Sboing4real: a real-time crowdsensing-based traffic management system

Theodoros Toliopoulos[a], Nikodimos Nikolaidis[a], Anna-Valentini Michailidou[a], Andreas Seitaridis[a], Theodoros Nestoridis[a], Chrysa Oikonomou[a], Anastasios Temperekidis[a], Fotios Gioulekas[a], Anastasios Gounaris[a], Nick Bassiliades[a], Panagiotis Katsaros[a], Apostolos Georgiadis[b], Fotis K. Liotopoulos[b]

[a] *Department of Informatics, Aristotle University of Thessaloniki, Greece*
[b] *Sboing, Thessaloniki, Greece*

Abstract

This work describes the architecture of the back-end engine of a real-time traffic data processing and satellite navigation system. The role of the engine is to process real-time feedback, such as speed and travel time, provided by in-vehicle devices and derive real-time reports and traffic predictions through leveraging historical data as well. We present the main building blocks and the versatile set of data sources and processing platforms that need to be combined together to form a fully-functional and scalable solution. We also present performance results focusing on meeting system requirements keeping the need for computing resources low. The lessons and results presented are of value to additional real-time applications that rely on both recent and historical data. Finally, we discuss the application of the aforementioned solution to a successful pilot study, where the full system has deployed and processed data from 800 taxis for a period of 3 months.

Keywords: vehicle traffic monitoring, IoT, stream processing, massive parallelism, OLAP, empirical evaluation

1. Introduction

Geographical Information Systems (GISs) and, more broadly, the development of applications based on or including geo-spatial data is a mature and hot

*Corresponding author
Email addresses: tatoliop@csd.auth.gr (Theodoros Toliopoulos), nikniknik@csd.auth.gr (Nikodimos Nikolaidis), annavalen@csd.auth.gr (Anna-Valentini Michailidou), sgandreas@csd.auth.gr (Andreas Seitaridis), nestorid@csd.auth.gr (Theodoros Nestoridis), oikochry@csd.auth.gr (Chrysa Oikonomou), anastemp@csd.auth.gr (Anastasios Temperekidis), gioulekas@csd.auth.gr (Fotios Gioulekas), gounaria@csd.auth.gr (Anastasios Gounaris), nbassili@csd.auth.gr (Nick Bassiliades), katsaros@csd.auth.gr (Panagiotis Katsaros), tolis@sboing.net (Apostolos Georgiadis), liotop@sboing.net (Fotis K. Liotopoulos)

area with several tools, both commercial and open-source, e.g., ArcGIS, PostGIS, GeoSpark [1], HadoopGIS [2], SpatialSpark [3]. In general, these tools and frameworks differ in the queries they support [4] and the quality of maps they utilize. For the latter, popular alternatives include Google Maps and OpenStreetMap[1], which can be considered as data-as-a-service. At the same time, urban trips is a big source of data; e.g., Uber Movement[2] shares data from more than 10 billion trips. Developing a system that can process real-time traffic data in order to report and forecast current traffic conditions combines all the elements mentioned above, e.g., modern GIS applications built on top of detailed world maps leveraging real-time big data sources, scalable stores and parallel processing platforms.

The aim of this work is to present architectural details regarding a novel back-end system developed on behalf of Sboing[3]. Sboing is an SME that implements innovative mobile technologies for the collection, processing and exploitation of location and mobility-based data. In particular, it implements an internationally patented, collaborative, crowdsourcing methodology for the collection, processing and distribution of traffic, road and sensory data, aiming to provide improved multi-GNSS (Global Navigation Satellite System) navigation and routing, globally. It offers an app, called UltiNavi, that can be installed on in-vehicle consoles and smartphones. Through this app, Internet-connected users share their location information in real-time and contribute to the collection of real-time traffic data. More specifically, Sboing collaborates with academia in order to extend their system and provide a crowdsensing solution [5] with a view to (i) continuously receiving feedback regarding traffic conditions from end users and process it on the fly; and (ii) providing real-time and accurate travel time estimates to users without relying on any other type of sensors to receive data apart from the data reported by the users. In order to achieve these goals, the back-end system needs to be extended to fulfill the following requirements:

R1: *Provide real-time information about traffic conditions.* This boils down to be capable of (i) providing speed and travel time conditions per road segment for the last few minutes and (ii) being capable to report incidents upon the receipt (and validation) of such a feedback.

R2: *Provide estimates for future traffic conditions.* This is important in order to provide accurate estimates regarding predicted travel times, which typically refer to the next couple of hours and are computed using both current and historical data.

R3: *Manage historical information* to train the prediction models needed by R2. This entails the need to store past information at several levels of granularity.

[1]`https://www.openstreetmap.org`
[2]`https://movement.uber.com/`
[3]`www.sboing.net`

R4: *Scalability.* Traffic reporting and forecasting can be inherently parallelised in a geo-distributed manner, i.e., each region to be served by a separate cluster of servers. A country as large as France is mapped using no more than 100M road segments. Therefore, the challenge is not that much in the volume of data to be processed at runtime but in the velocity of new update streams and the need to store historical data; i.e., the data volume becomes an issue when there is a need to store historical data for sophisticated prediction model building and further analyses, as in our case.

R5: *Fault-tolerance.* Any modules to be included in the back-end need to be capable of tolerating failures, so that the system availability is high and thus the solution can be used in practice.

There are several other tools that provide this type of information; e.g., Google Maps, Waze [4] and TomTom[5]. However none of these tools that are being developed by big companies have published information about their back-end processing engine. By contrast, we explain architectural details, while we also employ publicly available open-source tools, so that third parties can rebuild our solution with reasonable effort.

1.1. Contributions and Structure

This work makes the following contributions in relation to the requirements and the setting already described: (i) It presents an end-to-end crowdsensing-based solution for supporting real-time traffic reporting and forecasting as far as the communication subsystem and the back-end system engine is concerned. The architecture consists of several modules and integrates different tools and data stores. The different tools need to account for both streaming and batch processing, while storage needs to support diverse technical requirements, such as meeting real-time constraints and supporting analytical queries over historical data while keeping the volume of underlying data cubes limited. (ii) It discusses several alternatives regarding design choices in a manner that lessons can be transferred to other similar settings. For example, our work explains why using a main-memory database along with pushing the results of online analysis to persistent storage is important for real-time requirements and compares state-of-the-art scalable OLAP solutions. (iii) It includes indicative performance results that provide strong insights into the performance of each individual module in the architecture so that design choices can be evaluated, the efficiency in which requirements are met can be assessed, and bottlenecks can be identified. (iv) It presents a pilot case study, where the system has been successfully deployed on 800 taxis in the region of the city of Thessaloniki, Greece, for a period of three months.[6]

[4]`https://www.waze.com`

[5]`https://www.tomtom.com/automotive/products-services/real-time-maps/`

[6]An early version of this work has appeared in [6]; the main differences is that, in this manuscript, we present the comunication system along with its evaluation, we include more

From a system's point of view, the novelty of our work lies in (i) presenting a non-intuitive non-monolithic architecture encompassing three different data store types and two different stream processing platforms with complementary roles in order to meet the requirements; (ii) to the best of our knowledge, it is the first work that compares the two specific main alternatives regarding back-end analytics databases examined, namely Apache Kylin and Apache Druid; and (iii) the results presented are meaningful for software architects in different domains with similar requirements. Our architecture goes beyond the reference lambda architecture [7] and the main distinctive feature is that the stream and the batch processing components form an asynchronous pipeline and benefit from different data store types.

The remainder of this work is structured as follows. The next section gives the background on real-time traffic analysis. Section 3 presents the overall architecture. In the next section, we present the messaging component connecting the monitoring devices with the back-end system. Section 5 discusses the pre-processing of source data. In Section 6, we describe the underlying data warehouse and the queries that run over it. Indicative experiments showing the scalability of the system and its capability to support the requirements defined above are in Section 7. The pilot case study is presented in Section 8. We conclude in Section 9.

2. Background on real-time traffic reporting and forecasting

Real-time traffic reporting is an important part of navigation systems, which, nowadays, typically also provide a travel time prediction functionality; e.g., Google maps navigation annotates each proposed route with metadata about the length and the expected travel time. Effective forecasting of traffic can lead to accurate travel time prediction. Due to its practical applications, short-term traffic forecasting is a hot research field with many research works being published. Vlahogianni et al. [8] reviewed the challenges of such forecasting. These challenges refer to making the prediction responsive and adaptive to events (such as, weather incidents or accidents), identifying traffic patterns, selecting the best fitting model and method for predicting traffic and dealing with noisy or missing data. To forecast traffic, data can be collected in two manners, namely either through GPS systems deployed on vehicles or using vehicle detector sensors. The most common data features used in traffic prediction models are speed and travel time of vehicles along with the vehicle volume per time unit and occupancy of the roads. Djuric et al. [9] analysed travel speed data from sensors capturing the volume and occupancy every 30 seconds and advocated combining multiple predictors. Gao et al. [10] also used data from sensors but the main unit was vehicles per hour and the model they used is a neural network one. In our work, data come from user vehicles and, more specifically, we use

experiments for the parts already presented in [6] and we discuss the pilot case study, which has given rise to several technical challenges.

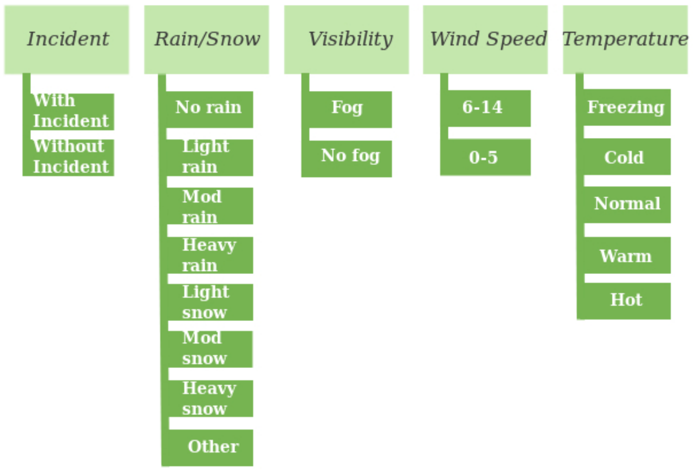


Figure 1: External condition metadata classification adopted by our approach: for each of the five categories considered, we present the possible values.

speed and travel time measurements. We show how we can build sophisticated models, but our solution can also rely on the latest measurements, implicitly assuming that the conditions in the next 5 minutes will remain similar to the current ones.

