Scalable Map Matching with Apache Spark Streaming

Achintya Kumar, Nishanth EV
Digital Engineering Project

***Abstract*—Recent times have seen a remarkable increase in the number of mobile devices connected to the Internet and a variety of uses based on their location data have come into existence. This increase in number necessitates a scalable map-matching solution to meet the ever-increasing demand. Map Matching is the problem of how to match recorded noisy geographic coordinates to a logical model of the real world. Thereby, measured GPS coordinates are matched onto a reference coordinate system (e.g. a map). The existing solutions for map matching do not scale and therefore are not suitable for Big Data. This paper is mainly based on research conducted by Newson & Krumm [1] and Mattheis et al [2]. Based on the above, this paper is aimed at developing a map matching application that runs on Apache Spark Streaming, where the map matching is done in a scaleable and in a streaming fashion.**

***Keywords*—Cluster Computing, map-matching, scalability.**

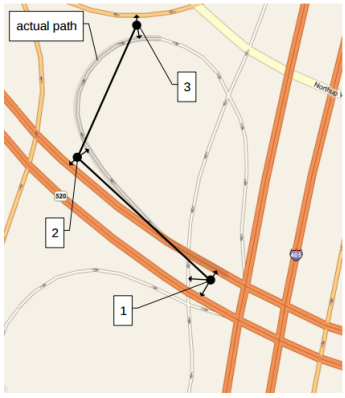
I. Introduction

The process of finding the most likely position of a vehicle on a given network of streets using noisy location data (eg GPS receiver) is called map matching. Newson and Krumm[1] have argued how map-matching has become increasingly important in real time as vehicles can be used as traffic probes for measuring road speeds and building statistical models of traffic delays. The constructed models can then be used to find time-optimal driving routes in order to avoid traffic jams. This paper at first describes the current state of the art algorithm used to perform map-matching given by Newson and Krumm[1]. This paper secondly discusses implementation of a fully open-source scalable map matching system using Apache Spark Streaming based on design suggested by Mattheis et al[3]. The paper also discusses some of the techniques to enhance processing capacity of the application through stateful transformations in Spark, increased resource allocation and sample-rate reduction in the application backend.

II. Literature Review

A. The Map-Matching Problem

According to [1] and as seen on Figure 1, despite location data being the sole indicator of the path, naively matching each point to the nearest road can lead to highly unreasonable paths and bizarre driving behavior. To avoid such unreasonable paths, it becomes necessary to introduce the knowledge of road network to prevent the aforementioned bizarre behavior. In order to integrate noisy location data with road networks, an algorithm based on Hidden Markov Model is proposed.
Hidden Markov Model is used to describe a system's behavior over time t where the sequence of system states(x) is hidden and can only be observed through a sequence of emissions(y).



Source: [1]
Fig. 1: According to [1], map matching consists of matching measured locations (black dots) to the road network in order to infer the vehicles actual path (light gray curve). Merely matching to the nearest road is prone to mistakes.

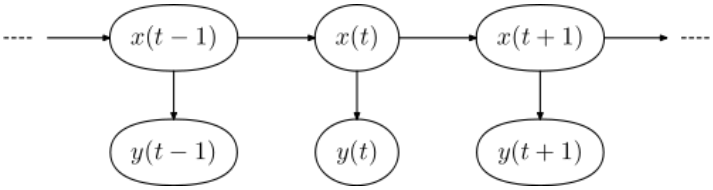
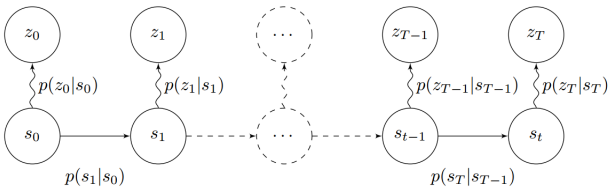


Fig. 2: A first-order Hidden Markov Model where sequence of hidden system states is denoted by x and sequence of emissions is denoted by y, over time t

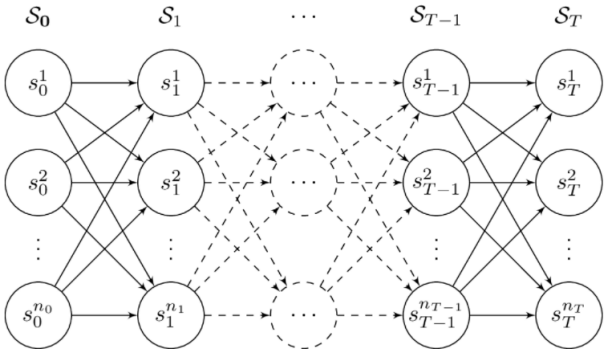
Using the principle of Hidden Markov Models, the problem of map-matching can be solved probabilistically.

B. HMM Map-Matching

This subsection describes in detail about the process of map-matching using Hidden Markov Models. The HMM can be used to model processes which involve a path through several possible states where certain state transitions are more likely than the rest and the state measurements are uncertain. According to [1], in the problem of map-matching, the states of the HMM are the individual road segments, and the state measurements are noisy and therefore uncertain vehicle location data. The goal is to match every location measurement with the correct road segment and this naturally fits the HMM as transitions between road segments are ruled by the connectivity of the road network.



Source: [3]
Fig. 3: According to [3], first-order Hidden Markov Model for map matching. Every emission z_t is the outcome of an observation (with wavy arrows) of the uncertain systems state s_t at time $t \in [0, T]$. The state transitions (straight arrows) are part of a stochastic process having transition probability $p(s_t|s_{t-1})$. A state refers to a map position of a vehicle and emissions refer to position measurements. The transition between system states corresponds to the connected road segments or the path between map positions.



