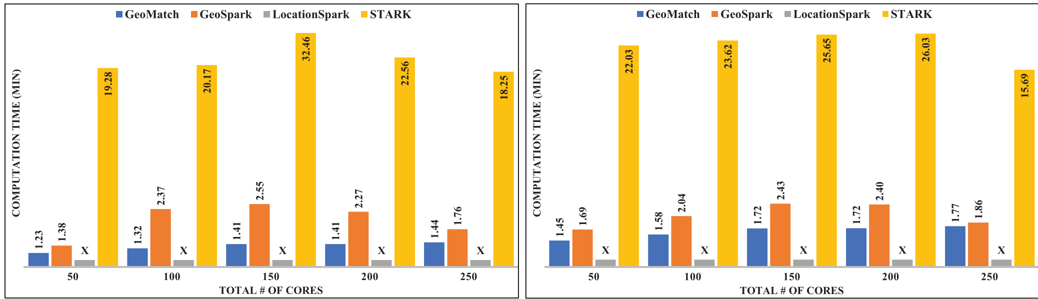
Upon examining the standard deviation of repeated experiments for both representations, we notice that GeoMatch exhibited the smallest variation of runtimes with a range of 0.00–0.03 minutes and 0.00–0.01 minutes using LineString and Rectangle, respectively. The small variation indicates good stability and reproducibility in performance for repeated experiments. GeoSpark exhibited higher runtime variation with a standard deviation range of 0.02–0.44 minute and 0.08–19.01 minutes, respectively. STARK performed worst with a standard deviation range of 0.14–9.39 minutes and 0.09–9.86 minutes.

Strong scalability test: The input size is fixed to the entire SMALL dataset (12 months) while gradually increasing the processing cores from 50 to 250 (in steps of 50). Figure 10 shows the average runtimes in minutes for all experiments using two street representations, and Table 7 shows their standard deviation.

When representing streets as LineString objects, GeoMatch completed all tests first while producing a complete output dataset. Its runtimes gradually decreased as the number of processing cores were increased, suggesting proper utilization of additional processing power and proper workload distribution. The standard deviation of GeoMatch's runtimes in these experiments was 0.66 minute. GeoSpark finished second with decreasing runtimes that took twice as long as GeoMatch. The standard deviation of its runtimes was 1.35 minutes. STARK finished last with runtimes decreasing except for the test with 200 cores; its runtime standard deviation was 37.36 minutes.

When representing streets as Rectangle objects, the order of task completion and performance were similar. GeoMatch's standard deviation was 0.81 minute; GeoSpark's standard deviation was 1.71 minutes. STARK's standard deviation was 38.33 minutes. Runtimes for LocationSpark are not provided either due to failure or termination.

The standard deviation of repeated experiments for the different representations were similar to those of the weak scalability. GeoMatch runtimes ranges were 0.01–0.03 minute and



(a) LineString (Error Correction with GeoMatch only) (b) Rectangle (With Error Correction)

Fig. 11. BUS dataset – weak scalability test (logarithmic scale). Runtime for LocationSpark (grey bar X) not provided due to (a) the lack of support of LineString objects, (b) consistent fails or over 180-minute limit.

Table 8. BUS Dataset – Standard Deviation

			Weak Scalability					Strong Scalability				
Framework			**50**	**100**	**150**	**200**	**250**	**50**	**100**	**150**	**200**	**250**
LineString		GeoMatch	0.01	0.02	0.01	0.02	0.02	0.02	0.04	0.03	0.02	0.02
	No Error Corr.	GeoSpark	0.03	0.05	0.14	0.37	0.28	0.39	0.34	0.11	1.18	0.90
		LocationSpark	-	-	-	-	-	-	-	-	-	-
		STARK	0.09	0.21	0.50	0.29	6.08	1.52	3.91	19.61	17.83	15.83
Rectangle		GeoMatch	0.01	0.02	0.01	0.01	0.03	0.03	0.03	0.05	0.01	0.02
	With Err. Corr.	GeoSpark	0.11	0.10	0.20	0.18	0.08	0.47	0.19	0.99	1.50	0.24
		LocationSpark	-	-	-	-	-	-	-	-	-	-
		STARK	0.31	0.15	0.12	0.28	0.92	0.48	0.33	6.88	21.14	0.20

A value of "-" indicates that either the spatial object is not supported or the task failed to complete.

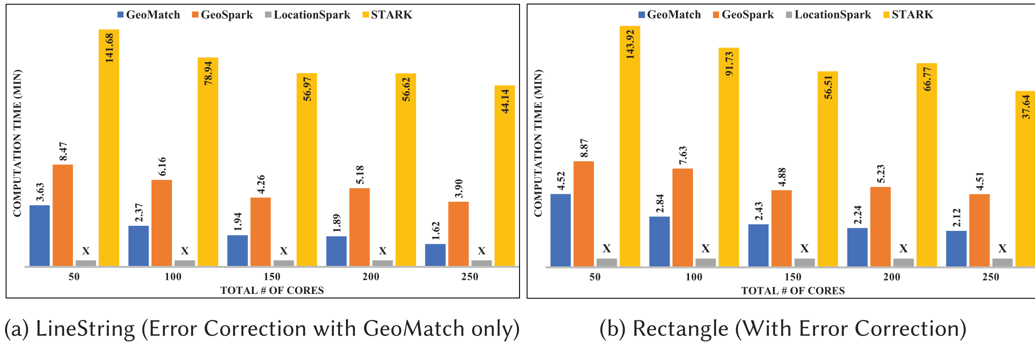
0.02–0.11 minute. For GeoSpark, ranges were 0.24–0.85 and 0.50–1.34. For STARK the ranges were 0.30–17.12 and 2.21–11.03.