Traffic may also be affected from other incidents and conditions that need to be taken into consideration during prediction. E.g., an accident may lead to traffic congestion that cannot be predicted in advance. Also, weather conditions play a key role. Qiao et al. [11] presented a data classification approach, where the data categories include information like wind speed, visibility, type of day, incident etc., all of which can affect the traffic. Their prediction model is based on a k-nearest neighbors algorithm and, apart from the previous categories, mean speed and travel time were also used. Li et al. [12] conclude that a model that includes historical travel time data, speed data, the days of the week, 5-min cumulative rainfall data and time encoded as either AM or PM can lead to an accurate prediction. Essien et al. [13] studied how rainfall and temperature impact on traffic. They also consider peak and off-peak time zones. Through experiments, they showed that speed was reduced in case of rainfall, high temperatures and during peak hours. Based on the above proposals, we also consider the presence of an incident, rain or snow, the visibility, the wind speed and the temperature. Our metadata classification approach regarding external conditions is presented in Figure 1. More specifically, the values are discretized as shown in the figure. All these data can lead to a more accurate traffic forecasting when incorporated in a prediction model and the proposed back-end is designed to support their acquisition, storage and manipulation (although the presentation of the development of an accurate traffic prediction model is out of the scope of this manuscript).

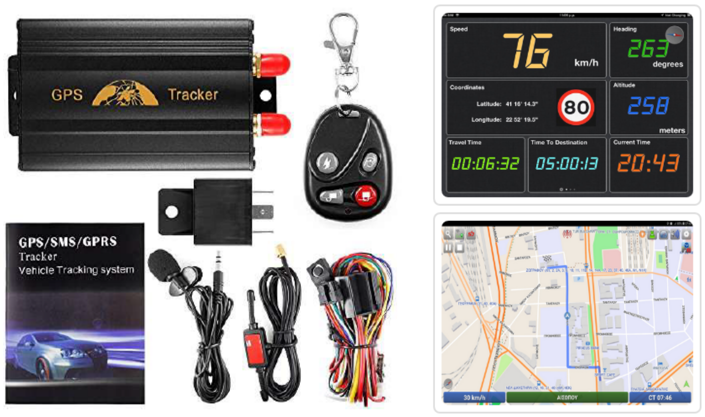


Figure 2: A TK103B GPS tracker used (left) and the more powerful monitoring and client app (right)

Finally, it is important to highlight that forecasting can be more accurate when there are large amounts of historical traffic data available [12]. These data include information collected over a long period of time (even years ago) and depart from simple models that consider the most recent measurements only, e.g., as in [9, 10, 11]. Historical data can help in the identification of traffic patterns on specific days or hours. For example, traffic on weekends typically varies from the traffic on weekdays. Thus it is important to incorporate this type of data in the traffic prediction model. The drawback is that this information can be very challenging to store and process efficiently due to its large size. Thus, suitable storage technologies must be used. In this work, we resort to a scalable data warehousing solution.

3. System Architecture

This section provides the high-level architecture and consists of two parts. The first part discusses the monitoring mechanisms and the communication between data sources and the back-end system. This part is further elaborated in Section 4. The second part provides the overview of the back-end pipeline that will be explained in detail in Sections 5 and 6.

3.1. Monitoring devices and communication subsystem

The raw data are transmitted from the users through user devices, such as GPS tracking devices and mobile navigation apps in a continuous stream. Figure 2 shows the two main options, namely SIM-enabled GPS trackers and mobile apps that can operate even without continuous internet connectivity. These apps are installed either on smartphones or on in-vehicle equipment. Every device periodically sends its current condition in a specific time interval; if there is no internet connectivity, the device may optionally send all the gathered data when connectivity is restored for historic analysis. The data sent through the messaging system contain information about the latitude and longitude of the device along with the timestamp that the measurements were taken and the speed of the user. Additional metadata about the position of the user such as elevation, course, road/segment id and direction are transmitted as well. That is, the device may have the capability to automatically map co-ordinates to road segments ids; explaining the details about how clients are developed are out of the scope of this paper. [7] Finally, tailored techniques for encryption and anonymization along with customized maps based on OpenStreetMap ones that allow for efficient road segment matching have been developed; these issues are not further analyzed in this work.

Overall, in real-time traffic monitoring based on crowdsensing, we distinguish between two types of data sources, according to their computing capabilities: (i) *thin* devices that only transmit data, e.g. GPS trackers, and (ii) *thick* devices that perform preprocessing. Thin devices transfer the specified information mentioned above more frequently, while thick devices allow for less frequent transmission of data that are already partially aggregated per segment id. For example, instead of reporting every second, a thick device can send reports every 10 seconds with the addition of the average speed over this interval for each road segment. Moreover, given that map information is installed on thick devices, these devices can report travel time per road segment. This is achieved by combining the co-ordinates information about the beginning of the segment with the timestamped locations of the vehicle while traveling through this segment; assuming that each segment id is a straight kine on the map, a travel time approximation can be provided. Our solution can work with both data source types, but it has been deployed using thick monitoring devices, namely the extended UltiNavi app.

The connection between the user devices and the back-end pipeline is materialized through the use of the MQTT [8] protocol. MQTT is a lightweight protocol that supports bi-directional data transfer while maintaining low latency. One of the key points of the protocol is the use of topics. Each client (device) subscribes to a topic and can both send data to that topic and receive any data sent from other clients (receiving feedback applies to thick clients only). In our broader solution, Mosquitto[9] has been chosen for the MQTT brokers. Moreover, each message is assigned to a specific region through the topics.

Scaling up the messaging system, e.g., due to both a transient and a steady increase in the volume of data transmitted, can be achieved by adding new brokers to the MQTT architecture; MQTT is scalable by design. The system is transparent from the end user's point of view, which means that the user

[7]Also, the vehicle type, e.g., car, truck, bicycle and so on, and the direction when traversing a segment id is identified; for simplicity, in the remainder of this work we do not distinguish between the directions when mentioning a specific segment or the vehicle types and we do not discuss the technical details to achieve this.

[8]`http://mqtt.org/`

[9]`https://mosquitto.org/`

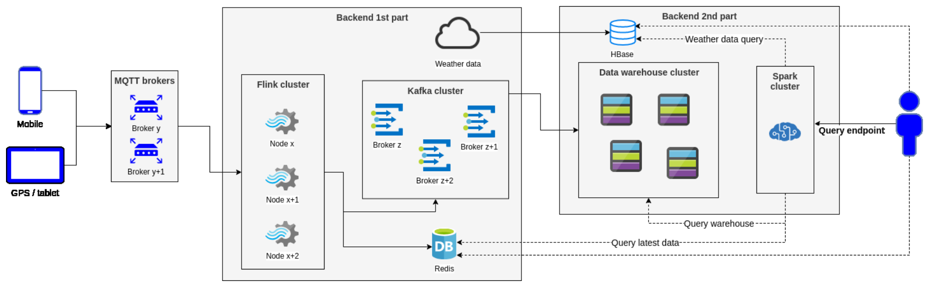


Figure 3: The diagram of the advocated architecture

devices do not need to know the number of the brokers or information about them in order to connect to a topic. On the other hand, when a new MQTT broker is added, the back-end system needs the information of its IP address to complete the connection. The process of updating the IP addresses pool is dynamic and does not slow down or suspend the rest of the pipeline.

3.2. Back-end architecture

The main responsibility of the back-end pipeline is to receive the raw data, deserialize, clean and process them, derive statistics for the last 5-minute tumbling window, and finally store the results in persistent storage for querying. This splits the pipeline into two conceptual parts. The first one handles the data cleaning and processing. The second part consists of the persistent data storage system and the querying engine. The main challenge regarding the first part of the solution is to handle a continuous intense data stream. This implies that the constituent modules need to share the following main characteristics: to be capable of fast continuous processing (to support R1 and R2 in Section 1) and to be scalable (which relates to R4) and fault tolerant (which relates to R5).

The streaming component comprises three main pieces of software, namely a streaming engine, a module to transfer results to persistent storage and a main memory database to support extremely fast access to intermediate results. The streaming engine handles the cleaning, transformation and processing of the raw data. It is implemented using the Apache Flink [10] framework. Flink is an open-source distributed continuous stream processing framework that provides real-time processing with fault-tolerant mechanisms called checkpoints. As such, R5 is supported by default. Flink can also easily scale up when the need arises to meet R4. As shown later, it can support efficiently R1 and R2. The second module that handles the transfer of the processed data to the persistent storage is built using Apache Kafka.[11]. Kafka is the most popular open-source streaming

[10]`https://flink.apache.org/`
[11]`https://kafka.apache.org/`

platform that handles data in real-time and stores them in a fault-tolerant durable way. Kafka uses topics to which other systems can publish data and/or subscribe to get access to those data. Kafka inherently meets R5 and does not become a bottleneck. Finally, the first part of the pipeline contains the main-memory database Redis.[12] This is due to the need for querying the latest time window of the stream (R1). Flink and Kafka can run on a small cluster serving a region or a complete country. Redis is a distributed database; in our solution it stores as many entries as the number of the road segments, which is in the order of millions that can very easily fit into the main memory of a single machine. Therefore, it need not be parallelized across all cluster nodes.

The second part of the pipeline is responsible for storing the processed data in a fault-tolerant way (which relates to R5) while supporting queries about the saved data at arbitrary levels of granularity regarding time periods, e.g., average speeds for the last day, for the last month, for all Tuesdays in a year, and so on, to support R2 and R3. For this reason, an OLAP (online analytical processing) data warehouse solution is required, which is tailored to supporting aggregate building and processing through operators such as drill-down and roll-up. To also meet the scalability requirement (R4), two alternatives have been investigated. The first is Apache Kylin [13] and the second is Apache Druid. [14] Both these systems are distributed warehouses that can ingest continuous streaming data. They also support high-availability and fault-tolerance. The main difference between the two systems is that Druid supports continuous ingestion of data, whilst Kylin needs to re-build the cube based on the new data at time intervals set by the user. To the best of our knowledge, no comparison of these two options in real applications, either in academic publications or in unofficial technical reports exists, and this work, apart from presenting a whole back-end system, fills this gap. Finally, Apache Spark [15] is the engine that is used for query processing as an alternative to standalone Java programs.

Figure 3 presents the complete back-end architecture. Starting off with the client's devices, such as mobiles and tablets, and the MQTT cluster of brokers that transfer the data, the data get ingested by the Flink cluster that forwards the processed results to the Kafka cluster and the Redis machine for temporary storage. Afterwards the results get ingested from Kafka to the data warehouse cluster. Since both Flink and the data warehouse alternatives examined are endowed by built-in Kafka connectors, there is no need for additional software components to write and read from Kafka. When an analytics query is submitted, the Spark cluster takes action by gathering data from the necessary storage (warehouse and/or Redis) and returns the query result. External conditions, such as weather, can be directly reported from external APIs and stored in a separate module, as explained later. Finally, road incidents are reported by the

[12]https://redis.io/
[13]http://kylin.apache.org/
[14]https://druid.apache.org/
[15]https://spark.apache.org/

monitoring devices.

