NSL SPINDLE
MAP-REDUCE AT THE EDGE IN A V2V ENVIRONMENT

By

William Rory Kronmiller

A Dissertation Submitted to the Graduate

Faculty of Rensselaer Polytechnic Institute

in Partial Fulfillment of the

Requirements for the Degree of

MASTER OF SCIENCE

Major Subject: COMPUTER SCIENCE

Examining Committee:

____
Dr. Stacy Patterson, Dissertation Adviser

____
Mohammed J. Zaki, Member

____
Magdon-Ismail Malik, Member

Rensselaer Polytechnic Institute
Troy, New York

March 7
(For Graduation May 2017)

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGMENT

TODO - right now I just wannt to thank whoever is proofreading this, as well as whoever invented Redbull

ABSTRACT

In the coming years, automobiles are expected to become increasingly connected and are also expected to become sources for vast amounts of data. Vehicles are increasingly becoming rich sensor platforms; each vehicle has a wealth of information about the local environment and groups of vehicles can convey valuable information about mobility patterns. By combining vehicle data with other data sources, one could potentially glean information not only about traffic conditions, but also weather and climate, infrastructure usage over time, and possibly even social and economic information derived from long term movement patterns.

We present Spindle, a hybrid vehicle-to-cloud architecture that allows its users to make the most of the available vehicle data while preserving limited bandwidth; it provides a seamless bandwidth-conserving system for streaming data from connected vehicles to one or more Apache Spark Streaming programs. To save bandwidth, Spindle leverages existing work on the use of ad-hoc wireless network clusters of vehicles. Spindle builds on Spark to provide an edge computing framework that can run Spark Streaming map/reduce functions on individual vehicles and inside vehicle clusters in order to increase bandwidth efficiency.

Spindle's novelty and power lies in its ability to allow an end-user to leverage edge computing optimizations with dynamic clusters of connected vehicles without having to leave the familiar Spark Streaming ecosystem and syntax.

Finally, we evaulate Spindle using a custom-built simulator that combines trace data with clustering information to provide a realistic facsimile of how a Spindle system would operate on the vehicle and vehicle cluster level. Our results show Spindle can produce bandwidth savings of over 75% through vehicle clustering.

1. INTRODUCTION

Spindle serves as a scalable hybrid vehicle-to-vehicle and vehicle-to-internet architecture for efficient near-real-time processing of streaming sensor data from network-connected vehicles. According to a 2015 Hitachi white-paper, some contemporary vehicles produce as much as 25 gigabytes of data every hour with prototype vehicles fitted with a variety of sensors and cameras producing up to 250 gigabytes per hour [2]. At the same time, work is underway to develop and test a variety of communications systems for connecting vehicles to one another and to road infrastructure and the internet [3].

One approach to marshaling data in networks with vehicles has been to create clusters of vehicle-to-vehicle networks [4][5][6][7]. In such systems, vehicles form temporary ad-hoc wireless networks in which they can exchange data with one another without the need to go over the internet; as such, vehicle-to-internet bandwidth is conserved.

We have developed Spindle, a hybrid vehicle-cloud architecture that addresses the problem of managing the vast quantities of available vehicle data by applying the map/reduce [8] paradigm to logical clusters of interconnected vehicles. Spindle exploits edge-computation by exchanging data over vehicle-to-vehicle communications systems in order to reduce the amount of data that must be sent over the internet to the cloud. Among Spindle's most powerful features is its integration with Apache Spark, which allows a user to write a Spark Streaming program with map/reduce functions that Spindle automatically serializes and processes as an edge computation. The end user writes a program that looks like a vanilla Spark Streaming application, but using Spindle automatically applies edge-computing optimizations.

Spindle is designed to support the use case of a developer or analyst writing a streaming map/reduce program that can then be deployed to the cloud and distributed to clusters of vehicles; vehicles in each cluster can apply the map operation to their incoming data streams and send the map outputs to a single leader (Clusterhead); the Clusterhead then applies the reduce operation to a buffer of incoming mapper outputs and sends the result of the reduce operation to the cloud, which forwards the data to the user's client program where the data can be displayed or further analyzed.

Contributions

- Library for running Spark DStream map and reduce operations outside of Spark, then re-integrating the results with the remainder of the Spark Streaming job
- Implementation of map/reduce on dynamic clusters of vehicles connected by ad-hoc networks
- Actor-model-based simulator for map/reduce queries on dynamic clusters of connected vehicles
- Edge computing architecture that supports multiple concurrent map/reduce queries and automatically eliminates duplicate queries and redundant messages (where a pair of redundant messages is a pair of messages containing the same values but associated with distinct map/reduce queries)

We demonstrate the bandwidth savings of performing map/reduce at the edge, in vehicle networks, by implementing key components of the Spindle architecture inside a simulation framework.

We implement the data Spindle processing pipeline for vehicles and Clusterheads, which we then integrate with a system that provides realistic mobility and connectivity data, such that we can simulate vehicles moving through space, forming clusters, processing map/reduce queries, and producing outputs to send to the cloud. We instrument the simulator to measure the number of bytes output by simulated Clusterheads, destined for the cloud. We work in collaboration with Mike Wittie of Montana State University to obtain realistic position and connectivity information for vehicles in the simulation. We also work in collaboration with Xiaotian Duan, from whose work we get vehicle-to-cluster assignments over time for our simulation [1]. In evaluating our experiments, we focus in particular on the effect of different amounts of time for which to buffer messages before performing reduce operations on the amount of data sent to the cloud. We explore the trade-off between resolution and latency versus bandwidth consumption. Having a smaller time window for reduction means that reduced messages are sent more frequently and client programs are given more data points to analyze, while having a larger time window for reduction means more messages can be batched together for reduction before data is transmitted to the cloud. We also explore the effects of regions with differing vehicle densities on bandwidth savings; regions of space that have a higher density of vehicles offer the opportunity to form larger clusters and to reduce the total number of clusters relative to the total number of vehicles present; a smaller number of clusters equates to a smaller number of Clusterheads transmitting data at any given time window, which has an effect on overall bandwidth usage.

Chapter 2 documents key technologies and vocabulary as well as relevant systems and research relating to the Internet of Things, streaming data, cloud computing, and connected vehicles. Chapter 3 describes they physical systems and software architectures pertaining to Spindle, as well as the simulation architecture. Chapter 4 provides information about our experimental design and setup, as well as our results. Chapter 5 draws conclusions from our research and lists ideas for future work.

2. Background and Related Work

2.1 Connected Vehicles

2.1.1 VANETs

The past few years have seen increasing interest in the area of Intelligent Transportation Systems (ITS) in which sensor technologies are put towards the improvement of transportation infrastructure. VANETs are wireless ad-hoc networks of vehicles and other road infrastructure [9]. Active research is underway to find applications for VANETs to improving transportation infrastructure and safety. VANETs have unique properties compared to general mesh and ad-hoc networks (MANETs) in that the total number of unique nodes that might join a network is quite large and information about the velocity and "trajectory" of each node is often available. As such, there are a number of low-level VANET protocols being explored including broadcast, unicast, and a variety of flooding protocols, however choosing an optimal low-level protocol is beyond the scope of this paper. To aid in this research, a number of simulators have come out. These simulators combine a mobility model, in which movement of vehicles through road networks under a variety of driving conditions, and a connectivity model which simulates the propagation of radio signals among vehicles in VANETs. OMNeT++ is one such network and connectivity simulator.

2.1.2 Analysis of Vehicle Data

A 2016 IEEE International Conference on Big Data paper titled "Supporting large scale connected vehicle data analysis using HIVE" illustrates practical use cases for analysis of vehicle data using big data technologies including Apache Spark [10]. The authors use the HIVE [11] query language in conjunction with Apache Spark [12] and Hadoop MapReduce [13] to perform analyses of a 2013 "connected vehicle" data-set from United States Department of Transportation. The authors combine the DoT data-set with geo-spatial data to validate and improve "traffic demand" models which "can drastically improve the planning of incident response strategies."

The authors also test travel time models for the generation of driving directions and create visualizations of the movement of traffic, to facilitate understanding of road conditions and usage. While the authors demonstrate the usefulness of applying tools such as Apache Spark to data coming from connected vehicles, the authors operate in a batched, offline, mode and do not perform edge computing optimizations.

2.1.3 Vehicle Clusters

Clustering Algorithms The creation of clusters of connected vehicles using ad-hoc wireless networks is an area of active research [5][4][6][**?**]. C-DRIVE is an approach for forming logical clusters of network-connected vehicles [5]. C-DRIVE is specifically designed to take into consideration the properties of VANETs with respect to vehicle movement. Specifically C-DRIVE attempts to form clusters of networked vehicles by taking into consideration both connectivity and direction of travel. These clusters are intended to be used to propagate information throughout groups of vehicles in a bandwidth -efficient manner.

2.2 Cloud Data Processing Technologies

The past few years have seen a number of critical developments in IT infrastructure and data processing software that, combined, provide a formidable platform for extracting useful information from vast quantities of data at scale.

Amazon Web Services Amazon Web Services (AWS) is a set of "Cloud Computing" offerings from Amazon.com that provide access to elastic computation and storage resources on an "on-demand" basis [14]. One particularly valuable service offered is EC2, which provides access to collections of virtual machines running on Amazon's cloud. These EC2 virtual machines (instances) can be used to perform resource-intensive cluster computing operations without the need for fixed infrastructure on the part of the end-user. One way in which a user can purchase access to EC2 instances is through Spot Requests, wherein the user places a bid for the maximum amount s/he is willing to pay per hour for a given EC2 instance configuration.

If the user is not outbid, then the user is given access to an instance of the specified configuration. As soon as the user is outbid (within approximately 2 minutes), the user's EC2 instance is shut down and its local data is lost. Spot Requests provide very low cost access to computing resources at the cost of reliability.Finally of note, Amazon provides an object storage layer - one can associate a key string with some binary blob that is stored in a replicated file system - called S3.