6.4 Bus Trips dataset (BUS)

In this experiment, we match points from the BUS dataset with streets from the NYC LION street dataset and report the average runtimes. This dataset is close in size to the SMALL dataset but covers half of the NYC streets and has a higher frequency of GPS pings along the bus routes. Testing with this dataset gives an insight as to how the framework handles query skews when certain indexes have significantly higher number of points than others. As with the SMALL dataset, streets are first represented as LineStrings then as Rectangles. Matches with distances larger than $150ft$ are removed before the final results are produced.

Weak scalability test: The input size and the number of processing cores are gradually increased as described earlier. Figure 11 shows the average runtimes in minutes of these repeated experiments and Table 8 shows their standard deviation.

When representing streets as LineString objects, GeoMatch completed all tests first while producing a complete output dataset and accounting for GPS errors. Its runtimes remained relatively close across all experiments with a standard deviation of 0.08 minute. GeoSpark finished second with runtimes slightly less close to each other and a standard deviation of 0.43 minute. STARK finished last with runtimes increasing over the first three experiments, then decreasing for the



(a) LineString (Error Correction with GeoMatch only) (b) Rectangle (With Error Correction)

Fig. 12. BUS dataset – strong scalability test (logarithmic scale).

final two. Its runtimes standard deviation was 5.16 minutes. Runtimes for LocationSpark are not presented for lack of support for LineString objects.

When representing streets as Rectangle objects, GeoMatch completed all tests first while, again, producing a complete output dataset. Its runtimes remained relatively close across the experiments with a standard deviation of 0.12 minute. GeoSpark finished second with standard deviation for its runtimes was 0.29 minute. STARK finished last with runtimes slightly *increasing* except for the last experiment with 250 cores. This suggests poor scalability when managing highly skewed datasets. Its standard deviation for these experiments was 3.75 minutes. Runtimes for LocationSpark are not provided, because its tests either failed or were stopped for exceeding the 180-minute limit.

The standard deviation ranges of repeated experiments for both representations for this experiment were 0.01–0.02 minute and 0.01–0.03 minute for GeoMatch, 0.03–0.37 minute and 0.08–0.20 minute for GeoSpark, and 0.09–6.08 minutes and 0.12–0.92 minute for STARK.

Strong scalability test: The input size is fixed to the entire BUS dataset while gradually increasing the processing cores from 50 to 250 (in steps of 50). Figure 12 shows the average runtimes in minutes for all experiments using two street representations, and Table 8 shows their standard deviation.

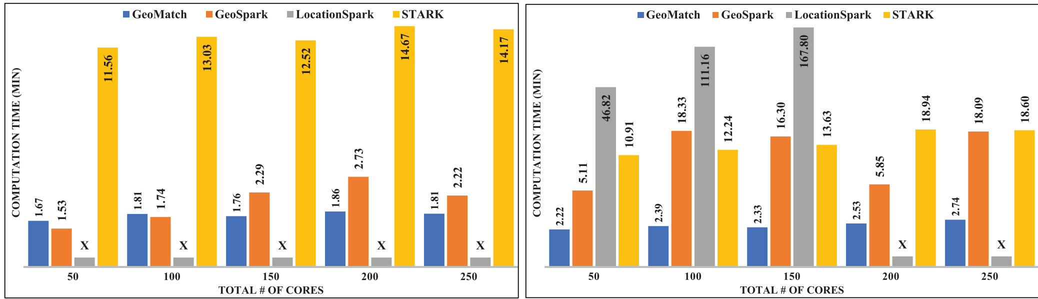
When representing streets as LineString objects, GeoMatch completed all tests first while producing a complete output dataset. Its runtimes gradually decreased as the number of processing cores were increased, indicating proper utilization of available resources. The standard deviation of GeoMatch's runtimes in these experiments were 0.71 minute. GeoSpark finished second with decreasing runtimes that took more than twice as long as GeoMatch. The standard deviation of its runtimes was 1.64 minutes. STARK finished last with runtimes decreasing, and the standard deviation was 34.86 minutes.

When representing streets as Rectangle objects, the order of task completion and performance were similar. GeoMatch's standard deviation was 0.88 minute; GeoSpark's standard deviation was 1.72 minutes. STARK's standard deviation was 36.72 minutes. Runtimes for LocationSpark are not provided either due to failure or termination.

The standard deviation ranges of repeated experiments for both representations for this experiment were 0.03–0.04 minute and 0.01–0.05 minute for GeoMatch, 0.11–1.18 minutes and 0.19–1.50 minutes for GeoSpark, and 1.52–19.62 minutes and 0.20–21.14 minutes for STARK.

6.5 Large Taxi dataset (LARGE)

In this experiment, we match points from the LARGE dataset with streets from the NYC LION street dataset and report the average runtimes. As this dataset is the largest, it requires better



(a) LineString (Error Correction with GeoMatch only) (b) Rectangle (With Error Correction)

Fig. 13. LARGE dataset – weak scalability test (logarithmic scale). Runtime for LocationSpark (grey bar X) not provided due to (a) the lack of support of LineString objects, (b) consistent fails or over 180-minute limit.

Table 9. LARGE Dataset– Standard Deviation

			Weak Scalability					Strong Scalability				
		Framework	**50**	**100**	**150**	**200**	**250**	**50**	**100**	**150**	**200**	**250**
LineString		GeoMatch	0.02	0.02	0.02	0.01	0.01	0.20	0.12	0.31	0.13	0.63
	No Error Corr.	GeoSpark	0.05	0.07	0.18	0.40	0.05	-	-	-	-	-
		LocationSpark	-	-	-	-	-	-	-	-	-	-
		STARK	0.47	0.18	1.38	1.00	0.14	-	-	-	-	-
Rectangle		GeoMatch	0.04	0.03	0.01	0.03	0.04	0.32	0.19	0.45	0.24	0.16
	With Err. Corr.	GeoSpark	0.16	4.46	2.79	0.07	0.75	-	-	-	-	-
		LocationSpark	16.20	31.89	-	-	-	-	-	-	-	-
		STARK	0.33	0.12	1.06	5.02	4.22	-	-	-	-	-

