ST4ML: Machine Learning Oriented Spatio-Temporal Data Processing at Scale

KAIQI LIU, Alibaba-NTU Singapore Joint Research Institute, Nanyang Technological University, Singapore
PANRONG TONG, Alibaba-NTU Singapore Joint Research Institute, Alibaba DAMO Academy, China
MO LI, Alibaba-NTU Singapore Joint Research Institute, Nanyang Technological University, Singapore
YUE WU, Alibaba-NTU Singapore Joint Research Institute, Alibaba DAMO Academy, China
JIANQIANG HUANG, Alibaba-NTU Singapore Joint Research Institute, Alibaba DAMO Academy, China

Data scientists and researchers utilize enormous spatio-temporal data and build machine learning models to solve practical problems in diverse domains including intelligent transportation, urban planning, epidemic prediction, and many more. Extracting application-specific features from big spatio-temporal data poses system requirements of heterogeneous data support, efficient and scalable computing over spatial and temporal dimensions, as well as a user-friendly programming interface. This paper presents ST4ML, a distributed spatio-temporal data processing system to support scalable machine-learning-oriented applications. We propose a three-stage pipelining computing framework, namely *"selection-conversion-extraction"* to abstract the distributed computing flow and implement it based on Apache Spark. To the best of our knowledge, ST4ML is the first of its kind to realize our design considerations. Extensive experiments with real-world datasets evidence that ST4ML outperforms straightforward extensions of existing ST data processing systems by up to an order of magnitude. ST4ML is open-sourced at https://github.com/Panrong/st4ml.

CCS Concepts: • **Computing methodologies** → **Distributed algorithms**; • **Information systems** → **Spatial-temporal systems**.

Additional Key Words and Phrases: big data, spatio-temporal data, Spark, system for AI

ACM Reference Format:
Kaiqi Liu, Panrong Tong, Mo Li, Yue Wu, and Jianqiang Huang. 2023. ST4ML: Machine Learning Oriented Spatio-Temporal Data Processing at Scale. *Proc. ACM Manag. Data* 1, 1, Article 87 (May 2023), 28 pages. https://doi.org/10.1145/3588941

1 INTRODUCTION

Large volumes of spatio-temporal (ST) data (e.g., GPS samples, location-based video footages, and remote sensing data) are increasingly collected and studied in diverse domains, including human mobility [73, 74], intelligent transportation [33, 37], urban planning [11, 75], epidemiology [2, 15], as well as environmental and climate science [12, 67]. While recent advances in Machine

Authors' addresses: Kaiqi Liu, kaiqi001@e.ntu.edu.sg, Alibaba-NTU Singapore Joint Research Institute, Nanyang Technological University, 50 Nanyang Avenue, Singapore, 639798; Panrong Tong, panrong.tpr@alibaba-inc.com, Alibaba-NTU Singapore Joint Research Institute, Alibaba DAMO Academy, No.1008 Dengcai Street, Hangzhou, China, 310030; Mo Li, limo@ntu.edu.sg, Alibaba-NTU Singapore Joint Research Institute, Nanyang Technological University, 50 Nanyang Avenue, Singapore, 639798; Yue Wu, matthew.wy@alibaba-inc.com, Alibaba-NTU Singapore Joint Research Institute, Alibaba DAMO Academy, No.1008 Dengcai Street, Hangzhou, China, 310030; Jianqiang Huang, jianqiang.hjq@alibaba-inc.com, Alibaba-NTU Singapore Joint Research Institute, Alibaba DAMO Academy, No.1008 Dengcai Street, Hangzhou, China, 310030.

2836-6573/2023/5-ART87
https://doi.org/10.1145/3588941

Learning (ML), especially Deep Learning (DL) see great benefits in leveraging big ST data to facilitate model training and inference, transforming the raw ST data into ML ingestible features, however, still faces major challenges. Such a problem weakens the utility of the large amount of available ST data in supporting various ML applications.

ML with ST data is distinct from general ML applications in its need for external feature extraction and derivation. Conventional ML applications directly take raw data as input: in computer vision applications, images are directly fed to ML models as matrices of pixel values, and data from relational databases also do not need special processing as they are well-structured and their input to ML models is explicitly defined on themselves. On the contrary, ML applications with ST data rely on derived features from the original data - which we define as *"Spatio-Temporal Data Machine Learning (STDML)"* in this paper. For example, a traffic forecast model takes average traffic speeds at different road segments as input, which may need to be derived from the original ST data with a large set of vehicle trajectories. The necessity of transforming the ST data and extracting derived features comes from the fact that spatio-temporal analysis often engages the multiple correlated dimensions contained within the ST data and thus needs joint transformation of those dimensions. ST data possess three dimensions: (1) *spatial* dimension defining the spatial scattering of data samples in locations, (2) *temporal* dimension defining the scattering across time, and (3) *data* dimension containing domain-specific measurements like event type, data volume, and intensity value. A set of ST data are often organized according to the spatial, temporal, or both dimensions, with the consideration of external reference structures like road maps and predefined timetables. One or many application-specific features suitable for ML input must be derived and extracted before being fed to the ML models. For example, from a set of vehicle GPS trajectories, one may extract the average speed, the frequently visited places, and the flow count on road maps, which may thereafter be used for different *STDML* applications.

For ML tasks that concern ST data at scale (e.g., millions of vehicle trajectories involving billions of GPS samples), fast and efficient extraction of application-specific features is essential to the performance. To the best of our knowledge, however, no comprehensive study has been performed in exploring a scalable and distributed system design to address such a need. Existing research and engineering efforts on big ST data mainly focus on conventional *query* operations including spatial or ST range query, *k*-nearest neighbor (kNN) query, and distance join [16, 24, 28, 58, 66, 68, 69]. The queried results from those systems often lack structuring or transformation to ML-ingestible features, and are thus not suitable for direct input to *STDML* applications.

While in principle, one may pipeline systems for general-purpose distributed storage and computation (e.g., Apache HBase [6] for distributed data storage, GeoMesa [28] for ST indexing, and Apache Spark [72] for in-memory computation) to build an end-to-end solution to this problem, in practice such an approach is highly inefficient. The segregation of different systems impairs the potential of optimally processing the data with high parallelism. The output from upstream systems may not favor the downstream systems in terms of data structure and locality. Excessive processing time may be taken due to data conversions across different platforms, as well as data exchanges in memory or through system I/O. Cross-platform development also leads to extra programming efforts, i.e., application programmers have to shift among different systems while deploying and maintaining code in different programming languages (e.g., command-line interface (CLI) for GeoMesa ingestion, SQL for querying, and Scala for computation tasks with Spark).

This state of the art calls for a new distributed system design that can extract ML ingestible features from large-scale ST data *efficiently* and *conveniently*. The system should integrate with the underlying computation engine to generally support diverse ST data and ML applications. High parallelism and little data movement among the machines should take place. In particular, we believe the system design has to fulfill the following major requirements:

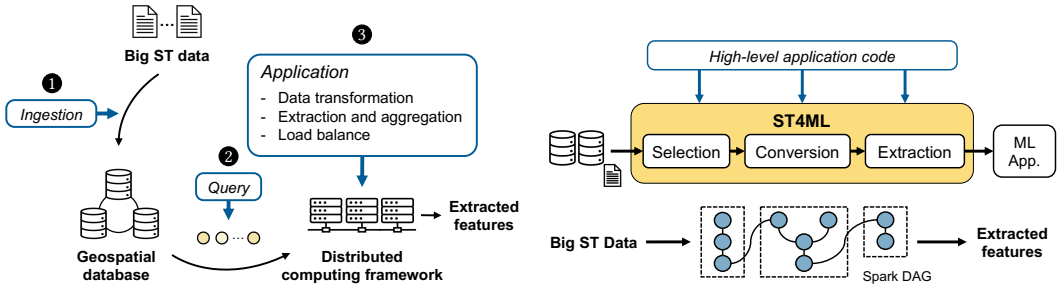
Support heterogeneous ST data. The system should provide a set of unified ST abstractions to support ST data with various physical meanings and representations. Those abstractions offer fundamental convenience for manipulating a data piece according to its spatial and temporal dimensions. In such a way multiple instances can be grouped based on the ST dimensions for collective processing.

Support efficient and scalable ST computation. Feature extractions for *STDML* involve complex computations over ST dimensions, all of which have to be efficiently executed in a scalable manner. The system needs to partition and replicate the ST data in a way that the workloads are evenly distributed across computing machines. If multiple ST-nearby data are involved in the computation (e.g., clustering and aggregating), they should locate on the same machine to ensure parallelism and reduce data transfer. Efficient data indexing in- and outside memory will greatly accelerate data retrieval and facilitate computations involving two or more large datasets.

Support user-friendly programming interface. The system should abstract a general and effective programming interface, in which most feature extraction applications can fit. Difficulties from distributed programming, such as task allocation and result aggregation, should be hidden from application programmers as much as possible. Since it is impossible to include all feature extraction functions in advance, the system needs to provide sufficient flexibility for application programmers to embed customized feature extraction logics, which the system can efficiently execute.

This paper presents ST4ML, a distributed ST data processing system to realize the above design considerations. We propose *Selection-Conversion-Extraction*, a three-stage pipelining computing paradigm, where various ML feature extraction problems can be abstracted and fit into. In the **Selection** stage, ST4ML retrieves an in-memory subset from gigantic on-disk ST data according to specified ST constraints (Section 3.1). ST datasets are of large scale while *STDML* applications are often applied on a portion of them. Loading all data into memory leads to a waste of memory and computation. A persistent metadata scheme is proposed, which groups and indexes on-disk ST data so only partial data are loaded into memory while the ST locality is preserved (Section 4.1). In-memory indexing is implemented for faster selection and multiple ST-partitioners are proposed to achieve ST-aware load balance during distributed computations. In the **Conversion** stage, ST4ML describes ST data with five ST instances: event, trajectory, time series, spatial map, and raster (Section 3.2). These instances provide representative abstractions of ST data and are suitable for different applications. Efficient conversions among the five ST instances are supported in ST4ML. The original ST data as one instance can be converted to the most appropriate instance according to the nature of the *STDML* applications. Specific optimizations are designed to speed up expensive conversions and benefit the computation pipeline (Section 4.2). In the **Extraction** stage, ST4ML executes feature extraction functions in parallel (Section 3.3). To provide different levels of flexibility, ST4ML pre-builds common extraction functions, supports application programmers to embed logics with instance-level APIs, as well as allows direct manipulation of RDDs. Such a paradigm transforms the ML feature extraction problem into scalable distributed executions, and makes the best use of the underlying distributed computing platform.

We implement ST4ML on top of Apache Spark [72] and evaluate its performance with extensive experiments based on public ST datasets of large scale. The results show that ST4ML significantly outperforms straightforward extensions of existing ST data processing systems including GeoMesa [28] and GeoSpark [69]. In end-to-end ML feature extraction applications, ST4ML performs up to 27× and 39× faster than the extended solutions based on GeoMesa and GeoSpark, respectively. We also adopt ST4ML in Alibaba's ongoing business and conduct case studies. With large-scale real-world data, the evaluation results show that even with the simple feature extraction logics in the current business, the adoption of ST4ML greatly outperforms the GeoSpark-based solution by up to 7×, which demonstrates the practical value of ST4ML in serving industry needs.



(a) ST feature extraction with existing systems.

(b) ST feature extraction with ST4ML.

Fig. 1. ST feature extraction workflows comparison. The blue boxes stand for the programming effort needed.

In summary, this paper makes the following contributions:

(1) We conduct the first thorough systematic study on extracting ML features from big ST data to facilitate *STDML* applications, which intrinsically differs from conventional ST query processing.

(2) We propose *"Selection-Conversion-Extraction"*, a three-stage paradigm to abstract the computing flow, which fully utilizes the power of the underlining distributed computing platform.

(3) We build ST4ML to implement the proposed framework based on Apache Spark, and devise a set of tailored optimization techniques, which to our knowledge is the first of its kind, and is proven efficient with extensive experimental evaluations.