2.2.1 Spark Streaming

Spark Streaming provides a computing framework and a set of abstractions which allow a developer to write map/reduce-style programs for streaming data with relative ease [15]. One framework that is capable of processing large quantities of data in a distributed fashion, such as on a cluster of EC2 instances, is Apache Spark. Spark offers a micro-batch package for processing streaming data called Spark Streaming. Spark Streaming allows a developer (or other analyst) to manipulate time-bounded batches of streaming data tuples by writing a program that performs immutable transformations such as map, filter, and reduce on a Spark Streaming abstraction called a DStream. Spark is able to take the client program, break it up into "stages" in which all processing across tuples can occur in parallel, then partition each stream and stage across nodes to be processed in parallel [15].

2.2.2 Kafka

Apache Kafka is primarily a high-throughput publish-subscribe message bus that runs as a cluster of message "brokers" which provide clients access to partitions of logical groups of message streams called topics; Kafka brokers support master/slave partition replication and support multiple producers and multiple consumers within and across topics and their partitions [16]. A recent addition to Kafka is Kafka Streams, a streaming data processing framework that provides some of the functionality offered by Apache Spark as it pertains to performing basic map/reduce-style transformations to streams of tuples. Though Kafka Streams provides per-message processing rather than Spark Streaming's micro-batch processing, Kafka Streams provides a similar programming interface in which a developer writes map, reduce, filter, and other transformations as function applications to data passing through KStreams [17].

2.3 Other Software

2.3.1 Akka

Akka is a JVM library that implements an actor-model system with an Erlang-like structure. Akka includes as key features programming abstractions that enable the quick and relatively painless development of parallel and concurrent programs which can be converted to distributed systems with relative ease [18]. Use cases include simulator software, data processing, and "Automobile and Traffic Systems" [18].

2.3.2 Postgres

Postgres is a relational database with a SQL language interface that supports view, aggregate functions, transactions, indices, and joins [19]. Postgres is a robust and, when properly configured, performant database.

2.4 Edge Computing Technologies

Some more recent cloud and related technologies take computations that would previously have been performed at a centralized system (e.g. a cloud computing platform) and run part or all of the computations on the devices at the edge of the network (e.g. IoT devices).

2.4.1 Edgent

Apache Edgent, formerly Apache Quarks, is a general edge computing and IoT data processing platform. Edgent provides connectors for a number of publish/subscribe systems such as MQTT and Apache Kafka as well as methods for performing transformations on streaming data on edge devices using pre-programmed analytical subroutines that can be activated and deactivated remotely through IBM's Watson IoT Platform. Edgent appears to be designed primarily to run on sensors for monitoring infrastructure and detecting exceptions, though it provides a programming model similar to that of Kafka Streams. Edgent appears to be designed along the lines of traditional IoT data collection platforms in which the edge device has some set of pre-configured features that can execute locally, while new and ad-hoc analytics are run server-side using a subset of the data the edge device was originally programmed to transmit. Edgent does not appear to be designed for executing runtime-generated analytic queries in the context of dynamic clusters of vehicles with a diverse array of sensor capabilities [20].

2.4.2 Greengrass

Greengrass is a new product from Amazon Web Services that is currently in preview. While available documentation is limited as of this writing, Greengrass appears to be designed to perform some event-driven logic offered by Amazon's cloud IoT platform on a Core device operating on the local area network [21]. Fixed IoT devices communicating with a shared Greengrass Core are a Greengrass Group. Greengrass offers a similar model to that of Spindle: a unified programming layer that mixes edge and cloud computing in an effort to optimize bandwidth use, Greengrass appears to be a vendor-exclusive platform and does not appear to support dynamically-formed clusters of large numbers connected devices. Greengrass appears designed for fixed groups of a few devices that share a common LAN, and is therefore a far more limited architecture than Spindle.

2.4.3 Cloud IoT Hubs

A number of cloud providers offer IoT Hubs, for processing streams of data from IoT devices in the cloud using a proprietary platform.

AWS IoT Platform The AWS IoT platform runs on top of the MQTT message bus hosted in the cloud and consists primarily of a message bus, rules engine, and a registry [22][**?**]. The registry contains information about every IoT device, as well as its current and desired states [23]. The message bus is used to publish data from the device and send messages to the device. The rules engine is used to trigger events based on data received from IoT devices, such as writing to cloud databases or performing operations on other AWS services such as SNS, and Lambda.

Azure IoT Microsoft has an offering similar to that of AWS IoT and Greengrass, in which IoT devices running client SDKs can publish messages directly to a cloud hub, or to a local network gateway where edge computing can take place.

Other IoT Platforms Similar systems are offered by Google [24], IBM [25], and a number of other companies [26].

3. System Architecture

Spindle provides a novel streaming data processing platform by building on existing cutting edge and industry standard distributed systems, primarily from the Apache ecosystem. Spindle applies these technologies to a hybrid vehicle and cloud architecture.

Spindle software consists of three major components: a set of Apache Spark streaming programs managed by one or more clients, a custom data ingestion and query management Middleware, and edge computing software running on network connected vehicles.

Spindle operates on the vehicle level as a Scala [27] program running on the Akka [18] framework; this program is responsible for handling cluster formation, data collection, and edge computation.

Data passes through components of the Vehicle software in the form of messages sent to Kafka [16] topics. Similarly, data is transferred at the cloud layer using Kafka.

Clients receive data from the Middleware as Kafka messages that a custom Spark [12] library processes into DStream tuples which can be further processed using any of the available Spark streaming operations written for DStreams and RDDs.

3.1 Theorized Architecture

Spindle is a hybrid vehicle-to-cloud architecture with three major components: a client Spark program, an in-cloud Middleware, and edge computing software.

3.1.1 Spark Clients

Spindle is ultimately designed to provide streaming data from network-connected vehicles to a streaming data analytics platform. We chose Apache Spark as our target platform. Apache Spark allows a user to analyze streams of tuples using the DStream programming abstraction previously described. In a traditional Spark

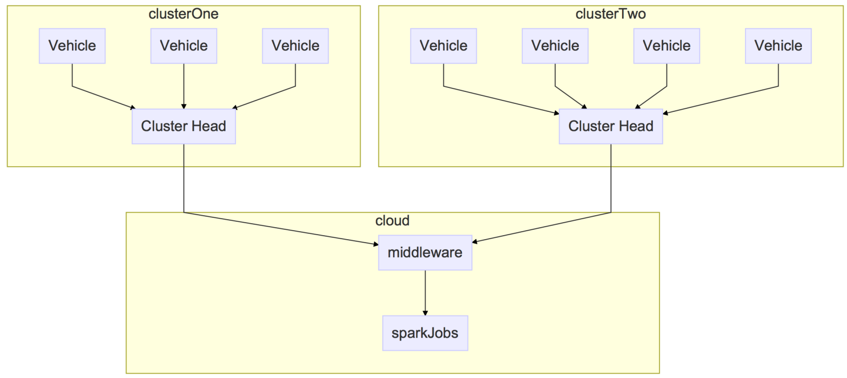


Figure 3.1: A simple diagram of the Spindle system components. Vehicles form ad-hoc clusters. Each vehicle cluster has one cluster head, which sends reduced data to the middleware. The middleware sends the processed results to client Spark Streaming programs

Streaming program, a user creates a connection to some message bus such as Apache Kafka and programs against a DStream representing the micro-batches of messages being received from the connected message bus. The user is then free to analyze the data by applying functional transformations such as map, filter, and reduce to the batches of streamed tuples.

A naive analytics architecture for vehicle data with no edge computing optimizations would likely have each connected vehicle send all of its data directly over the internet to a message bus such as Apache Kafka running in the cloud. The user would then write a Spark Streaming program to create a DStream to receive the stream of all data coming from the vehicles and apply a map operation to extract only the relevant values from the flood of incoming data. To perform an aggregate analysis of the data, the user would then apply a reduce (`reduceByKey`) operation, for example to get average fuel economy in real time for a given region or set of regions. While the naive example is extremely inefficient in terms of bandwidth consumption and compute resource utilization, the programming model is extremely simple and convenient. All the end-user writing the analysis has to do is connect, map, reduce, and output the results (or perform further analyses).

With the Spindle architecture, we attempt to preserve convenience of the naive model while greatly reducing bandwidth consumption and wasted cloud resources.

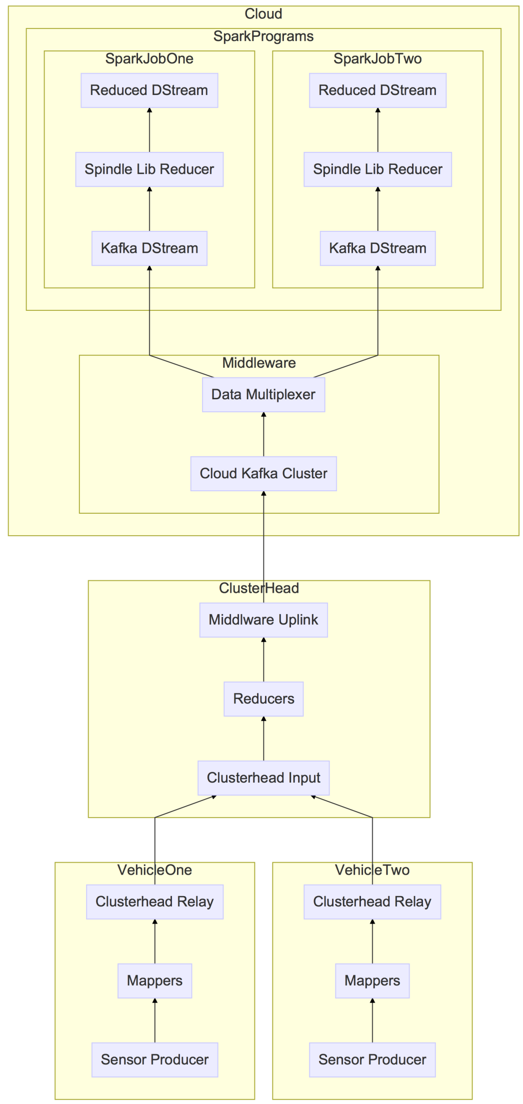


Figure 3.2: A diagram of the theorized system's major components. Vehicles send data to their Cluster Heads, which perform reduce operations over incoming mapped tuples. The resultant reduced tuples are then sent to the Middleware to be distributed to the connected Spark jobs.

