Developing a real-time traffic reporting and forecasting back-end system

Theodoros Toliopoulos[1], Nikodimos Nikolaidis[1], Anna-Valentini Michailidou[1], Andreas Seitaridis[1], Anastasios Gounaris[1], Nick Bassiliades[1], Apostolos Georgiadis[2], and Fotis Liotopoulos[2]

[1] Aristotle University of Thessaloniki, Thessaloniki, Greece
{tatoliop,nikniknik,annavalen,sgandreas,gounaria,nbassili}@csd.auth.gr
[2] Sboing, Thessaloniki, Greece
{tolis,liotop}@sboing.net

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 working 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.

1 Introduction

Geographical Information Systems and, more broadly, the development of applications based on or including geo-spatial data is a mature and hot area with several tools, both commercial and open-source, e.g., ArcGIS, PostGIS, GeoSpark [12] and so on.In general, these tools and frameworks are distinguished according to the queries they support [8] and the quality of maps they utilize. For the latter, popular alternatives include Google Maps and OpenStreetMap[3], which can be considered as data-as-a-service. At the same time, urban trips is a big source of data. 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, stores and processing platforms.

The aim of this work is to present architectural details regarding a novel back-end system developed on behalf of Sboing[4]. Sboing is an SME that implements innovative mobile technologies for the collection, processing and exploitation of location and mobility-based data. It offers an app, called UltiNavi, that can be

[3] https://www.openstreetmap.org
[4] www.sboing.net

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 with a view to (i) continuously receive feedback regarding traffic conditions from end users and process it on the fly; and (ii) provide real-time and accurate travel time forecasts 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 implies the need to store past information at several levels of granularity.

R4: scalability. Traffic forecasting can be inherently parallelised in a geo-distributed manner, i.e., each region to be served by a separate cluster of servers. Therefore, the challenge is not that much in the volume of data to be produced but in the velocity of new update streams to be produced by the system and the need to store historical data.

R5: fault-tolerance. Any modules to be included in the back-end need to be capable of tolerating failures.

There are several other tools that provide this type of information; e.g., Google Maps, Waze [5] and TomTom[6]. However none of these tools that are being developed by big companies have published information about their back-end processing engine. By contrast, we both explain architectural details and employ publicly available open-source tools, so that third parties can rebuild our solution with reasonable effort.

Background. 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 [11] 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

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

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

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.[1] analysed travel speed data from sensors capturing the volume and occupancy every 30 sec. and advocated combining multiple predictors Gao et al.[2] also used data from sensors but the main unit was vehicles per hour. 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. [9] 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. Li et al.[6] 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. Based on the above proposals, we also consider the presence of an incident, rain or snow, the visibility, the wind speed and the temperature.

It is important to note that forecasting can be more accurate when there are historical traffic data available [6]. The drawback is that this information can be very expensive to store due to its large size. Thus, suitable storage technologies must be used. In this work, we resort to a scalable data warehousing solution.

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 solution for supporting real-time traffic reporting and forecasting as far as the back-end system engine is concerned. The architecture consists of several modules and integrates different tools and data stores. (ii) It discusses several alternatives regarding design choices in a manner that lessons can be transferred to other similar settings. (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.

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; and (iii) the results presented are meaningful for software architects in different domains with similar requirements.

The remainder of this paper is structured as follows. Sec. 2 presents the overall architecture. In the next section, we discuss the pre-processing of source data. In Sec. 4 we describe the underlying data warehouse and the queries that run over it. Indicative experiments are in Sec. 5. We conclude in Sec. 6.

