Journal of Universal Computer Science, vol. 28, no. 1 (2022), 3-26
submitted: 15/7/2021, accepted: 6/12/2021, appeared: 28/1/2022

A Novel Real-Time Edge-Cloud Big Data Management and Analytics Framework for Smart Cities

Roberto Cavicchioli
(University of Modena and Reggio Emilia, Modena, Italy
https://orcid.org/0000-0003-0166-0898, roberto.cavicchioli@unimore.it)

Riccardo Martoglia
(University of Modena and Reggio Emilia, Modena, Italy
https://orcid.org/0000-0003-4643-6128, riccardo.martoglia@unimore.it)

Micaela Verucchi
(University of Modena and Reggio Emilia, Modena, Italy
https://orcid.org/0000-0003-3898-8571, micaela.verucchi@unimore.it)

Abstract: Exposing city information to dynamic, distributed, powerful, scalable, and user-friendly big data systems is expected to enable the implementation of a wide range of new opportunities; however, the size, heterogeneity and geographical dispersion of data often makes it difficult to combine, analyze and consume them in a single system. In the context of the H2020 CLASS project, we describe an innovative framework aiming to facilitate the design of advanced big-data analytics workflows. The proposal covers the whole compute continuum, from edge to cloud, and relies on a well-organized distributed infrastructure exploiting: a) edge solutions with advanced computer vision technologies enabling the real-time generation of "rich" data from a vast array of sensor types; b) cloud data management techniques offering efficient storage, real-time querying and updating of the high-frequency incoming data at different granularity levels. We specifically focus on obstacle detection and tracking for edge processing, and consider a traffic density monitoring application, with hierarchical data aggregation features for cloud processing; the discussed techniques will constitute the groundwork enabling many further services. The tests are performed on the real use-case of the Modena Automotive Smart Area (MASA).

Keywords: smart city framework, big data management, edge computing, cloud data management
Categories: I.4.8, C.3, H.2, H.4
DOI: 10.3897/jucs.71645

1 Introduction

In the context of smart cities, the use of combined data-in-motion and data-at-rest analysis provides efficient methods to exploit the massive amount of data generated from heterogeneous and geographically distributed sources including pedestrians, traffic, (autonomous) connected vehicles, city infrastructures, buildings, IoT devices, etc. Certainly, exposing city information to a dynamic, distributed, powerful, scalable, and user-friendly big data system is expected to enable the implementation of a wide range of new services and opportunities provided by analytics tools. Indeed, the potential of exploiting big data technologies to improve and define new smart city services has been shown, at least at the conceptual level, in several recent research works [Neilson et al., 2019, Honarvar

and Sami, 2019, De Gennaro et al., 2016, Kim et al., 2020]. However, there exist several challenges, not only related to the size and heterogeneity of data but also to its geographical dispersion, making it difficult to be properly and efficiently combined, analyzed, and consumed by a single system.

The CLASS project [cla, 2020], funded by the European Union's Horizon 2020 Programme[1] and in which context the research presented in this paper is performed, faces these challenges and proposes a novel software framework that aims to facilitate the design of advanced big-data analytics workflows, incorporating data-in-motion and data-at-rest analytics methods for efficiently collecting, storing and processing vast amounts of geographically-distributed data sources.

Differently from many other proposals, selectively focusing on either edge computing or cloud computing techniques, the CLASS software stack covers the whole compute continuum, from edge to cloud, and relies on a well-organized distributed infrastructure. The University of Modena and Reggio Emilia is one of the partners of the CLASS consortium, in charge of the edge software stack, with edges being both pole-mounted cameras and smart (or connected) vehicles.

In this paper we describe the software foundations of the innovative framework, involving different interdisciplinary aspects of scientific research, from cloud data management and analysis to high-performance edge computing and machine learning, first of all detailing its overall architecture and then focusing on its two main parts:

- *edge computing* solutions exploiting advanced computer vision technologies and enabling the real-time generation of "rich" data at the sensor level, supporting a vast array of sensor types including fixed smart cameras and smart vehicles equipped with cameras, GPSs, LiDARs and radars;
- *cloud data management* techniques offering efficient storage of the high-frequency incoming data at different granularity levels, together with its real-time querying and updating.