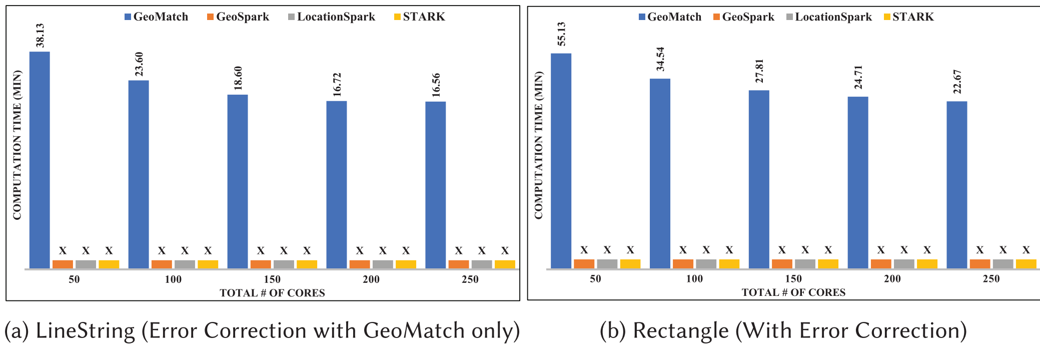
A value of "-" indicates that either the spatial object is not supported or the task failed to complete.

scalability than the previous datasets. As with the previous dataset, streets are first represented as LineStrings, then as Rectangles. Matches with distances large than 150ft are removed before the final results are produced.

Weak scalability test: The input size and the number of processing cores are gradually increased, as described earlier. Figure 13 shows the average runtimes in minutes of these repeated experiments and Table 9 shows their standard deviation.

When representing streets as LineString objects, GeoSpark finished slightly ahead of GeoMatch in the first two experiments only and was outperformed by GeoMatch when the number of cores increased. GeoMatch's runtimes were relatively closer to one another than those of GeoSpark; the standard deviation was 0.06 minute for GeoMatch and 0.42 minute for GeoSpark. STARK finished last with varying runtimes and standard deviation of 1.12 minutes. Runtimes for LocationSpark are not presented for lack of support for LineString objects.

When representing streets as Rectangle objects, GeoMatch completed all tests first while, again, producing a complete output dataset and accounting for GPS errors. Its runtimes remained relatively close across the experiments with a standard deviation of 0.18 minute. STARK finished second in the second (100 cores) and third (150 cores) experiments; GeoSpark finished second in the remaining experiments. The standard deviation was 3.30 minutes for STARK's experiments and 5.97 minutes for GeoSpark. LocationSpark was able to finish the first three experiments (50–150 cores) and failed the last two. Its runtimes standard deviation was very high at 49.42 minutes.



(a) LineString (Error Correction with GeoMatch only)

(b) Rectangle (With Error Correction)

Fig. 14. LARGE dataset – strong scalability test (logarithmic scale).

The standard deviation ranges of repeated experiments for both representations for this experiment were 0.01–0.02 minute and 0.01–0.04 minute for GeoMatch, 0.05–0.4 minute and 0.07–4.46 minutes for GeoSpark, and 0.14–1.38 minutes and 0.12–5.02 minutes for STARK.

Strong scalability test: The input size is fixed to the entire LARGE dataset while gradually increasing the processing cores from 50 to 250 (in steps of 50). GeoMatch was the only technique that successfully processed the dataset, while the other frameworks consistently failed or were stopped once exceeding the 180-minute limit. Figure 14 shows the average runtimes in minutes for all experiments using two street representations and Table 9 shows their standard deviation.

When representing streets as LineString objects, GeoMatch completed all tests while producing a complete output dataset and accounting for GPS errors. Its runtimes gradually decreased as the number of processing cores were increased, indicating, again, proper utilization of available resources. The time it took GeoMatch to complete the first test using 50 cores was noticeable 1.62× higher than its next highest runtime due to the small number of processing cores compared to the input dataset. The standard deviation was 8.11 minutes.

When representing streets as Rectangle objects, the results were achieved with higher runtimes than those of the weak scalability due to the larger size of the rectangle object. However, the behavior was the same: Runtimes decreased as the number of processing cores were increased. The standard deviation was 11.79 minutes.

The standard deviation ranges of repeated experiments for GeoMatch using the two different representations of LineString and Rectangle were 0.12–0.63 minute and 0.16–0.45 minute, respectively.

6.6 Output Accuracy

Using our accuracy benchmark framework described in Section 4, we compared the output results of GeoSpark, STARK, and GeoMatch to those of the baseline method. We set the parameter $n = 3$, which classifies a record as *Correctly Matched* if at least one of the technique's first three matches agrees with the first three matches in the baseline method. To achieve a more meaningful comparison, we report on map matching results for techniques that complete the strong scalability test (i.e., the entire input dataset). For each dataset, we report on the accuracy when using LineString street representations, then using Rectangles street representation. The results show that GeoMatch produces the most accurate results due to its ability to *natively* take error-correction measures using both representations.

SMALL Dataset: Table 10 shows the accuracy reports using LineString and Rectangle objects. When using LineString object representation of streets GeoMatch ranked best with error

Table 10. Accuracy Comparison – SMALL Dataset

Framework			Correct Matches	Neither Matched	FW Only Matched	FW Failed Matches	FW Over-match	FW under-matched	Excluded Records
LineString		GeoMatch	99.96%	2.37%	0.00%	0.02%	0.00%	4.03%	0.00%
	No Error Corr.	GeoSpark	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	100.00%
		LocationSpark	-	-	-	-	-	-	-
		STARK	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	100.00%
Rectangle		GeoMatch	87.95%	1.46%	0.91%	0.00%	14.50%	0.00%	0.00%
	With Err. Corr.	GeoSpark	42.39%	0.00%	0.22%	0.00%	9.74%	0.00%	51.36%
		LocationSpark	-	-	-	-	-	-	-
		STARK	80.10%	0.00%	0.48%	0.00%	13.91%	0.00%	1.89%

A value of "-" indicates that either the spatial object is not supported or the task failed to complete.