Source: [2]
Fig. 4: Illustration of HMM with matching candidates and state transitions

According to [3], every location measurement over time $t \in [0, T]$ can be denoted by z_t and system states over T can be denoted by sequence (s_0, ... , s_T). Since these states are hidden, they can be observed through a sequence of emissions (z_0, ... , z_T). Each position measurement z_t corresponds to a position s_t on the map. Since the position measurements are prone to noise, they may be matched to more than one state. This finite set of the most likely states (or in our case, road segments) is called state (or position) candidates S_t. Each emission's uncertainty due to measurement noise can be modelled with emission probability $p(z_t|s_t)$. The state transitions depicted with straight arrows as in Figure 3 are denoted as transition probability $p(s_t|s_{t-1})$.

[3] argue that for every incoming location measurement sample z_t, map matching is performed by finding the most likely state ${s_t}^i$ from the matching candidates $S_t = ({s_t}^1, ..., {s_t}^n)$. S_t is also known as candidate vector. Similarly, for every two consecutive candidate vectors S_t and S_{t+1}, there exists a transition between each pair of matching candidates ${s_t}^i$ and ${s_{t+1}}^j$ which signifies the route between the map positions. The figure 4 illustrates the aforementioned.

Based on [1] and [3], the emission probability $p(z_t|s_t)$ is given as a Gaussian distribution as Equation 1.

$$p(z_t|s_t) \sim \frac{1}{\sqrt{2\pi\alpha_z^2}} exp\left\{-\frac{\|z_t - s_t\|^2}{2\alpha^2}\right\}, \quad (1)$$

where α_z is the standard deviation of GPS measurements and $\|z_t - s_t\|$ is the geodesic distance between map location s_t and measurement z_t. Similarly, in order to compute transition probabilities, the routing from s_{t-1} to s_t must be done. The resultant path obtained is denoted as $\langle s_{t-1}, s_t \rangle$ and the length of the path is given as $|\langle s_{t-1}, s_t \rangle|$. Then, based on [1], "transition probabilities have been experimentally determined to fit a negative exponential distribution" and can be given by Equation 2.

$$p(s_t|s_{t-1}) \sim \lambda exp\left\{\lambda\left(\|z_t - z_{t-1}\| - |\langle s_{t-1}, s_t \rangle|\right)\right\}, \quad (2)$$

where the best estimate of λ for a specific sampling set remains to be calculated.

Now, according to [3], online map matching is performed by finding the most likely map position $\tilde{s}_t$ using Equation 3.

$$\tilde{s}_t = \underset{s_t}{\operatorname{argmax}}\, p\left(s_t \mid z_0, \ldots, z_t\right), \quad (3)$$

where $s_t \in S_t$ and $t \in [0, T]$. For a given set of position measurements containing up to the most recent value, i.e. $(z_0, \ldots, z_t)$, the part $p\left(s_t \mid z_0, \ldots, z_t\right)$ gives the probability that position candidate s_t corresponds to the real position of the vehicle on the road network and is referred to as filter probability of s^t. It can be calculated recursively as in Equation 4.

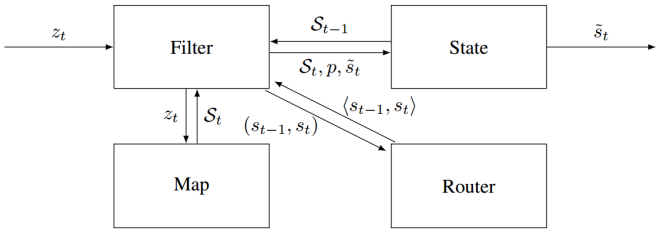
$$p(s_t|z_0 \ldots z_t) = \alpha \cdot p(z_t|s_t) \cdot \sum_{s_{t-1}}^{S_{t-1}} p(s_t|s_{t-1}) \cdot p(s_{t-1}|z_0 \ldots z_{t-1}), \quad (4)$$

where $p(s_t|z_0 \ldots z_t)$ is filter probability of $s_t \in S_t$, $p(s_{t-1}|z_0 \ldots z_{t-1})$ is the filter probability of s_{t-1}. Furthermore, $p(z_t|s_t)$ is the emission probability of position candidate s_t given z_t. The transition probability from state s_{t-1} to s_t is given by $p(s_t|s_{t-1})$. α is a normalizing constant. [3]

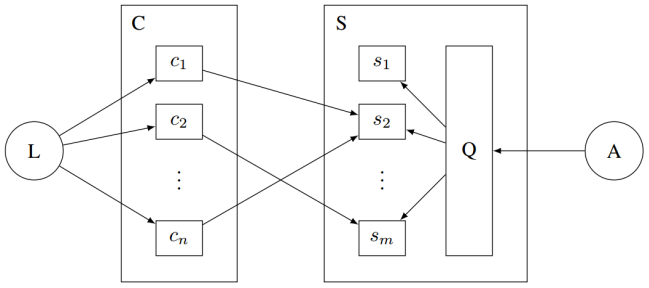
C. System Architecture: HMM Map Matching

Mattheis et al [3] propose and describe a system architecture that can be linearly scaled depending upon the requirements. According to them, the map matching system has to process a stream of position measurements $(z_0, \ldots, z_t)$ to determine the most likely position $\tilde{s}_t$ of a vehicle on a map that corresponds to the most recent position measurement z_t. [3]'s architecture consists of an online map matching component (Filter), a geometrical road map (Map), a routing unit with a separate road map topology (Router) and a memory unit that provides easy access to state information of the tracked vehicle (State).

According to [3], the following is the algorithm for the functioning of the proposed map matching system:



Source: [3]
Fig. 5: System architecture of a HMM map matching system consisting of a Filter/Matcher, Map, Router and State memory.