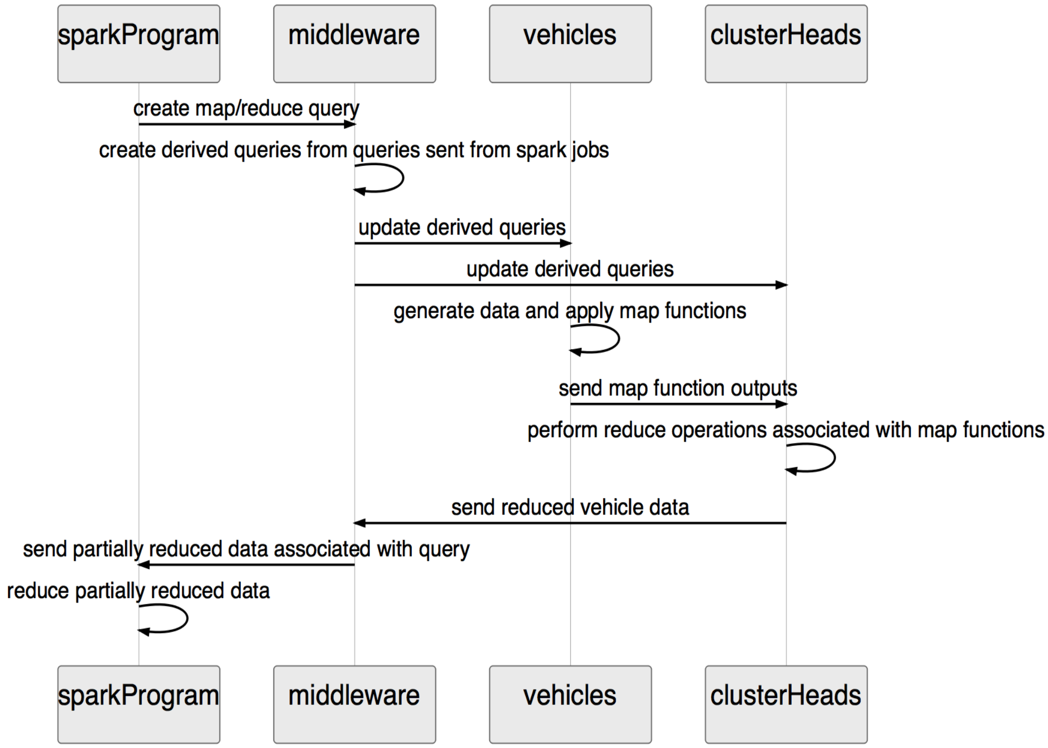


Figure 3.3: A sequence diagram showing how a query passes from a connected Spark job through the Middleware and vehicle systems and how the reduced results then return to the Spark job.

To do so, Spindle includes a small Scala library that can be imported to a Spark Streaming program. Rather than calling the Kafka Spark Streaming library to create a DStream, the user instead calls the Spindle Spark Streaming library to create their DStream. The user then writes their map and reduce functions as before, with the additional (current) requirement that the reduce function call be annotated with a descriptive enum to indicate the category of reduce operation taking place (e.g. min, max, mean, etc. . .).

The Spindle Spark Streaming library is then able to serialize the user's map and reduce functions and transmit them to the Middleware for de-duplication and distribution to vehicles and Clusterheads. The Middleware sends back partially reduced results which the library running on Spark then re-runs the reduce function over. The final output is a DStream of reduced tuples, which from the user's perspective has the same attributes as a DStream that would have been produced by running the naive implementation.

We have developed a proof-of-concept implementation of the Spindle library for Spark Streaming described above. A future version will include the ability to pass filter functions to the Middleware as well in order to support more selective queries.

3.1.2 Middleware

Spindle's cloud Middleware design consists of two components: a message bus and a query manager.

Kafka Message Bus Required vehicle data is published to a Kafka cluster, to which the Spindle Spark Streaming client can connect to consume partially reduced tuples for further processing as described above. Kafka provides a scalable platform with logical partitioning of messages (topics) that allows data from vehicle clusters for multiple Spark Clients to be ingested and routed to client-specific topics so that data for overlapping queries need only be sent once and so that each client can be required to process only the data required by its analysis.

Query Manager The query manager is responsible for storing, de-duplicating, and disseminating active Spark client queries. Duplicate map operations can be detected by their return types, while duplicate reducers can be detected by detecting duplicate inputs types, return types, and by their category annotations. Two queries can be de-duplicated if their mappers match, their reducers match, and their geographic regions overlap; if the geographic overlap is complete, then a single query is sent to the vehicles and Clusterheads. As Kafka requires Zookeeper, a consistent distributed key/value store, the Middleware will store its state and look for client queries in Zookeeper directories. Incoming data, from the cluster heads, will be loaded into Kafka. The query manager will then perform streaming map operations using Kafka Streams to route data to Kafka topics for the dependent queries.

3.1.3 Vehicles and Clusterheads

The vehicles and Clusterheads run a similar software stack to that of the Middleware, with data being processed through Kafka Streams. Each vehicle and Clusterhead is expected to have its own internal single-node Kafka cluster as well as supporting software for marshaling data, managing cluster membership, and applying map and reduce functions to the streaming data. Specific implementations of the protocols used to connected vehicles to one another, to Clusterheads, and Clusterheads to the internet are beyond the scope of this work, as are the specifics of how best to propagate queries from the Middleware query manager to vehicles and Clusterheads. Spindle's architecture requires only that vehicles be able to exchange binary data with Clusterheads and that Clusterheads be able to send binary data to the cloud Middleware. We anticipate that queries will be transmitted to vehicles by their Clusterheads, though a model in which vehicles periodically poll the cloud Middleware directly is also compatible with the Spindle architecture. Every vehicle can act as a Clusterhead, though a Clusterhead need not be a vehicle and can instead be some other piece of connected infrastructure. The Spindle architecture is also agnostic to clustering algorithms and protocols, but merely requires that each vehicle know the identity of and be able to communicate with its Clusterhead (where the vehicle may be its own Clusterhead).

3.1.3.1 Vehicle Components

Sensor Producer Each vehicle has a sensor producer, which is a Kafka message producer that periodically reads the current sensor data (e.g. speedometer, tachometer, fuel consumption, fuel level, tire pressure, O2 sensors, lat/lon, etc...) and publishes the data as a single serialized VehicleMessage object to the onboard Kafka cluster. The VehicleMessage class is what the Spark client map functions take as input.

Mappers Each vehicle has a set of map functions loaded from the Middleware. A local vehicle-level query manager periodically checks for updates to the set of active map functions. When a new map function arrives, the vehicle-level query manager launches a new Kafka Streams topology that reads data from the Sensor Producer topic, applies the map function to the deserialized data, and publishes the serialized map function output to a dedicated output topic.

Clusterhead Relay Finally, each vehicle has a Clusterhead relay, which reads data from the mapper output topics and relays the data to the vehicle's current Clusterhead. This is where data is transmitted from vehicle to vehicle or vehicle to infrastructure over a wireless communication protocol. Spindle is protocol agnostic, as described above.

3.1.3.2 Clusterhead Components

Clusterhead Input The Clusterhead Input is a Kafka producer, which receives messages transmitted by the member vehicle Clusterhead Relays and publishes the messages to a single local Kafka topic.

Reducers Just as on the vehicles, the Clusterheads have a local query manager which is responsible for loading and unloading query functions. Instead of loading and unloading map functions, the Clusterhead query manager loads and unloads reduce functions. Once again, the reduce functions run on Kafka Streams topologies, with the distinction that the reducer topologies include buffers, which cache intermediate reduce results until some pre-defined time window has passed, at which point the reduced data for a time window is published to a Middleware Uplink Kafka topic.

Middleware Uplink The Middleware Uplink serves a similar purpose to that of the vehicle's Clusterhead Relay. The uplink reads messages published to the Middleware Uplink Kafka topic and sends the messages to the cloud Middleware. It is at the Middleware Uplink that message compression and de-duplication can occur. If two reducers produce the same output in a given time window, the Middleware Uplink should send only one tuple, tagged with the UIDs of both reducers.

3.2 Simulation Architecture

We have developed a simulator to evaluate the vehicle and Clusterhead architecture under a variety of clustering, vehicle density, and query configurations.

The simulator implements the basic features of the vehicle and Clusterhead software described above, including Sensor producers, mappers, reducers, simulated Clusterhead relays, query managers, and simulated Middleware Uplinks. The core simulation software is implemented in Scala using the Akka actor framework.

3.2.1 SpindleSim

The Spindle simulator (SpindleSim) is implemented in several thousand lines of Scala [27] and is implemented on top of the Akka actor model framework. The Scala/Akka portions of the simulator implement as realistically as is practical the software that would run on a real-world deployment of Spindle. This is accomplished by creating a separate Akka actor for each simulated vehicle, where each actor is given a cache of time-series simulation events and data and a connection to a "World" actor responsible for keeping track of global simulation time; this global time-keeper can be thought of as a stand-in for a GPS clock running on each vehicle. Each vehicle actor is then responsible for managing its publish-subscribe streams and messages, as well as its simulated connections to other vehicles in its cluster. Messages are exchanged between vehicles and send from vehicles to "the cloud" by way of special "Stream Relay" Kafka Streams programs. The Stream Relay programs operate as simulated versions of the Clusterhead Relay and Middleware Uplink programs. Stream Relays are also responsible for filtering expired and "Canary" messages; the version of Kafka Streams being used (the latest version at the start of development) has implementation bugs that mean new message consumers must read old messages and that a topic may not fully initialize until a message is produced to it; our solution is to send "Canary" messages to ensure topic initialization and to filter out messages that have already been processed. Finally Stream Relays are responsible for keeping a running sum of the number of bytes sent on a per-relay basis and logging these sums to separate CSV files at run-time.