4. The messaging component and security considerations

For mobile networks and IoT systems, a common characteristic that affects design decisions is the inherent trade-off between performance and security, when a system scales to higher-load usage scenarios. In these cases, we expect the system to retain the same quality of service in terms of performance, while having to face increased computation/communication load due to the cryptographic protocols and authorization requirements that assure adequate security. Moreover, we also expect the system to exhibit high availability (i.e. tolerating faults) and to transparently scale through automated load balancing [14].

The Sboing4Real system architecture is based on the MQTT protocol for the communication between the UltiNavi clients and the backend cloud system. Specifically, we used an open source implementation of the MQTT broker services, the well-known Eclipse Mosquitto broker.[16] Furthermore, we also used the HAProxy open source software service[17] that offers a multitude of load-balancing algorithms for distributing a message load among a series of servers (Mosquitto brokers). Our aim was to achieve higher levels of availability and transparent load-balancing of increasing numbers of simultaneous UltiNavi connections assumed to generate publish-subscribe messages in various levels of frequencies. Fault tolerance was implemented using the Keepalived[18] open source service, which performs continuous health checking between the primary and a backup HAProxy service, thus ensuring that the whole system survives single service failures. Upon completing the recovery of a previously failed HAProxy, the same instance takes on again the primary role for servicing incoming MQTT messages. The whole system architecture is shown in Figure 4, where the Authorisation Database is also depicted, which is part of the security architecture, as well as part of the backend data processing engine (Apache Flink for distributed processing of data streams). Finally, an emailing notification mechanism (the POSTFIX service[19]) was also configured, to alert the system administrator for critical events that potentially require him/her to handle.

The HAProxy service receives messages and then filters and forwards them to a broker, while balancing the message load according to the applied algorithm. Our system architecture was configured to work using the so-called *source algorithm*, which is based on a hash function that generates hash codes from the clients IP addresses. Our load-balancing approach was motivated by the need to include support for unicast communication through MQTT. Thus, for each UltiNavi client, the broker handling its connection will be identified through the hash code, i.e., every UltiNavi client is always connected to the system using a

[16]`https://mosquitto.org`
[17]`https://www.haproxy.com`
[18]`https://www.keepalived.org/`
[19]`http://www.postfix.org`

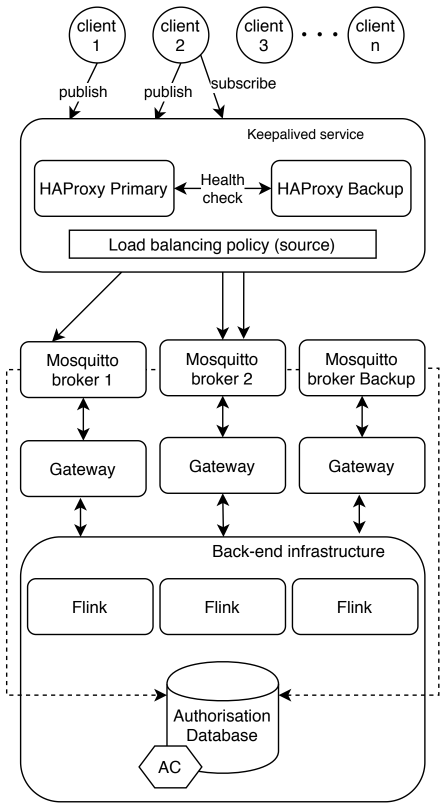


Figure 4: Sboing4Real communication subsystem architecture

single broker and just in case of a broker failure or system reconfiguration, the connection is handled by another broker. In case of a system reconfiguration (e.g. when adding extra broker), the load balancing algorithm updates automatically the weights used by the hash function, such that the load is transparently distributed to all available brokers, without any code modification.

Health-checks based on the KeepAlived service take place periodically (every 2 seconds) with negligible communication costs that cannot affect the overall system's performance. Upon a detected failure of the primary HAProxy, KeepAlived notifies the back-up HAProxy, which is pre-configured exactly as its twin HAProxy service and ready to take on the primary role. In overall, an HAProxy instance may be in one of the following distinct modes: primary, backup and faulty. Through the POSTFIX service, the system administrator is notified for each mode change.

In the cluster of available MQTT brokers, we employ at least one backup instance, which may be activated upon a detected error to any of the active brokers. The primary HAProxy is responsible for the broker status check and the decision to activate the backup broker(s) when needed.

At the security architecture level, our system is based on the Let's Encrypt Certificate Authority [20], which is a free and open authority that supports our security requirements, i.e. it provides certificates using the ECDSA algorithm with the secp384r1 elliptic curve and a 384-bit key length. More specifically, the brokers provide basic configuration options to control the access rights of clients to the topics based on Access Control Lists (ACLs). Publish-subscribe message communications can be authenticated and encrypted using the TLS (Transport Layer Security) protocol, in order to protect their confidentiality/integrity. However, special care should be paid towards using a comparatively light-weight TLS implementation (e.g. based on Elliptic Curve Cryptography (ECC) [15] instead of classical RSA), in order to avoid any scalability limitations due to the extra overhead implied by the underlying encryption algorithm. The Authorisation Database is connected to the Mosquitto broker using the open-source mosquitto-auth plugin, which supports username/password based authentication and authorisation for the access-controlled topics based on an external SQL database. We have chosen a synchronous multimaster replication architecture (e.g. PostgreSQL Database), thus enabling the provision of authentication and authorisation services over multiple sites. Finally, the gateway functionality in our system architecture is based on the Thingsboard IoT gateway[21], an open source implementation that we extended with additional functionality for the topics synchronization and the initialization of the ACL.

In order to further explain the topics hierarchy, we first need to elaborate on the functional details of the Sboing4Real crowdsensing system. For personalized delivery of location-based data updates at real-time and in regular intervals, we would need support for server-side identification of the road segments, on which the UltiNavi clients move. However, such a solution would cause redundant data traffic by potentially large groups of clients concentrated in busy urban areas. Instead of this, we have adopted an alternative approach that is also used by many other navigation systems, such as Bing Maps and OpenStreetMaps, which is based on a map segmentation into tiles. For a given zoom level Z, the map is partitioned horizontally and vertically into 2^Z zones. Then, an index is assigned to each zone, from 0 to $2^Z - 1$ and the distinct tiles can be uniquely identified by their index coordinates.

We identify two topic hierarchies to route messages with respect to the communication direction. Moreover, according to the structure of the topic hierarchies, each UltiNavi client can easily subscribe to one or more neighboring tiles simply by specifying the respective indexes. The categories of messages are the following:

- UltiNavi to back-end messages: the clients send a JSON message including the user's location, the tile's information and crowdsensing data (vehicle's speed, road segment, direction etc.), as well as road events such as traffic

[20]`https://letsencrypt.org/`
[21]`https://thingsboard.io/`

incidents.

- Back-end to UltiNavi messages: the back-end broadcasts events that have happened in a given tile or traffic-load estimations, by publishing the respective data to dedicated topics; all users that are currently moving to this specific tile - therefore subscribed to the corresponding topic - are then notified.

Further security protection is provided against UltiNavi users trying to abuse the two topic hierarchies, through the pattern option, which adds a 256-bit (32-byte) hash value to the topic string that is given as the result of the SHA256 cryptographic hash function.

The Sboing4Real system supports multitenancy, i.e. the system can simultaneously serve multiple tenants and each tenant represents a different group of users (e.g., UltiNavi clients for taxi companies, public transport operators, and so on) with specific privileges to publish and subscribe to the available topics. Authorisation management is based on RBAC combined with the access control for the topics hierarchy, thus grouping the management of large numbers of UltiNavi client access rights into well-defined groups.

This is not the first research work focusing on the architecture of crowdsesning systems. For instance, in [16], a publish-subscribe middleware for mobile crowdsensing is introduced that enables management of mobile sensor resources within the cloud and supports filtering and aggregation of sensor data on mobile devices prior to its transmission to the cloud. In our case, thick monitoring devices play exactly the same role. However, a key distinguishing aspect of our work is that we also focus on the scalability issues of crowdsensing systems, due to the trade-off between performance and security and the vital requirements of transparent load-balancing and high availability. Maintaining authorisation information in thousands of mobile devices is simply not feasible [17], and therefore we assume the existence of an authorisation service together with a series of additional security properties for the interactions with the authorisation server: mutual authentication, confidentiality and integrity of the credentials for the mobile devices [18]. Moreover, a lighweight security protocol should guarantee the mutual authentication, confidentiality and integrity of publish-subscribe interactions in the crowdsensing system. We therefore introduce a decentralised security architecture and evaluate its scalability prospects in the context of applying high availability and transparent load-balancing for crowdsensing systems.

5. The stream processing component

In this section, we present in detail the first part of the pipeline that includes the stream processing module, materialized by Flink, through which the raw data pass to be further processed, Kafka, which is the stream controller module that transfers the processed data to the storage, and Redis, which is a temporary main-memory database. We also present a fourth module that gathers weather data to complement the data reported by user devices.

5.1. The stream processing module

The data coming from the user devices create a continuous stream that goes through the MQTT brokers. The size of the data can quickly grow up in size due to the nature of the sources. For example, 800K of vehicles in a metropolitan area equipped with thick clients reporting once every 10 seconds, still generate 80K new sets of measurements per second, which amounts to approximately 7 billion measurements per day.[22] Overall, the pre-processing module needs to be able to handle an intense continuous stream without delays. Flink provides low-latency, high-throughput and fault-tolerance via checkpoints. It incorporates the *exactly-once* semantics, which means that, even in the case of node failures, each data point will be processed only once. In addition, scaling up can easily be completed through adding more worker machines (nodes).

In order to get the data that come from the user devices, Flink needs to connect with all of the MQTT brokers, which can adapt their number according to the current workload. Each Flink machine has a list of all available MQTT brokers along with their IP addresses. When a new broker is inserted in the pool, one of the available Flink nodes initiates the connection in order to start the ingestion. Flink can increase or decrease the number of its worker nodes without shutting down due to its built-in mechanisms.

In addition, deserialization takes place. That is, consecutive messages per vehicle are grouped by road segment id so that there is a single transmission. To save space, this type of messages contain a header with the segment id and the length of the measurement sequences. The body contains the time, speed, heading, distance from the beginning of the segment and location sequences in a compressed form. On average, in our pilot case study, each client transmits 21 bytes per 10 seconds, which occupy 40 bytes after deserialization.

After Flink starts ingesting the stream, it creates a tumbling (i.e., a non-overlapping) moving time window to process the data points. The measurements are aggregated according to the road segment they refer to. The size of the tumbling window is set to 5 minutes, since it is considered that the traffic conditions in the last 5 minutes are adequate for real-time reporting, and the traffic volume in each road segment in the last 5 minutes is high enough to allow for dependable statistics. The data points that fall into the window's time range are cleaned and several statistics, such as median speed, quartiles and travel time are computed. The results of every window are further sent downstream the pipeline to Kafka. In parallel, the data from the most recent time window are also saved to Redis overwriting the previous window. Continuously reporting real-time changes is possible, but it is rather distracting than informative.