Source: [3]
Fig. 6: Scalable system architecture for online map matching.

1) Matcher (or filter) receives a position measurement z_t. For this sample, the filter would find all the candidate vectors S_t from the map component.
2) The filter requests the path $\langle s_{t-1}, s_t \rangle$ for each pair of position candidates (s_{t-1}, s_t) with $s_{t-1} \in S_{t-1}$ and $s_t \in S_t$. The path is provided by the Router component and the candidate vectors S_{t-1} at the previous time interval $t-1$ are provided by the State memory component. If $t-1$ is invalid, then an empty candidate vector is returned.
3) The filter determines emission probabilities $p(z_t|s_t)$, transition probabilities $p(s_t|s_{t-1})$, and posterior probabilities $p(s_t|z_0...z_t)$ for each pair of position candidates (s_{t-1}, s_t) with $s_{t-1} \in S_{t-1}$ and $s_t \in S_t$. The vehicles state information, i.e. map position candidates S_t, probabilities $p(s_t|z_0...z_t)$ for each $s_t \in S_t$ (in figure 5 above denoted as p) and the most likely map position $\tilde{s}_t$, is saved to memory.

D. Strategies for Scalability

The number of vehicles that the map matching system tracks can vary over time. The number of position measurements that it processes per second is called system's load. The system must be able to process this varying load without any degradation in performance. According to [3], a system is scalable if it ensures a fixed response time independently of its load by increasing computing, storage and communication elements.

This can be achieved by horizontal scaling of the aforementioned system using Apache Spark Streaming. Apache Spark is defined as a general purpose open-source cluster-computing framework for big data processing. Spark Streaming is a built-in module of Spark for stream processing. In case of map matching, the stream of position measurements from vehicles is the data meant to be processed.

According to [3], the scalability can be achieved as shown in Figure 6 below.

As seen in Figure 6, according to [3], the scalable architecture uses two major clusters, i.e compute cluster (**C**), with n nodes, and storage cluster (**S**), with m nodes. Every node of the compute cluster **C** runs an instance of map matching system which consists of a matcher, a router and a map components. On the other hand, the state memory is distributed over the storage cluster (**S**) nodes $s_1...s_m$. A load-balancer (L) distributes incoming position measurement samples over compute nodes. Each compute node performs map matching as described for a single-node system, while it reads and writes state information from or to the storage cluster. The map matching described for a single-node system is performed on every compute node and the state information is read from or written to the storage cluster. Each state information is stored individually for every vehicle and is distinguished by an object identifier. On the right side of Figure 6, one can observe service applications (**A**) querying state information through query later (**Q**) provided by the storage cluster.

[3] propose that the following are the major design decisions to be made with this architecture:

1) The compute and storage cluster can be physically separated. This enables independent scaling of cluster, whichever is running out of resources, ie either computing or storage resources.
2) With advanced approaches, the clusters **C** and **S** may be separated only logically to enable physical locality of computing and storage resources.
3) In order to simplify load-balancing, every message can be processed by any of the compute nodes as the nodes of the compute cluster **C** are interchangeable.

III. Methods

For implementation purpose, Barefoot[2] has been chosen as to serve as the map-matching library. It is based on HMM map-matching discussed in the preceding sections.

A. Single Node Implementation Algorithm

Based on Barefoot[2], online map-matching implementation on a single-node system can be illustrated with the following flowchart. To achieve map-matching in streaming data, an empty state-memory must be initialized and updated with every incoming sample. This state-memory serves as the base for computations of correct positions for newly received GPS samples.

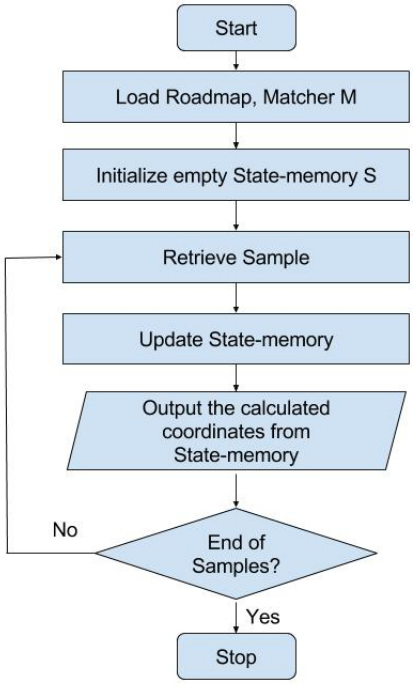


Fig. 7: Online map matching on a single node system, for a single client.

B. Implementation on a Spark Streaming Cluster

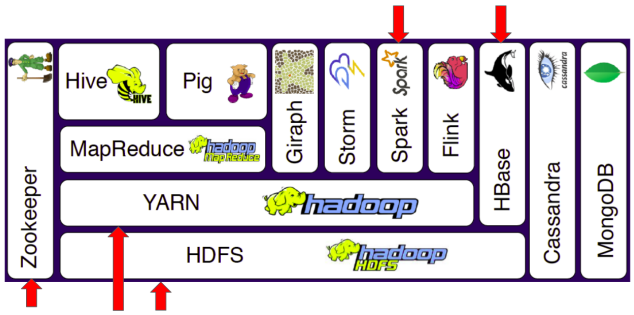


Fig. 8: Hadoop Ecosystem at a glance. The red arrows indicate the components selected for the implementation. *Source: cloudera.com*

S and **C** are used to refer to storage and compute clusters respectively. However, in our implementation, we have assumed that there is no physical separation of both clusters. **S** and **C** refer to the same cluster effectively.