In our discussion we specifically analyze the issue of obstacle detection and tracking for edge processing, and consider a traffic density monitoring application, with hierarchical data aggregation features for cloud processing; the discussed techniques will also constitute the groundwork enabling many further services, including traffic prediction, smart parking, air pollution estimation, and so on. The adopted real use-case is the Modena Automotive Smart Area (MASA), a 1 Km^2 area in the city of Modena (Italy), equipped with our sensing, communication, and computation infrastructure, on whose data we also report on the efficiency side with a number of tests.

To the best of our knowledge, this is the first time a software framework of this kind tackles the capabilities of powerful edge devices instead of simpler IoT sensors and processes the metadata in the cloud with real-time constraints. This frees cloud resources and allows to provide more enriched services that can be easily implemented from the gathered metadata.

The rest of the paper is organized as follows: after discussing related works (Section 2), we give an overview of the framework (Section 3) and then discuss in detail the edge- (Section 4) and cloud-processing (Section 5) techniques. Finally, we report on the performed tests (Section 6) and conclude the paper (Section 7).

[1] H2020 Grant Agreement: 780622

2 Related works

The framework presented in this paper involves different interdisciplinary aspects of scientific research, from cloud data management and analysis to high-performance edge computing and machine learning, all applied to a smart city context. In current best practices, we typically find these elements in separate research works or in different application scenarios. We start this related work analysis by considering (cloud) data management (Section 2.1) and IoT / edge computing proposals (Section 2.2) in similar smart city scenarios; then, we will continue our analysis from a technological point of view considering the state of the art of big data programming models (Section 2.3) and machine learning (Section 2.4).

2.1 (Big) Data management proposals in a smart city context

The potential of exploiting and managing Big Data to improve and define new smart city services has been shown, at least at the conceptual level, in several recent research works [Neilson et al., 2019, Honarvar and Sami, 2019, De Gennaro et al., 2016, Kim et al., 2020]. For instance, the survey [Neilson et al., 2019] discusses several proposals in the transportation systems domain, which is one of the most demanding and common scenarios in a smart city setting. One example is the work [Serrano et al., 2016], where the authors present a study of how cloud-computing and big-data management technologies can assist decision-making in this context. While there are several works showing the benefits of big data information extraction / analysis, in many cases the focus is mainly on the specific application and on the analysis of the possible benefits rather than on presenting actual data management solutions / architectures: one example is the paper [Honarvar and Sami, 2019], demonstrating very promising tests on an air pollution prediction scenario, where however the data collection and management aspect is not explicitly presented.

Among the works focusing on cloud data management / analytics, some of them propose architectures for offline-only data analysis, while others are devoted to pure real-time management. For instance, the work [De Gennaro et al., 2016] is an example of the first type of works, a promising data processing platform enabling the analysis of large datasets of GPS data for policy assessment; the data used for analysis are previously collected and the focus is not on their real-time storage and updating. Instead, the work [Kim et al., 2020] is a proposal of a management system working on real-time data in an energy saving scenario: in such kind of works the accent is on the managament of (very) recent data in real-time, while past data are typically not of interest. This is in line with what presented in pioneering works demonstrating the use of Data Stream Management Systems (DSMSs) in specific real-world contexts, including a real-time route planner in Lucerne [Özal et al., 2011] or a real-time traffic information management system in Stockholm [Biem et al., 2010]. In these systems, data is stored in the main memory only, and it is kept there as long as it is needed to solve the continuous requests that are currently in execution; then, data flows out of the system. This is not in line with the context envisioned in this paper and in the CLASS project for the development of a complex smart city framework, where real-time data must be retained beyond their real-time processing to offer extended knowledge for various service purposes.

One proposal offering data processing features both for real-time and historical data management is [Carafoli et al., 2016], which was based on previous data management experiences in actual smart city scenarios [Carafoli et al., 2012, Carafoli et al., 2013];

however, such work is not based on modern bigData/noSQL technologies and its completely centralized architecture makes it not adequate to efficiently support the wide range of services needed in a smart city.