2 BACKGROUND

2.1 Motivation

We take a typical application, traffic speed prediction, as an example to illustrate the existing gap in facilitating ML applications with big ST data. Dividing an urban area into grids, researchers take historical traffic speeds of each grid cell to train an ML (DL) model and predict the future speeds [33, 37]. The model input is usually formulated as a sequence of 2-d matrices, denoted by $[A^{t0}, A^{t1}, ...]$, where a matrix A^t records the traffic speeds of the grids at time t, and each element $a^t_{ij} \in A^t$ is the average speed of a grid cell. Since the actual traffic speed (across the grids and time slots) is often not directly available, researchers need to derive them from other attainable data, such as the trajectories of individual vehicles.

With existing ST computation support, a series of isolated data processing steps are needed for generating the ML input, as Figure 1a illustrates. The collected trajectory data are first imported into a geospatial database for management (❶). Application programmers often use CLI to ingest the data, as well as define the necessary attributes, indexing methods, etc. Each trajectory is represented as a linestring shape with an affiliated timestamp array and ID. To extract the ML features, trajectories within a specific ST range are queried with a SQL-like command (❷) and loaded into the distributed computing engine, e.g., Spark. Speed calculation needs sliding over consecutive sojourn points of a trajectory, which is not well supported by the representation used in the database. Therefore, the trajectories are reformatted by aligning their locations and timestamps, as Table 1 suggests. The left column displays an example of a typical vehicle trajectory in the database, while the right column displays the reformatted data with ST points. Last, the application programmer needs to program with the distributed computing framework to define the operations for deriving vehicle speeds and aggregating the average speeds across grids (e.g., programming over Spark RDDs in Scala (❸)). To achieve high efficiency, particular optimization over data placement and load balancing may also be taken.

Table 1. An example vehicle trajectory record and its reformatted version.

Original	Reformatted
ID: 6200589	ID: 6200589
Start time: 2013-07-01 0:00:58	Points:
Locations: (-8.618643, 41.141412), (-8.618499, 41.141376), ...	(-8.618643, 41.141412, 20130701 0:00:58),
Sampling rate: 15s	(-8.618499, 41.141376, 20130701 0:01:13) ...

Two apparent drawbacks exist: (1) stretching the solution over different databases and computing systems incurs overhead in data alignment and loses end-to-end opportunities in optimizing the computation flow; (2) application programmers face cross-system development which concerns programming with different languages, interfaces, and patterns. Both drawbacks significantly limit the efficiency and scalability when building practical applications at scale.

In this paper, we explore a unified computing framework to meet the efficiency and scalability requirements. Figure 1b illustrates the proposed three-stage pipelining paradigm. Revisiting the traffic speed prediction application, the relevant vehicle trajectories are first selected from the database. The application programmer indicates the ST ranges, based on which a load-balanced in-memory subset of trajectories will be generated. Second, an appropriately defined data structure is specified to reorganize the data, which in this example is ST raster. System-level support is provided for easy conversion of the original trajectories into an ST raster where each cell stores the relevant data from corresponding ST ranges. Third, the computing and aggregation logics for extracting the average speeds can be defined over the raster. The whole process is embedded into the distributed machines of Spark and executed in parallel. Such a unified computing framework enables joint optimizations: for each stage, not only can the executions be optimized for a shorter processing time, but the output can also be channeled to successive stages in terms of data locality and organization.

2.2 Spark Overview

Apache Spark [72] is a general-purpose in-memory distributed computing framework. Spark provides an efficient abstraction for in-memory calculation named Resilient Distributed Datasets (RDDs). An RDD is an immutable collection of Java objects, which are partitioned and distributed across a computer cluster. A Spark application is represented as a Directed Acyclic Graph (DAG), where each vertex is an RDD and the edges are predefined operations including `map`, `filter`, and `reduce`. After the DAG is generated and an action is triggered, Spark ships the operations to the worker nodes. Each worker node initializes several *executors* (i.e., processes) and each executor takes a portion of data (a *partition*) for parallel processing.

Building a Spark application requires the application programmer to have in-depth knowledge of functional programming and RDD operators. The performance relies on the application programmer's expertise as well. An application can be programmed with different sets of operators that lead to diverse performances. For example, `reduceByKey(_+_)` and `groupByKey.mapValues(_.sum)` return the same result but compute differently. The former operation performs a local aggregation and transfers the reduced results among machines, while the latter shuffles all data and computes slower. When the application becomes complicated, the choice and sequence of the operators notably affect the performance. Optimization strategies on maintaining load balance, reducing data shuffling, and increasing parallelism improve the performance yet requires advanced programming experience and the understanding of Spark rationale.

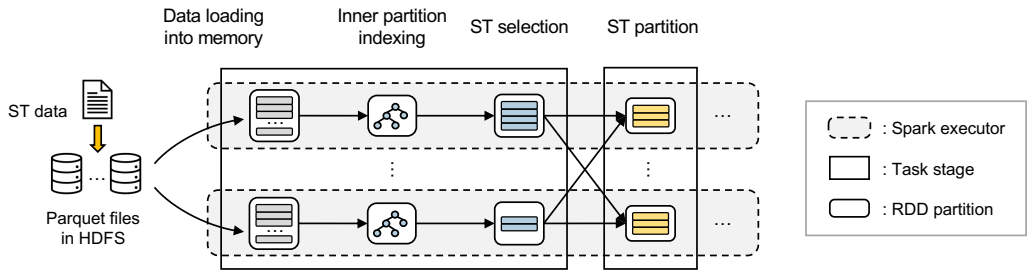


Fig. 2. Illustration of the Selection stage in ST4ML.

3 SYSTEM DESIGN

ST4ML is a three-stage pipelining framework built on top of Spark, as illustrated in Figure 1b. We see such a framework generally applicable to accommodate most feature extraction applications for *STDML*, where an application can be constructed in at most three stages with high-level operators. The **Selector** selects data subset of focus and loads them into memory; the **Converter** performs data instance conversion over Spark RDDs to organize ST data in the most suitable representation for the application; and the **Extractor** facilitates user-defined computing logics for feature extraction and aggregation. Those operators provide a unified easy-to-use interface for application programmers, and in the backend are optimized for high performance in the distributed setting.

3.1 Selection

ST4ML's selector loads ST data from persistent storage into memory, selects relevant data for the application, and distributes them in memory across the computer cluster. We implement per-partition R-tree indexing to efficiently select data based on ST requirements, and build multiple data partitioners to ensure load balance.

Preprocessing. ST4ML employs Parquet [7] files in HDFS [55] as its default data source to ensure high data loading efficiency. Several standard on-disk data structures, including `STEvent`, `STTraj`, and `STRaster` are defined. Application programmers are free to transform their datasets from external storage (e.g., cloud object storage and distributed NoSQL database) into ST4ML's data standard using their preferred methods. The preprocessing step incurs computation which however is one time off. In practical *STDML* scenarios, various features are usually extracted from the same set of data, where the one-off overhead is amortized across applications.[1]

After the data are transformed ST4ML-compatible, each time the application programmer writes an application, she may initiate a selector and select data as follows.

```
1  val selector = Selector[STTraj](cityArea, monthDuration,
2    index = false, partitioner = STRPartitioner(100))
3  val selectedRDD = selector.select(dataDir)
```

The usage of the selector generally applies to diverse applications, and we demonstrate it with the example of traffic speed extraction as provided in Section 2.1. In line 1, the application programmer constructs a selector by identifying the instance type (`STTraj`), as well as the spatial (`cityArea`) and temporal (`monthDuration`) ranges of interest. She may also toggle the optimization methods based on the application need (line 2). The application programmer then simply inputs the data directory to trigger the execution as in line 3.

[1]Existing systems are subject to similar preprocessing overhead, e.g., GeoMesa also performs data ingestion during preprocessing, and GeoSpark performs ad-hoc in-memory data ingestion for individual applications.

Behind the API, the selector composes multiple operations as shown in Figure 2, which can be split into two Spark stages: in the first stage the ST dataset is loaded into the memory pool of the computer cluster and each executor selects relevant data that locate on its local machine. In the second stage, ST partitioning is implemented to maintain load balance for successive operations. In conventional spatial query systems, a spatial partitioning is usually applied before indexing and querying [58, 66, 68, 69]. However, such a design is not suitable for our problem. The foremost partitioning leads to unbalance workload and data distribution: after spatial partitioning, only a portion of executors are invoked for the selection task while the rest are left idle. Consequently, the selected data only reside on a few machines. In ST4ML we instead utilize all computing power to select data and apply data partitioning afterward, so the following stages achieve high parallelism and take shorter processing time.

In the first Spark task stage, the selector loads data from the designated directory into the memory pool as an RDD. In parallel, the executors parse the data as ST instances (to be elaborated in Section 3.2.1). Filtering a large amount of data based on their ST attributes requires computations over shape and duration, which is expensive and may become the bottleneck as the data size increases. We employ the widely-used R-tree index [23] to reduce the complexity. A 3-d R-tree indexing the ST dimensions is built for every RDD partition on-the-fly, and the corresponding executor traverses the tree to select qualified ST data.

The selected data are randomly distributed over the cluster and may be unbalanced across partitions. ST4ML applies data partitioning in the second stage to (1) achieve load balance and (2) maintain ST proximity. Spark's native `repartition` function requires explicitly indicating a 1-d key for partitioning, and the resulting workload may still be imbalanced with skewed keys. Besides load balance, some extraction applications (e.g., hot spot extraction by event clustering) computes over multiple ST-nearby instances, so ST-proximity should be preserved after partitioning for performing as many computations locally as possible. Therefore, in ST4ML we build several ST-aware partitioners to serve different *STDML* applications. For applications where the ST-proximity is not concerned, we implement a new `Hash partitioner`, which uses the hash value of each data entry as the partition key to ensure randomness and load balance at the data record level. When the spatial-proximity should be preserved, ST4ML provides classic `STR partitioner` [32] and `Quad-tree partitioner` [53]. We also design a `T-balance partitioner` to partition the data by their temporal information with Spark's `approx_percentile` method. Moreover, we design a novel T-STR partitioner to preserve ST-proximity, which will be detailed in Section 4.1. A partitioner first samples data to calculate the partitioning boundaries, which takes much shorter time and only induces minor degradation in load balance. Next, each data entry is assigned to one or multiple partitions (based on the application need, e.g., whether duplication is required for correctness) with Spark's `flatMap` operation in parallel. Last, all data are shuffled among the worker machines, which possess balanced amounts of data.

3.2 Conversion

ST data can be organized from different perspectives to form different ST instances. For example, check-in records can naturally be viewed as events, and if grouped by the occurrence time, they can be viewed as a time series. Different *STDML* applications favor different data representations. If the desired representation does not align with the acquired data, an instance conversion may improve the feature extraction efficiency. In this section, we first present the design of ST instances for heterogeneous ST data support, and then elaborate on their conversions in the distributed environment.

Table 2. ST Instances, possible features to extract, and their applications. Those marked with * are experimentally evaluated in Section 5.2.

Instance	Features	Applications
Event	Anomaly*	Crime forecasting [26, 27]
	Clustering	Pattern mining [60, 61]
Trajectory	Stay point*	Travel recommendation [77, 79]
	Average speed*	Trajectory classification [31, 64]
Time Series	Periodical flow*	Demand prediction [52]
	Periodical speed	Traffic prediction [38]
Spatial Map	Regional flow	Site selection [10, 39]
	Regional speed*	Traffic forecasting [37, 41]
	POI statistics*	Tourism planning [62]
Raster	Transition*	Traffic forecasting [21, 33]
	Air quality*	Pollution prediction [78]

3.2.1 ST Instances. An ST instance contains spatial, temporal, and auxiliary information. ST4ML designs a base `Instance` class as a unified structure for different derivations, which is defined by an array of `entries` and a `data` field recording instance-level non-ST information (e.g., ID):

```
class Instance[S <: Geometry, V, D](entries: Array[Entry[S, V]], data: D){...}
```

An `Entry` has three fields: the `spatial` field can be a common 2-d geometry (e.g., point, linestring, and polygon); the `temporal` field is a duration (or an instant as a special case); and the `value` field records other entry-level attributes:

```
class Entry[S <: Geometry, V](spatial: S, temporal: Duration, value: V){...}
```

Inheriting the base class, we design five fundamental ST instances that can accommodate data types favored by different *STDML* applications. Table 2 presents suitable features to be extracted from different instances and possible applications that they can facilitate.