2 System Architecture

The raw data are transmitted from the users through user devices, such as GPS tracking devices and mobile navigation apps in a continuous stream. Every device periodically sends its current condition in a specific time interval; if there

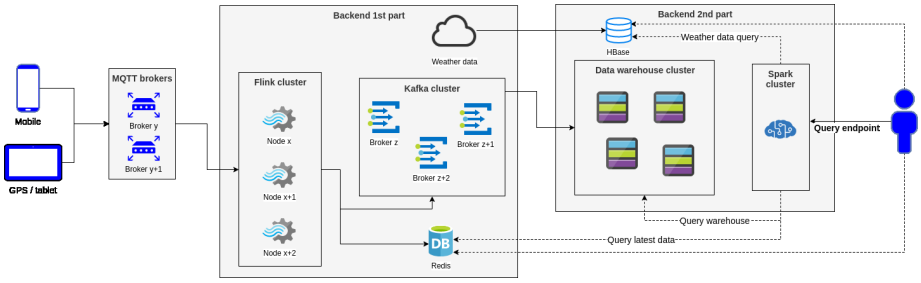

Fig. 1. The diagram of the advocated architecture

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. I.e., the device has 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.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. The connection between the user devices and the back-end pipeline is materialized through the use of the MQTTprotocol.

The main responsibility of the back-end pipeline is to receive the raw data, 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 framework. Flink is an open-source distributed continuous stream processing framework that provides real-time processing with fault-tole-rant 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. Kafka is the most popular open-source stream-

ing 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. 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 and the second is Apache Druid. 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 is the engine that is used for query processing as an alternative to standalone Java programs. Fig. 1 presents the complete back-end architecture. are reported by the monitoring devices.

3 The stream processing component

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 secs, still generate 80K new sets of measurements per sec, which amounts to approximately 7 billion measurements per day. 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. Loss of information is not acceptable; this implies that

Flink must dynamically connect to all new brokers without suspending data processing. Each Flink machine has a list of all available MQTT brokers along with their IP addresses. Each machine is responsible for one of those brokers in order to ingest its data. This implies that the solution needs to have at least the same number of Flink machines as the MQTT brokers. All of the Flink nodes that do not get matched with a broker remain available and keep checking the pool of brokers for updates. Note that all of the Flink nodes keep working on the data processing even if they do not get connected with a MQTT broker. When a new broker is inserted in the pool, one of the available nodes initiates the connection in order to start the ingestion. This process does not slow down or stop the job even if there are no available nodes. Flink can increase or decrease the number of its worker nodes without shutting down due to its built-in mechanisms.

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 plausible, 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 certain 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.

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. No more drtails are provided fue to space constraints.

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. 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.

The final module in this part of the pipeline is Redis. One of the data warehouse alternatives used in this work is Apache Kylin. Kylin does not have the capability to ingest a stream continuously and convert it into a cube but it needs to update the cube periodically (according to a user-defined time interval) with the new data of the stream. This means that by relying to a data warehouse, such as Kylin solely, R1 cannot be satisfied despite the fact that Flink can produce statistics for the most recent time window very efficiently. Redis solves this problem by saving the latest processed data from Flink. Druid does not have the limitations of Kylin, but still, imposes an unnecessary overhead to produce statistics almost immediately after the finish of each 5-minute window. 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.

Redis is a main-memory data structure store that can quickly save and retrieve data with a key-value format. It is fault-tolerant and can also scale up by adding more machines; i.e., it is suitable for meeting R1, R4 and R5. More specifically, each road segment forms a key and the statistics needed for real-time reporting (typically, mean speed) is stored as a value. Predicted travel times are stored in a similar manner. Remember that apart being in main-memory, the table size is inherently small, in the order of hundreds of MBs, even if a complete big country such as France is served. For this reason, the Redis table need not be parallelised.

4 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.

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) [3]. 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 cluster and goes beyond simple Hive, 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 [3]. To mitigate this limitation, Kylin encapsulates the HBase

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). There are two options in order to include them, either to add metadata conditions as dimensions or to consider them as another type of measure. Both options suffer from severe drawbacks. 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.

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 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 [4]). 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.

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 etc. by assigning a weight to each of the two types mentioned above. The model can also incorporate information for 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, 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., [5] 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:

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

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 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. Here, using the same setting as in Section 5, we give summary information about model building times for Eq. (1): a Spark job that retrieves the historical data of a specific segment from the last month, transforms the data 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 applies linear regression takes approximately 3 minutes. Different models for multiple segments can be computed in parallel at no expense on the running time. If the last 6 months are considered in training, the training takes 17 minutes. The coefficients are cached; Redis can be used to this end. Upon the completion of each 5-minute window, based on the precomputed co-efficients, predicted statistics are computed for each road segment for the next time window.