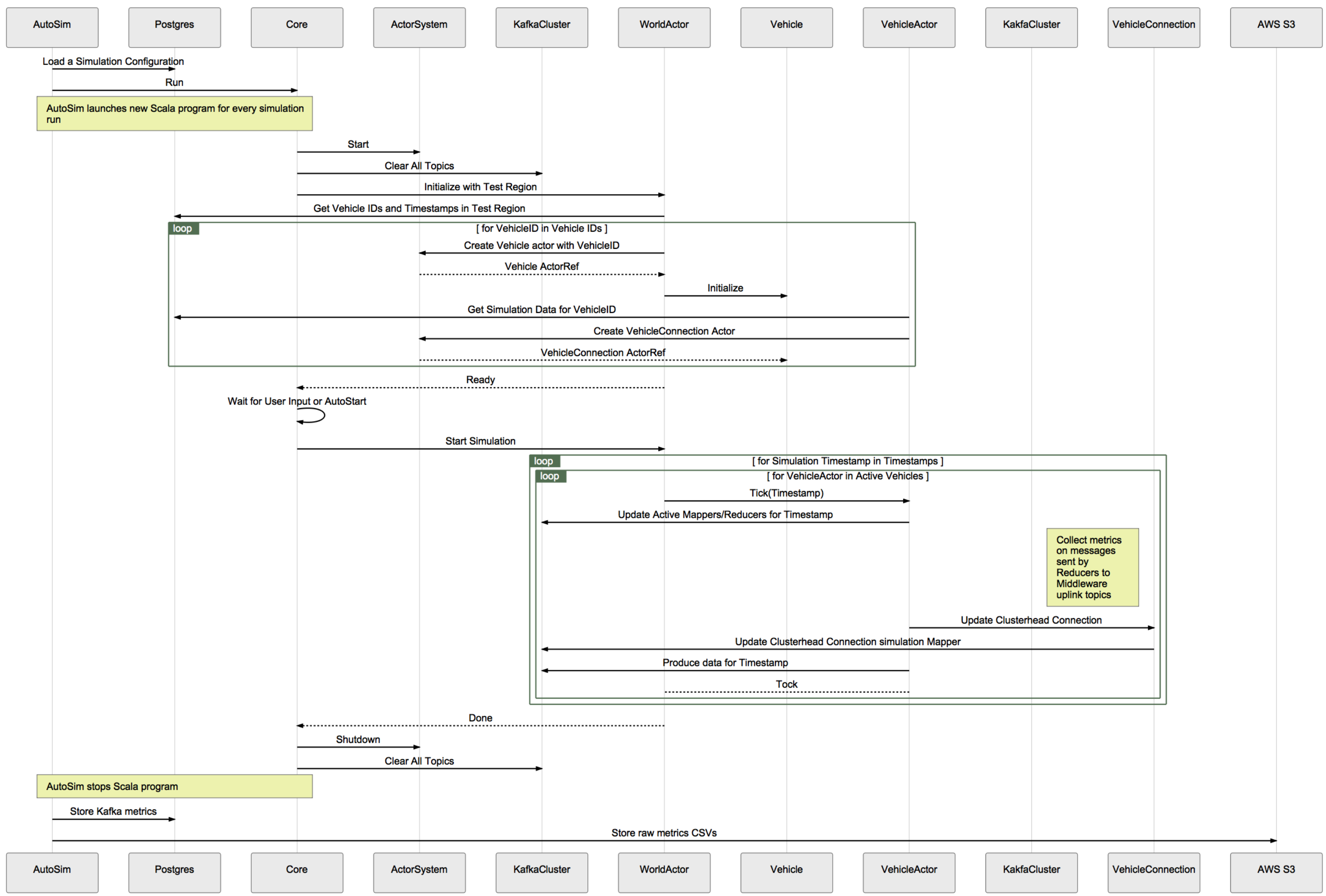


Figure 3.4: A sequence diagram showing the high-level interactions of the Spindle simulator components. A Postgres database stores the simulation configurations and pre-computed velocity and connectivity data, as well as simulation results.

3.2.2 AutoSim

The Scala/Akka simulator (SpindleSim) currently has a number of pre-configured map/reduce jobs written that can be turned on or off using the program's `application.conf` file. A secondary program, dubbed AutoSim, runs the simulator iteratively, writing to the conf file, launching SpindleSim, then gathering the results and uploading them to a database and permanent storage. In order to iterate through the desired test configurations, AutoSim reads from a table `sim_configs_vx` in a Postgres 9.6.1 database hosted on an AWS EC2 `m4.large` reserved instance. AutoSim will choose among the least-tested configurations in the configs table, write the configuration to the SpindleSim `application.conf`, and launch an instance of SpindleSim. While SpindleSim is running, AutoSim greps the console logs for exception messages. If an exception is detected, the current SpindleSim instance is killed and a notification is sent via AWS SNS topic reporting the error message in order to facilitate debugging. If the SpindleSim operation completes, AutoSim uploads the CSV logs to an AWS S3 bucket and parses the final message byte sums from the CSV files then writes the results to a `sim_results_vx` table in Postgres. AutoSim is a simple Node.JS ES6 application that is run from the Babel-Node [28] transpiler.

4. Experiments

4.1 Data Sets

4.1.1 Vehicle Traces

The Spindle experiments depend on a set of vehicle traces generated using "the open source vehicular network simulation framework" VEINS, which simulates vehicle movement and radio connectivity in a "realistic" manner [29]. The VEINS traces were generated by Mike Wittie of the Montana State University as part of a larger overall research project. The traces come in the form of sharded CSV files containing time-series mappings of three different types: speed, x-position, y-position, and connectivity to other vehicles. Each time-series mapping includes a timestamp, a vehicle uid, and a value (speed, position, reachable-vehicle-id). These shards have been parsed into a Postgres database with a table for each mapping type.

4.1.1.1 Test Regions

Original VEINS Traces The original VEINS traces span a region from $(41735, 85178.4)$ to $(63071.5, 114426)$ and includes 3340 distinct vehicles. This data-set includes more vehicles than the current simulator implementation can handle due to the large number of threads, network connections, and Kafka topics that would be required (the thread/topic count scales with the number of vehicles multiplied by a constant multiple of the average number of map/reduce queries running on each node). A real-world deployment would distribute the threads/topics over the set of vehicles in the system such that each vehicle and Kafka cluster would have threads and topics proportional to the number of active map/reduce queries. In essence, this is a temporary limitation of the simulation architecture, not the real-world architecture.

To overcome the simulator's limitations, we defined two test regions, one dense region and one sparse region being transited by approximately the same number of vehicles. The sparse and dense regions were selected based on the number of vehicles transiting them and based on the layout of the underlying road networks.

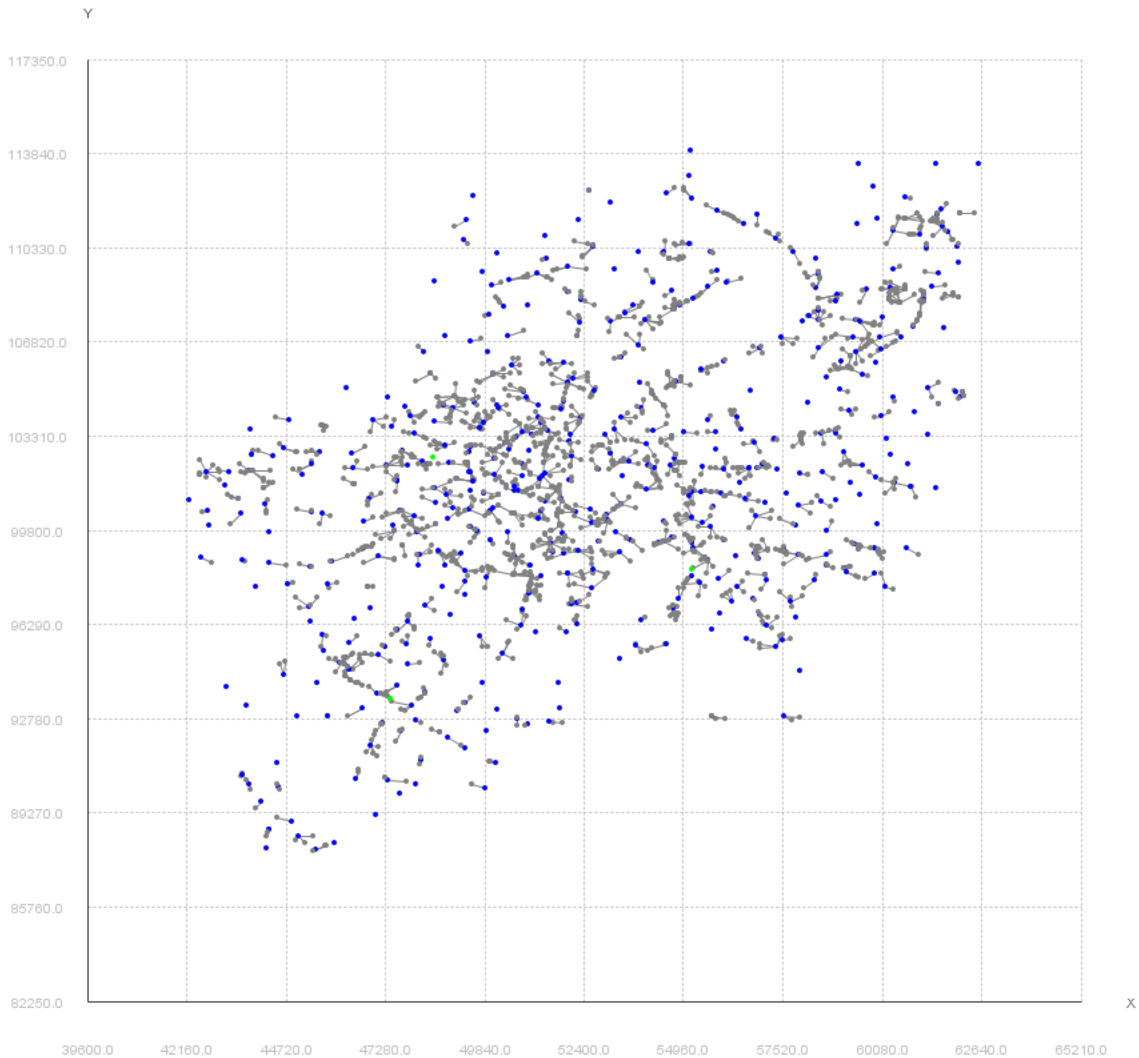


Figure 4.1: Illustration of VEINS Vehicles with MANET Clustering, Courtesy Mr. Duan [1]

Sparse Region The sparse region is transited by 148 distinct vehicles over the course of the simulation and defines a bounding box from $(47150, 106801)$ to $(53000, 112545)$. The sparse region covers an area on the outskirts of the city, with a relatively low density of vehicles. The sparse region is illustrated by a purple box in 4.2.