Event is an essential instance that contains only one geometry and one duration (i.e., the length of `entries` is limited to one). In typical cases, the geometry is a point and the duration is an instant, which can represent a camera snapshot or a check-in record.

Trajectory is prevalent in urban applications. A trajectory instance consists of a sequence of ST points, i.e., the `spatial` field of each entry is restricted to a point and the entries are sorted by their `temporal` field.

Time Series organizes ST data by time. Each entry represents a time slot and its `value` field records measurements or objects falling in that slot. Each entry's duration needs to be explicitly defined, while the `spatial` field is not a focus.

Spatial Map organizes ST data according to the spatial dimension. Contrarily to time series, the `spatial` field of each entry must be explicitly defined, and can be of any shape (e.g., linestring for representing road segments and polygon for districts).

Raster provides ST organization over data, and can be regarded as a collection of geometry shapes with temporal depths. In a raster, both `spatial` and `temporal` fields are used for computation and have to be explicitly defined.

While the main focus of ST4ML is on 2-d spatial data with temporal information, with the design of flexible `value` and `data` fields, the five instances can theoretically represent any data type. For example, 3-d mesh data can be represented as an event with the following abstraction. A mesh cell

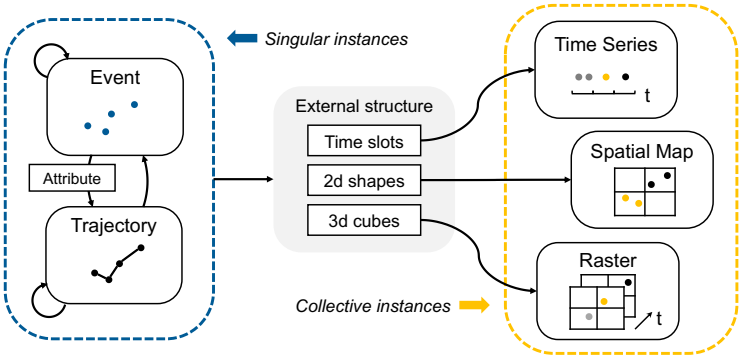


Fig. 3. Conversions between ST instances.

is projected to a reference surface and recorded in the spatial field. The detailed information such as vertices, edges, and faces are stored in the `value` field.

3.2.2 Instance Conversion. ST4ML supports conversions among all five ST instances and the commonly used ones are illustrated in Figure 3. The five instances can be divided into two categories: *singular* and *collective*. Events and trajectories fall into the singular category because they are atomic during processing and each instance represents one data record collected from the real world. Time series, spatial maps, and rasters are collective since each of them contains a set of parallel entries, and an entry may contain aggregated or a collection of real-world observations. We elaborate on the conversions by category:

Singular-to-collective conversions are most commonly used because acquired data usually appear singular while many *STDML* applications take collective features as input. The goal of the conversion is to allocate each singular instance to one or more cells of an external structure (e.g., a road network to obtain a spatial map or a timetable to obtain time series). In the distributed computing environment, the singular instances and the external structure are assigned to multiple machines for parallel processing, and there are two design options: (1) split the structure by cells, while each executor takes several cells and assigns corresponding singular instances; (2) each executor maintains a copy of the complete structure and allocates local singular instances to it. The first design requires a full shuffle over the singular instances, which takes extra processing time. Moreover, data bias may cause load imbalance and hinder the parallelism for the next stage. ST4ML employs the second design, where the cost of broadcasting an empty structure is low and the load balance is well managed.

The example below presents the use case of converting check-in *events* to a *spatial map* of districts. In line 1, the application programmer defines the spatial map as an array of polygons (`polygonArr`) to make the converter. The converter takes the `selectedRDD` and outputs a spatial map where each cell contains the events falling into it as in line 2.

```
1 val converter = new Event2SmConverter(polygonArr)
2 val convertedRDD = converter.convert(selectedRDD)
```

In addition, ST4ML provides two extensible points - `preMap` and `agg` - for application programmers to perform customized conversions with writing functions on a single instance, which eases the programming burden over structured and distributed data. We demonstrate the usage with the following example where the check-in events are converted to a spatial map of *regional per-type counts*.

```
1 type E = Event[Point, String, Map[String, String]]
2 val converter = new Event2SmConverter(polygonArr)
3 val preMap = (p: E) => p.mapData(x => x("type"))
4 def agg(arr: Array[E]): Map[String, Int] = {
5     val types = arr.map(_.data)
6     types.map(t => (t, 1)).groupBy(_._1).mapValues(_.length)
7 }
8 val convertedRDD = converter.convert(sRDD, preMap, agg)
```

In the example, only the "type" attribute of *each event* matters, so a `preMap` function is defined in line 3 to discard the rest of the attributes. `mapData` is a syntactic sugar of ST4ML to directly manipulate the `data` field of an instance while keeping the entries unchanged. The `agg` function defined in lines 4-7 aggregates the type count of *an array of events*. Finally, in line 8, the converter takes the RDD from the selection stage (`sRDD`) and the two extension functions to perform conversion. The produced spatial map contains the type counts inside each cell.

With the customized converter, the singular instances are first transformed with the `preMap` function in parallel. Next, the external structure is broadcasted to all executors, which allocate their local data to the designated structure cells. The naive implementation of allocation is expensive and an optimization is proposed in Section 4.2. Last, the `agg` function is applied on all cells of each executor in parallel to form the final collective instance. No data shuffling incurs in this process, and the balanced loads ensure high parallelism.

Singular-to-singular conversions are also widely used. *Trajectory-to-event* conversions take sojourn points out of trajectories, and are realized with a `flatMap` operation. *Event-to-trajectory* conversions group events by their `data` fields and order them by timestamps, which involve cross-machine joins. We implement it with *map-side join* mechanism to reduce data shuffling: events are first grouped locally and then shuffled to the reducers for a global merge. In the urban computing domain, trajectories are often manipulated within the road network, while the source data (e.g., GPS samples) may have sensing errors and need calibration. ST4ML provides a *trajectory-to-trajectory* conversion with the *map matching with Hidden Markov Model* based algorithm [43]. Given an input road graph, ST4ML executes map matching on raw trajectories and returns calibrated ones whose entry points lie on road segments. During execution, the road segments are indexed with an R-tree and broadcasted to all worker machines. The trajectories are map-matched in parallel while the R-tree accelerates the road candidates searching process. Similarly, ST4ML implements *event-to-event* conversion for calibration by projecting an event to its nearest road segment.

Collective-to-singular conversions require that the `value` field of the input has the type of `Array[SI]`, where `SI` is a singular instance. Similar to *trajectory-to-event* conversions, they are implemented with a `flatMap`-like operation in parallel with no data shuffling.

As for **Collective-to-collective conversions**, a general spatial map can only be converted to a time series with one slot or a raster with one cell. The temporal range of the converted instance is the union of the durations of the original spatial map cells. The rules of combining the `value` and `data` fields have to be explicitly defined by the application programmer. The same applies to time series. A raster can be converted to spatial maps or time series by grouping its cells by their temporal or spatial attributes, respectively. In distributed execution, the data of each executor are in parallel converted and no data shuffling incurs. ST4ML maintains a `HashMap` recording the cell indices and their spatial and temporal ranges to facilitate these conversions.

The converters output standard ST-Instance RDDs, which can be concatenated for specific applications. For example, a spatial map with values of `Array[Event]` can be converted to a time

Table 3. Built-in extractors in ST4ML.

Instance	Extractors
Event	EventAnomalyExtractor, EventCompanionExtractor, EventClusterExtractor
Trajectory	TrajSpeedExtractor, TrajOdExtractor, TrajStayPointExtractor, TrajTurningExtractor, TrajCompanionExtractor
Time Series	TsFlowExtractor, TsSpeedExtractor, TsWindowFreqExtractor
Spatial Map	SmFlowExtractor, SmSpeedExtractor, SmTransitExtractor
Raster	RasterFlowExtractor, RasterSpeedExtractor RasterTransitExtractor

Table 4. RDD extension interfaces for extraction.

API	Description
`cRDD.mapValue(f: V1 => V2)`	Map the value fields of all cells inside all instances in cRDD* according to function `f`. Require the value field having type of `Array[V1]`.
`cRDD.mapValuePlus(f: (V1, Polygon, Duration) => V2)`	Same as above but with entry spatial and temporal information involved. The `Polygon` and `Duration` variables will be replaced by the boundaries of each cell.
`cRDD.mapData(f: D1 => D2)`	Map the data field of cRDD according to function `f`.
`cRDD.mapDataPlus(f: (D1, Array[Polygon], Array[Duration]) => D2)`	Same as above but with instance spatial and temporal information involved. The `Polygon` and `Duration` variables will be replaced by the boundaries of the collective structure.
`cRDD.collectAndMerge(init: V1, f1: (V1, V) => T)`	Fetch cRDD to the master server and merge the distributed instances according to initial value `init` and function `f1`.

*cRDD stands for an RDD of collective instances.

series by reorganizing the events, which is implemented with a *spatial-map-to-event* conversion followed by an *event-to-time-series* conversion.

3.3 Extraction

The extractor finally extracts features for *STDML* applications based on the well-organized ST data output from the previous stage. For generality and flexibility, it is impossible to seal all extraction functions inside ST4ML. Instead, we provide three levels of extensions to facilitate application programmers in realizing their extraction logics.

Built-in extractors. ST4ML pre-builds a set of frequently-used extractors, as listed in Table 3. These extractors are carefully implemented with native RDD operations to ensure high efficiency, and also take input parameters to accept adjustments. We find these extractors very useful in serving numerous *STDML* applications and also expect to extend them over time.

RDD-level APIs. ST4ML designs RDD-level extension interfaces for translating code snippets into RDD operations as listed in Table 4. Many feature extraction applications are based on collective instances whose entries are arrays of singular instances, e.g., spatial maps of trajectories. The extraction logics are usually applied on the singular instances, e.g., extract stay points from each trajectory. When writing such applications, the objects of the extraction logic are deeply wrapped inside the RDD (RDD → spatial map → cell → trajectory). Implementing such extraction with native Spark API involves tedious and redundant nested programming. ST4ML abstracts these operations and provides APIs that allow application programmers to focus on designing the extraction logic over a single ST object and leave the distributed execution to ST4ML.

In the above stay point extraction example, the application programmer may first define the function to extract stay points falling in an ST range from *one trajectory* as:

```
def extractStayPoint(traj: Trajectory[Int, String],
    s: Polygon, t: Duration): Array[Point] = {...}
```

With the `mapValuePlus` API, the application programmer may construct an RDD operation, which extracts stay points from *all trajectories* in the distributed spatial maps (lines 1-2 below). The RDD operation `f` can then be translated to a customized extractor (line 3), and the `extract` function triggers the entire application to execute (line 4):

```
1 val f = (rdd: RDD[Raster[Polygon, Array[Traj], _]) =>
2     rdd.mapValuePlus(extractStayPoint)
3 val extractor = Extractor(f)
4 val extractedRDD = extractor.extract(convertedRDD)
```

ST4ML also provides an API for application programmers to merge the distributed collective RDD as follows:

```
val stayPointArr = extractedRDD.collectAndMerge(emptyPointArr, _++_)
```

Application programmers may only replace the most fundamental functions (`extractStayPoint` and `_++_` in the examples) and follow this pattern to conveniently make extractors for their applications.

Native Spark operations. Application programmers familiar with Spark can implement their logics by RDD programming with the highest level of freedom. ST-instance RDDs are compatible with native RDD operations and can be computed with other RDDs as well. For convenience, application programmers can utilize the plenty of functions over `Geometry` and `Duration` classes provided by ST4ML, such as `calcGeoDistance`, and `temporalSliding`.

The extracted feature outputs are stored as in-memory RDDs, which the application programmers may direct to Spark-affiliated ML modules like MLlib [42] and GraphX [22], or channeled to external ML engines, like TensorFlow [1] and PyTorch [49], in standard JSON or CSV data formats.

