International Journal of
Geo-Information

Article

A Scalable Architecture for Real-Time Stream Processing of Spatiotemporal IoT Stream Data—Performance Analysis on the Example of Map Matching

Marius Laska [1,*], Stefan Herle [1], Ralf Klamma[2] and Jörg Blankenbach [1]

1 Geodetic Institute and Chair for Computing in Civil Engineering & Geo Information Systems, RWTH Aachen University, Mies-van-der-Rohe-Str. 1, 52074 Aachen, Germany; herle@gia.rwth-aachen.de (S.H.); blankenbach@gia.rwth-aachen.de (J.B.)

2 Advanced Community Information Systems Group (ACIS), RWTH Aachen University, Lehrstuhl Informatik 5, Ahornstr. 55, 52074 Aachen, Germany; klamma@dbis.rwth-aachen.de

* Correspondence: marius.laska@rwth-aachen.de

Received: 12 March 2018; Accepted: 19 June 2018; Published: 21 June 2018

Abstract: Scalable real-time processing of large amounts of data has become a research topic of particular importance due to the continuously rising amount of data that is generated by devices equipped with sensing components. While existing approaches allow for fault-tolerant and scalable stream processing, we present a pipeline architecture that consists of well-known open source tools to specifically integrate spatiotemporal internet of things (IoT) data streams. In a case study, we utilize the architecture to tackle the online map matching problem, a pre-processing step for trajectory mining algorithms. Given the rising amount of vehicle location data that is generated on a daily basis, existing map matching algorithms have to be implemented in a distributed manner to be executable in a stream processing framework that provides scalability. We demonstrate how to implement state-of-the-art map matching algorithms in our distributed stream processing pipeline and analyze measured latencies.

Keywords: stream processing; IoT; spatiotemporal; data mining; map matching

1. Introduction

Scalable real-time processing of large amounts of data has become a research topic of particular importance due to the continuously rising amount of data that is generated by devices equipped with sensing components. In order to gain direct insights from this data, real-time analysis is required. Typically, each smart-phone is equipped with several sensing components like an accelerator, a camera and a Global Navigation Satellite System (GNSS) component. Due to the wide adoption of GNSS components in sensing devices, the data that is generated from so-called spatiotemporal data streams [1], where each event is tagged with a timestamp and a location.

In particular, generating these spatiotemporal data streams results in challenges regarding the knowledge extraction, real-time processing, scalability and data integration. The range of applications in the area of real-time processing of spatiotemporal data streams reaches from vehicle tracking applications such as fleet tracking to the abnormal event detection in sensor networks that monitor dike characteristics to prevent the bursting of a dike.

Spatiotemporal data stream mining comes with certain well known challenges regarding stream processing, such as scalability, high availability, fault-tolerance etc. discussed by Cherniack et al. and Stonebraker et al. [2,3], which have been tackled by existing frameworks like

Apache Storm or Kafka streams. However, the spatiotemporal property of data and the origination from IoT devices adds another challenge, which is data integration of those streams into one of the existing stream processing frameworks. Algorithms like traffic prediction might only be executed on range specific data, since there exists only a certain area of interest. Furthermore, sensing devices often lack sufficient computation power, which is why resource efficient communication protocols have to be adopted in order to support data integration from all sorts of IoT devices. Several communication protocols have been developed to match the IoT requirements such as the Extensible Messaging and Presence Protocol (XMPP), the Constrained Application Protocol (CoAP) or the Message Queuing Telemetry Transport (MQTT), which have been compared by Karagiannis et al. [4]. MQTT is a resource efficient publish/subscribe communication protocol, which offers several scalable implementations of message brokers, which makes it suitable to handle IoT data streams. However, it lacks the capability of specifically integrating spatiotemporal data streams, since the subscription mechanism is only topic based. The Geospatial Message Queuing Telemetry Transport (GeoMQTT) protocol published by Herle et al. [5,6], tackles this issue by adding the functionality of tagging published data with a timestamp and a location and allowing subscriptions to certain areas and/or temporal intervals. It preserves the resource efficiency of MQTT and thus presents a well-suited standard for integrating spatiotemporal data streams from IoT devices.

In this applied paper, we demonstrate how to deal with spatiotemporal IoT stream data by utilizing open source tools and frameworks to tackle data integration as well as scalable stream processing. Utilizing GeoMQTT, we can specifically integrate spatiotemporal data streams, which is especially important in IoT settings, since most of the originating sensor data is tagged with a timestamp and a location. In our case study, we demonstrate how to utilize the proposed architecture to tackle one typical stream processing task of spatiotemporal data, which is online map matching, a pre-processing step for trajectory mining algorithms. Given the rising amount of vehicle location data that is generated on a daily basis, existing map matching algorithms have to be implemented in a distributed manner to be executable in a stream processing framework that provides scalability. We demonstrate how to implement a state-of-the-art map matching algorithm [7] in our distributed stream processing pipeline and analyze measured latencies, which are currently, to the best of our knowledge, missing in research.

The architecture consists of several tools that are chained to build a pipeline. Data integration is provided by using a GeoMQTT broker as the entry point of published messages, which are then forwarded to a Kafka broker in order to match spout characteristics of fault-tolerant stream processing, such as message replay and parallel topic consumption. We utilize the stream processing framework Apache Storm to implement distributed adaptions of well known online map matching algorithms, which consume messages from the Kafka broker. Finally, the results of the online map matching algorithms are published again via GeoMQTT to our broker and can be consumed by anyone subscribing to the output topic. The architecture allows for offering any spatiotemporal streaming application as a service. Consumers only have to publish their data via GeoMQTT to our message broker and can subscribe to the post-processed output.

The main contributions of our work are the reusable pipeline architecture that is capable of real-time processing of spatiotemporal data streams and the performance measurements of the distributed implementations of state-of-the-art map matching algorithms in the proposed architecture.

The paper is organized as follows: Section 2 discusses related work in the field of scalable stream processing architectures. Section 3 describes the proposed architecture, while briefly introducing the required tools. In Section 4, we present a case study, where the stream processing pipeline is implemented to perform map matching on raw trajectory streams using state-of-the-art online map matching algorithms. The architecture is deployed on a cluster and evaluated regarding its latency. In Section 5, we discuss further usage of the proposed architecture, while emphasizing the ability to offer the pipeline as a service. Section 6 finally summarizes the results of the paper.

2. Related Work

Efficient and scalable stream processing is a highly active research field due to the growing amount of data, which requires real-time analysis. There are several recent publications in the area of stream processing for data that mostly originates from IoT scenarios. However, most of the publications lack the opportunity to integrate spatiotemporal data streams via resource efficient communication protocols, which is required for our proposed stream processing architecture. Furthermore, latencies of implementations of state-of-the-art map matching algorithms in a stream processing framework are currently missing.