Table 11. Accuracy Comparison – BUS Dataset

Framework			Correct Matches	Neither Matched	FW Only Matched	FW Failed Matches	FW Over-match	FW under-matched	Excluded Records
LineString		GeoMatch	99.99%	0.58%	0.00%	0.00%	0.00%	1.28%	0.00%
	No Error Corr.	GeoSpark	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	100.00%
		LocationSpark	-	-	-	-	-	-	-
		STARK	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	100.00%
Rectangle		GeoMatch	82.37%	0.43%	0.15%	0.00%	9.46%	0.00%	0.00%
	With Err. Corr.	GeoSpark	2.66%	0.00%	0.00%	0.00%	0.31%	0.00%	96.73%
		LocationSpark	-	-	-	-	-	-	-
		STARK	75.31%	0.00%	0.05%	0.00%	7.32%	0.00%	0.53%

A value of "-" indicates that either the spatial object is not supported or the task failed to complete.

correction with 99.96% correct matches, 0.00% invalid (FW Only Matched) and overmatched records, and a complete dataset (0.00% Excluded Records). STARK and GeoSpark completed their map matching tasks but failed to match any records correctly due to their inability to account for GPS errors.

When using Rectangle object representation, the MBR is expanded to account for GPS errors. GeoMatch ranked best with 87.95% correct matches, 0.91% invalid matches, 14.50% overmatched records, and a complete dataset. The lower accuracy is attributed to the inaccurate representation of streets (Rectangle vs. LineString) and the distance computation that uses the Rectangle's centroid. STARK ranked second with 80.10% correct matches, 0.48% invalid matches, 13.91% overmatched records, and 1.89% excluded records. GeoSpark ranked last with 42.39% correct matches, 0.22% invalid matches, 9.74% overmatched records, and excluded more than half of the input dataset.

BUS Dataset: Table 11 shows the accuracy reports using LineString and Rectangle objects. When using LineString object representation of streets, the results were similar to the SMALL dataset. GeoMatch ranked best with 99.99% correct matches, 0.00% invalid matches, 1.28% overmatched records, and a complete dataset. STARK and GeoSpark completed their map matching tasks but failed to match any records correctly due to their inability to account for GPS errors.

When using Rectangle object representation, GeoMatch ranked best with 82.37% correct matches, 0.15% invalid matches, 9.46% overmatched records, and a complete dataset. The lower

Table 12. Accuracy Comparison – LARGE Dataset

Framework			**Correct Matches**	**Neither Matched**	**FW Only Matched**	**FW Failed Matches**	**FW Over-match**	**FW under-matched**	**Excluded Records**
LineString		GeoMatch	99.66%	4.21%	0.00%	0.32%	0.00%	5.23%	0.00%
LineString	No Error Corr.	GeoSpark	-	-	-	-	-	-	-
LineString	No Error Corr.	LocationSpark	-	-	-	-	-	-	-
LineString	No Error Corr.	STARK	-	-	-	-	-	-	-
Rectangle		GeoMatch	80.23%	2.23%	1.98%	0.00%	15.68%	0.00%	0.00%
Rectangle	With Err. Corr.	GeoSpark	-	-	-	-	-	-	-
Rectangle	With Err. Corr.	LocationSpark	-	-	-	-	-	-	-
Rectangle	With Err. Corr.	STARK	-	-	-	-	-	-	-

A value of "-" indicates that either the spatial object is not supported or the task failed to complete.

accuracy is, again, attributed to the inaccurate representation of streets. STARK ranked second with 75.31% correct matches, 0.05% invalid matches, 7.32% overmatched records, and 0.53% excluded records. GeoSpark ranked last with 2.66% correct matches, 0.00% invalid matches, 0.31% overmatched records, and excluded almost the whole input dataset (96.73%).

LARGE Dataset: Table 12 shows the accuracy reports using LineString and Rectangle objects. When using LineString object representation of streets GeoMatch produced a complete dataset with 99.66% correctly matched records as compared to the baseline results. Unlike the previous datasets, it showed 0.32% invalid record matches and 5.23% undermatched records, which will require further improvements to the matching process mainly with the line rasterization algorithm, which fails to account for points located at the edge of the Hilbert cell. GeoSpark, LocationSpark, and STARK results are not available, since neither one was able to process the entire dataset.

When using Rectangle object representation, GeoMatch produced a complete dataset with 80.23% correct matches, 1.98% invalid matches, 15.68% overmatched records, and a complete dataset. The lower accuracy is, again, attributed to the improper representation of streets as Rectangles and hence the inaccurate distance calculation.

7 APPLICATIONS

To demonstrate the practical benefits of GeoMatch, we applied GeoMatch to two existing real-world urban management applications. These applications were implemented on top of the datasets described in Section 5, and were in collaborations with New York City agencies. Without GeoMatch, as our experiments have shown, the applications would struggle processing these datasets at scale. GeoMatch not only makes the analysis more efficient, but ensures performance constraints do not compromise accuracy or correctness of the analysis.

7.1 Unmet Taxi Demand

Unmet Trip Demand is a situation when a passenger would like to take a taxi, but is unable to get one for an extended period of time. In collaborations with New York Taxi and Limousine Commission, NYU-CUSP has helped the city with three types of analyses to infer unmet taxi demand: (i) identify the most under-served locations; (ii) explore contributing factors; (iii) and develop a model to predict demand in under-served areas across NYC [2]. This was done using the high-frequency GPS locations of the taxis, which consists of over 4B records. GeoMatch enabled the matching of the GPS data with street segments in a timely fashion before proceeding with the pipeline below.