Flink can also be used for more complex processing that involves data streams. One such example is continuous outlier detection on the streaming data from clients. Detecting an outlier can either indicate an anomaly in a cer-

[22]As reported at `https://www.marketdatapeaks.net/rates/usa/`, this is in the same order of the messages transmitted per day in real-time market data feeds from equities, futures, and options markets in the US.

tain road segment, i.e. an accident, or simply noisy data, i.e. a faulty device or a stopped vehicle. Especially in traffic forecasting, quickly detecting an accident can result in a decrease of congestion in the specific road. Further discussion on this is deferred to Section 8.

5.1.1. Weather data acquisition

In order to provide the user with more information about the road conditions as well as make more precise predictions of the future traffic and trip times, we gather weather data by using weather APIs including Dark Sky[23] and Open Weather Map [24]. Weather data are collected for wide geographical areas, each one comprising its own unique set of road segments. i.e., each geographical area is automatically mapped to a pre-computed and cached set of road segments. For each area, a request is sent every hour to one of the available APIs using its coordinates as parameters. Because each API returns a different set of metrics in its response, the received data are cleaned and transformed so that only a portion of the data is retained, namely the phenomena that are considered more likely to affect the road conditions. In more detail, these data are: (i) a text summary of the current condition (e.g. clear, rain), (ii) the visibility (in km), (iii) the wind speed (in km/h) and (iii) the temperature (in Celsius degrees). After the weather data of an area have undergone transformation, they are discretized to conform to the schema presented in the next section and are sent to persistent storage.

5.2. The stream controller and temporary storage modules

As depicted in the overall architecture, the processed data are forwarded to Apache Kafka. Kafka is one of the most popular distributed streaming platforms and is used in many commercial pipelines. It can easily handle data on a big scale and it is capable of scaling out by adding extra brokers; therefore it is suitable for meeting the R4 requirement. It uses the notion of topics to transfer data between systems or applications in a fault-tolerant way; thus it also satisfies R5. Topics have a partitioning and replication parameter. The first one is used in order to partition the workload of the brokers for the specific topic whilst the second one is used to provide the fault-tolerant guarantees. An additional useful feature is that it provides a retention policy for temporarily saving the transferred data for a chosen time period before permanently deleting them. We will explain later how we can leverage this feature to avoid system instability. The consumers of Kafka can read data from a topic starting either from a specific timestamp, or from the beginning of the topic (taking into account the retention policy) or from the latest received record. In our work, Kafka is used as the intermediate between the stream processing framework and the data warehouse. In our case, apart from receiving the output of Flink, it is also used for alerts received, such as an accident detection, by passing them through specific topics.

[23]`https://darksky.net/dev`
[24]`https://openweathermap.org/api`

The final module in this part of the pipeline is Redis, which stores the latest statistics so that we do not have to rely on the data warehousing module solely in order to retrieve them. Overall, the statistics aggregated by Flink are passed on both to Kafka for permanent storage and to Redis for live traffic conditions update. In addition, as explained in the next section, Redis holds the predicted travel time for each segment id, and, when combined with Kylin, it may need to store the two last 5-minute sets of statistics.

6. Data Storage and Querying

Here, we describe the OLAP solutions and the type of queries over such solutions and Redis to support R1, R2, and R3 in a scalable manner.

6.1. Scalable OLAP

OLAP techniques form the main data management solution to aggregate data for analysis and offer statistics across multiple dimensions and at different levels of granularity. From a physical design point of view, they are classified as ROLAP (relational OLAP), MOLAP (Multidimensional OLAP) and HOLAP (hybrid OLAP) [19]. Scalable OLAP solutions are offered by the Apache Kylin engine. An alternative is to leverage the Druid analytics database. We have explored both solutions.

Kylin is deployed on top of a Hadoop[25] cluster and goes beyond simple Hive[26], which is the main data warehousing solution offered by Apache. Hive allows for better scalability but does not support fast response time of aggregation queries efficiently [19]. To mitigate this limitation, Kylin encapsulates the HBase[27] NoSQL solution to materialize the underlying data cube according to the MOLAP paradigm. The overall result is a HOLAP solution, which can answer very quickly statistics that have been pre-computed, but relies on more traditional database technology to answer queries not covered by the materialized cube.

The important design steps are the definition of dimensions and measures (along with the appropriate aggregate functions). For the measures, we consider all Flink output, which is stored in Kafka, using several aggregation functions. For the dimensions, we employ two hierarchies, namely the map one consisting of road segments and roads, and the time one at the following levels of granularity: 5 minutes window, hour, day, week, month, quarter, year. Note that the time hierarchy is partially ordered, given that aggregating the values of weeks cannot produce the statistics per month. Overall, precomputed aggregates grouped by time or complete roads or individual road segments or combinations of road and time are available through Kylin. The cube, as defined above, does not consider external condition metadata (i.e., weather information and accidents).

[25]`https://hadoop.apache.org/`
[26]`http://hive.apache.org/`
[27]`http://hbase.apache.org/`

There are two options in order to include them, namely either to add metadata conditions as dimensions or to consider them as another type of measure. Both options suffer from severe drawbacks. Having external conditions as a set of different dimensions, e.g., one dimension for each feature in Figure 1 would result in a sparse physical cube, given that pre-aggregates are according to the MOLAP paradigm. Such sparsity would explode the cube size, since there would be a potentially empty entry for each combination of time slot, road segment id and weather condition type. Also, there would be many repeats, since the weather conditions would be copied for all road segments in the same region. On the other hand, having external conditions as measures makes little sense in terms of aggregation while not solving the data redundancy problem. Moreover, weather conditions do not evolve very rapidly and is adequate to monitor weather conditions less frequently than in 5-minute windows. Thus, we have opted to employ a third type of storage apart from Kylin and Redis, namely HBase. HBase is already used internally by Kylin; here we explain how we employ it directly. More specifically, we store all external condition metadata in a single column family in a HBase table. The key is a road and hour pair, i.e., weather conditions for a specific region are mapped to a set of roads (rather than road segments) and are updated every hour. The values from the weather APIs are transformed as presented below, based on the literature findings discussed in Section 1:
1. Current Condition ($Text \rightarrow Integer$): {0: no rain, 1: light rain, 2: moderate rain, 3: heavy rain, 4: light snow, 5: moderate snow, 6: heavy snow, 7: other}
2. Visibility ($Double \rightarrow Boolean$): $true$ if fog ($\geq 10km$), $false$ in any other case
3. Wind Speed ($Double \rightarrow Boolean$): $true$ if strong wind ($> 10.8km/s$), $false$ in any other case
4. Temperature ($Double \rightarrow Integer$): {1: freezing ($temp < 5C^{\circ}$), 2: cold ($5C^{\circ} \leq temp < 15C^{\circ}$), 3: normal ($15C^{\circ} \leq temp < 25C^{\circ}$), 4: warm ($25C^{\circ} \leq temp < 30C^{\circ}$), 5: hot ($temp \geq 30C^{\circ}$)}
5. In addition, a fifth column stores the presence of accidents taking values 0 or 1.

An alternative to Kylin is Druid. Druid can connect to Kafka and may be used to replace even Flink aggregation preprocessing to automatically summarize data by splitting them in 5-minute windows. In our solution, we keep using Flink for preprocessing (since this can also be enhanced with outlier detection) and we test Druid as an alternative to Kylin only. Contrary to Kylin, Druid has no HOLAP features and does not explicitly precompute aggregate statistics across dimensions (which relates to the issue of cuboid materialization selection [20]). However, it is more tailored to a real-time environment from an engineering point of view. Druid data store engine is columnar-based coupled with bitmap indices on the base cuboid, which is physically partitioned across the time dimension.

6.2. Query Processing

Supporting the real-time reports according to R1 relies on accessing the Redis database. R2 and R3 involve forecasts, and in order to forecast traffic an appropriate model needs to be implemented. This model acquires two main types of data; real-time statistics of the last 5 minutes and historical ones. The model analyses data like travel time, mean speed and so on, by assigning a weight to each of the two types mentioned above. The model can also incorporate information about weather or any occurred incidents.

Regarding real-time querying, the results include information for the last 5 minutes for all the road segments and are stored in a Redis database. We can retrieve them through Spark using Scala and Jedis[28], a Java-Redis library. In order to use this information that is in JSON string format, there is a need to transform it to a Spark datatype, for example DataSet. Overall, as will be shown in the next section, this is a simple process and can be implemented very efficiently thanks to Redis, whereas solely relying on Kylin or Druid would be problematic. Historical data can grow very large in space as they can contain information about traffic from over a year ago and still be useful for forecasting. Thus, historical querying is submitted to Kylin or Druid. To meet R2 and R3, we need to train a model and then apply it every 5 minutes. Developing and discussing accurate prediction models for traffic is out of the scope of this work. In general, both sophisticated and simpler models are efficient in several workload forecasting problems with small differences in their performance, e.g., [21]. But, as explained in the beginning, the important issue in vehicle traffic forecasting is to take seasonality and past conditions into account. Without loss of generality, an example function we try to build adheres to a generic template:

$$\tilde{X_{i,t}} = w_{i,1} \cdot X_{i,t-1} + w_{i,2} \cdot X_{i,t-2} + w_{i,3} \cdot (X_{i,t-1week+1} - X_{i,t-1week}), \qquad (1)$$

where $\tilde{X_{i,t}}$ is the next 5-minute metric, either expected speed or travel time of the i^{th} segment at time slot t, that we want to predict based on the values of the last two 5-minute windows and the difference in the values exactly 1 week ago. 1 week corresponds to $7 \times 24 \times 12 = 2016$ 5-minute windows. Using Spark jobs, we periodically retrain the model, which boils down to computing the weights $w_{i,1}$, $w_{i,2}$ and $w_{i,3}$. To provide the training data, we need to retrieve the non-aggregated base cube contents. We can train coarser models that are shared between road segments or train a different model for each segment. Obviously, the latter leads to more accurate predictions. In the next section, we provide detailed evaluation results regarding the times to retrieve cube contents.

Using a Spark job we retrieve the historical data of a specific segment, transform them to a set of tuples with 5 fields $(X_{i,t}, X_{i,t-1}, X_{i,t-2}, X_{i,t-1week+1}, X_{i,t-1week})$ and build a linear regression model. This procedure does not form a bottleneck as it can run in the background and its execution time is a few minutes. The coefficients are cached; Redis can be used to this end. Upon the completion of each

[28]`https://github.com/xetorthio/jedis`

CPU	**Cores/Threads**	**RAM**	**Storage**
Cluster A			
Intel(R) Xeon(R) CPU E5-2620 v2 @ 2.10GHz	6/12	64G	SSD
Intel(R) Core(TM) i7-3770K CPU @ 3.50GHz	4/8	32G	SSD
AMD FX(tm)-9370	8/8	32G	SSD
Intel(R) Xeon(R) CPU E5-2640 v2 @ 2.00GHz	8/16	64G	SSD
Cluster B			
Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz	6/12	64G	2 SSD & 2 HDD
Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz	6/12	64G	2 SSD & 2 HDD

Table 1: Cluster information

5-minute window, based on the precomputed co-efficients, predicted statistics are computed for each road segment for the next time window.