Dense Region The dense region defines a bounding box from $(50000, 100000)$ to $(51000, 102000)$ and is transited by 131 vehicles over the course of the simulation. The dense region covers a small area near the city center where the density of vehicles is higher. The dense region is illustrated by an orange box in 4.2.

4.1.2 Window Sizes

Kafka Streams can perform reduce operations over messages received over the course of a user-defined time-window and will write the results to a KTable, a logical structure whose API approximates that of a key-value store. Spindle's Kafka Streams reducer adds a `Batcher`, which transmits the final value of a time-window's KTable entry to some destination Kafka topic (in this case the Middleware). As such, Spindle supports micro-batch operations on Kafka streams with user-configurable window sizes. The sizes tested were 10, 15, and 30 seconds.

4.1.3 Test Clusters

The cluster head assignments for each node are pre-computed using either Mr. Duan's clustering algorithm or some base-line `single_clusterhead` (all vehicles in simulation share a single Clusterhead) or `self_clusters` (all vehicles act as their own cluster heads) [1].

4.1.4 Clustering Algorithms

The spindle architecture takes advantage of work done by fellow RPI student Mr. Duan on generating clusters of vehicles based on vehicle connectivity and lane position [1].

4.2 Test Software Environment

AutoSim and SpindleSim are packaged inside a docker image, `wkronmiller/nsl-spindle-simulator` that can be run from a laptop, desktop, or EC2 instance. The test framework and container are designed to survive the total loss of local storage and/or the running simulator container by storing the results of all completed simulation operations on S3 and a remote database. This design decision allows the test framework to be run on an AWS EC2 Spot Fleet which offers discounts of roughly 70-90% in exchange for the requirement that any software running on a Spot instance be designed to killed at any time with little or no warning. The architecture also enables trivial transfer of the simulator across different machines, easing debugging and mitigating problems related to transient EC2 network problems.

4.3 Test Configurations

4.3.1 Example Map/Reduce Programs

speedSum The most simple map/reduce operation tested is the "speedSum" job, which simply takes the sum of the speeds of all vehicles in the simulation in each time step. The map operation takes the vehicle state as input and returns only the vehicle speed. The reduce operation run at the Clusterheads takes the sum of the speeds of each of the cluster member vehicles.

geoFiltered The geoFiltered operation collects the data required to compute the average speed of all vehicles in a selected geographic region. This map/reduce job extends speedSum by performing a word-count-style map operation that maps each vehicle's sensors to the vehicle count and the vehicle speed: `(1, [speed])`. The reduce operation again simply sums the count and speed of each member vehicle `([numVehiclesInCluster], [sumOfSpeeds])`. The geoFiltered query also is selective, in that for a given test region the geoFiltered query operates only on a subset of the test region containing approximately half of the vehicles being tested.

geoMapped The geoMapped query performs a similar map/reduce operation to geoFiltered, but instead of filtering out half the test region, the geoMapped query maps half the test region to one region ID and maps the other half of the test region to a second region ID, then performs a reduceByKey. The map operation produces the following: `(regionId) -> (1, [speed])` and the reduce produces `(regionId) -> ([numVehiclesInCluster], [sumOfSpeeds])`.

4.4 Simulator Results

A total of 412 tests were carried out, across all configurations. Table **??** shows the byte counts and number of tests for each configuration evaluated. Figures 4.3 and

clustertable	region	window_secs	total_bytes	normed_bytes	num_runs
clusterinfo_dense_v1	dense_positions	10	325903.82	0.23	11
clusterinfo_dense_v1	dense_positions	15	229221.64	0.24	11
clusterinfo_dense_v1	dense_positions	30	137369.91	0.27	11
clusterinfo_dense_v1	dense_positions	10	204693.36	0.30	11
clusterinfo_dense_v1	dense_positions	15	147311.18	0.31	11
clusterinfo_dense_v1	dense_positions	30	86209.25	0.34	12
clusterinfo_dense_v1	dense_positions	10	315867.73	0.22	11
clusterinfo_dense_v1	dense_positions	15	221243.36	0.23	11
clusterinfo_dense_v1	dense_positions	30	129972.55	0.26	11
clusterinfo_sparse_v1	sparse_positions	10	1320204.83	0.46	12
clusterinfo_sparse_v1	sparse_positions	15	921419.36	0.47	11
clusterinfo_sparse_v1	sparse_positions	30	500452.75	0.50	12
clusterinfo_sparse_v1	sparse_positions	10	680126.75	0.55	12
clusterinfo_sparse_v1	sparse_positions	15	466748.58	0.56	12
clusterinfo_sparse_v1	sparse_positions	30	247315.00	0.58	11
self_clusters	dense_positions	10	1411252.83	1.00	12
self_clusters	dense_positions	15	957476.33	1.00	12
self_clusters	dense_positions	30	513734.82	1.00	11
self_clusters	sparse_positions	10	2854338.27	1.00	11
self_clusters	sparse_positions	15	1943005.25	1.00	12
self_clusters	sparse_positions	30	1005156.67	1.00	12
self_clusters	dense_positions	10	1405860.64	1.00	11
self_clusters	dense_positions	15	954038.36	1.00	11
self_clusters	dense_positions	30	496234.09	1.00	11
self_clusters	sparse_positions	10	2841283.83	1.00	12
self_clusters	sparse_positions	15	1938500.33	1.00	12
self_clusters	sparse_positions	30	996546.82	1.00	11
clusterinfo_sparse_v1	sparse_positions	10	1301719.45	0.46	11
clusterinfo_sparse_v1	sparse_positions	15	907851.58	0.47	12
clusterinfo_sparse_v1	sparse_positions	30	480055.50	0.48	12
self_clusters	dense_positions	30	254855.58	1.00	12
self_clusters	dense_positions	15	468076.36	1.00	11
self_clusters	dense_positions	10	675473.45	1.00	11
self_clusters	sparse_positions	30	426113.73	1.00	11
self_clusters	sparse_positions	15	831415.75	1.00	12
self_clusters	sparse_positions	10	1229630.83	1.00	12

Table 4.1: The rounded byte counts (total and normalized) from all test configurations.

4.4 show the bandwidth consumed by the speedSum and geoMapped map/reduce queries, respectively. Figure 4.5 shows bandwidth consumption for the geoFiltered query. We see from all of the aforementioned graphs the same pattern: that the higher density region has better performance than the lower density region, that larger window sizes have better performance than smaller window sizes, and that, critically, the use of clustering results in a 50-75% savings in bandwidth.

Figures 4.7 and 4.6 show the average bandwidth consumption over all tests for sparse and dense regions. Configurations that have clustering enabled and that have larger reducer time window sizes use less bandwidth. Furthermore, the geoFiltered query consistently uses less bandwidth than speedSum or geoMapped as a result of having fewer nodes sending data to the cloud.

The results make logical sense. A higher density region allows for better clustering - more vehicles to a cluster. A larger window size results in a smaller number of transmissions from the reducers to the Middleware, so even though the serialized messages may be larger (because the values contained within are larger), the larger message size is outweighed by the lower transmission frequency; this makes sense when one considers the inevitable boilerplate required in each message that gets wrapped around the changing values. Finally, the use of clusters makes sense for similar reasons to those of having larger window sizes - fewer overall messages are sent.