MapReduce on Taxi Record Counts Using GeoMatch: Unmet taxi demand operates using the yellow and green taxi TLC TPEP and LPEP data (Large) described in Section 5. These data include geographic coordinates of all taxis every two minutes and occupancy state of the vehicle. The large volume of data was stored on a Hadoop Distributed File System, and GeoMatch was used to preprocess the data as shown in Section 6.5. We then implement map and reduce jobs to infer pickups, dropoffs, and vacancies as follows:

- For pickups, a record is counted in the corresponding time slot when a taxi's passenger number increases from 0 to a number greater than 0.
- For dropoffs, a record is counted when a taxi's passengers drops to 0.
- For vacant taxis, a record is counted when a taxi's passenger number is 0.

After performing the mapping and reducing, the data format becomes the street IDs with the associated closest targeted counts in each evaluation duration for each of the tasks above. These intermediate products are then merged with our street dataset, which included street IDs, and for each street the corresponding IDs of the census tracts on both sides of the street. Therefore, for each street ID, the corresponding targeted counts were added in both associated census tracts.

Inferring Unmet Taxi Demand: If there are many free taxis roaming in a region, then there is no unmet demand in that region. We evaluate the frequencies of vacant taxis passing through each census tract for a given evaluation duration. These computations were performed on the TPEP and LPEP data of January 2015. Counts were divided into 11 batches showing the number of records with free status across NYC census tracts in each evaluation duration.

Based on the features of the yellow/green cab breadcrumb data, each ping is recorded every two minutes; therefore, it can be assumed that whenever a free taxi status is captured, the taxi has already been free for the past two minutes (under the optimal condition where everything is evenly distributed). This means that for each record of a free taxi status, it indicates availability for that two minutes. We treated each free status independently, disregarding the associated taxi IDs, as if every record represented a different taxi, then the corresponding free minutes is equal to the record interval, and the total free minutes can be represented as follows:

$$Total\ Free\ Taxi\ Minutes = \sum_{t \in T} (FreeCount_t \times RecordInterval_t).$$

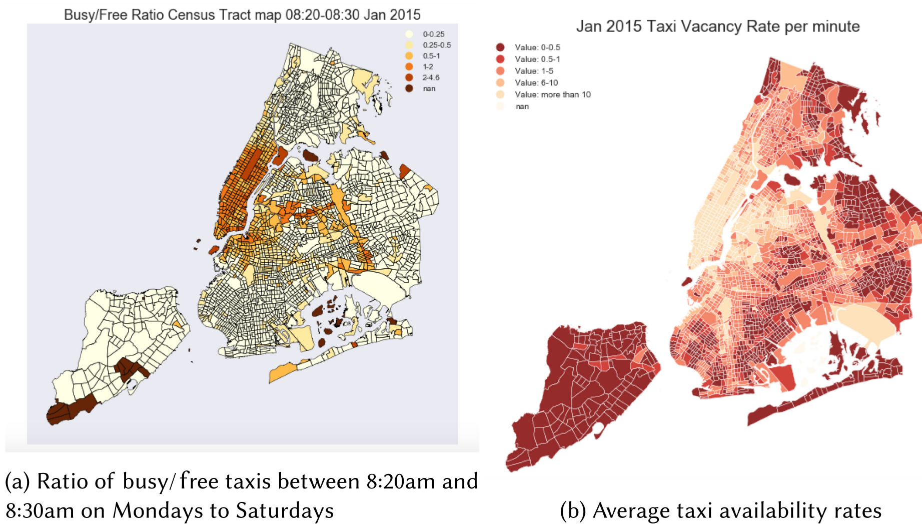
In the equation, T is the evaluation duration examined, and t is the moment of a pinned record within duration T. The record interval is a constant equal to two minutes. Therefore, the average free taxi availability rate is defined as the following formula:

$$Free\ Taxi\ Availability\ Rate = \frac{Total\ Taxi\ Free\ Minutes}{Evaluation\ Duration}.$$

If the availability rate is equal or more than 1, it means that there is always at least 1 vacant taxi available in every minute throughout the evaluation period in the census tract. Regions with high unmet trip demands are expected to have low availability rates.

We examine the unmet taxi demand by examining both the busy/free taxi ratio and the taxi availability rates. We visualize the busy/free ratio in Figure 15(a) to evaluate possible unmet demand as the relationship between free taxis and busy taxis (without and with passengers, respectively). The majority of Manhattan has higher busy/free ratios in the selected 10 minute time interval between 8:20am and 8:30am from Mondays through Saturdays. It means that there are more busy taxis than free taxis during the peak time weekdays and probably there is a high unmet taxi demand.

However, when applying to the whole dataset ignoring these special cases such as peak times before and after work, we observe that the taxi demand is fully served in the city center in NYC.



(a) Ratio of busy/free taxis between 8:20am and 8:30am on Mondays to Saturdays

(b) Average taxi availability rates

Fig. 15. Unmet taxi demand inferences in NYC [2].