Finally, we provide Spark queries to support the provision of meaningful feedback to drivers and users. More specifically, by retrieving data from the data warehouse (Kylin or Druid) we compute the average speed of every segment for every day of the week and hour of the day (using both simple and weighted mean values). This provides meaningful insights into the overall traffic conditions of the city. Also another important measurement when dealing with real-time traffic is the free flow speed of each segment; that is the average speed when there is no congestion. In order to obtain this speed for a segment, we calculate the median speed of the 5% highest speeds recorded for this segment at a defined time period (usually multiple months). This free flow speed when compared to the last 5-min speed can reveal which roads are currently congested. Finally, the results are stored in a database in order to be easily accessible. This whole process takes only a few seconds thus being suitable for running everyday in order to ingest the new data and update the statistics.

7. Performance Evaluation

In this section, we provide concrete experimental results regarding the architecture's efficiency and capability to handle large data volumes. We first focus on the stream processing component and we measure the resource consumption of Flink by simulating an intense input stream as well as the time it takes to write the intermediate data referring to the last 5-minute window to Redis. We continue by experimenting with both data warehouses using different settings and comparing their advantages and disadvantages. Next, we present the results of the query module using both Spark and a simple Java program as the query endpoint measuring the time it needs to gather the necessary data from the warehouses as well as Redis. Finally, we discuss end-to-end times including the latency due to the messaging component.

7.1. Experimental setting

All of our experiments, unless explicitly stated, are performed on two clusters, the technical characteristics of which are presented in Table 1. The first cluster, denoted as Cluster A, is deployed in private premises and comprises 4 heterogeneous machines both in CPU and RAM resources while the second one, denoted as Cluster B, has two identical powerful machines, rented from an established cloud provider. Both clusters are small in size and are meant to serve a limited geographic region, since it is expected each important municipality or region to have its own small cluster. The experiments on both of these clusters, as expected, confirm that a homogeneous cluster has better performance than an heterogeneous one even if the latter has more machines; however, we present experiments in both settings since, for some companies, the typically heterogeneous private cluster solution might be preferable, e.g., for a combinations of privacy, security and economic purposes.

For storage, both clusters run HDFS. In the second cluster that has both SSD and HDD storage types, the first one is used for persistent storage of the data warehouses' cubes while the HDD are used for temporary data with the help of HDFS's Heterogeneous Storage. The input stream is artificially generated in order to be continuous and intense reaching up to 80000 raw data tuples per second and 10 million total road segments, e.g., serving 800K vehicles reporting once every 10 seconds simultaneously in a region larger than half of Greece. This yields a stream of 288 million MQTT messages per hour with approximately 288 tuples per road segment for the whole hour. Note that having to update all road segments rather than a portion of them, as is more common, stresses more the system. As such, the artificial data traffic is higher than the expected traffic, even if we assume that our solution becomes the most popular GPS navigation product in the same area, i.e., it is not expected the number of simultaneously operating devices in the same area to exceed 800K.

7.2. Stream processing experiments

The objective of this experiment is to reveal the resources Flink consumes for the data processing step while meeting the real-time requirement R1 and the scalability requirement R4. The processing job is tested on both clusters using the YARN cluster-mode. For Cluster A, the YARN configuration comprises 4 nodes with 1 core and 8GB RAM per node. The job's total parallelism level is 4. For Cluster B, YARN uses 2 nodes with 2 cores and 16GB RAM per node with a total parallelism level of 4. Note that we aim not to occupy the full cluster resources, so that the components downstream run efficiently as well.

In the experiments, we keep the total road segments to 10 millions, while increasing the input rate of the stream starting from 40 tuples per second (so that each Flink node is allocated on average 10 records in Cluster A) and reaching up to 80000 tuples (i.e., 20000 per Flink node in Cluster A) per second. The reason for the constant number of road segments is to show the scalability of Flink in accordance to the R4 requirement regarding the volume of data produced per time unit keeping the underlying maps at the appropriate level for real world applications. For stress test purposes, the number of distinct devices that send data is also set to 1000 and kept as a constant. This means that each device sends multiple raw tuples, and thus the process is more intense due to the computation of the travel time for each device (i.e., if the same traffic is shared

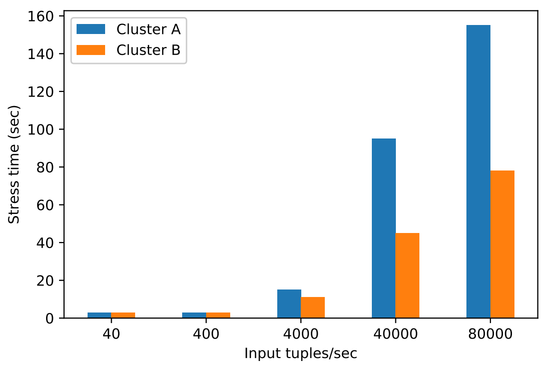


Figure 5: Flink's stress time during stream data processing

across 100K devices, then the computation would be less intensive). The process window is always a 5-minute tumbling one. This implies that the Flink job gathers data during the 5 minutes of the window while computing temporary meta-data. When the window's time-life reaches its end point, Flink completes the computations and outputs the final processed data that are sent through the pipeline to Kafka and Redis. The time between the window termination and the output of the statistics of the last segment is referred to as Flink stress time.

Figure 5 shows the results of the Flink process job on both clusters by varying the input stream rate. The measurements are the average of 5 runs. The plot deals with meeting R1 and shows the average stress time for the cluster upon the completion of the 5-minute window. As mentioned above, Flink computes meta-data and temporary data during the whole window in order to start building up to the final aggregations needed for the process. When the window expires after it has ingested all the data tuples that belong to the specified time period, the computations are increased in order to combine temporary meta-data and complete any aggregations needed in order to output the results. The stress time presented in the figure shows the running time of this process for each cluster in order to output the complete final results for a window to Kafka and Redis. As the rate of the input data increases, so does the stress time due to the increased number of complex computations. In cluster A, the stress time starts from 3 seconds for the lowest input rate and reaches up to 155 seconds during the maximum rate. On the other hand, the homogeneous cluster exhibits even better results, and its stress time duration does not exceed 80 seconds even for the highest workload. This means that after 80 seconds, the complete statistics of the last 5-minutes are available to Redis even for the last segment; since the whole process is incremental, many thousand segments have updated 5-minutes statistics even a few seconds after the window slide. After the results are in Redis, they can be immediately pushed or queried to update online maps. In [6], we also discuss CPU utilization per YARN node and memory consumption.

Tuples	**Local (sec)**	**Remote (sec)**
20000	3	3
100000	3	12

Table 2: Flink-to-Redis transfer

The process stage ends when the results are passed to Kafka and Redis. Due to Kafka's distributed nature and the fact that each Kafka broker is on the same machine as each Flink node, the data transfer between the two systems is negligible. On the other hand, Redis is used on a single machine as a centralized main-memory store according to the discussion previously. This incurs some overhead when transferring data from a remote machine, which is already included in the stress times presented.

Table 2 presents results of the data transfer between Flink and Redis. As shown in the table, local transfer is insensitive to the volume of data. However, the remote transfer incurs an overhead that grows almost linearly in the amount of data transmitted. Still, these times are hidden by the overall Flink process.

7.3. Persistent storage experiments

Persistent storage is the key component of the whole architecture. As explained in Section 6, the design choices include two alternatives, Kylin and Druid. Both require a Hadoop cluster to correctly function, whereas Kylin may also employ either Hadoop MapReduce or Spark as a cube building tool. In our experiments, we have chosen to use Spark because we also use it as the query mechanism. We have experimented with Kylin 2.6.1 and Druid 0.15.1. We used the output of the previous experiments to test the ingestion rate of both warehouses that indirectly affects the efficiency regarding R2 and R3. The total number of distinct road segments is kept at 10 million. Since Druid supports continuous ingestion while Kylin needs to update the cube at user-defined intervals, the two solutions are not directly comparable and thus their experimental settings differ. Nevertheless, the results are sufficient to provide strong insights in the advantages and drawbacks of each solution.

The following experiments present the total time that Kylin needs in order to update the cube at 3 different time intervals, namely every 5, 15 and 30 minutes. The first interval (5-minutes) means that Kylin rebuilds the cube after every window output from Flink, while 15 and 30 minute intervals imply that the cube is updated after 3 and 6 window outputs from Flink, respectively. As more windows are accumulated in Kafka before rebuilding, more data need to be processed by Kylin's cube building tool and incorporated into the cube itself. Based on the previous experiments, different input tuple rates in Flink provide a different number of processed output rows, which in turn are ingested into Kylin. For example, at the lowest rate of 40 tuples/sec arriving to Flink, on average 11961, 35891 and 71793 road segments are updated in 1, 3 and 6 windows, respectively. On the contrary, at the highest rate of 80K tuples/sec, the amount of updated segments is 5.19M, 21.47M and 48.86M, respectively. 5.19M implies that more than half of the map is updated every 5 minutes.

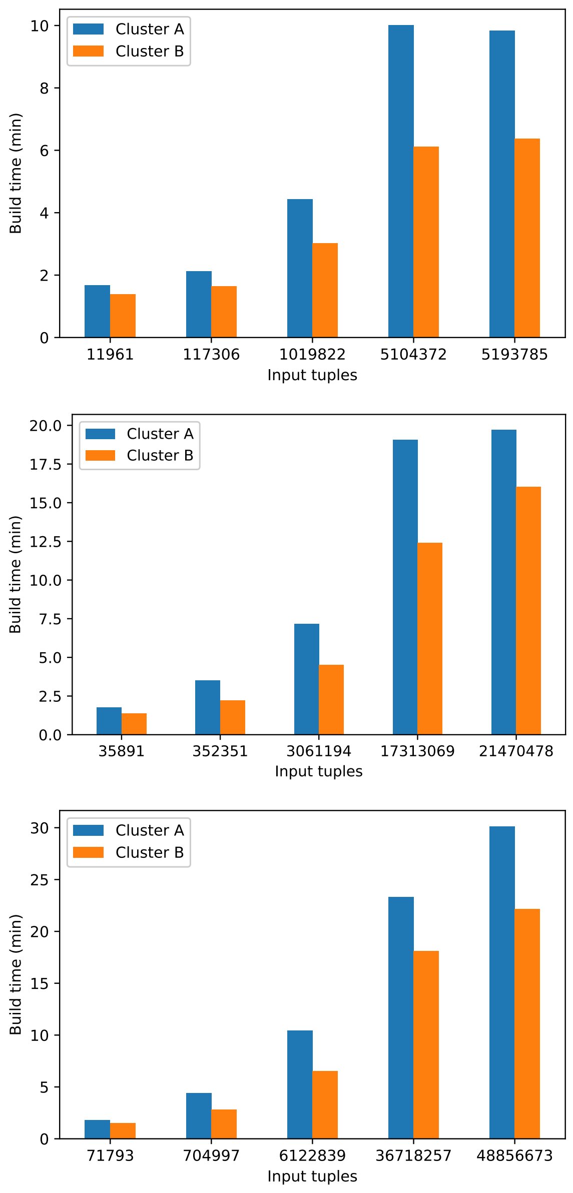


Figure 6: Kylin's average cube build time after ingesting 1 (top), 3 (middle) and 6 (bottom) 5-minute windows