5 Performance Evaluation

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 different 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.

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

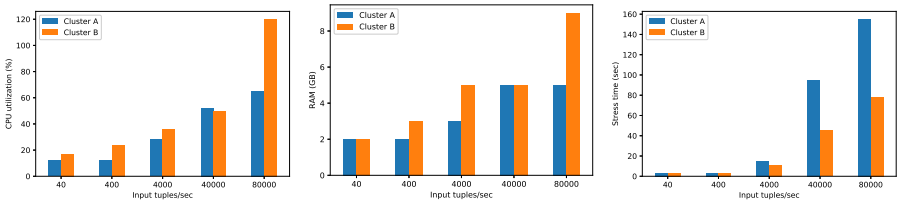

Fig. 2. Flink's average CPU utilization per YARN node (left), memory consumption (middle) and stress time (right) during stream data processing

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 secs simultaneously in a region larger than half of Greece. This yields a stream of 288 million MQTT messages per hour.

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

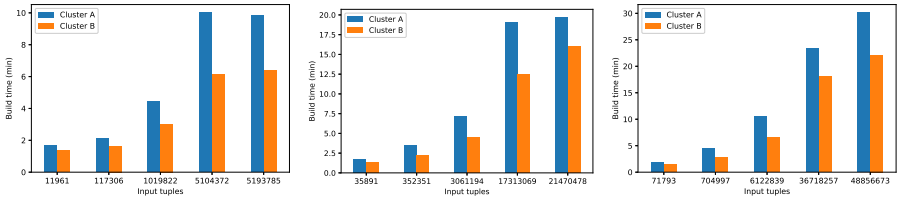

Fig. 3. Kylin's average cube build time after ingesting 1 (left), 3 (middle) and 6 (right) 5-minute windows

shared 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 2 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 left plot shows the average CPU utilization per YARN node for each cluster during the whole 5-minute tumbling window. Note that the 100% mark means that the job takes over a whole CPU thread. From this experiment, we can safely assume that Flink's CPU consumption scales up in a sublinear manner, but the current allocation of resources for Cluster B seems to suffer from resource contention for the highest workload. 120% utilization for Cluster B means that on average, 1.2 cores are fully utilized out of the 2 cores available, but the utilization increase compared to 40K tuples per second is 2.5X. In Cluster A, increasing the workload by three orders of magnitude results in a 5-fold increase in Flink demand for CPU resources. Cluster B exhibits lower utilization if we consider that each machine allocates two threads to Flink instead of one, up to 10000 records/sec per Flink node. Overall, the main conclusion is that Flink is lightweight and the example allocation of resources to Flink (a portion of the complete cluster capacity) is adequate to manage the workload.

The middle plot shows the average memory used by each machine during the 5-minute tumbling window. In the first cluster, even though the input tuples per second are increased 2 thousand-fold, the memory is increased by approximately 2.5 times only. Cluster B consumes up to 9GB of memory taking into account each machine has to process twice the amount of data compared to the physical machines in Cluster A. This further supports the conclusion that the homogeneous cluster exhibits better resource utilization than the heterogeneous, but both clusters can handle the workload.

Finally, the right 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 secs even for the highest workload. This means that after 80 secs, 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.

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.

Persistent storage experiments. Persistent storage is the key component of the whole architecture. 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. Fig. 3 shows the results for the two clusters employed. An initial observation is that the homoge-