Villari et al. [8] propose an enhancement of the AllJoyn IoT framework that originally suffers from scalability problems due to not supporting the communication among devices belonging to different broadcast domains and the lack of any data management system. They utilize the Lambda architecture, which was originally published by Marz and Warren [9]. Operations on data are split into a speed layer and a batch layer. The speed layer works only on aggregations of the data to enable stream processing, while not affecting the batch layer that recomputes batch views on the complete data set. For implementing the speed layer, they use Apache Storm, a fault-tolerant distributed stream processing framework, which allows them to integrate data from heterogeneous sources. MongoDB, a highly scalable NoSQL database, serves as storage and processing tool for the batch processing. However, the AllJoyn Lambda architecture is built for local area networks with its own routing component (AllJoyn routing component), which handles discovery, connection management and security. We think that a stream processing architecture should be able to integrate data not only from local area networks, but enable data integration through standard IoT communication protocols. Relying on existing publish/subscribe protocols and open source tools like an MQTT broker cluster provides better interoperability and high scalability.

Thakur et al. [10] propose a real-time streaming and spatiotemporal analytics platform for gathering geospatial intelligence from open source data. Oriented towards the Lambda architecture, their approach supports seamless integration of archived as well as streaming data. The streaming data input sources originate from social media and the IoT. Although mentioning that the architecture is capable of spatiotemporal data analytics, they do not address how to integrate spatiotemporal data streams sourced from IoT devices.

Kamburugamuve et al. [11] propose a scalable cloud-based real-time data processing architecture, which focuses on integrating sensor data. They connect sensing devices to the cloud by using a topic-based publish-subscribe message broker and enable scalable stream processing via a distributed stream processing engine. They demonstrate the capabilities of their system in a robotics application. As their message broker for data integration, they utilize Apache Kafka, which allows for good scalability but does not by default support any lightweight IoT communication protocols. Furthermore, since its subscription mechanism is only topic based, it is not ideal for spatiotemporal data stream integration in contrast to the GeoMQTT protocol used in our architecture.

Zhou et al. [12] propose a system for efficient streaming access and cleaning of spatiotemporal data, which is also based on Apache Storm. It uses Apache Kafka, a distributed, partitioned, replicated commit log service in order to integrate the data into the Storm cluster. The produced data, which can originate, for instance, from the Hadoop Distributed File System (HDFS), the Web or other sources, is stored in the Kafka cluster, such that Storm can access it subsequently. In contrast to our pipeline architecture, the streaming results of their system are written to a MongoDB database, which allows users to query the results in real time.

Several other recent contributions exist in the area of stream processing of IoT data. Dey et al. [13] propose a real-time streaming system called Namatad, which leverages machine learning to infer insights from sensors within buildings. In the follow up work of Dey et al. [14], the authors also study the impact of alternative IoT processing topologies for real-time processing pipelines, leading to improved latency and accuracy. Furthermore, in the area of trajectory mining, Sun et al. [15] provide a framework, which is capable of tackling several demands in this area (including map matching) in contrast to several publications that only tackle a specific trajectory mining task.

Mattheis et al. [16] focus on the trajectory preprocessing task of online map matching applied to raw trajectories. The objective is to determine the underlying position in a road network for a given raw trajectory point. They propose a logical separation of map matching algorithms into several components in order to distributively implement them in the stream processing framework Apache Storm. They also use Apache Kafka as middleware tool to integrate the data into Storm and store the state of their map matching algorithms in the scalable NoSQL database Apache Cassandra. They do not explicitly cover IoT data integration and implement an online map matching algorithm which only considers the last set of position candidates to determine ratings for position candidates. Furthermore, they do not provide any performance measurements of their deployment.

Almeida et al. [17] also propose a distributed map matching algorithm, which they implement in the analytics engine Apache Spark to tackle the map matching problem in the context of Big Data. The published experimental results demonstrate the achieved accuracy while satisfying the required scalability. However, the authors tackle the offline map matching problem, which assumes that the trajectory points are not revealed one by one, but are available as a whole.

3. Methodology

For building our architecture, we assume that IoT devices are equipped with an GNSS receiver and publish spatiotemporal data to a certain message broker. In order to allow spatiotemporal data integration, we have to build a scalable broker cluster to which the IoT devices send their data. Furthermore, we have to provide spout characteristics like message replay and parallel consumption in order to inject the data into the stream processing framework. This section gives an overview of how to use existing tools to match the stated requirements when designing our scalable stream processing architecture.

3.1. Tools

The architecture has to integrate the dynamic data stream and perform a predefined mining task in near real time. We choose Apache Storm as a stream processing engine, which provides high scalability out of the box. However, the specific spatiotemporal stream data integration that allows for consuming streams based on spatiotemporal subscriptions requires more tools. A scalable publish/subscribe broker cluster for handling spatiotemporal events (geoevents) is required. Therefore, we utilize GeoMQTT as messaging protocol, which is a specialized extension to the MQTT protocol. Furthermore, Apache Storm needs an input source that allows for reprocessing of already accessed tuples. The tool used to satisfy this demand is Apache Kafka, which works in-between the GeoMQTT broker cluster and the Storm stream processing engine. The technologies are briefly introduced subsequently to finally propose the complete pipeline with all components plugged together.

3.1.1. Geospatial Message Queuing Telemetry Transport Protocol (GeoMQTT)

GeoMQTT, originally published by Herle et al. [5,6], is a spatiotemporal extension to the widely known and used IoT protocol MQTT. It adopts the publish/subscribe interaction scheme, where publishers produce certain information and send them to a broker, such that consumers can subscribe to them via specifying certain filters.

The protocol is event based, which means that, as soon as a message arrives at the broker, it is forwarded to all of the consumers whose subscription filters are satisfied by the message. MQTT aims to be suitable for constrained devices, typically present in the IoT domain by reducing communication overhead. In GeoMQTT, a published message can be tagged with a timestamp/time interval and a geometry additionally to the ordinary MQTT topic name. This way, it is possible for a consumer to define a temporal- and/or a spatial filter in addition to the topic filter of the ordinary MQTT subscription. Only if all filters are satisfied is the message forwarded to the subscribed consumer.

The publish/subscribe scheme of GeoMQTT is illustrated by Figure 1. First, the consumers subscribe their interests at the broker by specifying the introduced filters. If a message arrives at the GeoMQTT broker, the stored subscriptions are matched against the message header and the broker notifies the respective consumers.