3.4 End-to-End Example

Last in this section, we present the essential code for implementing the running example: traffic speed extraction from trajectories.

```
1  // read raster structure
2  val raster = ReadRaster(rasterFile)
3  // initialize operators
4  val selector = Selector[STTraj](sQuery, tQuery, n = 100)
5  val converter = Traj2RasterConverter(raster)
6  val extractor = RasterSpeedExtractor(unit = "kmh")
7  // execute the application
8  val trajRDD = selector.select(dataDir)
9  val rasterRDD = converter.convert(trajRDD)
10 val speedRDD = extractor.extract(rasterRDD)
11 // save results
12 saveParquet(speedRDD, resDir)
```

In line 2, the helper function `ReadRaster` reads the raster structure from a CSV file (each line has fields *shape*, *t_min* and *t_max*). The three operators are initiated in lines 4-6. In line 4, `STTraj`

indicates that the original data type is trajectory, while `sQuery` and `tQuery` specify the spatial and temporal range of interest, which can be derived from the raster structure. `n` specifies the number of partitions, i.e., the parallelism of the application. For this feature extraction task, the most suitable data representation is raster, so a `Traj2Raster` converter is initiated as in line 5. Last, ST4ML's built-in `RasterSpeedExtractor` is invoked. After defining the operators, their execution functions are called in sequence. In line 8, the path to the trajectory data directory is passed to the selector, and subsequently the resulting RDDs are passed to the converter and extractor as a pipeline (lines 9-10). The final results are saved as Parquet files back to HDFS with the `saveParquet` helper function.

4 KEY OPTIMIZATIONS

Through ST4ML's pipeline, some common operations involve intensive computation and may take a long processing time. We devise the following optimizations to alleviate the overhead.

4.1 On-Disk Indexing with Metadata

Existing Spark-based spatial or ST data processing frameworks [24, 58, 66, 68, 69] load all data into memory for processing, and the intermediate results are discarded after each run. This design is not efficient in feature extraction scenarios for two reasons. First, loading all data into memory is time- and memory-consuming. When most data are pruned after the selection stage, loading all data into memory is unnecessary. Second, a dataset is usually repeatedly used for model training and inference in different occasions, and the results from data partitioning and processing are worth persistence for future reuse. To improve the data loading efficiency, we design an *on-disk data indexing with metadata* technique, which consists of two steps: offline index generation and index-facilitated selection.

In the offline preparation, we perform an in-memory ST partitioning over the data, persist the partitioned data on disk, and index the partitions. Considering the unique characteristic of ST data and its distinction from common 3-d data, we propose a new ST partitioning method.

T-STR partitioner. Since ST data's spatial and temporal dimensions have intrinsically different physical meanings and scales, they should not be coupled and partitioned together. For example, given a dataset that spans a year and an area of $5km \times 5km$, if the application programmer applies conventional multi-dimensional partitioners (e.g., K-D tree [9]) to split the data into 1000 partitions, each partition will span $500m \times 500m$ and 5 weeks (suppose the data are uniformly distributed). When the temporal query window is often of weekly scale, this partitioning performs ineffective temporal filtering [34]. With the 1000-partition constraint, it is better to segment the data to $1.25km \times 1.25km$ and one week span to achieve higher data loading efficiency, where the segment counts of spatial and temporal dimensions are 4 and 62, respectively.

We implement a simple yet effective extension on the 2-d STR partitioner to make it ST-aware. The original 2-d STR (sort-tile-recursive) [32] method partitions r samples into n groups by first uniformly partitioning them along one dimension into $\sqrt{n}$ groups and then partitioning each sub-group along the other dimension into $\sqrt{n}$ sub-groups. Each final partition contains (roughly) $\frac{r}{n}$ nearby samples. The 2-d STR neglects the temporal dimension. Directly extending it to 3-d mixes the ST dimensions, where the fixed granularity cannot favor applications of various needs as the previous example explains.

T-STR partitioner first segments the data along the temporal dimension into n_t partitions of equal size. Next, each temporal partition is split into n_s partitions with 2-d STR algorithm. The entire data is therefore segmented into $n_t \times n_s$ partitions, which contain ST-nearby data of similar sizes. Algorithm 1 presents the detailed steps. The design first partitions along the temporal dimension to achieve higher efficiency, because it divides the large data and workloads into chunks and the

Algorithm 1: T-STR paritioning

Input: ST data: $D : [d_1, d_2, ..., d_r]$, temporal granularity g_t,
spatial granularity g_s, flag for duplication *duplicate*, sampling rate: *sr*
Output: data with partition index: $P = [(p_1, d_1), (p_1, d_2), ..., (p_m, d_n)]$

```
// Find partition boundaries with sampled data
partitionBounds = [ ], P = [ ]
tBuckets = temporalPartition(sample(D, sr), g_t)
for t ← tBuckets do
    stBuckets = strPartition(t, g_s)
    for st ← stBuckets do
        partitionBounds.add(st.stBounds)
// Allocate all data to the partitions
for d ← D do
    for p ← partitionBoundaries do
        if d.overlaps(p) then
            P.add((p, d))
        if not duplicate then
            break
return P
```

expensive spatial partitioning can be executed in parallel to save processing time. Partitioning along the 2-d spatial dimensions is more expensive as it requires extracting the representative coordinates (e.g., center) from complex shapes for sorting. Such an idea can be extended with more dimensions according to the application needs. Any 1-d attribute of the ST data (e.g., the ID and the vehicle type) can be included for partitioning.

After partitioning, the data are written back to the persistent storage where objects belonging to the same partition are stored together. We maintain a metadata file on the master server to index the partitions. The metadata records boundaries of all indexing dimensions of each partition (e.g., the minimum bounding rectangle (MBR) for the spatial dimension and the endpoints for the temporal dimension). Application programmers write simple code as:

```
val p = TSTRPartitioner(gt, gs) // gt, gs are temporal and spatial granularities
val (pRDD, pInfo) = eventRDD.stPartitionWithInfo(p)
pInfo.toDisk(metaDataDir)
pRDD.toDisk(dataDir)
```

Once the data are reorganized and the metadata file is generated, each time a feature extraction application is performed on the same dataset, the application programmer specifies the metadata directory for optimized data selection as:

```
val trajRDD = selector.select(dataDir, metaDataDir)
```

The execution is illustrated in Figure 4. ST4ML first compares the query range(s) with all partitions to get the overlapping ones (❶). Next, ST4ML only assigns shortlisted partitions to the worker executors (❷) and performs in-memory fine-grained filtering on them (❸). Hence, both memory consumption and processing time are greatly reduced.

Discussions. (1) Compared to entry-based indexing methods used in conventional ST databases (e.g., GeoMesa [28]), such a partition-based index is more suitable for feature extraction applications as it saves storage for indices and possesses ST locality of loaded data. (2) ML and DL analyses are

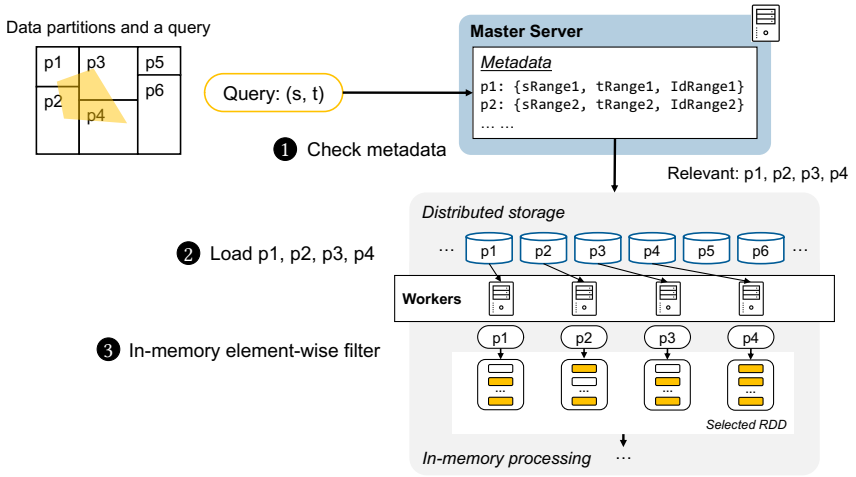


Fig. 4. Optimized data selection with on-disk indexing.

usually conducted on enormous historical data, and a set of data is repeatedly used for various applications. Therefore, we see the indexing time compensated in the long run. In scenarios where data are continuously generated, application programmers may periodically index the new group of data and merge the metadata file with the existing ones. (3) In practice, the indexing granularity is set according to the application programmer's heuristic on what *STDML* applications are conducted with the dataset (e.g., temporal-dominate or spatial-temporal-balance).

4.2 Optimizations on Instance Conversions

Singular-to-collective conversions are computationally expensive. Given m singular instances and a collective structure with n ST cells, the conversion assigns each singular instance to one or multiple cells by checking intersection. The naive implementation – iterating all pairs of instance and cell – results in $O(mn)$ time complexity. The computation is heavy when both m and n are large (e.g., city-wide trajectories into 1km × 1km × 1h raster). We propose the following optimization techniques to reduce the computational complexity.

Conversion with regular structures. A structure S is regular if its cells $[c_1, c_2, ..., c_n]$ have the same size and densely tile the space (i.e., no overlapping and no interspace). If a collective instance is regular, we sort the cells based on their boundaries: t_{start} for time series, $(lon_{\min}, lat_{\min})$ for spatial map, and $(t_{start}, lon_{\min}, lat_{\min})$ for raster. When allocating events or trajectories to a regular structure, we do not need to iterate all cells. Instead, the possibly intersecting cells can be obtained by derivation. In a regular structure, each dimension d with extreme values $d_{\min}$ and $d_{\max}$ is divided into segments with length $d_{interval}$. For a singular instance with extreme values $q_{\min}$ and $q_{\max}$ on the same dimension, we find the indices of possible intersecting segments as

$$[(\max(1, \frac{q_{\min} - d_{\min}}{d_{interval}}), \min(n, \frac{q_{\max} - d_{\min}}{d_{interval}})]$$

After calculating the indices for all dimensions, we have a smaller set of cells with size p that intersect the MBR of the singular instance, where $p \leq n$, and in practical cases $p << n$. To get an accurate result, we iterate the p cells and remove those not intersecting the actual instance. The time complexity remains $O(mn)$ but with a much smaller factor[2]. If the singular instance has a

[2]Although $p \leq n$, the number of intersection checks still grows with n. Therefore the big O notation does not change.

shape whose MBR equals itself (e.g., point and rectangle) or the conversion is to time series, the iteration step can be skipped and the complexity reduces to $O(m)$.

Conversion with irregular structures. In general cases, the structures are irregular: the cells may have different sizes and shapes, or even overlap. We design an R-tree-based conversion, which is specially tailored to our *Selection-Conversion* pipeline, and different from the original usage of R-tree [23]. Conventional ST range query solutions build tree-like indices over the *ST objects*, and the query range traverses the tree to find intersecting ones. In our problem, we index the *structure cells*, and each ST instance traverses the tree. The reason is twofold. First, indexing numerous instances is time-consuming but non-reusable. Second, building indices takes up significant memory: during indexing, both the original data and another tree structure (which has a bigger size than the original data) reside in memory. We employ R-tree of different dimensions to index the three collective structures: 1-d for time series durations, 2-d for spatial map elements, and 3-d for raster cells. The average time complexity can be reduced to $O(m \log(n))$. Recall that in ST4ML's design, all executors share the same collective structure. We first generate the in-memory R-tree index for the structure on the master machine, and then broadcast it to all workers to avoid duplicated computation. Each executor uses the R-tree instead of the raw structure for local conversion afterward.

5 EXPERIMENTAL EVALUATION

Datasets. We use the following *public* datasets for experiments so the results can be reproduced.

`NYC` [63] dataset consists of taxi pick-up and drop-off *events*. Each event has fields [*lon, lat, time, auxInfo*]. We take 337,865,116 events from New York, USA in 2013, with a size of 63.3GB in memory.

`Porto` [29] dataset consists of 1,674,160 vehicle *trajectories* collected in Porto, Portugal from 2013-07-01 to 2014-06-30. Each trajectory has fields [*tripId, Array((lon, lat)), startTime*] and the sampling interval is 15s. Although it is the largest public trajectory dataset, the size is still not challenging for cluster computation. We enlarge the dataset by duplicating it 20 times and adding Gaussian noise with deviations $\sigma_s = 20m$ and $\sigma_t = 2min$. The enlarged dataset contains 33,483,200 trajectories and has an in-memory size of 125GB.