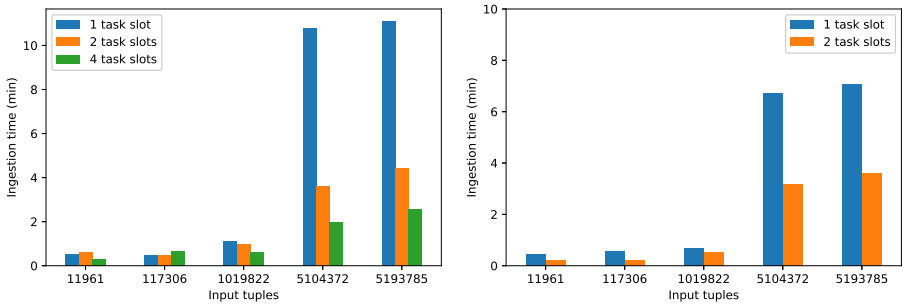

Fig. 4. Druid's average ingestion time in minutes for cluster A (left) and B (right)

neous cluster (Cluster B) consistently outperforms the heterogeneous one. This provides evidence that Kylin is sensitive to heterogeneity. The left plot from Figure 3 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 segment 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 right 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 suffer from instability when the device rate is 80K tuples/sec. In the right 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

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 2. Querying times to Redis using Spark

been tested in the left plot of Figure 3. 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 4 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 left plot shows, when the input stream is small in size, the difference between the task slots is negligible, whilst, in some 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.

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 2 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.

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.

No exact numbers are provided due to space constraints. In summary, 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-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.

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.

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 2(right), is totally hidden by the time taken to build the Kylin cube (see Figure 3) or ingest data into Druid (see Figure 4). The times to query Redis and the persistent storage for each window update are also fully hidden.

6 Lessons Learned and Conclusions

The main lessons learned can be summarized as follows: (1) To support our requirements, we need two big-data processing platform instantiations, one for streaming data and one for batch analytics, that should not interfere with each other in order not to compromise real-time requirements. In our system, we have chosen to employ Flink and Spark, respectively, instead of two instances of either Flink or Spark. (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.

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. No details are presented due to lack of space, but the relevant publications include [10,7].

Conclusions. Our work is on developing a back-end engine capable of supporting online applications that rely on both real-time sensor measurement 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 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.

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. Djuric, N., Radosavljevic, V., Coric, V., Vucetic, S.: Travel speed forecasting by means of continuous conditional random fields (2011)
2. Gao, Y., Sun, S., Shi, D.: Network-scale traffic modeling and forecasting with graphical lasso. In: Advances in Neural Networks – ISNN 2011. pp. 151–158 (2011)
3. Han, J., Kamber, M., Pei, J.: Data Mining: Concepts and Techniques, 3rd edition. Morgan Kaufmann (2011)
4. Harinarayan, V., Rajaraman, A., Ullman, J.D.: Implementing data cubes efficiently. In: Proc. of the 1996 ACM SIGMOD. pp. 205–216 (1996)
5. Kim, I.K., Wang, W., Qi, Y., Humphrey, M.: Empirical evaluation of workload forecasting techniques for predictive cloud resource scaling. In: 9th IEEE Int. Conf. on Cloud Computing, CLOUD. pp. 1–10 (2016)
6. Li, C.S., Chen, M.C.: Identifying important variables for predicting travel time of freeway with non-recurrent congestion with neural networks. Neural Computing and Applications **23** (11 2013)
7. Michailidou, A., Gounaris, A.: Bi-objective traffic optimization in geo-distributed data flows. Big Data Research **16**, 36–48 (2019)
8. Pandey, V., Kipf, A., Neumann, T., Kemper, A.: How good are modern spatial analytics systems? PVLDB **11**(11), 1661–1673 (2018)
9. Qiao, W., Haghani, A., Hamedi, M.: Short-term travel time prediction considering the effects of weather. Transportation Research Record: Journal of the Transportation Research Board **2308**, 61–72 (12 2012)
10. Toliopoulos, T., Gounaris, A., Tsichlas, K., Papadopoulos, A., Sampaio, S.: Parallel continuous outlier mining in streaming data. In: 5th IEEE International Conference on Data Science and Advanced Analytics, DSAA. pp. 227–236 (2018)
11. Vlahogianni, E.I., Karlaftis, M.G., Golias, J.C.: Short-term traffic forecasting: Where we are and where we're going. Transportation Research Part C: Emerging Technologies **43**, 3 – 19 (2014)
12. Yu, J., Wu, J., Sarwat, M.: Geospark: a cluster computing framework for processing large-scale spatial data. In: Proc. of the 23rd SIGSPATIAL. pp. 70:1–70:4 (2015)