From a cloud/edge computing perspective, we notice that a consistent part of the works in the smart city scenario focus primarily on one of these two complementary aspects. This is true for instance for the above discussed works, which do not present specific solutions in the IoT / edge computing context integrated with the proposed frameworks. Conversely, there are also many works solely focusing on IoT / edge computing aspects (they will be analyzed in the next section).

Instead, the framework proposed in this paper provides a combination of: a) edge computing solutions exploiting advanced computer vision technologies and enabling the real-time generation of "rich" data at the sensor level; b) cloud data management techniques offering efficient storage of incoming data at different granularity levels together with its real-time querying and updating. In this way, the framework enables cloud data management services which are both easy to implement, powerful (aiming at real-time processing of incoming data but also at full past data storage and availability) and with promising scalability performances, especially w.r.t. other architectures not exploiting cloud/edge hybrid computation.

2.2 IoT applications in a smart city context

In the context of smart cities, there are several works that consider the use of Internet of Things (IoT) sensors, together with edge computing techniques, to monitor traffic in the city areas. The work of Muthanna et al.[Muthanna et al., 2020], Akhter et al. [Akhter et al., 2019] and Barthelemy et al. [Barthélemy et al., 2019] are certainly relevant to this paper.

Muthanna et al.[Muthanna et al., 2020] propose a method of recognizing pedestrians in real-time on edge devices using a Raspberry Pi microcomputer to automate the process of signaling traffic lights and improve system mobility. The experiment was conducted to detect pedestrians in an image with the subsequent determination of their location at a pedestrian crossing. Akhter et al. [Akhter et al., 2019] designed and developed an IoT enabled intelligent sensor node for smart city applications. The fabricated sensor nodes count the number of pedestrians, their direction of travel along with some ambient parameters, such as temperature, humidity, pressure, Carbon di Oxide (CO2), and the total volatile organic component (TVOC). The monitored data are uploaded to the Internet server through the Long Range Wide Area Network (LoRaWAN) communication system. In the infrastructure, there are a total of 74 sensor nodes that have been installed around Macquarie University and continued working for several months. Barthelemy et al. [Barthélemy et al., 2019] proposed a sensor-based traffic monitoring system in the Australian city of Liverpool, with the following requirements: (i) the sensors need to be able to detect and track pedestrians, vehicles, and cyclists: (ii) the system should be privacy compliant; (iii) it should leverage existing infrastructures; (iv) scalability and interoperability of the sensors should be accounted for. The developed prototype has two core components: (i) an NVIDIA Jetson TX2 and (ii) a Pycom LoPy 4 module handling the LoRaWAN communications. The sensor performs iteratively 4 steps on average 20 times each second: (i) frame acquisition from an IP camera or a USB webcam; (ii) detecting the objects of interests in the frame via YOLOv3 [Redmon and Farhadi, 2018]; (iii) tracking the objects by matching the detections with the ones in the previous frame using SORT [Bewley et al., 2016]; (iv) updating the trajectories of objects already stored in the device database or creating records for the newly detected objects. In parallel to

those tasks, the sensor periodically transmits the results from the video processing to the IoT Core either via LoRaWAN or Ethernet.

Besides covering the aspects addressed by these other works, our proposed solution fits into a highly developed modular architecture designed to collect and analyze the big data produced by sensors. Moreover, in our case the sensors can also be very heterogeneous. Another important aspect to consider is that, in the majority of the presented literature, IoT is seen as a pervasive set of smart sensors, but our implementation is focused on the edge devices, that are more computationally capable and extract complex metadata from the signals that they process.

2.3 Big Data management and programming models

MapReduce [Dean and Ghemawat, 2008] has been one of the reference frameworks for writing data-centric parallel applications since the beginning of the big data revolution. In the most recent years, a large number of next-gen data management proposals have been proposed to revolutionize data management, offering extensible architectures able to cope with different kinds of data, including streams and tabular data, and offering newer and easier kinds of programming interfaces. Such proposals are aimed to provide complete solutions to real-time processing and batch processing needs, providing high levels of flexibility in single data management solutions: these include big data processing engines such as Apache Spark [Zaharia et al., 2016, spa, 2021], big data stores such as Apache HBase [hba, 2021], stream processing frameworks such as Apache Flink [Apa, 2021, Carbone et al., 2015] and Apache Samza [sam, 2021].