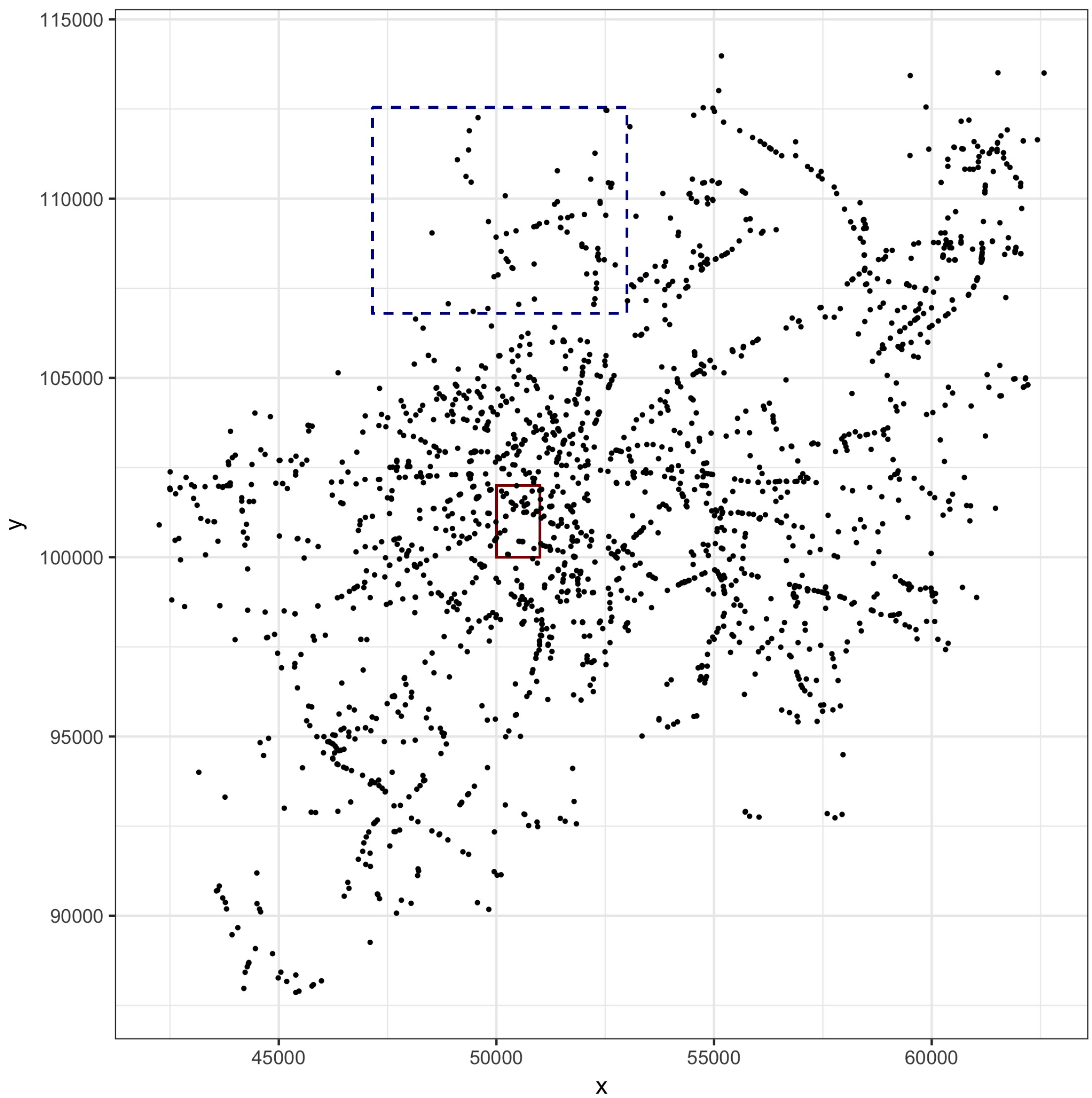
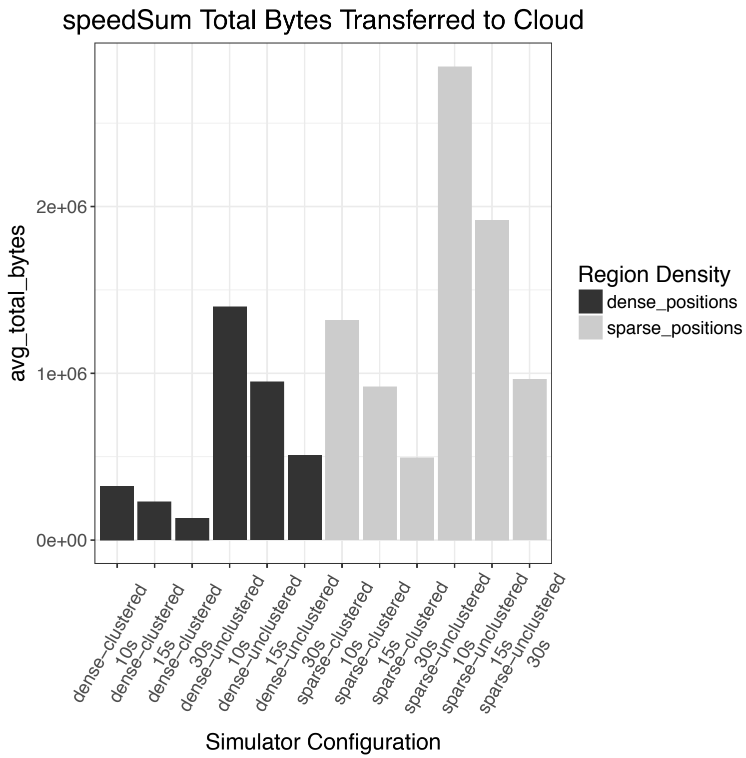
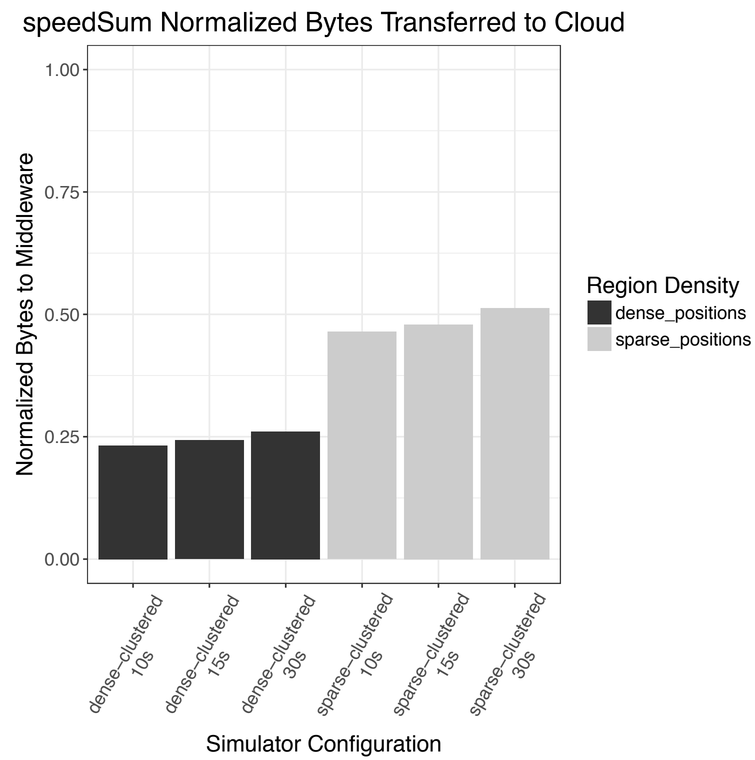


Figure 4.2: A Snapshot of Active Vehicles in VEINS Traces with Selected Regions Highlighted. The dense region is marked by the dark red box near the center of the graph. The sparse region is marked by the dashed blue box towards the center-top of the graph.

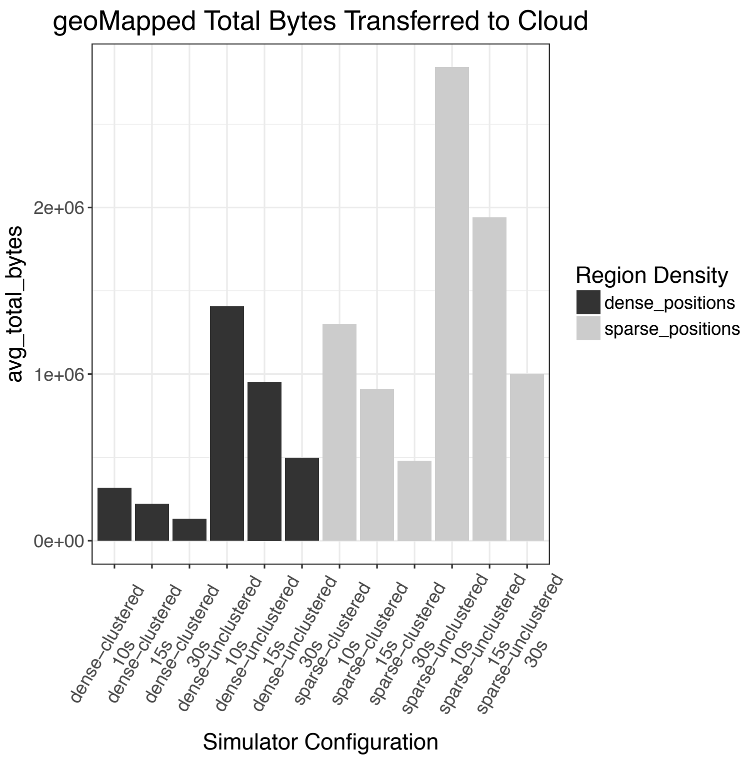


(a) The total number of bytes sent to the Middleware, averaged over multiple trials for the speedSum map/reduce job.

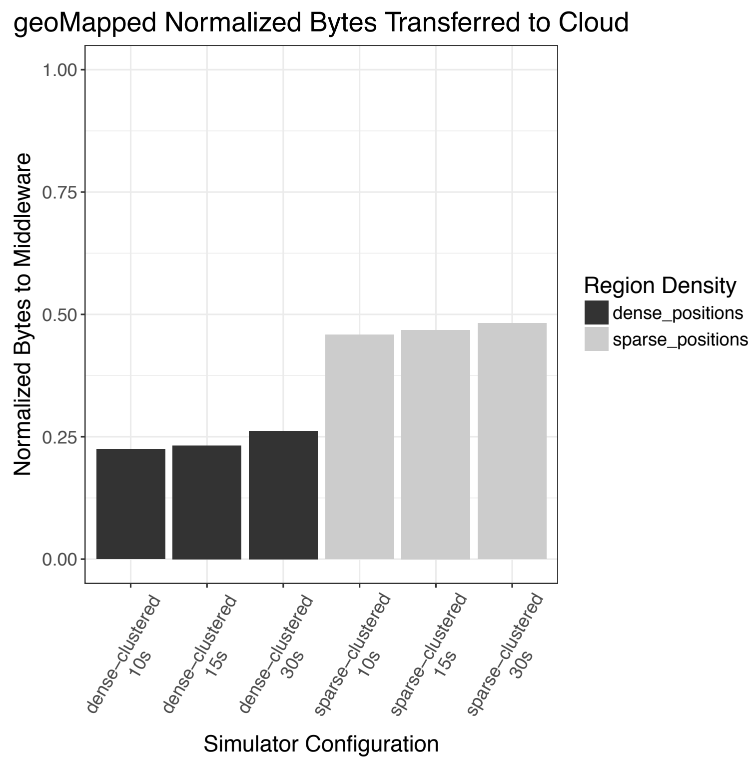


(b) The scaled number of bytes sent to the Middleware in a given region and window size configuration where y=1 indicates the number of bytes sent without clustering for a given region and reduce window size.

Figure 4.3: Results from running the speedSum map/reduce query. The x-axis contains different configurations of regions, clusterings, and reduce window sizes (10, 15, and 30 seconds). Smaller values in the same geographic region indicate better performance, where the best performance comes from using clustering with a 30 second window size. Demonstrates the data savings that occur as a result of using vehicle clusters. Also illustrates how data reduction is affected by regional vehicle density - areas of high vehicle density can take better advantage of clustering to get more savings.