Fig. 6 shows the results for the two clusters employed. An initial observation is that the homogeneous cluster (Cluster B) consistently outperforms the heterogeneous one. This provides evidence that Kylin is sensitive to heterogeneity. The top plot from Figure 6 shows the build times when the input tuples vary from approximately 11K to 5M, all referring to the same 5-minute window. The two rightmost pairs of bars are similar because the number of updated segments does not differ significantly. The main observation is twofold. First, when the input data increases in size, the build time increases as well but in a sublinear manner. Second, for more than 5M tuples to be inserted in the cube (corresponding to more than 40K tuples/sec from client devices), the cube build time is close to 6 minutes for Cluster B, and even higher for Cluster A. In other words, in this case, Kylin takes 6 minutes to update the cube according to the preprocessed statistics from a 5-minute window. This in turn creates a bottleneck and instability in the system, since Kafka keeps accumulating statistics from Flink at a higher rate that Kylin can consume them. The middle and bottom plot have similar results regarding the scalability. While the input data increases in size, the time needed to update the cube is also increased but in a sublinear manner. Regarding the time Kylin takes to consume the results from 3 5-minute windows, from the middle plot, we can observe that Cluster A suffers from instability when the client devices send more than 40K tuples/sec, whereas Cluster B suffers from instability when the device rate is 80K tuples/sec. In the bottom figure, which corresponds to the statistics in the last 30 minutes split in 5-minute slots, Kylin does not create a bottleneck using either Cluster A or Cluster B.

What is the impact of the above observations regarding the efficiency in supporting R2? The main answer is that we cannot rely on Kylin to retrieve the statistics of the penultimate 5-minute window. But to support real-time forecasts based on the already devised prediction models, such as the one in Eq. (1), Redis should store statistics from the two last 5-minute windows rather than the last one only. Otherwise, R2 cannot be met efficiently, or requires more computing resources than the ones employed in these experiments.

Unlike Kylin, Druid can continuously ingest streams and provides access to the latest data rows. To assess Druid's efficiency and compare against Kylin in a meaningful manner, we proceed to slight changes in the experimental setting. More specifically, we test Druid with exactly the same input rows that Kylin has been tested in the top plot of Figure 6. Also, Druid can have a different number of ingestion task slots with each one being on a different cluster machine. We experimented with the task slot number in order to detect the difference when choosing different levels of parallelism in each cluster. Figure 7 presents the results of the experiments. As expected the ingestion time increases as the input data size is increased in both clusters. But even for the bigger inputs, Druid can perform the ingestion before the statistics of the new 5-minute window become available in Kafka. In any case, Druid still needs Redis for efficiently supporting R1; otherwise the real-time traffic from the last 5-minutes would be available only after 2-3 minutes rather than a few seconds.

Another important remark is the difference in the ingestion time when the task slot number changes. In the homogeneous cluster, when the number of tasks increases, the ingestion time decreases. There are exceptions of this in the heterogeneous cluster. As the top plot shows, when the input stream is small in size, the difference between the task slots is negligible, whilst, in some

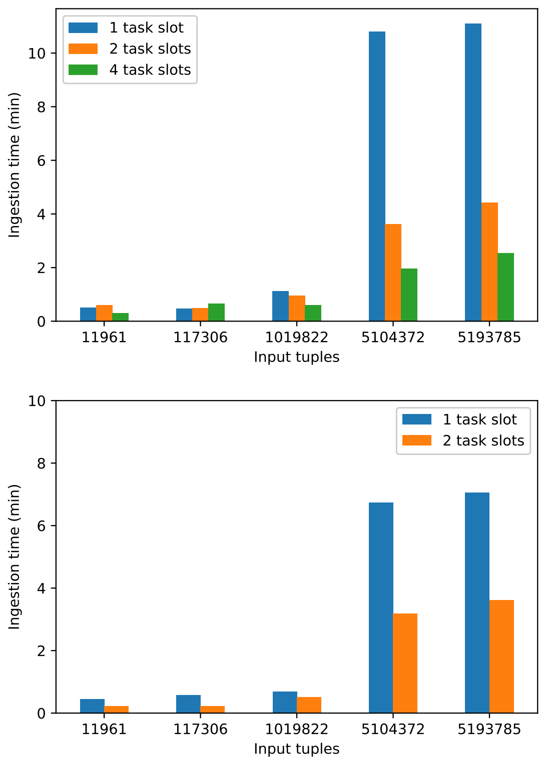


Figure 7: Druid's average ingestion time in minutes for cluster A (top) and B (bottom)

cases, when the task slots increase, the ingestion time increases as well. This is due to the fact that each machine is different and the size is small, which incurs communication and computation overheads that, along with imbalance, outweigh parallelism benefits. Also, when using 1 slot in Cluster A for high client device data rates, there is severe resource contention.

Finally, Druid is more robust to hardware heterogeneity compared to Kylin. When the task slots are equal to the number of machines in the heterogeneous cluster, the performance is better than in the homogeneous cluster. This means that in Druid, the total number of task slots and machines plays a bigger role during the ingestion than machine heterogeneity. On the contrary, Kylin does not seem capable of balancing the workload across heterogeneous nodes efficiently, therefore the homogeneous cluster exhibits better behavior than the heterogeneous one.

7.4. Storage requirements

According to the results presented above, Kylin cannot support well R2 and we need to resort to Redis. There is no such limitation for Druid. But this is the one side of the coin. Table 3 shows the storage capacity needed for the cube

Input Size	**Kylin**	**Druid**
11961	8.34	0.79
117306	78.59	7.85
1019822	511.15	62.25
5104372	1935.36	245.78
5193785	1966.08	249.44

Table 3: Kylin and Druid cube sizes in MBs

Rows	**Retrieval (sec)**	**Transformation (sec)**
1	0.012	3.3
10	0.013	3.35
100	0.016	3.39
1000	0.052	3.45
20000	0.78	4.27

Table 4: Querying times to Redis using Spark

itself for each warehouse depending on the input size. Kylin's cube is many times bigger than that of Druid. This stems from the fact that Kylin computes more aggregations and metrics depending on the user's preferences when the cube is created whilst Druid only stores the data (i.e., the base cuboid) and does not pre-compute rollups. This in turn has an effect when the user needs to query the warehouse in a complex manner. Complex queries for analytics and prediction model building is out of the scope of this work though. Kylin needs to compute small aggregations over the time hierarchy on the already computed metrics. On the other hand Druid takes more time in order to compute the metrics that are needed for the query.

7.5. Query experiments

In the following experiments, Spark is used as a standalone engine on a single machine outside the cluster where the warehouses and Redis are installed. Testing the scalability of Spark on more machines is out of our scope. Also, the warehouse contents refer to more than 1 year of data in an area consisting of 20K segments (overall more than 2 billions of entries).

Table 4 presents the results of the experiments when Spark pulls data from Redis. Because Redis returns data in JSON format, Spark needs to transform them into a dataframe in order to process them and return its results. The second column represents the time that Spark needed to fetch data from the cluster machine in seconds, whilst the third column displays the time needed to transform from JSON to a dataframe. The results show that fetching data is very fast and even if Spark is used in the front-end to create new maps ready to be asked by clients (R1), the whole process is ready a few seconds after the 5-minute window terminates. Also, fetching the results to update the predicted speed/travel times per segment (R2) every five minutes, takes only a few seconds.

Time Period Y	**No. Segments X**	**Retrieval (sec) (Spark-Druid)**	**Retrieval (sec) (Spark-Kylin)**	**Retrieval (sec) (Java-Kylin)**
1 year	**1**	6.31	4.10	0.97
	10	6.43	4.51	1.37
	100	6.57	8.07	5.02
1 month	**1**	6.25	4.12	0.92
	10	6.25	4.51	1.26
	100	6.33	7.41	4.31
1 week	**1**	6.52	4.07	0.93
	10	6.21	4.52	1.26
	100	6.67	7.41	4.29
1 day	**1**	6.24	4.16	0.93
	10	6.25	4.81	1.30
	100	6.27	7.44	4.31
1 hour	**1**	6.14	4.09	0.92
	10	6.12	4.61	1.27
	100	6.30	7.29	4.25
5 minutes	**1**	6.18	4.48	0.91
	10	6.55	4.40	1.28
	100	6.30	7.42	4.28

Table 5: Retrieval times for the *Aggregation Query*

Time Period Y	**No. Segments X**	**Returned Rows**	**Retrieval (sec) (Spark-Druid)**	**Retrieval (sec) (Spark-Kylin)**	**Retrieval (sec) (Java-Kylin)**
1 year	**1**	105120	8.00	4.91	1.34
	10	1051200	19.96	7.82	5.76
	100	10512000	147.63	-	-
1 month	**1**	8928	6.82	4.18	0.98
	10	89280	7.34	4.80	1.62
	100	892800	18.81	11.36	9.16
1 week	**1**	2016	6.44	4.17	0.96
	10	20160	6.50	4.58	1.40
	100	201600	9.02	8.40	5.25
1 day	**1**	288	6.29	4.11	0.95
	10	2880	6.02	4.48	1.33
	100	28800	6.75	7.71	4.42
1 hour	**1**	12	6.24	4.09	0.94
	10	120	6.21	4.42	1.29
	100	1200	6.39	7.54	4.32

Table 6: Retrieval times for the *Stress Query*

For the data warehouses, two different queries were used. The first one, called *Aggregation Query*, asks for the aggregated minimum speed of X road segments over a time period Y returning X rows of data. The second one, called *Stress Query*, asks for the speed of all of the rows of X road segments over a time period Y and may be used for more elaborate prediction models. The objective is to show that such queries take up to a few seconds and thus are appropriate to update segment information every 5 minutes; this is confirmed by our experiments even for the most intensive queries.

Table 5 presents the results for the *Aggregation Query*. Druid times seem constant regardless of the number of groups and the number of values that need to be aggregated. On the other hand, Kylin takes more time when the query needs to aggregate values over increased numbers of segments, while the aggregation cost does not seem to be increasing when the number of rows for each road segment increases due to a larger time window benefiting from pre-

Bytes per message	40	80
1 Broker - Mean	0.006329	0.014851
1 Broker - Min	0.003414	0.005066
1 Broker Max	0.120508	0.17830
2 Brokers - Mean	0.005516	0.0095731
2 Brokers - Min	0.000627	0.0036495
2 Brokers - Max	0.028637662	0.0382677

Table 7: Latency (in secs) during MQTT stress test with 20K connected clients

computations. Finally, the Java standalone program is significant faster for retrieval; however Spark can be easier parallelised and perform sophisticated computations after the retrieval to fulfill R2 and R3.