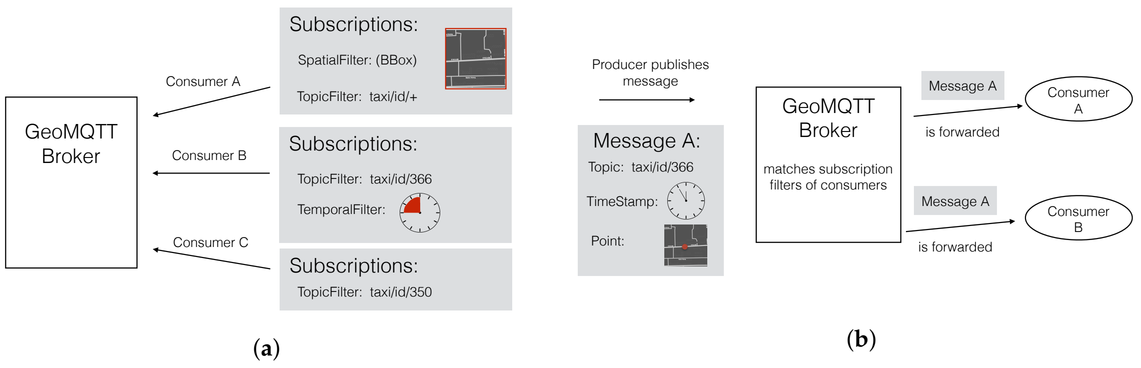
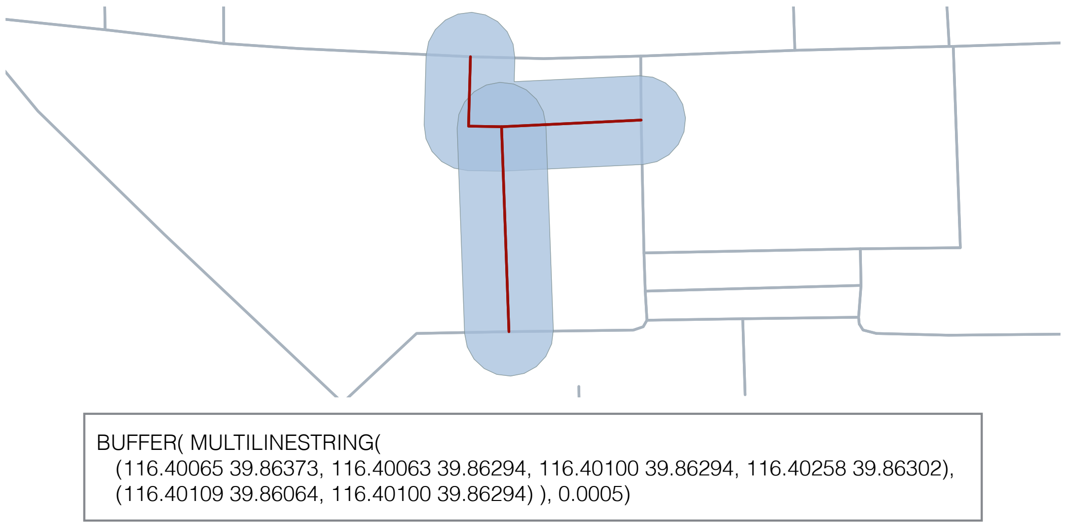


Figure 1. Publish/subscribe scheme of GeoMQTT. (**a**) subscription process; (**b**) notification process.

This subscription mechanism allows for easier integration of published data. Consider the use case, where a specific trajectory mining algorithm is meant to only operate on data which is located in a specific area, for example, to detect traffic congestions around a recently built intersection. By subscribing to a buffer around the relevant area, we can specifically integrate the data of interest. This use case is illustrated by Figure 2. The red lines depict an intersection, where, for example, new traffic lights have been installed. By specifying the spatial filter of GeoMQTT as given in the figure, we integrate all trajectory points of vehicles of the light blue area.



```
BUFFER( MULTILINESTRING(
  (116.40065 39.86373, 116.40063 39.86294, 116.40100 39.86294, 116.40258 39.86302),
  (116.40109 39.86064, 116.40100 39.86294) ), 0.0005)
```

Figure 2. Example of an intersection of interest (red lines) and the corresponding buffer with size 0.0005 (light blue). The corresponding spatial filter applied in GeoMQTT is depicted in the legend.

3.1.2. Apache Storm

Apache Storm is a stream processing engine, which has proven to be highly scalable and fault tolerant [9,18,19]. By defining the computation logic via so-called *topologies*, corresponding tasks are assigned on predefined worker nodes. This way, the computation logic and the actual assignment of tasks onto physical machines are separated, such that a machine failure will be automatically recovered by assigning new tasks according to the computation logic among the remaining physical machines.

The topologies mainly consist of two components. *Spouts* define the input of tuples into the streaming graph, whereas *Bolts* represent intermediate computation steps. Each Bolt can consume and output an arbitrary amount of input (respectively output) streams. Storm offers stream groupings for Bolts, such as spreading incoming tuples among all tasks of a Bolt using *shuffle grouping*, or splitting the stream over Bolt's tasks by hashing a subset of the incoming tuples using *field grouping*.

3.1.3. Apache Kafka

Since basic implementations of MQTT broker typically just forward any incoming message to clients that have subscribed to that topic, the consumer can only access each message once. Storm Spouts require the ability to reprocess tuples if a failure occurs inside the computation graph. Therefore, Apache Kafka [20] is introduced, a distributed, partitioned, replicated commit log service that provides message queuing as well as publish/subscribe semantics. It maintains a partitioned log for every topic. Messages inside the topic are stored for a certain period of time and are supplied with an offset that indicates the message's position inside the topic. Consumers are responsible for keeping their specific offset so they can access the correct messages. That enables the capability for consumers to re-access the message feed from a previous position for the sake of reprocessing messages. Storm provides Spout implementations to consume messages from a Kafka cluster, which makes Apache Kafka a suitable intermediate tool for the data integration into the stream processing engine Apache Storm.

3.2. Architecture of Stream Processing Pipeline

Starting from the assumption that there exist multiple dynamic streams of spatiotemporal data, which are transmitted via the publish/subscribe protocol GeoMQTT, we have to realize a first layer that processes this data and forwards it to subscribed consumers, namely a GeoMQTT broker cluster with a flexible amount of nodes that support the GeoMQTT protocol. The published GeoMQTT messages are subsequently forwarded to a Kafka cluster, such that a Storm Spout can consume them later on. This Kafka cluster should only contain messages that are specifically processed by the stream processing engine, which is realized by the early topic- and geospatial separation of data implemented in the GeoMQTT-Kafka bridge. Furthermore, we need a load balancing service among the entry point of the data streams. The final architecture is illustrated by Figure 3. We use the notations from [9] for the Storm components. The arrows indicate the data flow and, on the bottom of the figure, the target that is tackled by the specific architecture part is listed. The data is published to a GeoMQTT broker cluster of variable size. Via a Kafka bridge (for each GeoMQTT broker), the data is transferred to a Kafka cluster of variable size. Storm fetches the data from the Kafka cluster via the Kafka Spout. After the data is processed, it is published again to the Kafka cluster for subsequent processing or to the GeoMQTT broker such that it can be consumed.