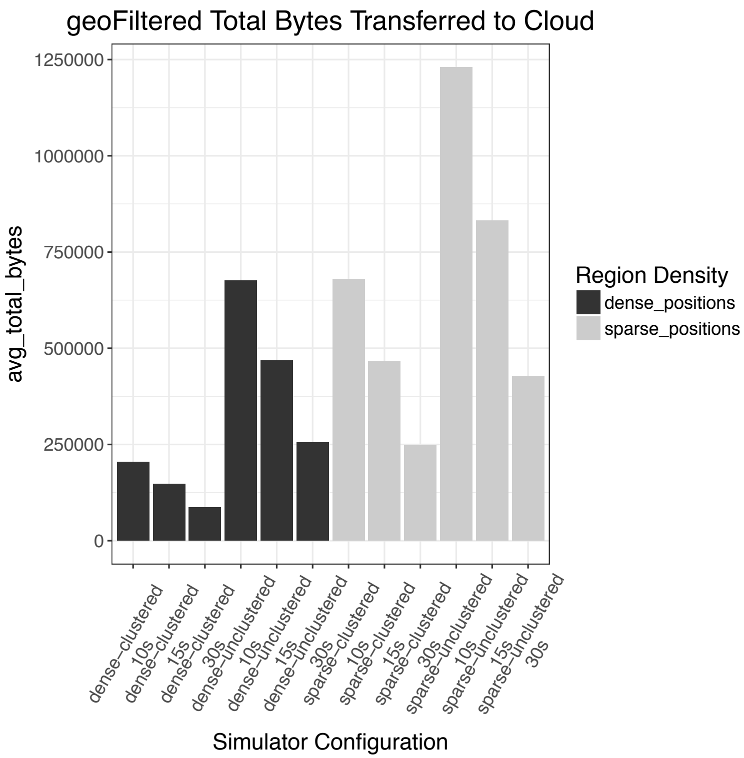


(a) The total number of bytes sent to the Middleware, averaged over multiple trials for the geoMapped map/reduce job.

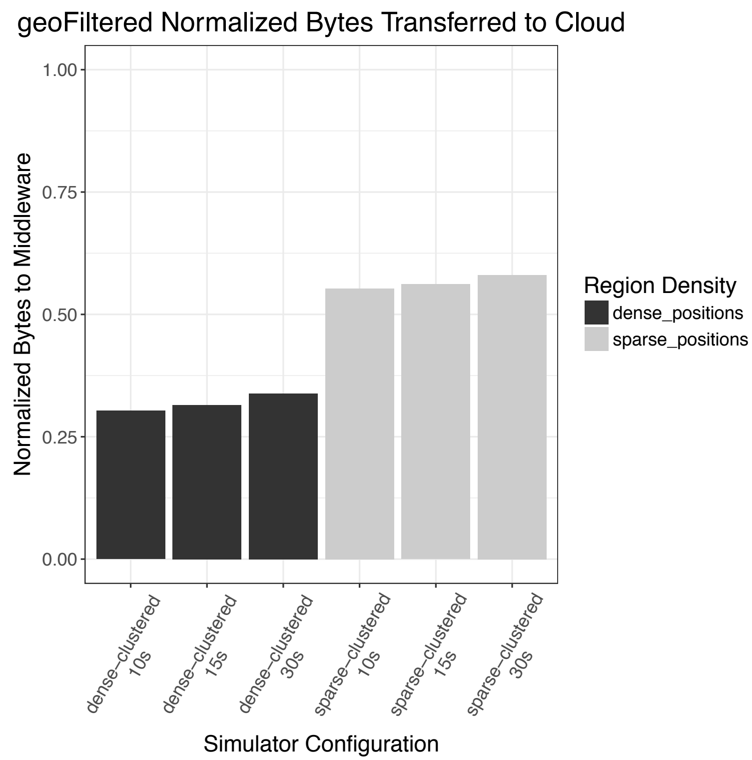


(b) shows the scaled number of bytes sent to the Middleware in a given region and window size configuration where y=1 indicates the number of bytes sent without clustering for a given region and reduce window size.

Figure 4.4: Results from running the geoMapped map/reduce query. The x-axis contains different configurations of regions, clusterings, and reduce window sizes (10, 15, and 30 seconds). Smaller values in the same geographic region indicate better performance, where the best performance comes from using clustering with a 30 second window size. Demonstrates the data savings that occur as a result of using vehicle clusters. Also illustrates how data reduction is affected by regional vehicle density - areas of high vehicle density can take better advantage of clustering to get more savings.



(a) The total number of bytes sent to the Middleware, averaged over multiple trials for the geoFiltered map/reduce job.



(b) shows the scaled number of bytes sent to the Middleware in a given region and window size configuration where y=1 indicates the number of bytes sent without clustering for a given region and reduce window size.

Figure 4.5: Results from running the geoFiltered map/reduce query. The x-axis contains different configurations of regions, clusterings, and reduce window sizes (10, 15, and 30 seconds). Smaller values in the same geographic region indicate better performance, where the best performance comes from using clustering with a 30 second window size. Demonstrates the data savings that occur as a result of using vehicle clusters. Also illustrates how data reduction is affected by regional vehicle density - areas of high vehicle density can take better advantage of clustering to get more savings.

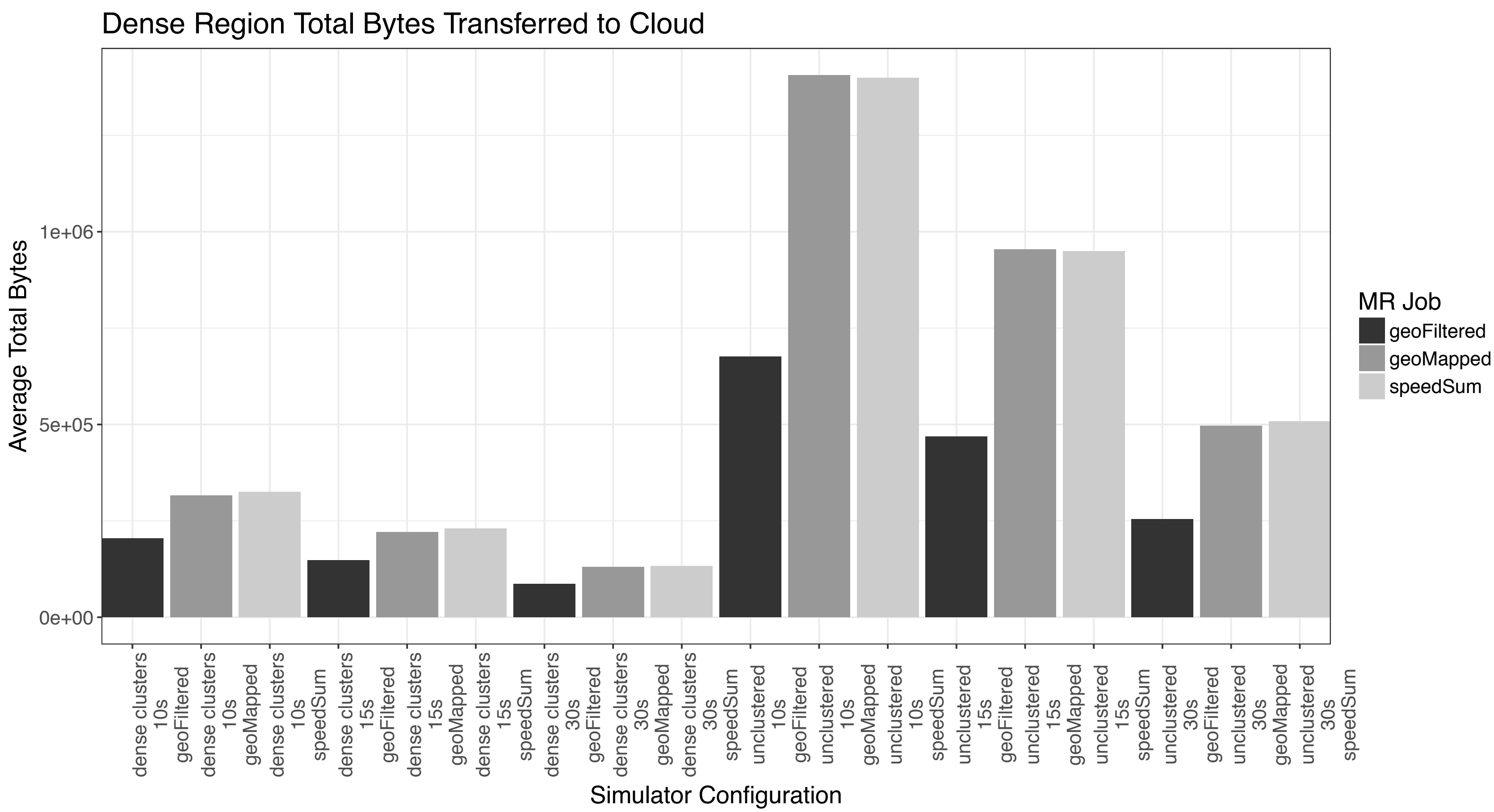


Figure 4.6: The total number of bytes sent to the Middleware from the dense region, averaged over all trials of a given configuration. The geoMapped and speedSum queries use approximately the same amount of data, while the geoFiltered query uses less as a result of fewer vehicles producing data for the query.