Table 6 presents the experiments for the *Stress Query.* The third column in this table represents the number of rows returned (no aggregation is performed). Regarding the *Stress Query*, the results are mixed. In most of the cases, Druid has the slowest retrieval times whilst the Java program has the fastest. Druid's retrieval performance is greatly affected by the number of rows that it returns. Kylin is also affected but less. A small remark in this experiment is that both Spark and the Java program crashed during the query that needed to return 10512000 rows due to Kylin's restrictions; retrieving such a high number of records should be executed in rounds. As both of these experiments reveal, Druid and Kylin behave differently in the query phase. Druid is more affected by the number of rows that it needs to return. Meanwhile, a simple Java program shows in both cases that is even better than Spark for the same job, if we are interested in retrieval only. This is due to the fact that Spark is a distributed framework and is not suitable for querying a small size of data.

7.6. End-to-end performance

Previously, we investigated the performance of individuals components in a manner that no end-to-end processing evaluation results are explicitly presented. However, in fact, the time taken by the streaming processing engine, which ouputs its temporary results into both Redis and Kafka, as shown in Figure 5, is totally hidden by the time taken to build the Kylin cube (see Figure 6) or ingest data into Druid (see Figure 7). The times to query Redis and the persistent storage for each window update (e.g., Table 4) are also fully hidden.

For completeness, we provide evidence that the latency due to the messaging component is negligible. Table 7 shows the results of a MQTT stress test with 20K connected clients generating 600K messages per minute, where each message is either 40 or 80 bytes long. As can be observed, the incurred overhead is negligible, well below 1 second therefore we can argue that our deployment is suitable for real-time applications.

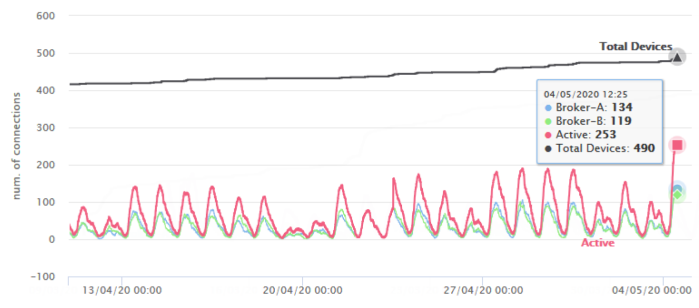


Figure 8: Devices connected during the pilot case study

8. Pilot Case Study

To test the solution, we deployed our clients on 800 taxis in the period of March-May 2020 in the area of Thessaloniki, Greece's second largest city. Unfortunately, this period largely overlapped with the lockdown due to COVID-19, and most of the taxis resumed their operation near beginning of May when the COVID-19 lockdown measures were first relaxed in Greece, as shown in Figure 8. In this figure, the messages are sent to two MQTT brokers and the solid black line shows the total devices activated at least once; the red and green plots show the active devices at a specific time point.

In Figure 9, we see an example of how the solution can identify congestion; in the example, the congestion was real and was due to roadworks; the system could successfully detect abnormal slow speeds and thus, the corresponding road segments are colored in red. To determine whether the speed is slow, it is compared to the free flow speed that is computed as described previously based on historical data.

In general, we have noticed that our solution could provide real-time statistics at a finer level of granularity compared to other existing solutions (see an example in Figure 10), which resulted in different estimated travel times; e.g. an example is shown in Figure 11, where the estimates between multiple providers differ. In this work, we do not examine whether the estimated travel times are more or less accurate than other, but we mention our observation that despite the relatively small number of sensing devices, we were able to cover the road network at the level of road segments.

Technically, in the deployed solution, we have chosen Druid over Kylin. A major additional problem we had to face regarded outlier detection. Outlier detection is incorporated along with the main Flink processing module to clean data from both faulty sensors and stopped vehicles that continue reporting speed. To this end, we resorted to Flink-based continuous distance-based outlier detection, based on the work in [22]. The key point is that, if in the same time window, a vehicle reports widely different speed data than other vehicles for the same segment, then such data should be discarded. To this end, we employed distance based outliers, which define a point as an outlier if it has less than

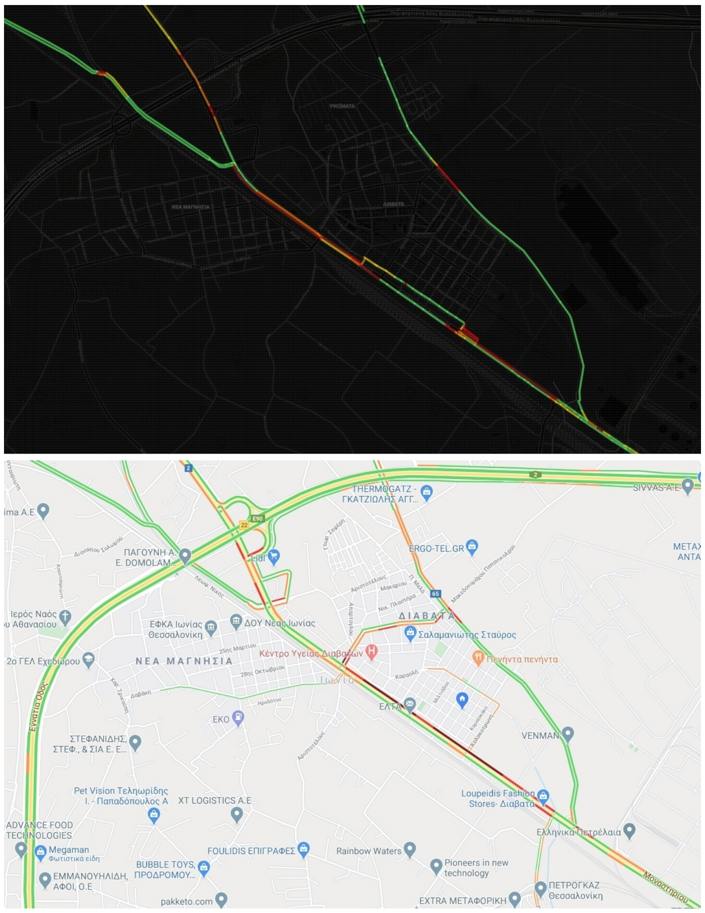


Figure 9: Example screenshot (bottom) during the pilot case study

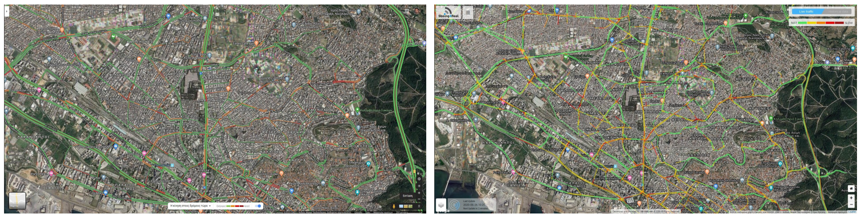


Figure 10: Example real-time statistics of Google (left) and our solution (right)

k neighbors with distance at most R. Distance-based outliers are suitable for a streaming case and are capable of pruning widely different measurements of some vehicles in the same time window in an unsupervised learning manner, but they are sensitive to their parameters; therefore, running streaming distance-

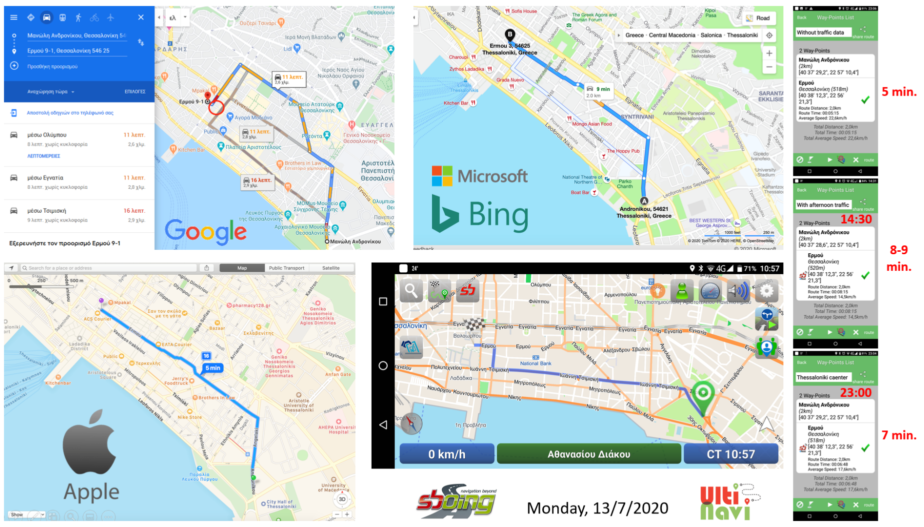
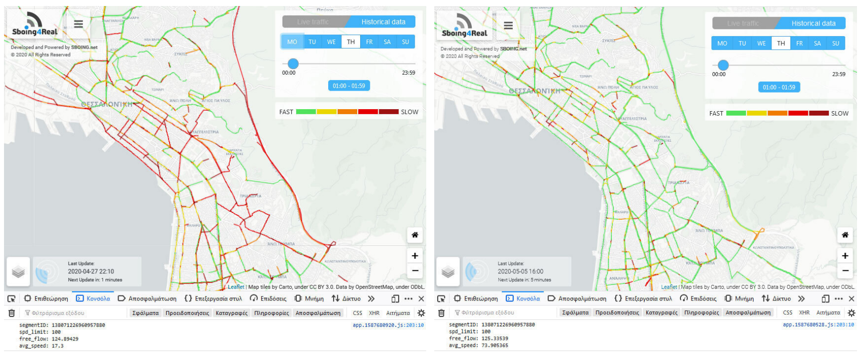


Figure 11: Example of different travel time estimates



```
segmentID: 138071226960957880
spd_limit: 100
free_flow: 124.89429
avg_speed: 17.3
```

```
segmentID: 138071226960957880
spd_limit: 100
free_flow: 125.33539
avg_speed: 73.905365
```

Figure 12: Example real-time statistics before applying outlier detection (left) and after (right)

based outlier detection with multiple parameters is advocated in [23]. This type of outlier detection can discard short-term stops of taxis, e.g., to drop a customer. However, it cannot deal with stopped taxis for longer period. Thus, expert knowledge needs to be incorporated in the form of simple rules, e.g., it is safer to include a rule establishing that when the reported speed is repeatedly close to 0 while other vehicles move, the recordings should be discarded rather than relying on outlier detection solely. Example benefits of applying outlier detection enhanced by rules are shown in Figure 12, where on the figure on the right, the enhancements are also due to the manner free flow estimated as discussed at the end of Section 6.