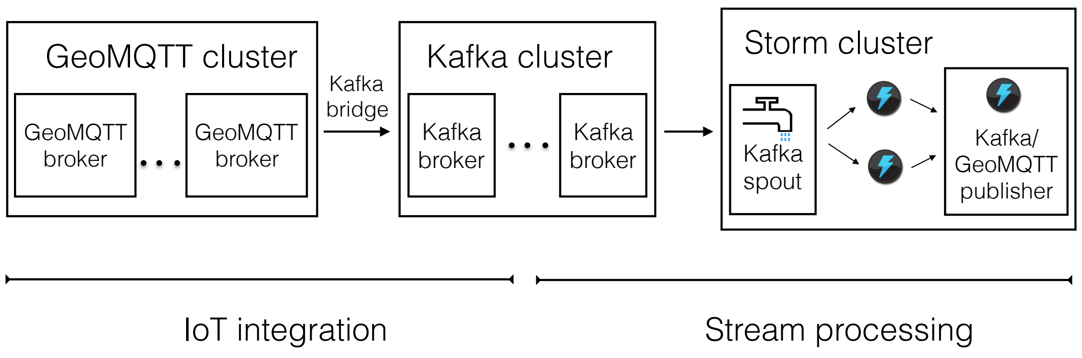


Figure 3. Complete architecture with focus on IoT data integration.

4. Case Study: Map Matching

Trajectory mining is one typical field, where real-time analysis of spatiotemporal data is required and thus is perfectly suitable as a case study to evaluate the proposed architecture. We will simulate data streams of GNSS locations of a dynamic amount of vehicles, which are published via GeoMQTT. Distributed state-of-the-art map matching algorithms are implemented for the usage in a Storm topology and finally the complete architecture is deployed on a local cluster and evaluated regarding its latency. We will first introduce the problem of map matching and present recent work regarding algorithms. Afterwards, we will present our data set that we use for evaluation and finally analyze the obtained results.

Typically, GNSS trajectories of vehicles are bound to errors, such that the determined locations do not always match the underlying road network. However, trajectory mining algorithms need to know on which road a vehicle is located at a certain point in time. According to Zheng [21] and Newson and Krumm [22], the map matching problem can be formulated as follows. Given a trajectory $\widetilde{T} = (\widetilde{tp}_1, \widetilde{tp}_2, ..., \widetilde{tp}_n)$ of raw trajectory points. For each $\widetilde{tp}_i \in \widetilde{T}$, determine a road segment e, such that e intersects the ground truth trajectory point $\overline{tp}_i \in \overline{T}$ of the ground truth route $\overline{T}$. The notations are illustrated in Figure 4. Depending on whether the complete trajectory is available or trajectory points are revealed over time, map matching algorithms are split into offline respectively online algorithms. Based on the deployed architecture, we focus on the online map matching approach.

In general, map matching algorithms can be classified according to the additional knowledge they apply [21]. *Geometric* algorithms specify the simplest category since they just snap the raw trajectory point to the closest road segment. *Topological* algorithms utilize additional knowledge such as connectivity of the road network. Another group of map matching algorithms can be classified as *probabilistic* algorithms. They generate a set of possible points on the road network that could match the corresponding raw trajectory point and rate them based on specified metrics. Finally, *advanced* map matching algorithms are a combination of all groups. Several position candidates are generated and rated based on geometric as well as connectivity based metrics. A probabilistic model serves as basis for the position candidate evaluation. In the map matching domain, especially the Hidden Markov Model (HMM) is widely adopted, since its model naturally fits the map matching problem [22].

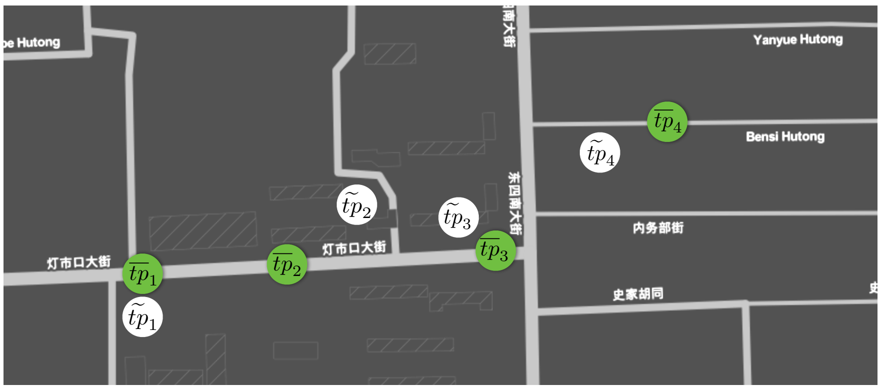


Figure 4. Notations for map matching. The white points depict the raw trajectory points $\widetilde{tp}_i$ received by the algorithm. The green points depict the corresponding ground truth trajectory points $\overline{tp}_i$, where the vehicle was actually located during a specific point in time.

4.1. Recent Work on Map Matching

Several map matching algorithms have been published in recent years, proposing online as well as offline algorithms. Most of the publications implement advanced map matching algorithms using the Hidden Markov Model (HMM) in order to provide a probabilistic model as a foundation [7,16,22,23]. Besides this, there exist other optimization approaches as suggested for example by Li et al. [24], who improve existing map matching algorithms by applying trajectory simplification prior to map matching them, which results in improved latency and accuracy.

Newson and Krumm [22] are some of the first researchers to invent an offline map matching algorithm based on the HMM. They use the HMM to obtain ratings for possible position candidates for a raw trajectory point. For any incoming trajectory point, a set of candidates (positions on possible road segments) is generated and subsequently evaluated according to the probabilistic model.

For the ratings, they incorporate the great circle distance between raw trajectory points and their associated position candidate. Furthermore, the shortest route between consecutive sets of position candidates is analyzed with the assumption that the shortest route between consecutive position candidates should be close to the great circle distance of their corresponding raw trajectory points.

Goh et al. [7] implement an HMM online map matching algorithm by extending existing solutions for an optimal sliding window approach to apply an online Viterbi algorithm. Their aim is to find an incremental solution that corresponds to a global solution in the end. The authors use the fact that when the current surviving paths converge at some point (convergence point) in the Markov chain, paths selected in the future will contain the same sub-path up to the convergence point. That implies that the outputs of partial solutions have to match the global one.

4.2. Storm Topology Design for Map Matching

Subsequently, we present the design of our Storm topology, which performs online map matching on raw trajectory data and thus serves as a proof of concept for the whole stream processing pipeline as proposed in this paper. In order to achieve good scalability of a Storm topology, the implemented map matching algorithms have to be split into several components allowing for scaling each of them individually. The following architecture of splitting the map matching algorithms into components is partially adopted from Mattheis et al. [16]. Map matching algorithms require mostly geospatial information related to a raw trajectory point, such that they can evaluate, for instance, distance measures regarding the underlying road network. The query process of this spatial data can be separated from the algorithm, such that it can be implemented in a single Bolt, which we will call Mapper Bolt subsequently. The map matching algorithm, which evaluates certain measures using the queried spatial data from the Mapper Bolt, is encapsulated in the so-called Matcher Bolt. An overview of the map matching topology is depicted by Figure 5.

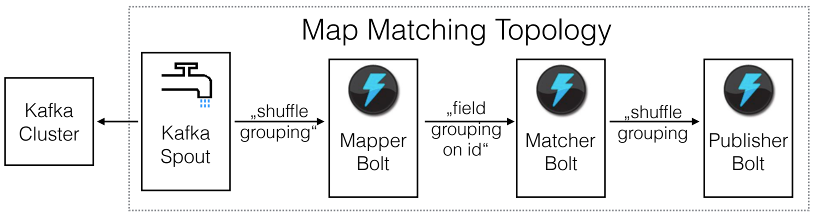


Figure 5. Map matching topology.