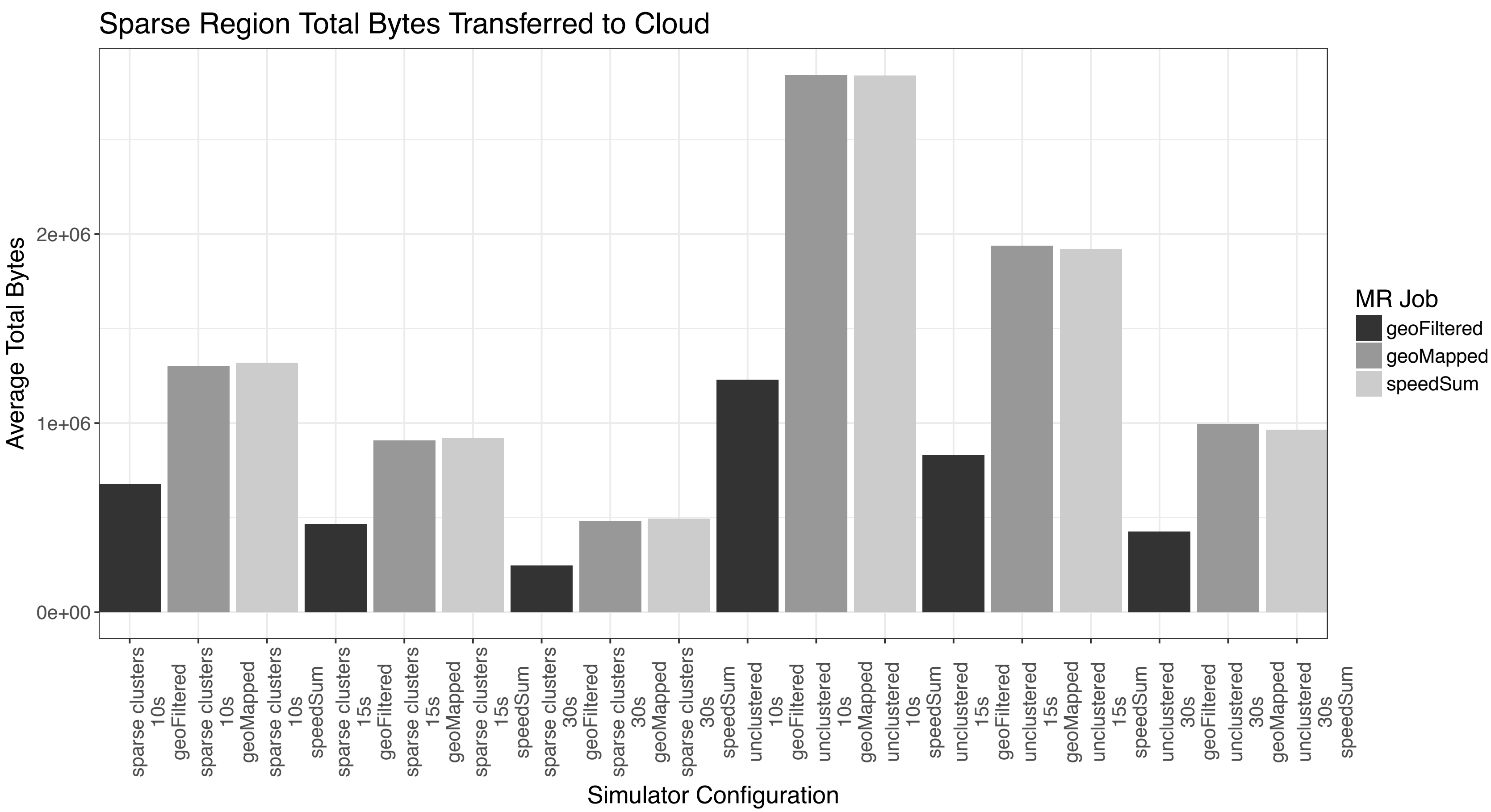


Figure 4.7: The total number of bytes sent to the Middleware from the sparse region, averaged over all trials of a given configuration. The geoMapped and speedSum queries use approximately the same amount of data, while the geoFiltered query uses less as a result of fewer vehicles producing data for the query.

5. Conclusion

In this paper, we discuss the state of the art in connected vehicle technologies as well as the problem of managing the vast quantities of data these vehicles are expected to produce. We further discuss some of the use cases that are being explored for the wealth of information that researchers expect to be available in these new connected cars. Finally, we review current and upcoming cloud technologies related to data processing, particularly with respect to the Internet of Things. Next, we provide an overview of a novel solution to the problem of managing vehicle data by extending cloud and big data technologies such as the map/reduce paradigm and Apache Kafka to the edge using the Spindle architecture. We discuss how existing tools such as Apache Spark can interfaced with edge computing systems through the use of the Spindle library for Spark Streaming.

We demonstrate the power of our paradigm by creating a simulator for our vehicle and Clusterhead software, which we test under a number of different workloads. We then demonstrate that the combination of clustering and message reduction used in our architecture produces significant bandwidth savings.

Future Work

In this work, we have demonstrated the efficacy of the core of the Spindle architecture - dynamic map/reduce on vehicles. There are a number of directions for future work, including extensions to the simulator and, more critically, extensions to the architecture. It should be possible in the future to implement a full-stack simulator that launches spark clients, runs the cloud Middleware, and interfaces with the existing SpindleSim software. We also believe it is worth investigating what other functions can be run at the edge, such as join and group functions, as well as how multiple simultaneous queries with such functions can be optimized to minimize the duplication of messages. By developing a more rich edge computation system, it will be possible to run more sophisticated algorithms at the edge, for example with the addition of stateful operators it might be possible to run machine learning algorithms at the vehicle and Clusterhead level.

It might also be worthwhile to integrate not only with Spark but also with Apache Flink [30] and/or Heron [31].

Beyond tweaking Spindle, itself, we believe there is now an opportunity for further research that builds upon the framework. Particularly, there are opportunities for improvements to vehicle clustering algorithms to improve cluster size and stability by integrating available information. Furthermore, the topic of what useful information can be gathered from connected vehicles will likely remain a perpetually open question, now that it is possible to combine sensor and traffic information with other online data sources; we saw how one group combined the data with geographic information, but there may be other interesting data-sets to pull in - everything from data on road construction to public event lists to economic data could be correlated with traffic flows. It might, for example, be possible to glean social and economic information from traffic flows based on variation in travel among seasons (indicating which areas host people with full time versus seasonal jobs). The real-time element might also allow real-time response to potential accidents and obstructions based on live traffic data.Furthermore, as has is being addressed in other research projects vehicles are themselves rich sensor platforms that can be used for gathering environmental data; it must now be determined what data is worth storing and to what degree of granularity it should be kept.

LITERATURE CITED

[1] X. Duan, Personal Communication.

[2] "Internet on wheels." [Online]. Available: https://www.hds.com/en-ous/pdf/white-paper/hitachi-white-paper-internet-on-wheels.pdf

[3] N. Lu, N. Cheng, N. Zhang, X. Shen, and J. W. Mark, "Connected vehicles: Solutions and challenges," *IEEE Internet of Things Journal*, vol. 1, no. 4, pp. 289–299, Aug 2014.

[4] A. Ahizoune and A. Hafid, "A new stability based clustering algorithm (sbca) for vanets," in *37th Annual IEEE Conference on Local Computer Networks - Workshops*, Oct 2012, pp. 843–847.

[5] N. Maslekar, M. Boussedjra, J. Mouzna, and H. Labiod, "C-drive: Clustering based on direction in vehicular environment," in *2011 4th IFIP International Conference on New Technologies, Mobility and Security*, Feb 2011, pp. 1–5.

[6] H. Wu, Z. Zhong, and L. Hanzo, "A cluster-head selection and update algorithm for ad hoc networks," in *2010 IEEE Global Telecommunications Conference GLOBECOM 2010*, Dec 2010, pp. 1–5.

[7] M. M. C. Morales, C. S. Hong, and Y. C. Bang, "An adaptable mobility-aware clustering algorithm in vehicular networks," in *2011 13th Asia-Pacific Network Operations and Management Symposium*, Sept 2011, pp. 1–6.

[8] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, 2004. [Online]. Available: https://research.google.com/archive/mapreduce.html

[9] E. Spaho, L. Barolli, G. Mino, F. Xhafa, and V. Kolici, "Vanet simulators: A survey on mobility and routing protocols," in *2011 International Conference on Broadband and Wireless Computing, Communication and Applications*, Oct 2011, pp. 1–10.

[10] W. Xu, N. R. Juri, A. Gupta, A. Deering, C. Bhat, J. Kuhr, and J. Archer, "Supporting large scale connected vehicle data analysis using hive," in *2016 IEEE International Conference on Big Data (Big Data)*. Institute of Electrical and Electronics Engineers (IEEE), Dec 2016, pp. 2296–2304.

[11] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Liu, and R. Murthy, "Hive - a petabyte scale data warehouse using

hadoop," in *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*, March 2010, pp. 996–1005.

[12] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, ser. HotCloud'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 10–10. [Online]. Available: http://dl.acm.org/citation.cfm?id=1863103.1863113

[13] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, ser. MSST '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–10. [Online]. Available: http://dx.doi.org/10.1109/MSST.2010.5496972

[14] *Getting Started with AWS*. [Online]. Available: http://docs.aws.amazon.com/gettingstarted/latest/awsgsg-intro/awsgsg-intro.pdf

[15] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: Fault-tolerant streaming computation at scale," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13. New York, NY, USA: ACM, 2013, pp. 423–438. [Online]. Available: http://doi.acm.org/10.1145/2517349.2522737

[16] G. Wang, J. Koshy, S. Subramanian, K. Paramasivam, M. Zadeh, N. Narkhede, J. Rao, J. Kreps, and J. Stein, "Building a replicated logging system with apache kafka," *Proceedings of the VLDB Endowment*, vol. 8, no. 12, pp. 1654–1655, aug 2015. [Online]. Available: http://dx.doi.org/10.14778/2824032.2824063

[17] "Kafka streams." [Online]. Available: http://docs.confluent.io/3.0.0/streams/

[18] *Akka Scala Documentation*. [Online]. Available: http://doc.akka.io/docs/akka/2.4/AkkaScala.pdf

[19] *PostgreSQL 9.6.2 Documentation*. [Online]. Available: https://www.postgresql.org/files/documentation/pdf/9.6/postgresql-9.6-US.pdf

[20] "Edgent." [Online]. Available: https://edgent.apache.org

[21] "Greengrass." [Online]. Available: https://aws.amazon.com/greengrass/faqs/

[22] "What is aws iot?" [Online]. Available: http://docs.aws.amazon.com/iot/latest/developerguide/what-is-aws-iot.html

[23] "Iot on aws overview." [Online]. Available: https://www.slideshare.net/AmazonWebServices/getting-started-with-aws-iot?next_slideshow=1

[24] "Internet of things (iot) solutions." [Online]. Available: https://cloud.google.com/solutions/iot/

[25] "Watson iot platform: Recognized as a leader in internet of things platforms." [Online]. Available: https://www.ibm.com/internet-of-things/platform/watson-iot-platform/

[26] R. Kearny, "Four things to watch for in your cloud blind spots," *Forbes*, 2017. [Online]. Available: https://www.forbes.com/sites/f5/2017/01/25/four-things-to-watch-for-in-your-cloud-blind-spots

[27] Scala language specification. [Online]. Available: http://www.scala-lang.org/files/archive/spec/2.11/

[28] "Babel the compiler for writing next generation javascript." [Online]. Available: https://babeljs.io

[29] "Veins." [Online]. Available: http://veins.car2x.org

[30] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flinktm: Stream and batch processing in a single engine," in *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 2015.

[31] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja, "Twitter heron: Stream processing at scale," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '15. New York, NY, USA: ACM, 2015, pp. 239–250. [Online]. Available: http://doi.acm.org/10.1145/2723372.2742788