For our scalable implementation, Hadoop Ecosystem has been chosen. A typical stream-processing cluster application consists of the following:

1) Distributed Coordination Service
2) Cluster Resource Manager
3) Data Ingestion
4) Fault-tolerant Data Storage
5) Computation Engine

According to the use-cases discussed so far, all the requirements can be satisfied through the tools (marked with an arrow) from the open-source Hadoop Ecosystem. The components used to meet the aforementioned requirements are described as follows:

Distributed Coordination Service - Apache Zookeeper

Apache Zookeeper is a server that distributed processes can reliably depend on for coordination. Some of the important functions performed by Zookeeper are as follows:

- Distributed configuration service
- Distributed coordination service
- Naming registry for distributed systems

In a growing distributed environment, it becomes increasingly difficult to keep configurations across all the nodes in sync with each other. Similarly, synchronizing the process execution in order to know that start and end of a computation is also a challenge. Zookeeper's functionalities listed above are the reasons why it is chosen as the mechanism for coordination among nodes in a Hadoop stack.

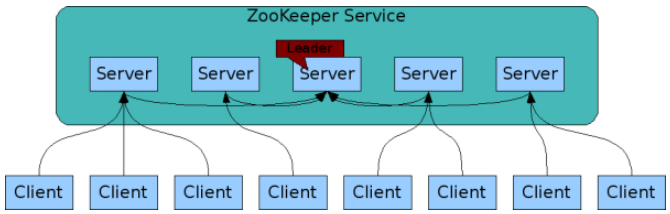


Fig. 9: Zookeeper Service. *Source: zookeeper.apache.org*

As seen in the figure above, ZooKeeper is replicated. Like the distributed processes it coordinates, it itself is replicated over a set of hosts called an ensemble. The ensemble consists of an odd number of hosts and a leader. In the event of leader becoming unresponsive, a new leader is elected.
A client connects to a single Zookeeper server at once. If the connection is lost, the client reconnects to a different server this time, as the service is replicated anyway.

Cluster Resource Manager - YARN

Yet Another (or YAhoo!) Resource Negotiator, is the resource management layer in Hadoop Ecosystem. YARN can be used to specify the memory and processing requirements of an application. In a cluster based on master-slave architecture and controlled by YARN, there are 3 major functionalities provided by YARN:

- NodeManager: It is a slave daemon that launches and tracks processes started on slave nodes.
- ResourceManager: It is a master daemon responsible for arbitrating resources among all applications in the system.
- ApplicationMaster: It is a per-application process acting as an intermediary by negotiating resources from the ResourceManager and working with NodeManagers to execute and track the progress of tasks.

The ResourceManager and the NodeManager together form the data-computation framework.

Computation Engine - Apache Spark Streaming

Apache Spark Streaming is built on the top of Apache Spark. Some of the salient features of Apache Spark are as follows:

- It is a fast and general purpose cluster computing engine developed at UC Berkeley.
- It provides lightning speed of computation ranging from 10 to 100 times that of Hadoop MapReduce.
- It consists of Driver and Worker. Each worker node may consist of multiple Executor processes which are responsible for computations. Spark cluster mode can be achieved in 3 ways: Standalone mode, Mesos or by using YARN.
- Invoking an action inside a Spark application leads to the launch of a job to fulfill it. Spark examines the dataset on which that action depends and formulates an execution plan. The execution plan assembles the dataset transformations into stages. A stage is a collection of tasks that run the same code, each on a different subset of the data.

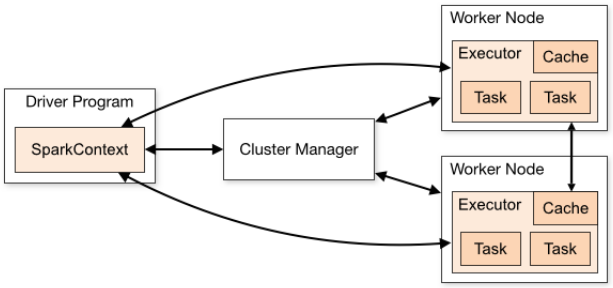


Fig. 10: Apache Spark in cluster mode. Driver program is the main program in a Spark application. Executors on the other hand are worker nodes' processes in charge of running individual tasks in a given Spark job. *Source: spark.apache.org*

Spark Streaming being an extension of core Spark Engine is capable of lightning fast discretized stream processing. Through its micro-batch execution model, it processes a relatively small amount of data at a defined interval (repeatedly) called batch interval. It can also balance the load on its resources dynamically, help in straggler (slow nodes) identification and fast failure recovery.

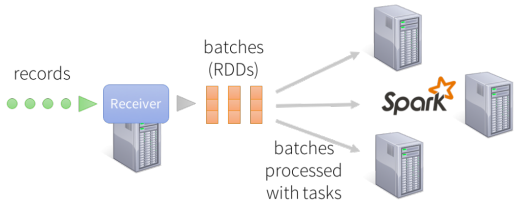


Fig. 11: Spark Streaming Architecture. *Source: databricks.com*

Spark Streaming divides incoming streams into discretized chunks of data. This abstraction is termed as DStreams. These DStreams are then forwarded to the executors for processing inside Spark Core Engine.

Data Ingestion Layer - Apache Kafka

Apache Kafka is a publish/subscribe messaging system to process streams of data efficiently in real time. It achieves real time processing by writing the streams of messages as they arrive. Unlike other message queues Apache Kafka distributes the incoming streams of messages across the Kafka cluster that consists of one or more Kafka servers, thus making itself a distributed message broker. An Apache Zookeeper quorum with one or more zookeeper instances manages the metadata required for publishing and subscribing messages through Kafka server. The Apache Kafka core architecture consists of:

- Kafka Server/Broker: A single server in a Kafka cluster is called a broker. Once the broker receives the messages, it assigns offset to them and stores it on the disk.
- Kafka Topic : It is a subdivision of a broker which is similar to a table or a folder in a file system where each topic represents a category of messages.
- Kafka Producer : Producer publishes messages to topics on Kafka broker which will be then subscribed by the consumers.
- Kafka Consumer : Consumer subscribes to a topic on Kafka broker and starts consuming the published messages.