Since the query process of a Mapper Bolt is independent of its related vehicle, it can consume the messages of the Kafka Spout via the *shuffle grouping* paradigm. However, the Matcher Bolt computation might rely on previous results of regarding the same vehicle, which is why each Matcher Bolt's task has to receive all tuples of a specific vehicle. This can be realized by using Storm's *field grouping* paradigm. Finally, in the proposed architecture, the map matched trajectory points are again published using GeoMQTT together with the original timestamp and topic but now also with the map matched position. Alternatively, they can be published to the Kafka broker cluster for using them as input for subsequent analysis. This component is called the Publisher Bolt. Its task is independent of the related vehicle, which allows it to consume the stream via *shuffle grouping*.

4.3. Evaluation

We evaluated our architecture by deploying it on several virtual machines at a local server cluster. We simulated a subset of taxi trajectories gathered by Yuan et al. [25,26] and measured the latency inside the stream processing framework using the evaluation data provided by Storm, as well as the complete period, which a trajectory point spends in the whole architecture (including all components such as the GeoMQTT broker). In the following section, the deployment setup is described and the test data set is introduced. Relevant implementation details of the storm components are introduced, which are important for subsequent parameter tuning to achieve the best possible throughput.

4.3.1. Deployment Setup

In order to evaluate the architecture, we deployed it on a local cluster consisting of nine virtual machines with 2 GB RAM each, which are launched on a server with 64 cores. Two virtual machines are used for the databases that store the geometries such that the querying load can be split among them. The Kafka cluster and the GeoMQTT cluster only consist of one virtual machine each, since they did not turn out to be the limiting factor regarding overall throughput. Finally, the Storm topology consists of a Nimbus, a Zookeeper and three supervisor virtual machines. The simulation of the GeoMQTT trajectory data stream and the bridge to the Kafka cluster are operated on the same virtual machine the GeoMQTT broker is running on.

4.3.2. Test Data Set

The dataset used to evaluate our architecture contains the GNSS trajectories of 10,357 taxis captured during the period of 2 February to 8 February 2008 in Peking. The total number of points in this dataset is about 15 million and the total distance of the trajectories reaches 9 million kilometers. The average sampling interval between two consecutive points lies at about 177 s with a distance of about 623 m [25,26]. Since the used map matching algorithms are optimized towards low to medium frequencies of subsequent trajectory points, we picked a subset of vehicles for which the sampling rate is below 10 s in between consecutive sampling points. During the simulation, we set the publishing frequency to one trajectory point per taxi per second to obtain comparable results regarding latencies.

4.3.3. Total Latency Estimation

In the following, we describe the latency inside the Storm topology, which we capture via the feedback that Storm provides (Storm UI). Furthermore, we are interested in the total time a trajectory point needs to complete all stages of the pipeline. This includes the GeoMQTT broker, the forwarding to the Kafka cluster, the subsequent processing inside the Storm topology, where the map matching algorithms are implemented, plus the final publishing delay to the GeoMQTT broker. In order to capture this total delay of one trajectory point, we compare the timestamp right before the trajectory point is published via GeoMQTT with the timestamp recorded when the map matched result arrives at the GeoMQTT broker. When we simulate multiple taxis, we plot the complete latency on the vertical axis, whereas the index of the map matched output is present on the horizontal axis. Thus, we are able to analyze the total latency during certain time periods (independent from the specific vehicle, since we are only interested in the overall latency of our architecture over time).

4.3.4. Implementation and Parameter Tuning

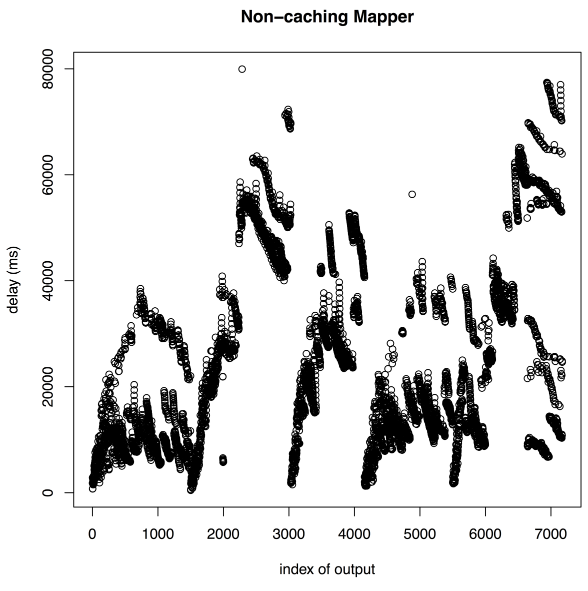
For further understanding regarding the upcoming description of the parameter estimation, we introduce relevant implementation details of the Mapper and the Matcher Bolt components. For both of them, we realized two implementations.

Mapper Bolt

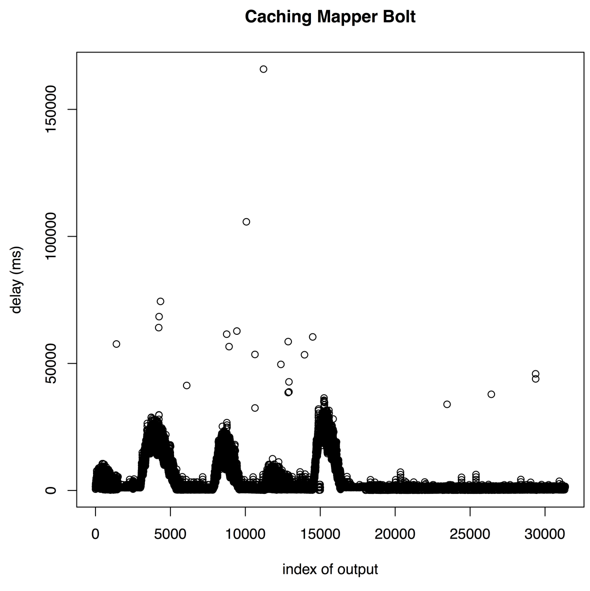
The Mapper Bolt is responsible for providing the geometries so that the Matcher Bolt is able to compute ratings for possible position candidates including topological information such as road connectivity. The first realization uses a PostgreSQL database with spatial support via PostGIS. For each raw trajectory point, the database is queried for relevant geometries. Thus, the load on the database increases linearly with the amount of trajectory points. One possibility to overcome this bottleneck is by replicating the PostGIS database and, therefore, load balancing the queries. In the deployment setup, the database has been replicated once.