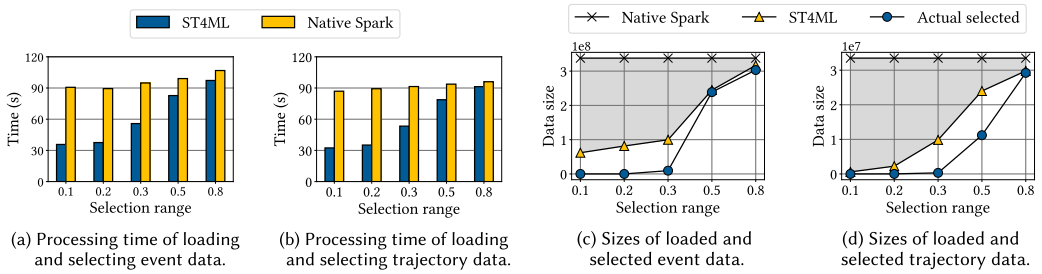
`Air` [78] consists of 2,891,393 air quality records hourly collected from 437 stations in China from 2014-05-01 to 2015-04-30. Each record contains location, time, and six air quality indices. Considering its small scale, we enlarge it by (1) replicating the stations 20 times with Gaussian noise ($\sigma = 500m$), and (2) interpolating the records to bring down the sampling interval to $5min$. The enlarged dataset has 743,701,800 records and consumes 45.1GB of memory.

`OSM` [47] contains the global map data. We take 147,331,044 points of interest (POIs) and 218,785 postal code areas updated in 2021 around the world. Each POI contains 2-d coordinates with `String`-typed attributes, and each area has a polygon shape. This dataset contains no temporal information. The in-memory sizes of the POI and area data are 19.2GB and 1.5GB, respectively.

Environment. We conduct experiments with a small cluster to demonstrate the comparative advantage of ST4ML with throttled computing resources. The cluster consists of 5 machines, each of which is equipped with a 32-core processor (Intel Xeon Platinum 8163 2.5GHz) and 128GB RAM. All machines are connected to a 10 Gigabit Ethernet switch and run a CentOS 7.6 system with Hadoop 3.3.1 and Spark 3.1.2. One machine is chosen as the master node while the other four are workers. We allocate 8 CPU cores, 32GB memory, and 256GB SSD for experiments in each worker node. The implementation of ST4ML comprises $\sim$ 14k lines of Scala code.

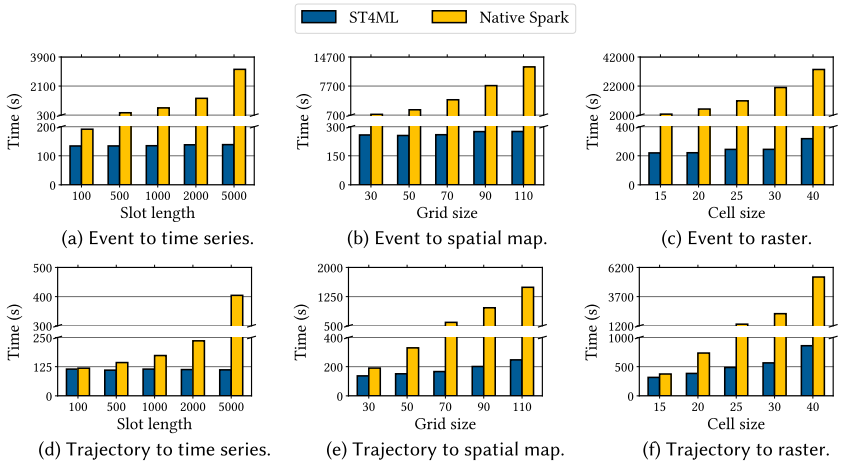
5.1 Microbenchmarks

We first evaluate the effectiveness of the optimizations proposed in Section 4: on-disk indexing for data loading, indexing-based conversion optimization, as well as the effectiveness of the T-STR partitioner. In this section, we experiment with the `NYC` and `Porto` datasets as they are of larger



(a) Processing time of loading and selecting event data.
(b) Processing time of loading and selecting trajectory data.
(c) Sizes of loaded and selected event data.
(d) Sizes of loaded and selected trajectory data.

Fig. 5. Processing time and memory usage of loading and selecting event and trajectory data.



(a) Event to time series.
(b) Event to spatial map.
(c) Event to raster.
(d) Trajectory to time series.
(e) Trajectory to spatial map.
(f) Trajectory to raster.

Fig. 6. Processing time of instance conversions.

scale and higher complexity. All experiments are conducted 5 times and the average performance is reported.

On-disk indexing with metadata. We split NYC and Porto datasets with T-STR partitioner into 180 and 258 partitions respectively, which are determined by the on-disk data sizes divided by HDFS block size. The spatial and temporal granularities are heuristically set to (36, 5) and (86, 3). We perform data loading and selection on the indexed data with the comparison of native Spark, which loads all data into memory and performs parallel in-memory filtering over them. Figure 5a and 5b compare the processing time for loading and selecting the event data and trajectory data. The on-disk index saves up to 60% time for both datasets. The time saving is more notable on smaller query ranges. Figure 5c and 5d compare the sizes of the data loaded into memory. The lines with cross and triangle makers are the data sizes loaded with native Spark and ST4ML, while the circle-marked line is the actual selected data size. The shaded area indicates the gain from the optimization: 42% to 98% of the irrelevant data have been pruned for the two cases. Because real-world data are not uniformly distributed in the ST space and the queries are randomly generated, the gain is not linearly correlated to the query range. For event data, when the selection range is small, more irrelevant data are loaded due to the unbalanced distribution of the data. This result represents the worst case where the query range overlaps multiple partitions and only a few data are selected. Nonetheless, more than 80% of irrelevant data are pruned when the selection range is smaller than 0.2.

Table 5. Load balance evaluation (CV: coefficient of variation, OV: overlap).

	CV_event	OV_event	CV_traj	OV_traj
Native Spark	**0.0018**	454.63	**0.0057**	72.19
GeoSpark	0.15	1.56	0.22	0.41
GeoMesa	0.81	13.44	0.052	283.1
ST4ML (T-STR)	0.063	**0.86**	0.045	**0.074**

Conversion optimization with in-memory indexing. We conduct experiments on all six singular-to-collective conversions with R-tree-based indexing, compared with the default solution in Spark (a Cartesian product of the singular instance set and the collective cells). Figure 6 presents the processing time of conversions from both datasets to time series, spatial map, and raster with varying granularity. The grid size x of a spatial map indicates splitting the whole spatial range evenly into $x \times x$ grids and the raster size y indicates splitting the 3-d ST space into $y \times y \times y$. ST4ML's optimizations achieve up to 23×, 45×, and 105× faster in conversions from events to time series, spatial map, and raster, respectively. When converting trajectories, the optimization achieves up to 6× faster. We have three observations: (1) The gain is more notable on the point-shaped event dataset where higher pruning efficiency is achieved. (2) The effectiveness increases with the dimension of the collective structure as objects are more distinguishable in higher dimensions. (3) The gain is more evident with finer-grained structures, where the ratio of overlapping tree nodes to the total tree nodes is smaller. To summarize, our optimization is very effective and with more complicated conversion tasks (structures of higher dimensions and more cells), the optimization brings high gains.

T-STR partitioner effectiveness. We evaluate the proposed T-STR partitioner from two aspects: (1) the load balance, and (2) its efficiency in facilitating the computing pipeline.

We first compare T-STR partitioner's load balancing effectiveness with Spark's native partitioner and the methods employed in GeoSpark [69] and GeoMesa [28] (which will be detailed in Section 5.2). We evaluate the performance with two metrics:

Coefficient of variation:

$$CV = \frac{\sigma_P}{\mu_P}$$

where P is the set of partitions, σ is the standard deviation of the partition sizes, and μ is the mean of the partition sizes. A smaller CV implies a more balanced load distribution.

Overlap:

$$OV = \frac{\sum_{p \in P} V_p}{V_{all}}$$

where V_p is the ST MBR of a partition $p \in P$, and V_{all} is the ST MBR of all data. An ST-aware partitioner results in a small OV.

We set the number of partitions to 1024 for all cases, while the spatial and temporal granularties for T-STR partitioner are both set to 32. Table 5 presents the comparison of different partitioning methods on event and trajectory datasets. ST4ML's T-STR partitioner achieves the overall best performance in terms of load balance and ST-locality awareness. Spark's native partitioner randomly partitions the dataset without considering ST locality. Though with the lowest CV, native Spark leads to the highest OV and cannot generally benefit diverse ST applications. GeoSpark and GeoMesa only preserve spatial locality, which leads to higher OVs in the ST space.

We next evaluate how T-STR partitioner facilitates the computing pipeline with comparison to the original 2-d STR partitioner in two representative scenarios.

Table 6. Efficiency comparison between T-STR and 2-d STR (unit: *minute*).

	Data loading		**Companion extraction**	
	Event	Traj	Event	Traj
2-d STR	5.53	2.36	57.52	71.57
T-STR	**0.98**	**0.91**	**19.35**	**8.92**

Index construction for data loading. For the two datasets, we build on-disk indices by partitioning them into 1024 partitions with our T-STR and the original 2-d STR methods. We perform 10 randomly generated data selection tasks and record the processing time. As the left columns in Table 6 suggest, with the consideration of the temporal dimension, the T-STR-based index performs 4.6× and 1.6× faster in completing the selection tasks for the two datasets.

Companion feature extraction. We apply the built-in companion extractors on the two datasets to evaluate how the T-STR partitioner helps in end-to-end applications. The extractors find all data pairs within an ST threshold of $1km$ and $15min$ in a day. The data is first partitioned into 1024 so the tasks can be executed in parallel with only inner-partition comparison. The right columns in Table 6 show that extraction with T-STR partitioner performs 2× and 7× faster in completing the end-to-end feature extraction as the ST-aware partitioning leads to fewer inner-partition comparisons.

5.2 End-to-End Performance

We evaluate the end-to-end performance of ST4ML on processing time and lines of code (LoC) of the application. We compare ST4ML with straightforward extension of two existing systems, both of which are widely adopted in the industry and actively maintained:

GeoSpark [69] (now known as Apache Sedona [8]) is a Spark-based system for processing big spatial data. According to [48], GeoSpark has a competitive performance compared to similar works. In GeoSpark, a piece of ST data is represented as a geometry with `String`-typed attributes (including the temporal information, ID, etc.). For end-to-end applications, after loading data into memory, we apply its range query function (with optimizations such as data indexing and K-D-tree partitioning) to select related data and write customized feature extraction logic over RDDs.

GeoMesa [28] is a distributed ST index built on top of NoSQL databases. In our experiments, we use HDFS backend, which is the same as used by ST4ML and GeoSpark. We first ingest the datasets by representing them as GeoMesa's `SimpleFeature` and save them in Parquet format. Meanwhile, indices are built on geometries using `XZ2-8bit` and also on the start timestamp. The evaluation does not count its data ingestion time to give the benefit to GeoMesa. For end-to-end applications, we program over its SparkSQL connector to load data into memory (with the help of the indices and grid-based partitioning) and execute extraction logics.