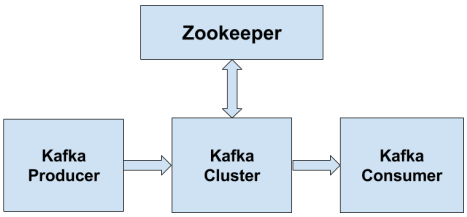


Fig. 12: Apache Kafka Architecture. The Kafka cluster relies on Zookeeper for metadata management. Producers produce messages to a predefined topic and consumers read messages from that topic.

A Kafka Topic is further divided into partitions. Partitions parameter is a key factor for performance consideration while creating a topic. The number of partitions determines the number of executors a spark streaming program will deploy for parallel processing. Replication-factor is another configurable parameter that determines the number of replicas that are created for a partition. The minimum value for partitions and replication-factor is 1. Maximum replication-factor depends on the number of available brokers in the cluster.

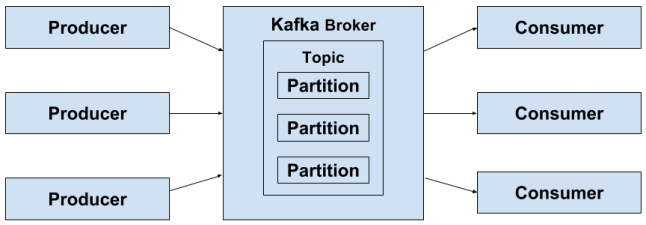


Fig. 13: Detailed functioning of Apache Kafka. As the message arrives they are written to the partitions in round robin fashion and an unique offset value is assigned to each record in the partition. The messages are distributed evenly across all the partitions and then they are consumed by the consumers.

Fault-tolerant Data Storage Layer - HBase

HBase is a Hadoop database which is scalable, non-relational and provides low-latency random access to its very large key-value stores. It is designed to handle planet-size data.
Such handling of immensely large data-stores is possible through automatic sharding of the HBase tables into horizontal splits called 'Regions'. Each region is hosted by a region-server. Each region-server can host one or more regions. A client requesting value for a given key connects to the HBase Master node. The HBase master subsequently redirects the client to the relevant region-server where the data is present.

HBase is called Hadoop database as it is built on the top of Hadoop Distributed File System (HDFS). As the name suggests, HDFS is a distributed and fault tolerant file storage. The fault tolerance is achieved through breaking down of a single large file into blocks of 128MB (configurable) and distributing the blocks across the cluster, followed by the replication of those blocks. The default replication factor is 3. This ensures that even if certain nodes on the cluster are down, the availability of a file can still be guaranteed if at least one copy of each of its blocks exists. It consists of 2 types of nodes:

- NameNode: It handles file metadata management.
- DataNode: This is where the data blocks are actually stored.

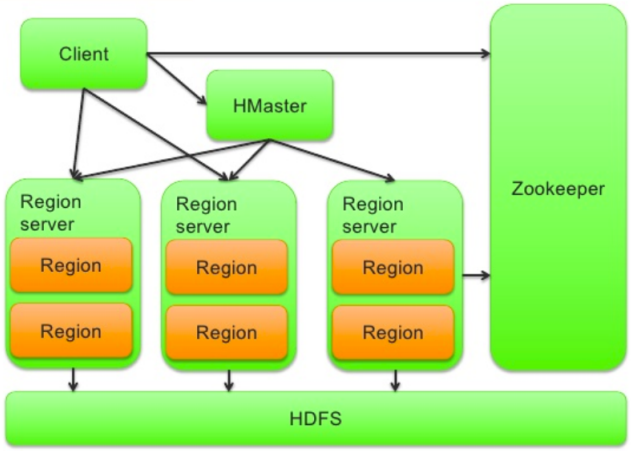


Fig. 14: HBase Architecture *Source: hortonworks.com*

The following section describes the relationship between various components of the map-matching architecture shown in Figure 5 and Figure 6 and the selected Hadoop tools.

1) Map Data

Map block consists of the road networks and provides the candidate vectors necessary to find the most likely position and consequently the path. This block is serializable in nature and is required for the creation of matcher instance.
Barefoot offers two ways of loading the map data. They are described as below:

(i) A Docker-based PostgreSQL/PostGIS database server running on a dedicated server can be used to load the map data. Upon first invocation, Barefoot copies the road network into a local file with *.bfmap* extension.
(ii) The *.bfmap* file created in the first approach can be used to directly load the road network directly from the local filesystem, even if the PostGIS database server is not running.

2) Matcher

For the matcher/filter block from the system architecture in Figure 6, each node in the compute cluster **C** shall run an instance of the matcher. There are 2 ways to achieve this:

1) Package and run Barefoot matcher server on every node manually. All the requests to and responses from the server can be handled through socket connections.
2) Distribute Matcher inside a serializable wrapper to all the nodes in the cluster as a Broadcast Variable. In Apache Spark Streaming, Broadcast variables are read-only copies of object that are sent to all the nodes upon starting the application.

Although it is more convenient to distribute the map-matcher as a Broadcast variable, for the reasons of demonstration and surety, we have packaged and run map-matcher as a multi-threaded server on every node. Because of the multithreaded nature of the server, a greater degree of parallelism can be achieved with this approach.

3) State Memory

The state memory holds the results consisting of most likely position candidates so far for every streaming device. The storage cluster **S** described in the system architecture in Figure 6 shall be used to store these state memory instances.