In all these next-generation platforms, functional programming is adopted as the emerging paradigm [Wu et al., 2017]: functional interfaces facilitate programmers to write data applications in declarative ways and the computation is treated as a calculation of functions. Among the next-gen data management solutions, Spark is one of the most powerful and largely adopted solutions, and is also the one considered in this paper for cloud data management. Spark provides a set of operators that can be called by the applications, which internally are optimized to be executed in distributed environments through the Spark runtime. Spark is an extension and generalization of the MapReduce paradigm and its popular open-source implementation Apache Hadoop [White, 2012]. The Spark data model is based on finite Resilient Distributed Datasets (RDDs) and is extended towards streaming (infinite) data by means of the SparkStreaming [Zaharia et al., 2013] extension. Moreover, further extensions (SparkSQL [Armbrust et al., 2015] in primis) allow to extend the data and querying model towards the relational world, including for instance aggregates and joins.

2.4 Machine Learning

There exist multiple libraries or environments for machine learning which are very popular: Tensorflow [Abadi et al., 2016], PyTorch [Paszke et al., 2017], to mention a few. Most of these environments usually provide a Python interface and are easy to use. Some of these environments support some type of parallelism, like Caffe that runs on Graphic Processing Units (GPUs), or Tensorflow, which supports data and model parallelism. Also based in Python, Keras [Gulli and Pal, 2017] is a high-level neural networks API that was designed to offer a simple and intuitive interface for the users and to enable fast experimentation. MLlib [Meng et al., 2016] is Spark's machine learning library, and like Spark is based on RDD data structure. MLlib is composed of several algorithms for

classification, regression, collaborative filtering, clustering, and decomposition. Another popular machine library in Python is Scikit-learn [Pedregosa et al., 2011] which provides simple and efficient tools for data mining and data analysis. The library is built on top of the optimized Python libraries NumPy, SciPy, and matplotlib and it is open source under BSD license. However, only coarse-grain parallelism is supported.

For our purpose, all the previous methods were not compliant with the real-time constraint imposed by the use cases of the CLASS project, therefore we deployed our machine learning data analytics tool exploiting CUDA [NVIDIA et al., 2020], cuDNN [NVIDIA, 2021] and TensorRT [Vanholder, 2016]. In particular, for the neural networks we employed tkDNN [Verucchi et al., 2020a, Verucchi et al., 2020b] a open-source Deep Neural Network library built with cuDNN and TensorRT primitives, specifically thought to work on NVIDIA Jetson Boards, whose main goal is to exploit those boards as much as possible to obtain the best inference performance. These tools allowed us not only to exploit GPU for neural network inference, but also to tune the performance of the network to be faster, even with less precision, on the edge devices that are deployed in the area.

3 Framework overview

This section describes the overall big-data analytics workflow we present responsible for managing the different heterogeneous data sources and processing the data incoming from them, on whose knowledge the different applications of the CLASS use cases are based.

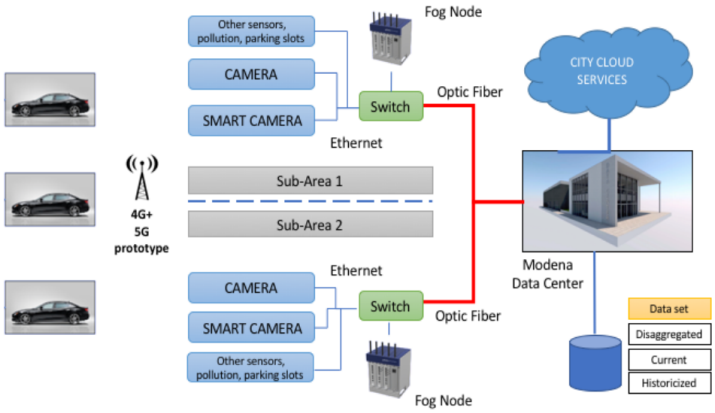


Figure 1: The proposed framework architecture, including data sources (cameras, sensors, vehicles), computing (edge, fog and cloud) and communication (antennas, fiber) infrastructures.