However, with increasing size of trajectory points that have to be processed in real time, this could lead to very high demands regarding the size of the database cluster. This is why we implemented a second Mapper Bolt that uses caching to tackle this issue. The caching Mapper Bolt builds an index on already queried geometries such that it can check its cache for relevant geometries before querying the database. We claim that this approach yields to a higher latency at the beginning, since the cache has to build up first; nevertheless, once enough geometries have been collected, the latency of the Mapper Bolt decreases. The caching Mapper Bolt can be configured with three parameters: the spatial extent for which the database is queried in case of a cache fault, the minimum amount of geometries that the cache has to return such that the result is not considered to be a cache failure and the area for which the cache is queried. The difference between the extents of the queries for the cache and the database queries results from the idea that, in case of a cache failure, it is beneficial to query more geometries than required, to reduce the amount of upcoming cache failures.

When analyzing the total latency of trajectory points applying the non-caching mapper while simulating a relatively high amount of 100 taxis, the results we obtain diverge (Figure 6a) using 100 taxis with a publishing frequency of 1/s. However, if we run the same test while utilizing the caching Mapper Bolt, we obtain a total latency that converges to a value of approximately one second over time. After the cache has sufficiently built up, we arrive at a stable low latency (see Figure 6b).



(**a**)



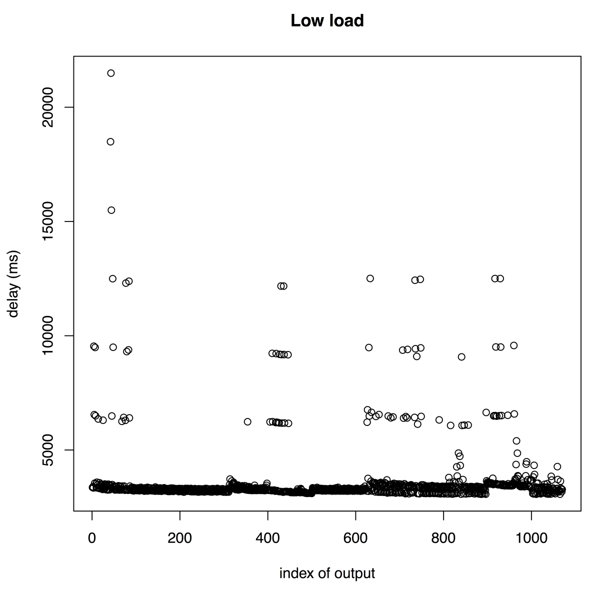
(**b**)

Figure 6. Total latency comparison between non-caching and caching Mapper Bolt on throughput of 100 trajectory points per second. (**a**) non-caching mapper bolt; (**b**) caching mapper bolt.

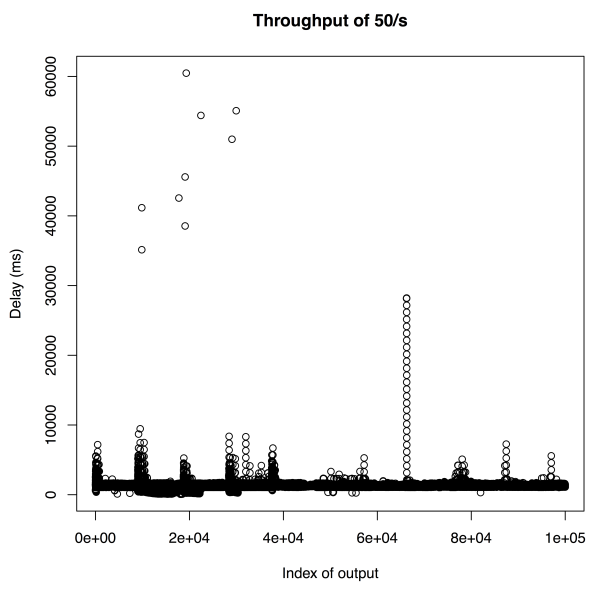
The optimal parametrization for the caching mapper was obtained by multiple test runs, where we analyzed the amount of database requests required to serve the matcher with a sufficient amount of geometries. We discovered that the spatial extent for which we query the database in case of a cache failure is optimally twice as large as the spatial extent that is delivered to the matcher, which is set to 5×10^{-4}. This helps to reduce subsequent database requests due to caching more geometries than needed for the current request. The minimum amount of geometries that we specify as sufficient when searching the cache for stored geometries influences the average amount of geometries that the matcher obtains in order to rate its position candidates. We obtained that setting this value to 50 exhibits best results regarding the amount of database requests and the average amount of geometries.

Matcher Bolt

The Matcher Bolt implements the determination of ratings for possible position candidates. We chose to implement the algorithms of Mattheis et al. [16] and Goh et al. [7], which have been described above. The first one emits a mapped trajectory point immediately for every incoming raw trajectory point, while generating ratings for possible position candidates and using connectivity to the previous set of position candidates. The latter one uses a sliding window approach, which only emits a mapped trajectory point if a convergence point in the computed Markov chain is found. Thus, the major difference is that the maximizing path along the Markov chain for two or more sets of position candidates is determined, while the other one just uses Markov chain path maximization for two sets of position candidates. This means that the Window Matcher Bolt has a delay that depends on the frequency of incoming trajectory points. If we set the frequency of incoming trajectory points to 1/3 s, we expect latencies to be a multiple of 3 s plus a processing delay depending on the size of the current Markov chain for which the path is maximized. This is illustrated by Figure 7a, where we set the frequency of published trajectory points to 1/3 s for a test set of 10 taxis.



(**a**)



(**b**)

Figure 7. Total latency comparison on low and medium throughput. (**a**) latency on low throughput of 10 trajectory points per second; (**b**) latency on throughput of 50 trajectory points per second.

4.3.5. Topology under Load

In this section, we will focus on the evaluation of the topology using the Matcher Bolt that implements the algorithm of Goh et al. [7] and the caching Mapper Bolt, since the sliding window approach achieves better accuracy results when comparing the two algorithms. We noticed that, when increasing the load on the topology, the total latency of the tuples of the topology increases and exceeds the sum of the latencies of the individual components, which is due to the time tuples spent in internal buffers of the topology. This phenomenon can be controlled by introducing a back pressure in the Storm Spout. Setting the MAX_SPOUT_PENDING value, we can force the Spout to stop emitting tuples in the topology if the amount of unacknowledged tuples in the topology exceeds this value. As a result, published messages that arrive in the Kafka cluster are not directly emitted into the topology, such that, when looking at the complete processing latency of trajectory points (including all stages of the pipeline), we notice a heavy increase whenever the Storm Spout is throttled. This is illustrated in Figure 6b, where a publishing frequency of 1/s is used to simulate 100 taxis. As soon as the Kafka Spout is throttled, the complete processing delay of trajectory points increases additionally to the delay, if no overload of the topology occurs. After we arrive at the peak delay, we have multiple trajectory points stored that can be processed without waiting for upcoming ones, since they are already present, which is the reason why the delay decreases again. After a certain time, the total latency has stabilized, due to more efficient geometry provision of the Mapper Bolt. Its cache has built up, such that the time consuming database queries are reduced. The higher we set the MAX_SPOUT_PENDING value, the higher are the delays of the peaks, however the faster the system stabilizes.