Finally, some deployment details are as follows. The complete application has been containerized with the help of 12 Docker images which correspond to MQTT, Flink, Redis, Kafka, HBase, Kylin, Druid, Spark, Zookeeper (needed by Kafka, Druid, Kylin), Hadoop (need by Druid, Kylin, HBase), Hive (needed by Kylin), PostgreSQL (needed by Druid) and our query processing programs, respectively. Then, we have employed Docker Swarm to install the containers to the cluster nodes.

9. Lessons Learned, Additional Issues and Conclusions

This work presented the back-end system in the context of the Sboing navigator to support real-time traffic reporting and forecasting. We presented a modular architecture and explained in detail all the main design choices so that our solution can be easily re-engineered by third parties. The main lessons learned can be summarized as follows: (1) To support our requirements, we need a scalable messaging component and two big-data processing platform instantiations, one for streaming data and one for batch analytics. In our system, MQTT-based solutions are adequate for messaging and we have chosen to employ Flink and Spark for streaming and batch processing, respectively. (2) We require three types of storage: a main-memory storage for quick access to recently produced results, a persistent data warehousing storage supporting aggregates at arbitrary granularity of grouping (e.g., per road, per weekday, per week, and so on), and a scalable key-value store. We have chosen Redis, Kylin or Druid, and HBase, respectively. (3) Kafka can act as an efficient interface between the stream processing and permanent storage. In addition, Flink is the main option for the stream processing platform. (4) Redis, used as a cache with advanced querying capabilities, is a key component to meet real-time constraints. Solely relying on back-end analytics platforms such as Kylin or Druid, can compromise real-time requirements. (5) Druid is more effective than Kylin regarding ingestion. However, this comes at the expense of less aggregates being pre-computed. (6) Using HBase for metadata not changing frequently and shared across multiple segments can reduce the cube size significantly; otherwise cube size may become an issue.

In this work, we have explicitly compared Kylin against Druid. However, Redis and HBase could have been replaced by other similar stores, even though we have not examined their alternatives in this work. More specifically, any fast main-memory database that can play the role of a caching mechanism can be employed instead of Redis; we expect different choices not to have a significant impact on the overall system behavior. Similarly, any other key-value store can replace HBase. A question may arise as to whether NewSQL solutions, especially geo-distribution-aware ones like CockroachDB [24] can be employed. However, such solutions focus on distributed transactions, which are not the main query processing use case in our scenarios. Moreover, they are not tailored to analytic queries and data cube materialization. By contrast, we need solutions for OLAP queries while rendering the results of online analysis available as soon as possible after every window slide. Such requirements are better met by combining a system like Redis with a scalable data warehousing solution, whereas the auxiliary key-value store helps in keeping the volume of the data cube low.

Developing a back-end system for real-time navigation systems involves several research issues. In our context, we have focused on three areas: outlier detection, quality assessment and geo-distributed analytics. We have discussed outlier detection in the previous section. Regarding data quality, again, outlier detection can be deemed as one technique needed for assessing and ensuring high data quality. Scalable data quality measurement and enforcing tools are in their infancy, with main current options including Apache Griffin[29] and Databricks DDQ[30]. The scalability of such tools has not been tested and a research challenge is to devise an efficient methodology to incorporate data quality tools in our architecture without compromising the scalability requirement. Finally, large-scale analytics over geographically distributed clusters is an emerging and challenging area [25], where massive parallelism has to be combined with awareness of data transmission across WANs. As reported in [26], such jobs in Alibaba may result in PBs of data being transmitted across multiple data centers. In our setting, given that each area is covered by a different cluster, analytics tasks that span multiple clusters raise the need for more efficient usage of bandwidth while keeping latency low. Our preliminary work on this area optimizes task placement so that to minimize data transfer without degrading performance compared to solutions that solely focus on response time minimization [27].

In summary, our work is on developing a back-end engine capable of supporting online applications that rely on both real-time sensor measurements and combinations with historical data. This gives rise to several requirements that can be addressed by a non-monolithic modular architecture, which encapsulates several platforms and data store types. We have shown how to efficiently integrate MQTT, Flink, Spark, Kafka, Kylin (or Druid), Hbase and Redis to yield a working and scalable solution. The lessons learned are explicitly summarized and are of value to third parties with similar system requirements for real-time applications that continuously ingest data and perform complex analysis.

Acknowledgements. This research has been co-financed by the European Union and Greek national funds through the Operational Program Competitiveness, Entrepreneurship and Innovation, under the call RESEARCH - CREATE - INNOVATE (project code:T1EDK-01944).

References

[1] J. Yu, J. Wu, M. Sarwat, Geospark: a cluster computing framework for processing large-scale spatial data, in: Proc. of the 23rd SIGSPATIAL, 2015, pp. 70:1–70:4.

[2] A. Aji, F. Wang, H. Vo, R. Lee, Q. Liu, X. Zhang, J. H. Saltz, Hadoop-

[29]`https://griffin.apache.org/`

[30]`https://github.com/databricks/drunken-data-quality-1`

gis: A high performance spatial data warehousing system over mapreduce, PVLDB 6 (11) (2013) 1009–1020.

[3] S. You, J. Zhang, L. Gruenwald, Large-scale spatial join query processing in cloud, in: 31st IEEE Int. Conf. on Data Engineering Workshops, 2015, pp. 34–41.

[4] V. Pandey, A. Kipf, T. Neumann, A. Kemper, How good are modern spatial analytics systems?, PVLDB 11 (11) (2018) 1661–1673.

[5] B. Guo, C. Chen, D. Zhang, Z. Yu, A. Chin, Mobile crowd sensing and computing: when participatory sensing meets participatory social media, IEEE Communications Magazine 54 (2) (2016) 131–137.

[6] T. Toliopoulos, N. Nikolaidis, A. Michailidou, A. Seitaridis, A. Gounaris, N. Bassiliades, A. Georgiadis, F. Liotopoulos, Developing a real-time traffic reporting and forecasting back-end system, in: Research Challenges in Information Science - 14th International Conference, RCIS 2020, Limassol, Cyprus, September 23-25, 2020, Proceedings, 2020, pp. 58–75.

[7] M. Kiran, P. Murphy, I. Monga, J. Dugan, S. S. Baveja, Lambda architecture for cost-effective batch and speed big data processing, in: 2015 IEEE International Conference on Big Data, Big Data, IEEE Computer Society, 2015, pp. 2785–2792.

[8] E. I. Vlahogianni, M. G. Karlaftis, J. C. Golias, Short-term traffic forecasting: Where we are and where we're going, Transportation Research Part C: Emerging Technologies 43 (2014) 3 – 19.

[9] N. Djuric, V. Radosavljevic, V. Coric, S. Vucetic, Travel speed forecasting by means of continuous conditional random fields, Transportation research record 2263 (1) (2011) 131–139.

[10] Y. Gao, S. Sun, D. Shi, Network-scale traffic modeling and forecasting with graphical lasso, in: Advances in Neural Networks – ISNN 2011, 2011, pp. 151–158.

[11] W. Qiao, A. Haghani, M. Hamedi, Short-term travel time prediction considering the effects of weather, Transportation Research Record: Journal of the Transportation Research Board 2308 (2012) 61–72.

[12] C.-S. Li, M.-C. Chen, Identifying important variables for predicting travel time of freeway with non-recurrent congestion with neural networks, Neural Computing and Applications 23 (6) (2013) 1611–1629.

[13] A. Essien, I. Petrounias, P. Sampaio, S. Sampaio, The impact of rainfall and temperature on peak and off-peak urban traffic, in: Database and Expert Systems Applications, 2018, pp. 399–407.

[14] H. Guo, J. Ren, D. Zhang, Y. Zhang, J. Hu, A scalable and manageable iot architecture based on transparent computing, Journal of Parallel and Distributed Computing 118 (2018) 5 – 13.

[15] D. Johnson, A. Menezes, S. Vanstone, The elliptic curve digital signature algorithm (ecdsa), International Journal of Information Security 1 (1) (2001) 36–63.

[16] A. Antonić, M. Marjanović, K. Pripužić, I. P. Žarko, A mobile crowd sensing ecosystem enabled by cupus: Cloud-based publish/subscribe middleware for the internet of things, Future Generation Computer Systems 56 (2016) 607 – 622.

[17] W. Feng, Z. Yan, H. Zhang, K. Zeng, Y. Xiao, Y. T. Hou, A survey on security, privacy, and trust in mobile crowdsourcing, IEEE Internet of Things Journal 5 (4) (2018) 2971–2992.

[18] V. Beltran, A. F. Skarmeta, Overview of device access control in the iot and its challenges, IEEE Communications Magazine 57 (1) (2019) 154–160.

[19] J. Han, M. Kamber, J. Pei, Data Mining: Concepts and Techniques, 3rd edition, Morgan Kaufmann, 2011.

[20] V. Harinarayan, A. Rajaraman, J. D. Ullman, Implementing data cubes efficiently, in: Proc. of the 1996 ACM SIGMOD, 1996, pp. 205–216.

[21] I. K. Kim, W. Wang, Y. Qi, M. Humphrey, Empirical evaluation of workload forecasting techniques for predictive cloud resource scaling, in: 9th IEEE Int. Conf. on Cloud Computing, CLOUD, 2016, pp. 1–10.

[22] T. Toliopoulos, A. Gounaris, K. Tsichlas, A. Papadopoulos, S. Sampaio, Continuous outlier mining of streaming data in flink, Inf. Syst. 93 (2020) 101569.

[23] T. Toliopoulos, A. Gounaris, Multi-parameter streaming outlier detection, in: IEEE/WIC/ACM Int. Conf. on Web Intelligence, 2019, pp. 208–216.

[24] R. Taft, I. Sharif, A. Matei, N. VanBenschoten, J. Lewis, T. Grieger, K. Niemi, A. Woods, A. Birzin, R. Poss, P. Bardea, A. Ranade, B. Darnell, B. Gruneir, J. Jaffray, L. Zhang, P. Mattis, Cockroachdb: The resilient geo-distributed SQL database, in: Proceedings of the 2020 International Conference on Management of Data, SIGMOD, 2020, pp. 1493–1509.

[25] S. Dolev, P. Florissi, E. Gudes, S. Sharma, I. Singer, A survey on geographically distributed big-data processing using mapreduce, IEEE Trans. Big Data 5 (1) (2019) 60–80.

[26] Y. Huang, Y. Shi, Z. Zhong, Y. Feng, J. Cheng, J. Li, H. Fan, C. Li, T. Guan, J. Zhou, Yugong: Geo-distributed data and job placement at scale, PVLDB 12 (12) (2019) 2155–2169.

[27] A. Michailidou, A. Gounaris, Bi-objective traffic optimization in geo-distributed data flows, Big Data Research 16 (2019) 36–48.