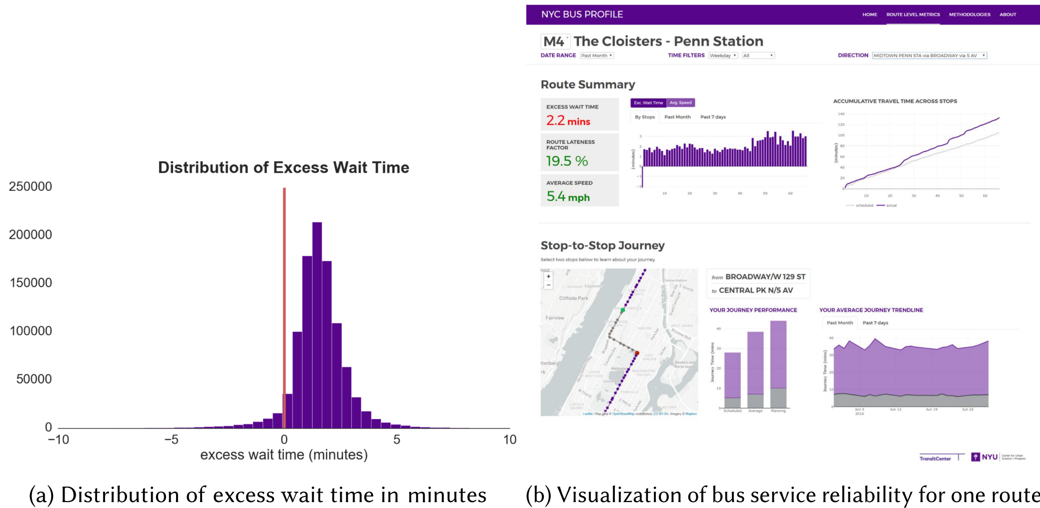
Instead, it is these outer boroughs of NYC having high unmet taxi demand. Figure 15(b) demonstrates the taxi availability rates for each census tract of the whole month. Regions in light colors (orange and cream), such as midtown and downtown Manhattan, are places where there is at least one vacant taxi available during every minute of the evaluation period. Areas in dark colors (red and burgundy) have fewer vacant taxis roaming, so unmet trip demands are more likely to exist in these regions. Our unmet taxi demand analysis results has helped the New York City Taxi and Limousine Commission to "fill in the gaps left by taxi records," thus potentially improving their taxi services to citizens living in NYC outer boroughs [22].

7.2 Bus Service Reliability Analysis

Bus ridership has been on a decline in New York City (NYC) even as population has increased, in part due to perceived unreliability, slow travel times, and an increase in other options. Advocacy groups such as the TransitCenter have pushed for a public-facing dashboard that reports objectively and transparently on performance trends. The NYC Metropolitan Transportation Authority collects and publishes raw data with bus locations. In collaboration with the TransitCenter, NYU-CUSP has examined the possibility of transforming the raw data to evaluate long-term bus reliability performance. We identified a set of metrics that would matter to travelers and visualized the results in an accessible dashboard [8]. However, the proof-of-concept dashboard[12] can only handle a limited amount of data (one month) due to the map matching performance. GeoMatch has allowed the framework to scale to the full historical dataset when holistically analyzing the city bus performance.

Bus Service Reliability. We define bus service reliability as the invariability of service attributes that influence the decisions of travelers and transportation providers [1]. This definition has been expanded to encompass schedule adherence, where increased deviation between scheduled and actual bus arrival times indicate poorer adherence.

[12] http://www.busstat.nyc/.



(a) Distribution of excess wait time in minutes (b) Visualization of bus service reliability for one route

Fig. 16. Bus reliability analysis in NYC [8].

Given this definition, we evaluate bus reliability in NYC using Excess Wait Time (EWT) at the route and borough levels, in line with recommendations by Trompet et al. [30]. This metric can be defined as the difference between scheduled wait times (SWT) and actual wait times (AWT) given a random distribution of arrivals. In the case of this random distribution of arrivals, average wait time is equal to half of the headway between buses. It is given by the following equation:

$$EWT = SWT - AWT,$$

where

$$AWT = \frac{\sum_{i=1}^{N} AHway_i^2}{2 * \sum_{i=1}^{N} AHway_i}, and$$

$$SWT = \frac{\sum_{i=1}^{N} SHway_i^2}{2 * \sum_{i=1}^{N} SHway_i},$$

$Hway_i$ represents the ith headway between buses and N is the total number of headways in one route.

We use GeoMatch to map match bus location data against the street map for route RBT analysis, as shown in Section 6.4. This yields locations on streets, ready for large-scale timetable comparison and reliability visualization. We plot the distribution of EWT in Figure 16(a). We found that excess wait time was 1.76 minutes on average, with Manhattan performing the worst at 1.93 minutes and Staten Island performing the best at 1.53 minutes. Staten Island also had the fastest buses with an average speed 8.38 mph, while Manhattan buses were slowest with an average of 5.62 mph. To further analyze the cause of the excess wait time, we also calculate the average speed in NYC. We observe that the average speed could vary greatly on different routes and even route segments, with buses in Manhattan yielding the lowest speeds and bus segments on arterial streets in the outer boroughs moving much more quickly. Overall, the average speed was 7.41 mph, which is consistent with TransitCenter's findings in 2014.[13]

[13] http://transitcenter.org/publications/whos-on-board-2014/.

8 DISCUSSION

Performance: GeoMatch has superior performance to all of the techniques discussed in this article, including the baseline. Using proper object representation, the baseline method can match NYC streets to the SMALL, BUS, and LARGE datasets in 3.31, 5.85, and 71.5 minutes, respectively. GeoMatch can complete the same task with near 100% accuracy using proper object representation and error correction in a fraction of the time, namely, 1.62, 1.61, and 16.56 minutes, respectively. The other studied frameworks either failed to process the datasets or required more time than the baseline method. For those that were able to process the datasets, a benchmark of their accuracy results ranked them poorly especially when using LineString representations.

Index Accuracy: In rare cases, GeoMatch can fail to find the optimal match if the currently matched point is located on the edge of its Hilbert cell and the optimally matching road passes through a neighboring cell. The failure rate in our experiments was rare; namely, 0.02%, 0.00%, and 0.32% for the SMALL, BUS, and LARGE datasets, respectively. A potential way to remove these errors is to use a secondary index with coarser granularity (e.g., lower-order Hilbert index) for cases where the point is close to a boundary and the shortest distance to a road is higher than the distance between the point and the boundary of the index cell.

Partitioning: Currently, GeoMatch aims to balance partitions, but allows larger partitions to keep points of the same index together. We demonstrated that this was sufficient for datasets containing 3.78B points, but when the dataset size increases, further optimization may be needed. When sufficiently many computing nodes are available, we can construct a secondary nested index for partitions with a large number of points and spatial objects. Alternatively, when fewer computing nodes are available, the optimal solution is to increase the order of the Hilbert curve to achieve a better-balanced distribution of computations.