If we increase the throughput to a certain level (above 150 taxis which publish at a frequency of 1/s), the total latency of map matched trajectory points does not stabilize after the initial peaks but shows peaks of roughly the same size again, which can be seen in Figure 8b. Thus, going above this level of throughput is not reasonable for our test deployment. Since our deployment setting consists only of a single server, further investigations into the performance of the architecture in a real distributed setup have to be undergone in future work. However, the architecture is designed to be easily scalable by choosing open source components that can each be replicated separately.

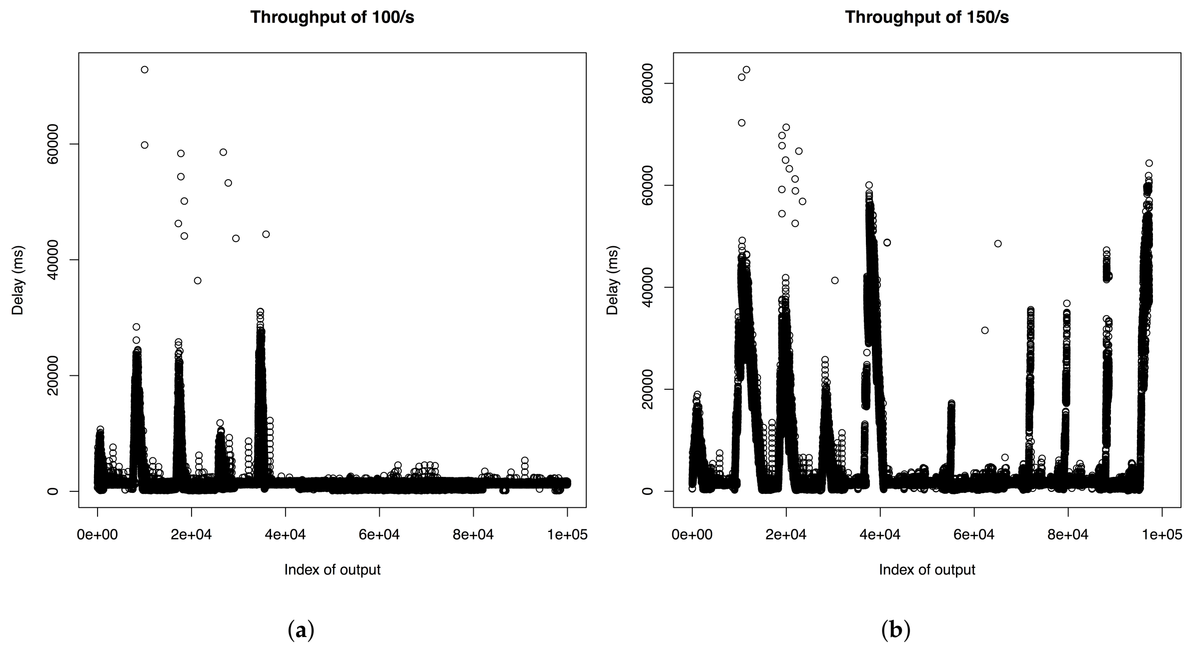


Figure 8. Total latency comparison on medium and high throughput.(**a**) latency on throughput of 100 trajectory points per second; (**b**) latency on throughput of 150 trajectory points per second.

5. Outlook

In this paper, we realized the implementation of our proposed real-time stream processing architecture for one application field of spatiotemporal stream mining, namely online map matching. However, the architecture is perfectly suited for implementing any other spatial or non-spatial stream mining algorithm in the Apache Storm framework and reusing the data integration layer of the architecture. The architecture can be offered as a service, since it is implemented as a pipeline with GeoMQTT as input and output. Herle and Blankenbach proposed the concept of GeoPipes [5], which matches the interfaces of our pipeline architecture. They demonstrate the integration of the GeoPipes idea in the Web Processing Framework (WPS) interface to expose standardized real-time geo-processing services [27]. As a proof of concept, we implemented a WPS map matching service that utilizes our architecture to offer the algorithms as a standardized real-time geo-processing service and offered the map matching algorithm via a web application.

For understanding how our pipeline architecture can be used in practice, consider the following use case: A company wants to offer a fleet tracking platform, where they provide real-time services to their customers such as map matching. Customers provide the gathered location data of their vehicle fleet, which is equipped with GNSS sensors. In order to simplify the data provision process, the company wants to rely on open standard communication protocols. By utilizing our proposed architecture, the company only needs to implement the streaming algorithms in the Apache Storm framework while reusing the other components of the architecture. While customers publish all the location data to the GeoMQTT broker, the GeoMQTT protocol allows the company to specifically integrate data, which is relevant for a specific stream processing algorithm. Customers like bus companies might be interested in analyzing traffic congestions regarding a recent route optimization. By only subscribing to the target area using spatial filters, relevant data can be specifically fed into the stream processing algorithm.

6. Conclusions

In this paper, we proposed a real-time stream processing pipeline that allows for spatiotemporal data stream integration from IoT devices. The data integration layer enables geospatial subscriptions, utilizing the GeoMQTT protocol. This allows for target specific data integration while preserving capabilities of gathering data from IoT devices due to the resource efficiency of GeoMQTT. We utilize the state-of-the-art stream processing framework Apache Storm as the core tool for our architecture and Apache Kafka as tool for message processing between GeoMQTT broker and Apache Storm. We demonstrated the capabilities of the proposed architecture by implementing efficient map matching algorithms and evaluated them on a distributed deployment on a local cluster. In contrast to existing approaches that implement map matching algorithms in a stream processing framework, we provide detailed latency evaluations of a distributed implementation of the online map matching algorithm proposed by Goh et al. [7]. This algorithm achieves more accurate results than comparable algorithms based on the HMM by using a sliding window approach.

Simulating a sample set of taxi trajectories from Peking, we obtained stable latencies in the area of milliseconds, when not exceeding a simulated publishing ratio of 100 trajectory points per second. In our current setting, we only evaluated our pipeline architecture on a single server. However, the architecture is deployed using several virtual machines and relies on open source tools that can be replicated easily. In further investigations of the architecture, we will incorporate more rigorous checks regarding the latency in a more powerful cloud setting and also check the robustness of the system, respectively, of the algorithms under distributed failover.

The proposed architecture can be used to implement applications for several use cases of deploying and evaluating distributed stream processing algorithms that operate on spatiotemporal data streams originating from IoT devices.

Author Contributions: M.L., S.H., R.K. and J.B. designed the methodology; M.L. and S.H. conceived and conducted the experiments; R.K. and J.B. administrated and supervised the research project; and M.L. wrote the paper.

Funding: This research has been conducted as part of the EarlyDike project (http://www.earlydike.de/), which is funded by the German Federal Ministry of Education and Research (BMBF) within the GEOTECHNOLOGIEN programme (03G0847A).