Figure 1 depicts a schematic overview of the underlying physical architecture, composed of several data sources (traditional and smart cameras, pollution and parking sensors, connected vehicles), a computing infrastructure (edge/fog nodes and the city cloud) and a communication infrastructure (4G antenna, optical fiber network).

We will now describe the whole combined big-data analytics workflow (shown in Figure 2), to let the reader have a complete overview of the proposed framework. Nevertheless, in the remainder of the paper, we will focus on the implementation of a fundamental subset of these analytics tasks (the numbers of the list of are used to identify the tasks, in bold the ones that we focus on):

1. *Sensor fusion*, responsible for object identification and computation of its position based on the raw data collected from sensors included in the vehicles;
2. ***Object detection***, responsible for object identification and computation of its position based on raw data from the cameras of the city;
3. ***Object tracking***, tracks the movement of objects identified with task 2, across multiple video frames;
4. *Data deduplication*, identifies and removes objects identified with task 3 by multiple cameras of the city;
5. *Trajectory prediction*, predicts the trajectory of vehicles and cyclists identified with task 3 based on previous tracked positions;
6. *Pollution computation*, estimates emissions from the drive cycles of vehicles identified in task 3 based on the emission class (e.g., Euro 1, Euro 2, Euro 3, etc) and vehicle speed over time;
7. ***Data aggregation***, responsible for aggregating all the information generated in the previous analytics tasks into the common database;
8. *Collision prediction*, identifies potential collisions, based on the intersection of the trajectory predictions of two objects (computed by task 5);
9. *Warning Area (WA) generation*, obtains all the information included in the database surrounding a connected vehicle;
10. *Alert visualization*, visualizes into the connected vehicle the following information: (i) the potential collision of the connected vehicle with other vehicles within the WA, and (ii) the available parking slots in the database;
11. *Parking counters*, computes the available parking slots based on cameras from the city;
12. ***Dashboard visualization***, responsible for visualizing the information included in the database;
13. *Predictive models*, implements the digital traffic signs application on the MatSim simulator.

The combined big-data analytics workflow is shown as a direct acyclic graph (DAG) in which nodes represent the analytics tasks described above (the number in the node identifies the task) and edges represent the data exchange (and so dependencies) among the different analytics. The workflow includes several data-sources; concretely, three city cameras and one connected vehicle are considered as input sources, and two connected vehicles and the city control room are receiving the information. The figure also identifies where data analytics tasks can be executed, i.e., in the edge/fog, in the cloud, or in both locations, aiming to efficiently distribute the workload across the compute continuum.