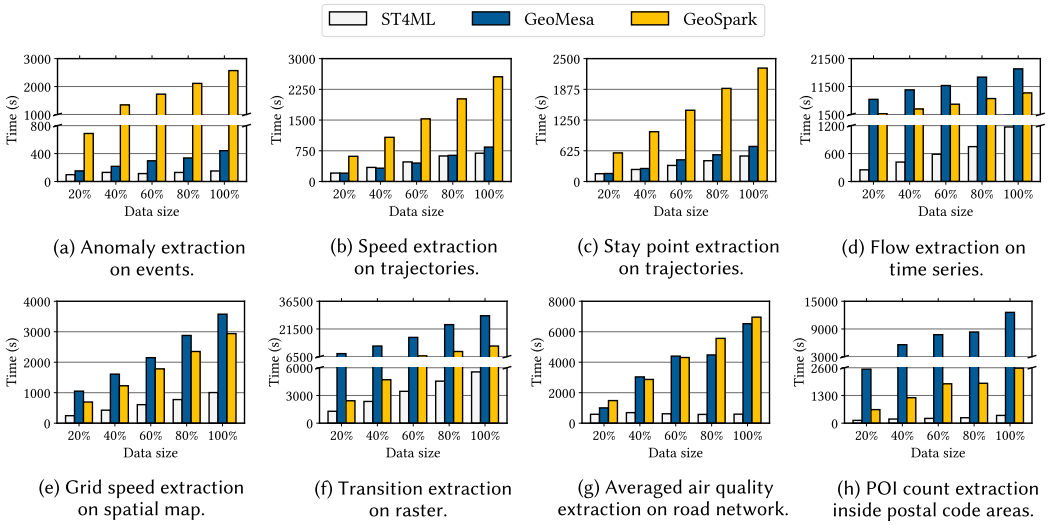
Eight applications are chosen from Table 2 to perform evaluation: three of which adopt direct feature extraction from the source data instance, and the rest require data instance conversions in ST4ML. A detailed description is presented in Table 7. Since Spark is lazily evaluated and the other systems do not employ the concept of stages, we report the end-to-end processing time.

Figure 7 compares the processing time of the applications at different data scales. Each application is performed on 10 randomly-generated ST ranges in sequence, and the total processing time is presented. We discuss the experiment results by category:

Feature extraction without data instance conversion. For feature extractions directly applied on the input data, ST4ML has the best overall performance as shown in Figure 7a to 7c. The advantage of ST4ML is more obvious when extracting features from the event dataset: GeoSpark and GeoMesa take up to 17× and 3× processing time, respectively. When extracting features from the trajectory dataset, ST4ML performs similarly to GeoMesa on a smaller data scale but saves more than 20%

Table 7. Details of the extractions for experiments. Experimental datasets: ◊: NYC, †: Porto, ‡: Air, §: OSM.

Feature	Conversion	Description
Anomaly◊	-	Extract **events** occurring 23-4hrs daily.
Average speed†	-	Extract the average speed of each **trajectory**.
Stay point†	-	Extract stay points from **trajectories** with threshold (200m, 10min).
Hourly flow◊	Event2Ts	Extract the number of events in a **time series** of 1 hour interval.
Grid speed†	Traj2Sm	Extract average speed of each cell in a **spatial map** with size 1m × 1m.
Transition†	Traj2Raster	Extract in/out flow of each cell in a (10km × 10km, 1h) **raster**.
Air over road‡	Event2Raster	Extract the daily averaged air quality indices over urban **road network**.
POI count§	Event2Sm	Extract the POI count inside each **postal code area**.



(a) Anomaly extraction on events. (b) Speed extraction on trajectories. (c) Stay point extraction on trajectories. (d) Flow extraction on time series.

(e) Grid speed extraction on spatial map. (f) Transition extraction on raster. (g) Averaged air quality extraction on road network. (h) POI count extraction inside postal code areas.

Fig. 7. Processing time of end-to-end feature extractions.

time on the complete dataset. GeoSpark takes on average 3.5× time of ST4ML. As the data size increases, all solutions take longer processing time but ST4ML grows much slower, indicating higher scalability. GeoSpark loads all data into memory, which takes longer and consumes more resources. As more RDDs are generated in consequent operations, the inadequate memory hinders the performance. On the other hand, GeoMesa selects relevant data to process and supports queries over the timestamp, thus it takes an overall shorter time than GeoSpark. However, GeoMesa does not optimize in-memory data processing (e.g., ST joining) and the performance degrades as the operations get complex. Both baselines store the timestamps in a trajectory as a `String`, which needs additional reformation to facilitate feature extraction and takes extra time.

Feature extraction with data instance conversion. Figure 7d to 7h present the feature extractions that need converting source data to collective instances, where ST4ML provides a more noticeable gain. In all experiments, ST4ML employs the R-tree-based optimized conversion while the baselines have no specific optimization. ST4ML outperforms GeoMesa and GeoSpark by up to 27.6× and 9.6× in the flow extraction on time series, 4.2× and 3× in the speed extraction on spatial maps, and 6.3× and 2.2× in the transition extraction on rasters. For `Air` and `Osm` datasets, ST4ML outperforms GeoMesa by 11× and 39×; and outperforms GeoSpark by 11.8× and 7×. Besides the

Table 8. Lines of code implementing end-to-end applications.

	Abnormal event	Average speed	Stay point	Hourly flow	Grid speed	Raster transition	Air over road	POI count	**Average**
ST4ML-B	**44**	**44**	**46**	**45**	**47**	**70**	**50**	**44**	**100%**
ST4ML-C	48	45	67	50	57	90	62	**44**	119%
GeoMesa	66	78	98	77	124	138	87	84	193%
GeoSpark	112	79	100	116	133	158	97	61	219%

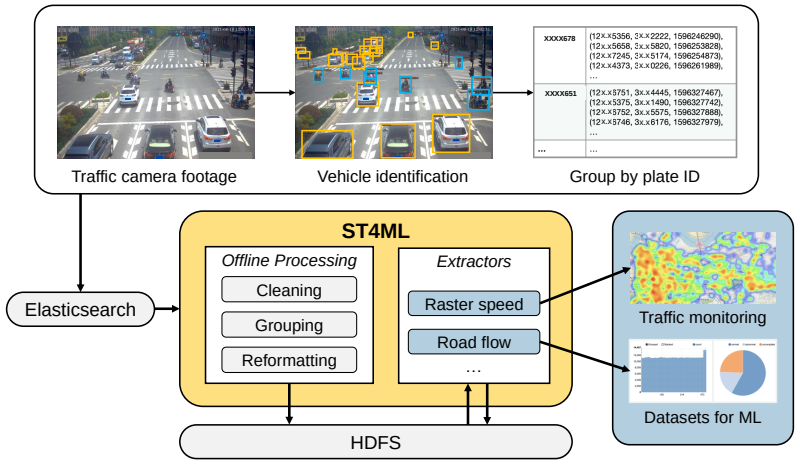


Fig. 8. ST4ML interfaced with existing computing modules in Alibaba business.

pipeline design and data loading optimization, the gain of ST4ML also comes from the conversion optimization, which is more prominent in applications with finer-grained and less-overlapping structures (hourly flow, transition, and POI count).

The processing time comparison suggests that ST4ML has the best performance compared with the two baselines in all cases, and the gain is more pronounced when the data size gets larger and the computations involved are more complicated (e.g., involving data grouping and aggregation).

Ease-of-use. Table 8 reports the LoC implementing the end-to-end experiments with different solutions (all include the same glue code like environment setting and time recording). For ST4ML, we present the LoC of invoking the built-in functions when available (ST4ML-B) and writing customized functions with provided APIs (ST4ML-C). The LoC of data ingestion for GeoMesa is not included. ST4ML takes the least programming effort across applications and domains. The two baselines require 93% and 119% more LoC than ST4ML, respectively. With the built-in APIs, application programmers take subtle efforts (19% more code) to write functions.

6 CASE STUDIES

We have adopted ST4ML to support ongoing urban data analytics applications in Alibaba City Brain Lab. Figure 8 depicts how ST4ML is interfaced with existing computing modules. A large amount of ST data are collected from various sources and stored in storage (e.g., Amazon S3) or search (e.g., Elasticsearch) engines. ST4ML periodically takes data from those engines, processes them, and persists the formatted data in HDFS for downstream applications including traffic monitoring and generation of ML datasets.

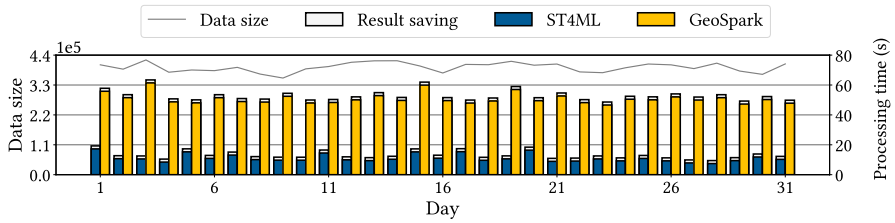


Fig. 9. Performance of traffic speed extraction.

We present two case studies with large-scale trajectories captured by thousands of traffic cameras in a city. The vehicle plate numbers are identified with the deployed computer vision module. The output data consist of plate ID, location, timestamp, and other log information, which are then grouped by the plate ID to form vehicle trajectories. ST4ML takes trajectories from ElasticSearch, partitions them with the T-STR partitioner, and persists them as Parquet files. The applications are continuously supporting business cases, and we quantitatively demonstrate the performance with vehicle trajectories collected from one month in Hangzhou city (with an amount of 12,251,168 and an in-memory size of 25GB). Note that the proprietary data is of much higher spatial and temporal density (287 trajectories/day/km^2) than the public dataset used in the previous experiments (38 trajectories/day/km^2 for enlarged `Porto`). The computer cluster for deployment consists of 1 master and 8 worker machines with the same configuration as those used in the previous evaluation.

Traffic speed extraction on rasters. Various ML applications make use of time-evolving regional traffic speeds for applications including traffic forecasting [33] and route recommendation [13]. In this study, we extract the regional average speed at different times. After taking data from a day, we divide the city into 100 polygon-shaped districts and construct a raster with cells of (*district*, *one-hour duration*). The extraction application returns the number of vehicles appearing inside each cell and the average speed of them, which are saved as CSV files back to HDFS. We compare ST4ML with GeoSpark, which outperforms GeoMesa in aggregation-involved applications as evidenced in the previous section.

Figure 9 presents the data sizes and the processing times when applying ST4ML and GeoSpark for each day in a month. Since the extracted high-level features are not large in size, persisting the results takes insignificant time. The extraction time grows for both systems as the data size increases. Working with the moderate computer cluster, ST4ML can extract daily city-wide speed profiles in tens of seconds, which is 3-7× faster than the adoption of GeoSpark.

Traffic flow extraction on the road network. Many urban applications study the ST data on the road network, and the traffic flow (the amount of passing vehicles during a given time duration) on each road segment is a significant feature to extract. Two challenges are faced in this business scenario: (1) the original trajectory may deviate from the road network topology due to sensing errors and need to be matched to the road segments; (2) the sparse map-matched points need to be connected based on the graph information so that the flow count of road segments not covered by cameras can be inferred. We show the results obtained from a district that contains 2899 road segments. Our application program leverages the built-in *trajectory-to-trajectory* conversion in ST4ML to perform map matching and connect the projected road segments to form complete trajectories. The map-matched trajectories are further converted to a raster where the spatial cells are road segments and the temporal slots have a duration of one hour.

Table 9 lists the trajectory data information and the processing time of two days, which shows ST4ML's ability in performing complex computation and aggregation over large-scale ST data. The average number of points and duration of the trajectories imply long intervals between location

Table 9. Performance of road network flow extraction.

Date	Amount	Avg. number of points	Avg. duration	Processing time
2020-08-02 (Sun)	882,817	9.03	26.95 min	55 min
2020-08-03 (Mon)	810,855	8.74	27.25 min	52 min

(a) Peak hour. (b) Off-peak hour.

Fig. 10. Visualization of road flow at different times.

samples, which incur high computation intensity in map matching. Figure 10 visualizes the derived traffic flows across the road network at different times, showing clear spatial and temporal patterns and can be used as input to various ML applications. This type of application cannot be supported by simply extending GeoSpark or GeoMesa so no comparative study can be performed.

7 RELATED WORKS

NoSQL databases with ST support. [25, 28, 35, 44, 46, 51, 59] extend distributed NoSQL databases with ST indices and predicates for efficient on-disk ST data management and query. Databases do not support data analytics so the extensions usually provide APIs to connect with analytic systems like Spark. JUST [34] builds a customized NoSQL database with Spark computation framework as an urban ST data engine. It predefines some ST analytic functions including trajectory noise filtering and spatial clustering, which is similar to our feature extraction but is non-extensible.