It is important to mention that each device streaming their position measurements to the map-matching system shall be assigned one of these state memory instances and stored with an identifier (e.g. unique device ID).

Upon receiving a new position sample, the state-memory record for the sample's identifier shall be looked up in a key-value store (in our case, Apache HBase) installed on storage cluster **S**. The procedure followed for creation and update of state memory object is as follows:

1) If a record is found for the sample's identifier in key-value store, it is updated with the computation results of the new sample. The result is put back into the store cluster **S**.
2) If an instance is not found, a new and empty state memory object is created for the given device-id and it is updated with computation results of the new sample. The result upon completion is converted into a state-memory record and stored into the storage cluster **S**.

It is also important to perform a grouping of samples by their identifiers because of the possibility that more than one sample per identifier may exist per Spark Streaming's batch interval. After the samples are grouped, they can be map-matched on the same executor in ascending order of their timestamps. In

order to group the samples together, Spark Streaming API offers 2 options: *reduceByKey* and *groupByKey*.
Unlike the latter, *reduceByKey* groups the items together first on every executor, followed by shuffle (network transfer) operations. This results in less time spent on shuffle operations and also less network traffic.
An alternative to reading from HBase datastore for every batch would be to use stateful streaming API within Spark Streaming. Stateful transformations like *updateStateByKey* and *mapWithState* facilitate creating, maintaining and updating states for multiple identifiers (in our case, deviceIDs) and removing the bottleneck associated with HBase datastore lookups for every device per batch processing. Our application shall make use of *mapWithState* as it is an improvement over *updateStateByKey* and provides a relative performance boost of up to 10x [4].

4) Load Balancer

For this layer, a fault-tolerant distributed message broker (or queue) is needed. Apache Kafka is the preferred solution for our application. Since the devices shall be streaming their position coordinates over the internet, the message broker would accept these requests along with the received data and queue them for further processing.
Kafka consists of multiple *brokers*, hosting one or multiple *partitions* for a *topic*. These brokers are capable of storing messages coming from message-producers in a fault-tolerant and partitioned manner. The consequences of a broker's failure is safeguarded through replication.

5) Query Layer

This layer **Q** as shown in Figure 6 would be used to query the results of the position computations. This enables the client to receive the results of the map-matching process. The key-value store used here, Apache HBase also provides a REST interface which can be used to query the results. The architecture of HBase ensures scalability as the clients deal directly with the RegionServer which holds the relevant Region(table-split), hence distributing read operation load.
The operations in red in the Figure 15 are potential bottlenecks. Therefore, they are handled in the following way:

- Read state record operations from HBase datastores are delegated to Spark Streaming's stateful transformation, i.e. *mapWithState*. Hence, creation, maintenance and updating of states are taken care of within Spark Streaming.
- Reading map data to instantiate matchers locally on every worker can delay first matching task by 10 seconds or more. Therefore, it is necessary that map data is loaded beforehand, thereby reducing the map data initialization time. We achieve this by running a map-matcher server on every node manually.

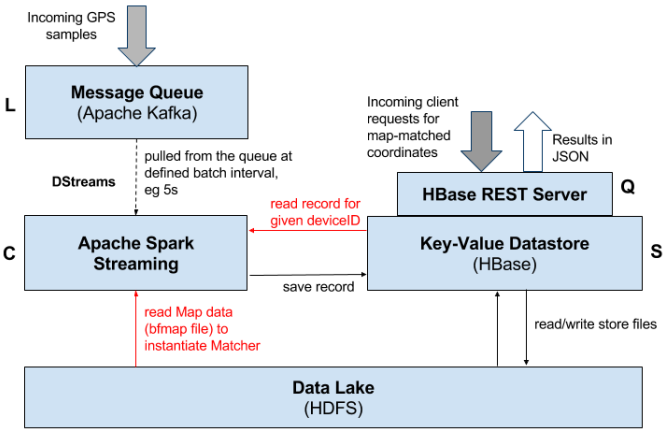


Fig. 15: Online Map Matching using Apache Spark Streaming. Block **C** consists of multiple Executors, each with an instance of Matcher **M**. Block **S** consists of multiple RegionServers, each serving one or many Regions(table-splits). The **Q** block exists on the top of datastore **S** and serves results to the incoming requests from the clients. Each request is forwarded to individual RegionServer.

IV. Evaluation

For the evaluation of our application, we have used a proof-of-concept cluster with the following specifications:

1) Master Node
   - Count: 1
   - Processor: 16 cores, Intel Xeon E5-2660 @ 2.20GHz
   - RAM: 64GB
   - HDD: 200GB
   - Network: 10Gb/s
2) Slave Node
   - Count: 5
   - Processor: 8 cores, Intel Xeon E5-2660 @ 2.20GHz
   - RAM: 32GB
   - HDD: 200GB
   - Network: 10Gb/s

In order to minimize issues with version-incompatibility among Hadoop components, we have used Cloudera Distribution Hadoop. The installation requires certain pre-installation configuration which are included in the source code [5]. In a typical production-environment cluster, the master node is lightly loaded and all the storage and computing loads are handled by the slave nodes. However, in order to get the most out of the hardware at our hand, storage and computing roles are also assigned to the master node.
In order to run Java code with lambdas, JDK 8 is installed on all the nodes and also configured as cluster JDK inside Cloudera configurations.
The cluster consists of 3 Zookeeper servers, which is also the recommended minimum in the Zookeeper ensemble. As long as the majority of the Zookeeper ensemble is up, the service will be available. Since Zookeeper requires majority for operation, it is recommended to use an odd number of nodes as servers.

For data ingestion, Apache Kafka is installed on all the slave nodes and the master node. A message topic *gps* is created and divided into 6 partitions, so as to distribute the incoming streams onto all 6 of the brokers.