Trajectory Matching: Our experiments focused exclusively on matching the GPS measurements against the street network, whereas many applications of map matching require matching complete movement trajectories. As discussed, we focused on this step, as it is the most time- and resource-consuming part of map matching, and as finding the optimal trajectory can be performed efficiently once the candidate road segments have been found. Indeed, the first phase of map matching effectively requires comparing each measurement against each road segment, whereas the second phase simply requires finding the best path through the best N matches of each point.

Spatial Frameworks: GeoMatch has been designed for supporting large-scale map matching instead of being a fully fledged spatial framework. Hence, currently, only a limited set of spatial objects, operations, and coordinate formats are supported. Compared to the previous version, we have improved the ability of GeoMatch to support new objects and operations. The next release already includes support for Point, LineString, Rectangle, and Polygon spatial objects and a *spatialJoinKNN* and *JoinDistance* operation. We plan to continue improving GeoMatch and extend its support to include other spatial operations and data structures.

Routing: GeoMatch implements the first step in a spatial analysis pipeline; namely, transforming individual location points to traversed streets in the street network. This is a necessary part of analyzing trajectories, such as taxi trips and bus journeys, and optimizing the path they take within the road network. This has implications on transport planning, such as where taxi pickups happen and which routes buses should take, as well as private transport, giving drivers optimal routes to take according to the time of day and congestion conditions of the road network.

9 SUMMARY

We presented GeoMatch, an accurate, scalable, and fast map matching framework for very large spatial datasets, based on Apache Spark. We demonstrated the performance and accuracy of GeoMatch through rigorous and extensive benchmarks that considered datasets containing large-scale urban spatial datasets ranging from 166,253 to 3.78B location measurements. In our benchmark experiments, GeoMatch had up to 27.25-fold runtime improvement to state-of-the-art large-scale spatial data processing frameworks. We described two real-world applications where GeoMatch was utilized. Both GeoMatch and our benchmark framework are available as open-source.

REFERENCES

[1] Mark Abkowitz et al. 1978. *Transit Service Reliability.* Technical Report. Cambridge, MA.

[2] Anita Ahmed, Alexey Kalinin, Pooneh Famili, Xin Tang, Ziman Zhou, Kaan Ozbay, and Huy Vo. 2017. *Predicting Unmet Trip Demand.* Technical Report. In *2016 IEEE International Conference on Big Data (Big Data).* 833–842.

[3] Ablimit Aji, Fusheng Wang, Hoang Vo, Rubao Lee, Qiaoling Liu, Xiaodong Zhang, and Joel H. Saltz. 2013. Hadoop-GIS: A high performance spatial data warehousing system over MapReduce. *PVLDB* 6, 11 (2013), 1009–1020.

[4] A. M. R. Almeida, M. I. V. Lima, J. A. F. Macedo, and J. C. Machado. 2016. DMM: A distributed map-matching algorithm using the MapReduce paradigm. In *Proceedings of the IEEE 19th International Conference on Intelligent Transportation Systems (ITSC'16).* 1706–1711.

[5] Jie Bao, Ruiyuan Li, Xiuwen Yi, and Yu Zheng. 2016. Managing massive trajectories on the cloud. In *Proceedings of the 24th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (SIGSPATIAL'16).* Association for Computing Machinery, New York, NY.

[6] W. J. Bouknight. 1969. *An Improved Procedure for Generation of Half-tone Computer Graphics Presentations.* Coordinated Science Laboratory Report no. R-432 (1969).

[7] Jack E. Bresenham. 1965. Algorithm for computer control of a digital plotter. *IBM Syst. J.* 4, 1 (1965), 25–30.

[8] Hongting Chen, Francis Ko, Shay Lehmann, Nurvita Monarizqa, Ian Wright, Kaan Ozbay, and Huy Vo. 2017. *Performance Analysis and Tracking for NYC's Transit System.* Technical Report. In Center for Urban Science and Progress (CUSP), New York University.

[9] Ahmed Eldawy. 2014. SpatialHadoop: Towards flexible and scalable spatial processing using MapReduce. In *Proceedings of the SIGMOD PhD Symposium (SIGMOD'14 PhD Symposium).* ACM, New York, NY, 46–50.

[10] Ahmed Eldawy, Mohamed F. Mokbel, et al. 2016. The era of big spatial data: A survey. *Found. Trends® Datab.* 6, 3–4 (2016), 163–273.

[11] Stefan Hagedorn, Philipp Götze, and Kai-Uwe Sattler. 2017. The STARK framework for spatio-temporal data analytics on Spark. In *Datenbanksysteme für Business, Technologie und Web (BTW'17).* 123–142.

[12] Yaobin He, Haoyu Tan, Wuman Luo, Shengzhong Feng, and Jianping Fan. 2014. MR-DBSCAN: A scalable MapReduce-based DBSCAN algorithm for heavily skewed data. *Front. Comput. Sci.* 8, 1 (2014), 83–99.

[13] Yan Huang and Jason W. Powell. 2012. Detecting regions of disequilibrium in taxi services under uncertainty. In *Proceedings of the ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (SIGSPATIAL'12).* ACM, New York, NY, 139–148.

[14] Environmental Systems Research Institute. 2018. GIS Tools for Hadoop by Esri. Retrieved from http://esri.github.io/gis-tools-for-hadoop/.

[15] International Data Corporation (IDC). 2019. IDC Forecasts Revenues for Big Data and Business Analytics Solutions Will Reach $189.1 Billion This Year with Double-Digit Annual Growth through 2022. Retrieved from https://www.idc.com/getdoc.jsp?containerId=prUS44998419.

[16] Ibrahim Kamel and Christos Faloutsos. 1994. Hilbert R-tree: An improved R-tree using fractals. In *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB'94).* Morgan Kaufmann Publishers Inc., San Francisco, CA, 500–509.