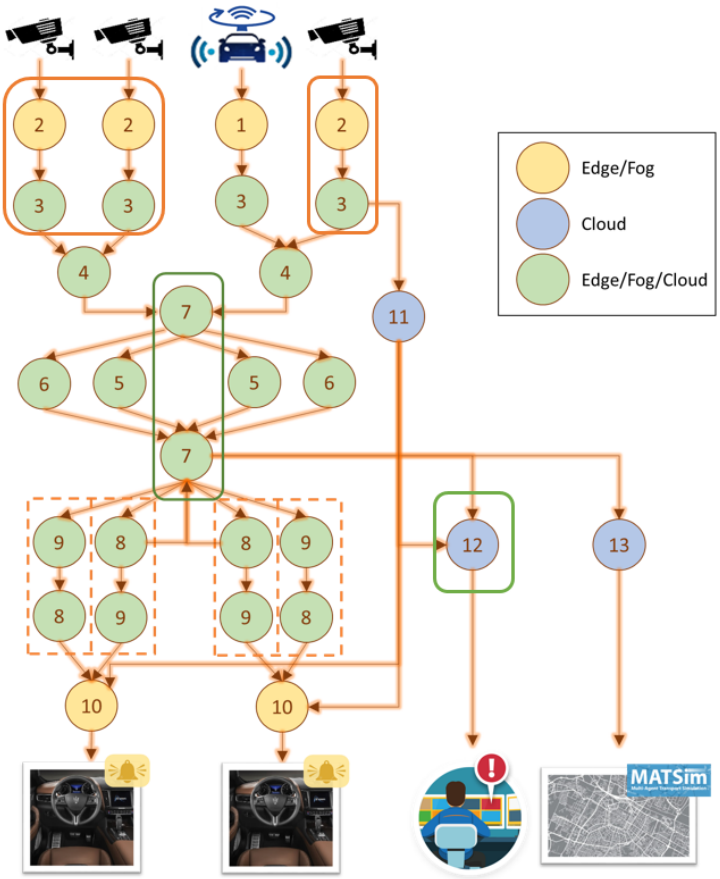


Figure 2: CLASS combined big-data analytics workflow when considering multiple data-sources. The non-dashed orange boxes are highlighting the contribution of Section 4 while the green ones the contribution in Section 5

4 Edge processing: object detection and tracking

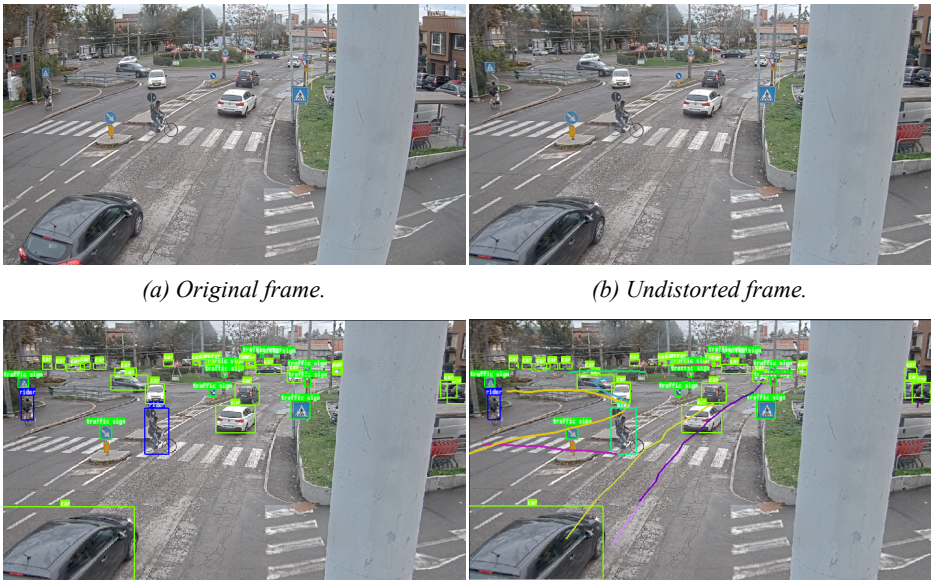
From cameras located in the streets, objects can be detected, classified, and tracked. All the cameras belonging to the infrastructure are supposed to perform those tasks in real-time and send the extrapolated information to a data aggregator. This aggregator will then de-duplicate repeated information and send messages to the connected cars, which will receive only objects that are relevant to their surroundings.

Geo-localization. To perform the above operations, it is necessary to know the real-world position of each object detected from the cameras. To do so, an extrinsic calibration of each camera in the MASA has been performed, in order to have a mapping for each pixel in the camera frame to its GPS position on a geo-referenced map.

Real-time requirements. Data is *constantly* being produced and processed and it is extremely important to guarantee that the results are meaningful by the time they

are computed. This is especially relevant for the *Obstacle Detection and Tracking* use case since alerts must raise within a time interval that is useful for the driver to react. A reasonable metric, considered in the scope of the CLASS project, is to get updated results at a rate between 10 and 100 milliseconds. Assuming that the maximum speed of a vehicle within the city is 60 km/h, vehicles will advance between 0.17 and 1.7 meters. This level of granularity is enough to implement the proposed use-cases.