The *NameNode* of HDFS and *HBaseMaster* of HBase are installed on the Master Node. Similarly, the *DataNodes* and *RegionServers* of HDFS and HBase respectively are installed on all the nodes.
YARN's *ResourceManager* is installed on the Master node and the *NodeManagers* are installed on every node in the cluster.
Now that storage layer and cluster resource management layers are configured, Apache Spark is installed on the cluster. This would also include Spark Streaming as it is a part of Apache Spark. Apache Spark applications can be run in 2 modes:

1) Dynamic Allocation mode: In this mode, Spark decides the resources required based on the incoming load. This is also the default mode of execution.
2) Static Allocation mode: In this mode, the resources required can be specified in advance in order to reduce time required for resource allocation. Resources that can be specified are as follows:
   - Number of executors
   - Number of processing cores per executor
   - Amount of memory

Static Allocation mode helps in determining what parameters assist in achieving best performance.
The Java application is submitted to Spark via terminal command *spark-submit* in YARN-client mode. The application shall run on every Slave node, read from the Kafka broker partitions in parallel, pre-process the samples, map-match the samples (via Matcher inside Broadcast Variables) and finally store them in HBase datastore for client-consumption via the HBase REST server.
In our trials, it is observed that map-matching a single GPS sample takes approximately 40-50ms. Hence, in order to analyze application throughput, the application is deployed in static allocation mode with different numbers of executors and processing cores per executor and the results are described in the next section.

V. Results

To test the performance of the application, a large batch-interval of 20 seconds is defined. The application's performance is tested by connecting up to 250 devices simultaneously via internet to the Kafka brokers.

A Spark Streaming job in YARN-client mode can be launched as shown:

```
spark-submit --class package.Driver
--master yarn --deploy-mode client
--num-executors 12 --executor-cores 2
--executor-memory 4G target/fatJar.jar
```

After deploying the application in YARN client mode with various executor counts, the following results are obtained.
As can be seen from the chart in Figure 17 above, the processing time reduces with increase in the number of executors. The increased parallelism facilitates map-matching operation to conclude in a shorter time interval. However, it is also observed that increasing the number of executors from 12 to 18 does not significantly reduce processing time. This is highly likely due to bad I/O throughput resulting from excess running threads. 18 executors effectively means running 3 executors per node.

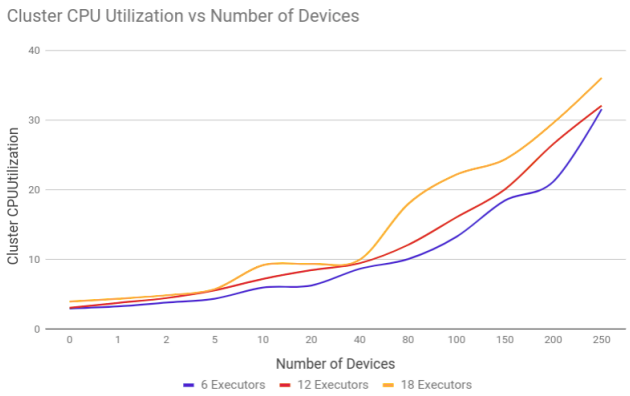


Fig. 16: Chart depicting Cluster CPU Utilization vs Number of connected devices.

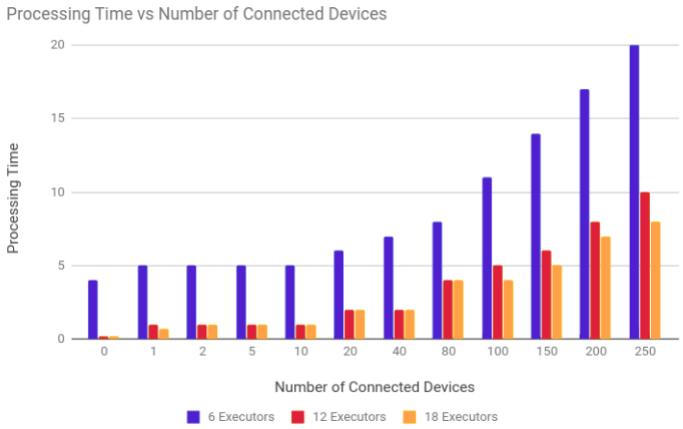


Fig. 17: Chart depicting Processing time vs Number of connected devices. Each executer is allocated with 2 processing cores. Processing 250 connected devices with 20 seconds batch interval amounts to 5000 GPS samples. Running 18 executors can process this amount in over 8 seconds. Processing time for 6 executors for 0 device case is 4 seconds due the fact that the executors are not uniformly distributed over the cluster. This creates issue with data locality as the executors wait before moving onto the next level of locality. The default is 3 seconds and can be configured by setting *spark.locality.wait* parameter to a lower value during *spark-submit* launch.

Another reason why running 12 executors has yield a reduction in processing time is because in such case, there is at least one executor running per node. This allows Spark Streaming to take advantage of *data locality*, i.e. the data is either PROCESS_LOCAL or NODE_LOCAL. Other supported locality levels are RACK_LOCAL and ANY. The data read from Kafka broker on every node is processed locally by the executor(s) running on the same node. Since lesser amount of data is needed to be transferred over the network, lower execution time is achieved.
In addition to that, a processor-count of more than 5 per executor is not recommended as it can result in bad HDFS I/O throughput [6]. Our application only uses 2 cores per executor. Also, no noteworthy differences were found in running executors with 2 cores against running executors with 5 cores. This is because the transformations carried out within Spark Streaming are lightweight operations and the bulk of the processing is being handled by multithreaded map-matcher servers installed on every node.
For the test case with 12 executors, it processes 250*20=5000 samples in 10 seconds, despite the fact that the batch interval is 20 seconds. This proves that the application with 2 cores per executor and 12 executors in total is capable of processing input stream faster than it arrives. Subsequently, the application is examined for a significantly lower batch interval, 3 seconds and is again found to successfully complete processing within the batch interval. The allocated executor memory can become a bottleneck and it can get used up rather quickly at a lower batch interval and with stateful transformations in use. It is therefore recommended to allocate sufficient memory [6] before launching Spark Streaming jobs. Allocating enough memory is also crucial so as to minimise delays due to Spark's garbage collection.