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Galić, Z. *Spatio-Temporal Data Streams*; Springer: Berlin, Germany, 2016.
2. Cherniack, M.; Balakrishnan, H.; Balazinska, M.; Carney, D.; Cetintemel, U.; Xing, Y.; Zdonik, S.B. Scalable distributed stream processing. In Proceedings of the First Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, 5–8 January 2003; Volume 3, pp. 257–268.
3. Stonebraker, M.; Çetintemel, U.; Zdonik, S. The 8 requirements of real-time stream processing. *SIGMOD Rec.* **2005**, *34*, 42–47. [CrossRef]
4. Karagiannis, V.; Chatzimisios, P.; Vazquez-Gallego, F.; Alonso-Zarate, J. A survey on application layer protocols for the internet of things. *Trans. IoT Cloud Comput.* **2015**, *3*, 11–17.
5. Herle, S.; Blankenbach, J. GeoPipes using GeoMQTT. In *Geospatial Data in a Changing World*; Springer: Berlin, Germany, 2016; pp. 383–398.
6. Herle, S.; Becker, R.; Blankenbach, J. Bridging GeoMQTT and REST. In Proceedings of the Geospatial Sensor Webs Conferenc, Münster, Germany, 29–31 August 2016; pp. 1–5.
7. Goh, C.Y.; Dauwels, J.; Mitrovic, N.; Asif, M.T.; Oran, A.; Jaillet, P. Online map-matching based on hidden markov model for real-time traffic sensing applications. In Proceedings of the 2012 15th International IEEE Conference on Intelligent Transportation Systems (ITSC), Anchorage, AK, USA, 16–19 September 2012; pp. 776–781.
8. Villari, M.; Celesti, A.; Fazio, M.; Puliafito, A. AllJoyn Lambda: An architecture for the management of smart environments in IoT. In Proceedings of the 2014 International Conference on Smart Computing Workshops (SMARTCOMP Workshops), Hong Kong, China, 5 November 2014; pp. 9–14.
9. Marz, N.; Warren, J. *Big Data, Principles and Best Practices of Scalable Real-Time Data Systems*; Manning Publications: Shelter Island, NY, USA, 2015; Volume 37, pp. 1–23.
10. Thakur, G.S.; Bhaduri, B.L.; Piburn, J.O.; Sims, K.M.; Stewart, R.N.; Urban, M.L. PlanetSense: A real-time streaming and spatio-temporal analytics platform for gathering geo-spatial intelligence from open source data. In Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems, Seattle, WA, USA, 3–6 November 2015; ACM: New York, NY, USA, 2015; pp. 11:1–11:4. [CrossRef]
11. Kamburugamuve, S.; Christiansen, L.; Fox, G. A framework for real time processing of sensor data in the cloud. *J. Sens.* **2015**, *2015*, 468047. [CrossRef]
12. Zhou, L.; Chen, N.; Chen, Z. Efficient streaming mass spatio-temporal vehicle data access in urban sensor networks based on Apache Storm. *Sensors* **2017**, *17*, 815. [CrossRef] [PubMed]
13. Dey, A.; Ling, X.; Syed, A.; Zheng, Y.; Landowski, B.; Anderson, D.; Stuart, K.; Tolentino, M.E. Namatad: Inferring occupancy from building sensors using machine learning. In Proceedings of the 2016 IEEE 3rd World Forum on Internet of Things (WF-IoT), Reston, VA, USA, 12–14 December 2016; pp. 478–483. [CrossRef]
14. Dey, A.; Stuart, K.; Tolentino, M.E. Characterizing the impact of topology on IoT stream processing. In Proceedings of the 2018 IEEE 4th World Forum on Internet of Things (WF-IoT), Singapore, 5–8 February 2018; pp. 505–510.
15. Sun, W.; Zhu, J.; Duan, N.; Gao, P.; Hu, G.Q.; Dong, W.S.; Wang, Z.H.; Zhang, X.; Ji, P.; Ma, C.Y.; Huang, J.C. Moving object map analytics: A framework enabling contextual spatial-temporal analytics of internet of things applications. In Proceedings of the 2016 IEEE International Conference on Service Operations and Logistics, and Informatics (SOLI), Beijing, China, 10–12 July 2016; pp. 101–106. [CrossRef]
16. Mattheis, S.; Al-Zahid, K.K.; Engelmann, B.; Hildisch, A.; Holder, S.; Lazarevych, O.; Mohr, D.; Sedlmeier, F.; Zinck, R. Putting the car on the map: A scalable map matching system for the open source community. In Proceedings of the Informatik 2014, Stuttgart, Deutschland, 22–26 September 2014; pp. 2109–2119.
17. Almeida, A.M.R.; Lima, M.I.V.; Macedo, J.A.F.; Machado, J.C. DMM: A distributed map-matching algorithm using the MapReduce paradigm. In Proceedings of the 2016 IEEE 19th International Conference on Intelligent Transportation Systems (ITSC), Rio de Janeiro, Brazil, 1–4 November 2016; pp. 1706–1711. [CrossRef]
18. Apache Software Foundation. *Apache Storm*; Apache Software Foundation: Forest Hill, MA, USA, 2015.
19. Ranjan, R. Streaming big data processing in datacenter clouds. *IEEE Cloud Comput.* **2014**, *1*, 78–83. [CrossRef]
20. Apache Software Foundation. *Apache Kafka*; Apache Software Foundation: Forest Hill, MA, USA, 2016.
21. Zheng, Y. Trajectory data mining: An overview. *ACM Trans. Intell. Syst. Technol.* **2015**, *6*, 29. [CrossRef]

22. Newson, P.; Krumm, J. Hidden markov map matching through noise and sparseness. In Proceedings of the 17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, Seattle, WA, USA, 4–6 November 2009; ACM: New York, NY, USA, 2009; pp. 336–343. [CrossRef]
23. Lou, Y.; Zhang, C.; Zheng, Y.; Xie, X.; Wang, W.; Huang, Y. Map-matching for low-sampling-rate GPS trajectories. In Proceedings of the 17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, Seattle, WA, USA, 4–6 November 2009; ACM: New York, NY, USA, 2009; pp. 352–361. [CrossRef]
24. Li, H.; Kulik, L.; Ramamohanarao, K. Spatio-temporal trajectory simplification for inferring travel paths. In Proceedings of the 22nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, Dallas, TX, USA, 4–7 November 2014; ACM: New York, NY, USA, 2014; pp. 63–72. [CrossRef]
25. Yuan, J.; Zheng, Y.; Zhang, C.; Xie, W.; Xie, X.; Sun, G.; Huang, Y. T-drive: Driving directions based on taxi trajectories. In Proceedings of the 18th SIGSPATIAL International Conference on Advances in Geographic Information Systems, San Jose, CA, USA, 2–5 November 2010; ACM: New York, NY, USA, 2010; pp. 99–108. [CrossRef]
26. Yuan, J.; Zheng, Y.; Xie, X.; Sun, G. Driving with knowledge from the physical world. In Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Diego, CA, USA, 21–24 August 2011; ACM: New York, NY, USA, 2011; pp. 316–324. [CrossRef]
27. Herle, S.; Blankenbach, J. Enhancing the OGC WPS interface with GeoPipes support for real-time geoprocessing. *Int. J. Digit. Earth* **2017**, *11*, 48–63. [CrossRef]