4.1 Flow of the application



(a) Original frame. *(b) Undistorted frame.*

(c) Object detection. *(d) Object tracking.*

Figure 3: Results on various steps of the application: starting from the original frame (a), undistortion is applied (b), then objects are detected (c) and tracked (d).

We have implemented an application called `class-edge` and we have released the code open source[2].

`class-edge` takes in input $N\_streams$ streams and for each of them perform four main steps:

- First, frames are retrieved via the Real-Time Streaming Protocol (RTSP). Given that the image is taken from a camera, it is very likely that it is affected by distortion. To correct that, undistortion, based on the intrinsic calibration of the camera, is applied. Figure 3a shows an original frame while Figure 3b shows the output of undistortion.
- Object detection is then performed on the undistorted image. For this projects we picked the tkDNN implementation of Yolov4 [Bochkovskiy et al., 2020]. This task

[2] https://github.com/mive93/class-edge

is divided into three parts, i.e., (i) pre-processing to convert the image in the NN input, (ii) NN inference, (iii) post-processing to convert the output of a NN into bounding boxes (BBs). An example is given by Figure 3c.

– The detection gives in output a list of BBs. For each of them, a single point is picked to represent the whole object, namely the center of the bottom side of the BB. This pixel is converted first, into a GPS position, and then in meters. This is the format required from the next step: the tracker. Indeed, to track and predict the position of the detected objects an Extended Kalman Filter (EKF) [Kalman, 1960] on the real-world position of the object has been applied. Objects between frames are matched together only if the class corresponds and their distance is under a user-defined threshold. Tracking is not only used to have a more robust detection, but also to have a history of the objects. The idea of history is given by the lines in Figure 3d.

– Finally the information can be sent both to the data aggregator, in an anonymized form, and to the optional viewer.

4.2 Implementation details

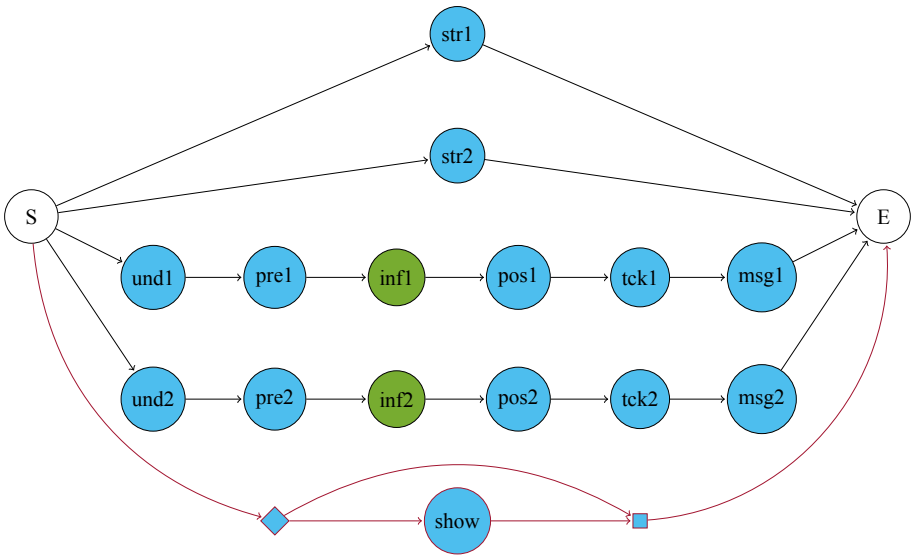


Figure 4: class-edge represented as a HC-DAG composed of 5 threads (top to bottom): retrieve stream 1 and 2, process stream 1 and 2, visualize (optional).

The whole application is reported as a directed acyclic graph in Figure 4. It is actually composed of 5 threads: 2 threads, one for stream, that retrieve the stream ad store the image (*str1|2*); 2 threads that execute the flow of undistortion (*und1|2*), detection (*pre1|2*, *inf1|2*, *pos1|2*), tracking (*tck1|2*) and message sending (*msg1|2*); and an optional thread for visualization purposes (*show*).