In addition to the above results, it is observed that if tracking the path of the device is the only objective, fractional map-matching can be used. Fractional map-matching takes every fifth sample from a device and matches it to the road network. Upon placing the map-matched coordinates on the road network, it produces the exact same results as the previous approach. The results are shown as following:

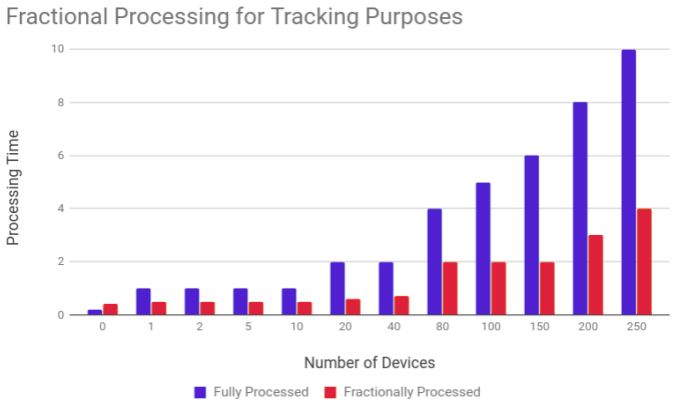


Fig. 18: Chart depicting Processing Time for Full vs Fractional Processing.

For tracking purposes, it is not necessary to process all the samples. Reduction in samples' size to certain extent helps reduce execution time and nevertheless is verified to produce the same results (path) when placed on the map as shown in Fig 19. 12 executors with 2 cores each are used.

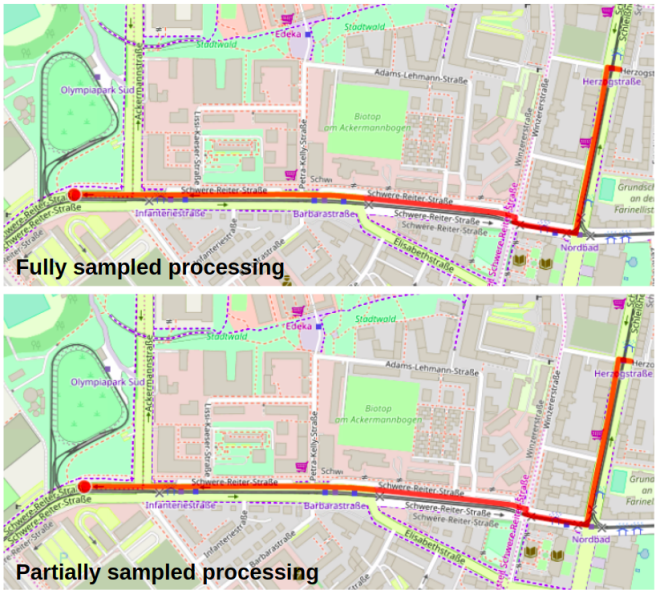


Fig. 19: Result of fully-sampled processing compared against partially sampled processing. By taking endpoints of the list of samples and every other intermediate 5th sample, nearly identical results are obtained.

VI. Conclusion

Through this project implementation[5], we have addressed the issue of how an existing map-matching solution can be integrated with open-source Big Data tools. Various state-of-the-art Hadoop tools are introduced, explained and employed to solve the problem at hand. We have subsequently analysed the performance of the Spark Streaming application and and were successfully able to map-match upto 250 devices simultaneously on a 6 node cluster. Apart from full-processing of samples, a tracking-oriented alternative is introduced which is capable of running over 2x faster.

Acknowledgment

We would like to sincerely thank Dr Robert Neumann for the project idea, guidance and motivation throughout the duration. We would also like to express our gratitude to Prof. Sebastian Zug for his profound encouragement in successful completion of this project.

REFERENCES

[1] P. Newson and J. Krumm, "Hidden markov map matching through noise and sparseness," November 2009, pp. 336–343. [Online]. Available: https://www.microsoft.com/en-us/research/publication/hidden-markov-map-matching-noise-sparseness/

[2] Mattheis, "Bmwcarit barefoot wiki." [Online]. Available: https://github.com/bmwcarit/barefoot/wiki

[3] S. Mattheis, K. K. Al-Zahid, B. Engelmann, A. Hildisch, S. Holder, O. Lazarevych, D. Mohr, F. Sedlmeier, and R. Zinck, "Putting the car on the map: A scalable map matching system for the open source community." [Online]. Available: http://subs.emis.de/LNI/Proceedings/Proceedings232/2109.pdf

[4] T. Das and S. Zhu, "Faster stateful stream processing in apache spark streaming." [Online]. Available: https://databricks.com/blog/2016/02/01/faster-stateful-stream-processing-in-apache-spark-streaming.html

[5] A. Kumar and N. EV, "Scalable map matching with apache spark streaming." [Online]. Available: https://github.com/achintya-kumar/SMM-with-Spark-Streaming

[6] S. Ryza, "How-to: Tune your apache spark jobs (part 2)." [Online]. Available: https://blog.cloudera.com/blog/2015/03/how-to-tune-your-apache-spark-jobs-part-2/