[17] Scott T. Leutenegger, Mario A. Lopez, and Jeffrey Edgington. 1997. STR: A simple and efficient algorithm for R-tree packing. In *Proceedings of the 13th International Conference on Data Engineering.* IEEE, 497–506.

[18] Bin Li, Daqing Zhang, Lin Sun, Chao Chen, Shijian Li, Guande Qi, and Qiang Yang. 2011. Hunting or waiting? Discovering passenger-finding strategies from a large-scale real-world taxi dataset. In *Proceedings of the IEEE International Conference on Pervasive Computing and Communications (PerCom 2011).* IEEE, Los Alamitos, CA, 63–68.

[19] LocationTech. [n.d.]. LocationTech JTS Topology Suite. Retrieved from https://projects.eclipse.org/projects/locationtech.jts.

[20] Yin Lou, Chengyang Zhang, Yu Zheng, Xing Xie, Wei Wang, and Yan Huang. 2009. Map-matching for low-sampling-rate GPS trajectories. In *Proceedings of the 17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. ACM, 352–361.

[21] L. Moreira-Matias, J. Gama, M. Ferreira, J. Mendes-Moreira, and L. Damas. 2013. Predicting taxi-passenger demand using streaming data. *IEEE Trans. Intell. Transport. Syst.* 14, 3 (Sept. 2013), 1393–1402.

[22] New York City Taxi and Limousine Commission. 2020. TLC Mentors Students Using Big Data. Retrieved from https://medium.com/@NYCTLC/students-use-tlc-data-to-study-unmet-taxi-demand-and-find-ideal-spots-for-taxi-relief-stands-644e40ebe11a.

[23] Varun Pandey, Andreas Kipf, Thomas Neumann, and Alfons Kemper. 2018. How good are modern spatial analytics systems? *Proc. VLDB Endow.* 11, 11 (2018), 1661–1673.

[24] Douglas Alves Peixoto, Hung Quoc Viet Nguyen, Bolong Zheng, and Xiaofang Zhou. 2019. A framework for parallel map-matching at scale using spark. *Distrib. Parallel Datab.* 37, 4 (2019), 697–720.

[25] Meng Qu, Hengshu Zhu, Junming Liu, Guannan Liu, and Hui Xiong. 2014. A cost-effective recommender system for taxi drivers. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'14)*. ACM, New York, NY, 45–54. DOI: https://doi.org/10.1145/2623330.2623668

[26] S. Ruan, R. Li, J. Bao, T. He, and Y. Zheng. 2018. CloudTP: A cloud-based flexible trajectory preprocessing framework. In *Proceedings of the IEEE 34th International Conference on Data Engineering (ICDE'18)*. 1601–1604.

[27] P. Shimonti. 2015. What Is Geospatial Industry's Value and Impact in World Economy? Retrieved from https://www.geospatialworld.net/blogs/geospatial-industrys-value-world-economy/.

[28] Ram Sriharsha. 2018. Magellan: Geospatial Analytics Using Spark. Retrieved from https://github.com/harsha2010/magellan.

[29] Mingjie Tang, Yongyang Yu, Qutaibah M. Malluhi, Mourad Ouzzani, and Walid G. Aref. 2016. LocationSpark: A distributed in-memory data management system for big spatial data. *VLDB Endow.* 9, 13 (Sept. 2016), 1565–1568.

[30] Mark Trompet, Xiang Liu, and Daniel J. Graham. 2011. Development of key performance indicator to compare regularity of service between urban bus operators. *Transport. Res. Rec.* 2216, 1 (2011), 33–41.

[31] Hoang Vo, Ablimit Aji, and Fusheng Wang. 2014. 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 (SIGSPATIAL'14)*. ACM, New York, NY, 545–548.

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

[33] Simin You, Jianting Zhang, and Le Gruenwald. 2015. Large-scale spatial join query processing in cloud. In *Proceedings of the 31st IEEE International Conference on Data Engineering Workshops (ICDEW'15)*. IEEE, 34–41.

[34] Jia Yu, Jinxuan Wu, and Mohamed Sarwat. 2015. 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 (SIGSPATIAL'15)*. ACM, New York, NY.

[35] Jing Yuan, Yu Zheng, Chengyang Zhang, Wenlei Xie, Xing Xie, Guangzhong Sun, and Yan Huang. 2010. T-drive: Driving directions based on taxi trajectories. In *Proceedings of the 18th ACM SIGSPATIAL International Symposium on Advances in Geographic Information Systems (ACM-GIS'10)*. ACM, New York, NY, 99–108.

[36] Jing Yuan, Yu Zheng, Chengyang Zhang, Xing Xie, and Guang-Zhong Sun. 2010. An interactive-voting based map matching algorithm. In *Proceedings of the 11th International Conference on Mobile Data Management*. IEEE Computer Society, 43–52.

[37] Ayman Zeidan, Eemil Lagerspetz, Kai Zhao, Petteri Nurmi, Sasu Tarkoma, and Huy T. Vo. 2018. GeoMatch: Efficient large-scale map matching on Apache Spark. In *Proceedings of the IEEE International Conference on Big Data (BigData'18)*. IEEE, 384–391.

[38] Daqing Zhang, Nan Li, Zhi-Hua Zhou, Chao Chen, Lin Sun, and Shijian Li. 2011. iBAT: Detecting anomalous taxi trajectories from GPS traces. In *Proceedings of the 13th International Conference on Ubiquitous Computing (UbiComp'11)*. ACM, New York, NY, 99–108.

[39] Yu Zheng, Yanchi Liu, Jing Yuan, and Xing Xie. 2011. Urban computing with taxicabs. In *Proceedings of the 13th International Conference on Ubiquitous Computing (UbiComp'11)*. ACM, New York, NY, 89–98.

Received June 2019; revised March 2020; accepted May 2020