Distributed spatial or ST processing systems. Several systems are developed to process spatial or ST data based on distributed computing frameworks. These systems support common operations, including spatial range query, kNN query, and distance join. [3, 4, 18, 40] build systems on top of Hadoop [5], a distributed disk-based data processing framework, and extend it with spatial indexing and partitioning techniques to support the above operations within the MapReduce [14] paradigm. Hadoop-based systems frequently dump data to the disk, which hinders performance. Spark [72] improves computational efficiency by moving data and intermediate results to memory. Spark-based spatial processing systems [56–58, 66, 68, 69] implement the query operations with techniques including data partitioning, in-memory indexing, and query scheduling to achieve high performance. These systems cannot process data according to their temporal attributes, which constrains the applicable scenarios. A handful of works [24, 36, 45, 50] are proposed to process ST data, which support the same query operations as above but also consider temporal information. ST-indexing methods are implemented to accelerate querying over ST ranges. For systems in this category, neither complex computations for ML feature extraction nor representation of heterogeneous ST data types is supported. Straightforward extension on them induces programming burden and performance degradation. Same as ST4ML, most such systems focus on data querying and analytics while assuming data are collected complete. Tools like [19, 70] can be applied to fill missing values to incomplete data beforehand.

Distributed trajectory analytics systems. [16, 17, 20, 30, 54, 65, 71, 76] thoroughly study the representation and characteristics of trajectories, a specific ST data type. Besides spatial range queries, these systems study query and join operations over different trajectory similarity metrics, which involves complex computations. However, these one-off systems do not support other data instances or extract customized ML features such as speed.

Existing systems cannot smoothly bridge big ST data with ML applications as they lack the capability of complex ST computations over heterogeneous data types, and the flexibility of accepting user-defined logics.

8 CONCLUSIONS

This paper presents ST4ML, the first ML-oriented distributed ST data processing system. ST4ML employs a three-stage pipelining framework to abstract the computing flow of ML feature extraction. Aiming at critical computationally-expensive operations, ST4ML employs optimizations including on-disk ST partitioning with metadata for data loading and in-memory indexing for instance conversion. Extensive experiments demonstrate ST4ML's superior performance compared to straightforward extensions of existing systems.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive comments and suggestions. This research is supported by the National Research Foundation, Singapore under its Industry Alignment Fund – Pre-positioning (IAF-PP) Funding Initiative, Alibaba Group through Alibaba Innovative Research (AIR) Program and Alibaba-NTU Singapore Joint Research Institute (JRI), and the Singapore Ministry of Education AcRF Tier 1 RT13/20. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not reflect the views of National Research Foundation Singapore, and other funding agencies.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, USA) *(OSDI'16)*. USENIX Association, USA, 265–283.

[2] Bijaya Adhikari, Xinfeng Xu, Naren Ramakrishnan, and B. Aditya Prakash. 2019. EpiDeep: Exploiting Embeddings for Epidemic Forecasting. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (Anchorage, AK, USA) *(KDD '19)*. Association for Computing Machinery, New York, NY, USA, 577–586. https://doi.org/10.1145/3292500.3330917

[3] Ablimit Aji, Fusheng Wang, Hoang Vo, Rubao Lee, Qiaoling Liu, Xiaodong Zhang, and Joel Saltz. 2013. Hadoop GIS: A High Performance Spatial Data Warehousing System over Mapreduce. *Proc. VLDB Endow.* 6, 11 (aug 2013), 1009–1020. https://doi.org/10.14778/2536222.2536227

[4] Louai Alarabi and Mohamed F Mokbel. 2017. A demonstration of st-hadoop: A mapreduce framework for big spatio-temporal data. *Proceedings of the VLDB Endowment* 10, 12 (2017), 1961–1964.

[5] Apache. 2006. Apache Hadoop. https://hadoop.apache.org/.

[6] Apache. 2007. Apache HBase. https://hbase.apache.org/.

[7] Apache. 2013. Apache Parquet. https://parquet.apache.org/.

[8] Apache. 2022. Apache Sedona. https://sedona.apache.org/.

[9] Jon Louis Bentley. 1975. Multidimensional Binary Search Trees Used for Associative Searching. *Commun. ACM* 18, 9 (sep 1975), 509–517. https://doi.org/10.1145/361002.361007

[10] Chu Cao and Mo Li. 2021. Generating Mobility Trajectories with Retained Data Utility. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining* (Virtual Event, Singapore) *(KDD '21)*. Association for Computing Machinery, New York, NY, USA, 2610–2620. https://doi.org/10.1145/3447548.3467158

[11] Chao Chen, Daqing Zhang, Zhi-Hua Zhou, Nan Li, Tülin Atmaca, and Shijian Li. 2013. B-Planner: Night bus route planning using large-scale taxi GPS traces. In *2013 IEEE international conference on pervasive computing and communications*

(PerCom). IEEE, 225–233.

[12] Weiyu Cheng, Yanyan Shen, Yanmin Zhu, and Linpeng Huang. 2018. A Neural Attention Model for Urban Air Quality Inference: Learning the Weights of Monitoring Stations. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence and Thirtieth Innovative Applications of Artificial Intelligence Conference and Eighth AAAI Symposium on Educational Advances in Artificial Intelligence* (New Orleans, Louisiana, USA) *(AAAI'18/IAAI'18/EAAI'18)*. AAAI Press, Article 262, 8 pages.

[13] Jian Dai, Bin Yang, Chenjuan Guo, and Zhiming Ding. 2015. Personalized route recommendation using big trajectory data. In *2015 IEEE 31st international conference on data engineering*. IEEE, 543–554.

[14] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM* 51, 1 (jan 2008), 107–113. https://doi.org/10.1145/1327452.1327492

[15] Songgaojun Deng, Shusen Wang, Huzefa Rangwala, Lijing Wang, and Yue Ning. 2020. Cola-GNN: Cross-Location Attention Based Graph Neural Networks for Long-Term ILI Prediction. In *Proceedings of the 29th ACM International Conference on Information & Knowledge Management* (Virtual Event, Ireland) *(CIKM '20)*. Association for Computing Machinery, New York, NY, USA, 245–254. https://doi.org/10.1145/3340531.3411975

[16] Xin Ding, Lu Chen, Yunjun Gao, Christian S. Jensen, and Hujun Bao. 2018. UlTraMan: A Unified Platform for Big Trajectory Data Management and Analytics. *Proc. VLDB Endow.* 11, 7 (March 2018), 787–799. https://doi.org/10.14778/3192965.3192970

[17] Ahmed Eldawy, Vagelis Hristidis, Saheli Ghosh, Majid Saeedan, Akil Sevim, A.B. Siddique, Samriddhi Singla, Ganesh Sivaram, Tin Vu, and Yaming Zhang. 2021. *Beast: Scalable Exploratory Analytics on Spatio-Temporal Data.* Association for Computing Machinery, New York, NY, USA, 3796–3807. https://doi.org/10.1145/3459637.3481897

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

[19] esri ArcGIS. 2022. Fill Missing Values (Space Time Pattern Mining). https://pro.arcgis.com/en/pro-app/2.8/tool-reference/space-time-pattern-mining/fillmissingvalues.htm.

[20] Ziquan Fang, Lu Chen, Yunjun Gao, Lu Pan, and Christian S Jensen. 2021. Dragoon: a hybrid and efficient big trajectory management system for offline and online analytics. *The VLDB Journal* 30, 2 (2021), 287–310.

[21] Ziquan Fang, Lu Pan, Lu Chen, Yuntao Du, and Yunjun Gao. 2021. MDTP: A Multi-Source Deep Traffic Prediction Framework over Spatio-Temporal Trajectory Data. *Proc. VLDB Endow.* 14, 8 (apr 2021), 1289–1297. https://doi.org/10.14778/3457390.3457394

[22] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. 2014. GraphX: Graph Processing in a Distributed Dataflow Framework. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Broomfield, CO) *(OSDI'14)*. USENIX Association, USA, 599–613.

[23] Antonin Guttman. 1984. R-Trees: A Dynamic Index Structure for Spatial Searching. *SIGMOD Rec.* 14, 2 (jun 1984), 47–57. https://doi.org/10.1145/971697.602266

[24] 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 2017)*, Bernhard Mitschang, Daniela Nicklas, Frank Leymann, Harald Schöning, Melanie Herschel, Jens Teubner, Theo Härder, Oliver Kopp, and Matthias Wieland (Eds.). Gesellschaft für Informatik, Bonn, 123–142.

[25] Huajun He, Ruiyuan Li, Jie Bao, Tianrui Li, and Yu Zheng. 2021. JUST-Traj: A Distributed and Holistic Trajectory Data Management System. In *Proceedings of the 29th International Conference on Advances in Geographic Information Systems* (Beijing, China) *(SIGSPATIAL '21)*. Association for Computing Machinery, New York, NY, USA, 403–406. https://doi.org/10.1145/3474717.3483990

[26] Chao Huang, Chuxu Zhang, Peng Dai, and Liefeng Bo. 2020. Cross-Interaction Hierarchical Attention Networks for Urban Anomaly Prediction. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI-20*, Christian Bessiere (Ed.). International Joint Conferences on Artificial Intelligence Organization, 4359–4365. https://doi.org/10.24963/ijcai.2020/601 Special track on AI for CompSust and Human well-being.

[27] Chao Huang, Chuxu Zhang, Jiashu Zhao, Xian Wu, Dawei Yin, and Nitesh Chawla. 2019. MiST: A Multiview and Multimodal Spatial-Temporal Learning Framework for Citywide Abnormal Event Forecasting. In *The World Wide Web Conference* (San Francisco, CA, USA) *(WWW '19)*. Association for Computing Machinery, New York, NY, USA, 717–728. https://doi.org/10.1145/3308558.3313730

[28] James N Hughes, Andrew Annex, Christopher N Eichelberger, Anthony Fox, Andrew Hulbert, and Michael Ronquest. 2015. Geomesa: a distributed architecture for spatio-temporal fusion. In *Geospatial informatics, fusion, and motion video analytics V*, Vol. 9473. International Society for Optics and Photonics, 94730F.

[29] Kaggle. 2015. Predict the destination of taxi trips based on initial partial trajectories. https://www.kaggle.com/c/pkdd-15-predict-taxi-service-trajectory-i/data.

[30] Hai Lan, Jiong Xie, Zhifeng Bao, Feifei Li, Wei Tian, Fang Wang, Sheng Wang, and Ailin Zhang. 2022. VRE: A Versatile, Robust, and Economical Trajectory Data System. *Proc. VLDB Endow.* 15, 12 (2022), 3398–3410. https:

//www.vldb.org/pvldb/vol15/p3398-bao.pdf
[31] Jae-Gil Lee, Jiawei Han, Xiaolei Li, and Hector Gonzalez. 2008. TraClass: trajectory classification using hierarchical region-based and trajectory-based clustering. *Proceedings of the VLDB Endowment* 1, 1 (2008), 1081–1094.
[32] S.T. Leutenegger, M.A. Lopez, and J. Edgington. 1997. STR: a simple and efficient algorithm for R-tree packing. In *Proceedings 13th International Conference on Data Engineering*. 497–506. https://doi.org/10.1109/ICDE.1997.582015
[33] Mingqian Li, Panrong Tong, Mo Li, Zhongming Jin, Jianqiang Huang, and Xian-Sheng Hua. 2021. Traffic Flow Prediction with Vehicle Trajectories. *Proceedings of the AAAI Conference on Artificial Intelligence* 35, 1 (May 2021), 294–302. https://ojs.aaai.org/index.php/AAAI/article/view/16104
[34] Ruiyuan Li, Huajun He, Rubin Wang, Yuchuan Huang, Junwen Liu, Sijie Ruan, Tianfu He, Jie Bao, and Yu Zheng. 2020. JUST: JD Urban Spatio-Temporal Data Engine. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. 1558–1569. https://doi.org/10.1109/ICDE48307.2020.00138
[35] Ruiyuan Li, Huajun He, Rubin Wang, Sijie Ruan, Yuan Sui, Jie Bao, and Yu Zheng. 2020. Trajmesa: A distributed nosql storage engine for big trajectory data. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 2002–2005.
[36] Ruiyuan Li, Rubin Wang, Junwen Liu, Zisheng Yu, Huajun He, Tianfu He, Sijie Ruan, Jie Bao, Chao Chen, Fuqiang Gu, Liang Hong, and Yu Zheng. 2021. Distributed Spatio-Temporal k Nearest Neighbors Join. In *Proceedings of the 29th International Conference on Advances in Geographic Information Systems* (Beijing, China) *(SIGSPATIAL '21)*. Association for Computing Machinery, New York, NY, USA, 435–445. https://doi.org/10.1145/3474717.3484209
[37] Yaguang Li, Rose Yu, Cyrus Shahabi, and Yan Liu. 2018. Diffusion Convolutional Recurrent Neural Network: Data-Driven Traffic Forecasting. In *International Conference on Learning Representations (ICLR '18)*.
[38] Binbing Liao, Jingqing Zhang, Ming Cai, Siliang Tang, Yifan Gao, Chao Wu, Shengwen Yang, Wenwu Zhu, Yike Guo, and Fei Wu. 2018. Dest-ResNet: A Deep Spatiotemporal Residual Network for Hotspot Traffic Speed Prediction. In *Proceedings of the 26th ACM International Conference on Multimedia* (Seoul, Republic of Korea) *(MM '18)*. Association for Computing Machinery, New York, NY, USA, 1883–1891. https://doi.org/10.1145/3240508.3240656
[39] Dongyu Liu, Di Weng, Yuhong Li, Jie Bao, Yu Zheng, Huamin Qu, and Yingcai Wu. 2017. SmartAdP: Visual Analytics of Large-scale Taxi Trajectories for Selecting Billboard Locations. *IEEE Transactions on Visualization and Computer Graphics* 23, 1 (2017), 1–10. https://doi.org/10.1109/TVCG.2016.2598432
[40] Jiamin Lu and Ralf Hartmut Güting. 2012. Parallel secondo: boosting database engines with hadoop. In *2012 IEEE 18th International Conference on Parallel and Distributed Systems*. IEEE, 738–743.
[41] Zhongjian Lv, Jiajie Xu, Kai Zheng, Hongzhi Yin, Pengpeng Zhao, and Xiaofang Zhou. 2018. LC-RNN: A Deep Learning Model for Traffic Speed Prediction. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*. International Joint Conferences on Artificial Intelligence Organization, 3470–3476. https://doi.org/10.24963/ijcai.2018/482
[42] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, Doris Xin, Reynold Xin, Michael J. Franklin, Reza Zadeh, Matei Zaharia, and Ameet Talwalkar. 2016. MLlib: Machine Learning in Apache Spark. *J. Mach. Learn. Res.* 17, 1 (jan 2016), 1235–1241.
[43] Paul Newson and John Krumm. 2009. Hidden Markov Map Matching through Noise and Sparseness. In *Proceedings of the 17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems* (Seattle, Washington) *(GIS '09)*. Association for Computing Machinery, New York, NY, USA, 336–343. https://doi.org/10.1145/1653771.1653818
[44] Jan Kristof Nidzwetzki and Ralf Hartmut Güting. 2018. BBoxDB - A Scalable Data Store for Multi-Dimensional Big Data. In *Proceedings of the 27th ACM International Conference on Information and Knowledge Management* (Torino, Italy) *(CIKM '18)*. Association for Computing Machinery, New York, NY, USA, 1867–1870. https://doi.org/10.1145/3269206.3269208
[45] Panagiotis Nikitopoulos, Akrivi Vlachou, Christos Doulkeridis, and George A Vouros. 2018. DiStRDF: Distributed Spatio-temporal RDF Queries on Spark.. In *EDBT/ICDT Workshops*. 125–132.
[46] Shoji Nishimura, Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. 2011. Md-hbase: A scalable multi-dimensional data infrastructure for location aware services. In *2011 IEEE 12th International Conference on Mobile Data Management*, Vol. 1. IEEE, 7–16.
[47] OpenStreetMap. 2022. Planet.osm. https://wiki.openstreetmap.org/wiki/Planet.osm.
[48] Varun Pandey, Andreas Kipf, Thomas Neumann, and Alfons Kemper. 2018. How Good Are Modern Spatial Analytics Systems? *Proc. VLDB Endow.* 11, 11 (July 2018), 1661–1673. https://doi.org/10.14778/3236187.3236213
[49] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019), 8026–8037.
[50] Maria Patrou, Md Mahbub Alam, Puya Memarzia, Suprio Ray, Virendra C. Bhavsar, Kenneth B. Kent, and Gerhard W. Dueck. 2018. DISTIL: A Distributed in-Memory Data Processing System for Location-Based Services. In *Proceedings of the 26th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems* (Seattle, Washington) *(SIGSPATIAL '18)*. Association for Computing Machinery, New York, NY, USA, 496–499. https://doi.org/10.1145/

3274895.3274961

[51] Jiwei Qin, Liangli Ma, and Jinghua Niu. 2019. THBase: A Coprocessor-Based Scheme for Big Trajectory Data Management. *Future Internet* 11, 1 (2019), 10.

[52] Filipe Rodrigues, Ioulia Markou, and Francisco C. Pereira. 2019. Combining time-series and textual data for taxi demand prediction in event areas: A deep learning approach. *Information Fusion* 49 (Sep 2019), 120–129. https://doi.org/10.1016/j.inffus.2018.07.007

[53] Hanan Samet. 1984. The Quadtree and Related Hierarchical Data Structures. *ACM Comput. Surv.* 16, 2 (June 1984), 187–260. https://doi.org/10.1145/356924.356930

[54] Zeyuan Shang, Guoliang Li, and Zhifeng Bao. 2018. DITA: Distributed In-Memory Trajectory Analytics. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 725–740. https://doi.org/10.1145/3183713.3183743

[55] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The Hadoop Distributed File System. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST) (MSST '10)*. IEEE Computer Society, USA, 1–10. https://doi.org/10.1109/MSST.2010.5496972

[56] Samriddhi Singla, Ahmed Eldawy, Tina Diao, Ayan Mukhopadhyay, and Elia Scudiero. 2021. The Raptor Join Operator for Processing Big Raster + Vector Data. In *Proceedings of the 29th International Conference on Advances in Geographic Information Systems* (Beijing, China) *(SIGSPATIAL '21)*. Association for Computing Machinery, New York, NY, USA, 324–335. https://doi.org/10.1145/3474717.3483971

[57] Ram Sriharsha. 2017. Magellan: Geospatial Analytics Using Spark. https://github.com/harsha2010/magellan

[58] 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. *Proc. VLDB Endow.* 9, 13 (Sept. 2016), 1565–1568. https://doi.org/10.14778/3007263.3007310

[59] Vu Tin, Eldawy Ahmed, Hristidis Vagelis, and Tsotras Vassilis. 2022. Incremental Partitioning for Efficient Spatial Data Analytics. *PVLDB* 15, 3 (2022), 713–726. https://doi.org/10.14778/3494124.3494150

[60] Panrong Tong, Mingqian Li, Mo Li, Jianqiang Huang, and Xiansheng Hua. 2021. Large-Scale Vehicle Trajectory Reconstruction with Camera Sensing Network. In *Proceedings of the 27th Annual International Conference on Mobile Computing and Networking* (New Orleans, Louisiana) *(MobiCom '21)*. Association for Computing Machinery, New York, NY, USA, 188–200. https://doi.org/10.1145/3447993.3448617

[61] Sheng Wang, Zhifeng Bao, Shixun Huang, and Rui Zhang. 2018. A Unified Processing Paradigm for Interactive Location-Based Web Search. In *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining* (Marina Del Rey, CA, USA) *(WSDM '18)*. Association for Computing Machinery, New York, NY, USA, 601–609. https://doi.org/10.1145/3159652.3159667

[62] Sheng Wang, Mingzhao Li, Yipeng Zhang, Zhifeng Bao, David Alexander Tedjopurnomo, and Xiaolin Qin. 2018. Trip Planning by an Integrated Search Paradigm. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 1673–1676. https://doi.org/10.1145/3183713.3193543

[63] Chris Wong. 2019. Pickup and drop-off locations of taxi rides in New York City. https://doi.org/10.6086/N13J3B0G Retrieved from UCR-STAR https://star.cs.ucr.edu/?NYCTaxi&d.

[64] Zhibin Xiao, Yang Wang, Kun Fu, and Fan Wu. 2017. Identifying different transportation modes from trajectory data using tree-based ensemble classifiers. *ISPRS International Journal of Geo-Information* 6, 2 (2017), 57.

[65] Dong Xie, Feifei Li, and Jeff M Phillips. 2017. Distributed trajectory similarity search. *Proceedings of the VLDB Endowment* 10, 11 (2017), 1478–1489.

[66] Dong Xie, Feifei Li, Bin Yao, Gefei Li, Liang Zhou, and Minyi Guo. 2016. Simba: Efficient In-Memory Spatial Analytics. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 1071–1085. https://doi.org/10.1145/2882903.2915237

[67] Jianpeng Xu, Xi Liu, Tyler Wilson, Pang-Ning Tan, Pouyan Hatami, and Lifeng Luo. 2018. MUSCAT: Multi-Scale Spatio-Temporal Learning with Application to Climate Modeling.. In *IJCAI*. 2912–2918.

[68] Simin You, Jianting Zhang, and Le Gruenwald. 2015. Large-scale spatial join query processing in Cloud. In *2015 31st IEEE International Conference on Data Engineering Workshops*. 34–41. https://doi.org/10.1109/ICDEW.2015.7129541

[69] 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* (Seattle, Washington) *(SIGSPATIAL '15)*. Association for Computing Machinery, New York, NY, USA, Article 70, 4 pages. https://doi.org/10.1145/2820783.2820860

[70] Qing Yu and Jian Yuan. 2022. TransBigData: A Python package for transportation spatio-temporal big data processing, analysis and visualization. *Journal of Open Source Software* 7, 71 (2022), 4021. https://doi.org/10.21105/joss.04021

[71] Haitao Yuan and Guoliang Li. 2019. Distributed In-memory Trajectory Similarity Search and Join on Road Network. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. 1262–1273. https://doi.org/10.1109/ICDE.2019.00115

[72] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. USENIX Association, San Jose, CA, 15–28. https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia

[73] Chao Zhang, Keyang Zhang, Quan Yuan, Luming Zhang, Tim Hanratty, and Jiawei Han. 2016. GMove: Group-Level Mobility Modeling Using Geo-Tagged Social Media. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (San Francisco, California, USA) *(KDD '16)*. Association for Computing Machinery, New York, NY, USA, 1305–1314. https://doi.org/10.1145/2939672.2939793

[74] Junbo Zhang, Yu Zheng, and Dekang Qi. 2017. Deep spatio-temporal residual networks for citywide crowd flows prediction. In *Thirty-first AAAI conference on artificial intelligence*.

[75] Yipeng Zhang, Yuchen Li, Zhifeng Bao, Songsong Mo, and Ping Zhang. 2019. Optimizing Impression Counts for Outdoor Advertising. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (Anchorage, AK, USA) *(KDD '19)*. Association for Computing Machinery, New York, NY, USA, 1205–1215. https://doi.org/10.1145/3292500.3330829

[76] Bolong Zheng, Lianggui Weng, Xi Zhao, Kai Zeng, Xiaofang Zhou, and Christian S Jensen. 2021. REPOSE: Distributed Top-k Trajectory Similarity Search with Local Reference Point Tries. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 708–719.

[77] Yu Zheng and Xing Xie. 2011. Learning Travel Recommendations from User-Generated GPS Traces. *ACM Trans. Intell. Syst. Technol.* 2, 1, Article 2 (jan 2011), 29 pages. https://doi.org/10.1145/1889681.1889683

[78] Yu Zheng, Xiuwen Yi, Ming Li, Ruiyuan Li, Zhangqing Shan, Eric Chang, and Tianrui Li. 2015. Forecasting Fine-Grained Air Quality Based on Big Data. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Sydney, NSW, Australia) *(KDD '15)*. Association for Computing Machinery, New York, NY, USA, 2267–2276. https://doi.org/10.1145/2783258.2788573

[79] Yu Zheng, Lizhu Zhang, Zhengxin Ma, Xing Xie, and Wei-Ying Ma. 2011. Recommending Friends and Locations Based on Individual Location History. *ACM Trans. Web* 5, 1, Article 5 (feb 2011), 44 pages. https://doi.org/10.1145/1921591.1921596

Received July 2022; revised October 2022; accepted November 2022