The reason behind this split is to improve the end-to-end latency of the application, which should not exceed $100ms$. Indeed, the first bottleneck of this application is to retrieve and read the frame from the stream, especially because the minimum resolution of the considered stream in our infrastructure is HD (1920x1080). The second bottleneck is the complete undistort-detect-track-send flow. Having these two operations in sequence can lead to work on very old data, while separating them improves the performance of the system (see also the experimental evaluation in Section 6).

We want to stress that the real-time requirement for the use case depicted here are not obtainable with extra-low power computing platform such as typical IoT devices, since the need to elaborate the video streams requires both computer vision and neural network computation capabilities.

5 Cloud data management: data aggregation and querying

Cloud data management enables the real-time processing of incoming data to allow its fast querying and visualization. The novel hybrid cloud/edge architecture of the CLASS framework provides the foundation to design and implement a data management layer which has a number of peculiarities w.r.t. typical (big) data management proposals in the smart city context:

- the ingested data go beyond raw data which is typically available from standard sensors, since they have already been elaborated in the edge in real-time: in this way, data processing can seamlessly take in input data coming from heterogeneous sources, enriched with high-level information (e.g., type of detected objects, coordinate of objects, etc.);
- starting from the above privileged standpoint and exploiting big data technologies, the framework enables cloud data management services which are both easy to implement and powerful. In particular, in this section we discuss a traffic monitoring service enabling its density computation and visualization at different granularities and for different vehicle types (which are not usually considered variables in similar systems);
- the aim is at real-time processing of incoming data but also at full past data storage and availability, differently from many proposals focusing either on offline-only [De Gennaro et al., 2016] or real-time recent-data-only [Kim et al., 2020, Biem et al., 2010] data analysis;
- scalability performances are promising, especially w.r.t. other architectures not exploiting cloud/edge (in Section 6.2 we show that processing an hour of data is done in 15 minutes on a very standard PC hardware).

5.1 General overview

On average (see Figure 5), every hour, the cameras and cars in the MASA produce 15 million records. Incoming data is in the form shown in Figure 6, where vehicle_type denotes the type of the identified object. Typical values are 1 for bicycles, 5 for buses, 6 for cars, 13 for motorbikes, and 14 for pedestrians, following the COCO dataset [Chen et al., 2015] indexes for detected objects.

Data processing is mainly performed in Spark and its goals are the following:

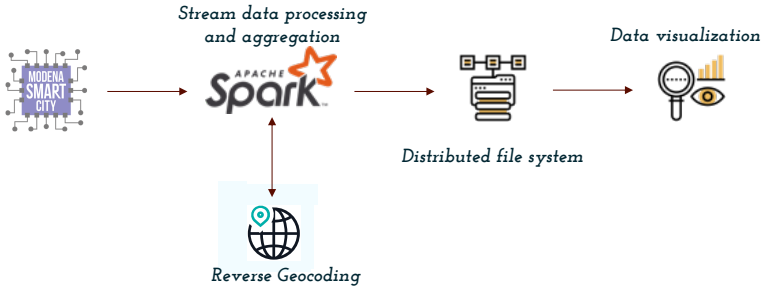


Figure 5: Data processing flow overview: incoming data from the MASA infrastructure (left) are processed, aggregated and stored in Spark (center), then data visualization is applied (right).

timestamp	vehicle_type	yaw	speed	latitude	longitude
1542626100013	6	170.00	13.00	44.656635284	10.934630394
1542626100013	6	0.00	0.00	44.686763763	10.959928513
1542626100013	6	254.00	2.00	44.655197144	10.934746742

Figure 6: Example of input data: from left to right, time stamp, vehicle type ('6' stands for car), yaw, speed and coordinates of each detected object.

1. Store the full raw data stream, as received in real-time, on a suitable data structure; this allows us to always have the full data available for future analyses but also to immediately process it to enable subsequent tasks;
2. Always in real-time and as a parallel operation, perform reverse geocoding so to spatially aggregate the input data w.r.t. route sections;
3. Perform time aggregation on the reverse geocoded data so to obtain data at different time granularities: 1 minute, 15 minutes, 1 hour and 1 day.

The final goal is to obtain, for each section of each route, the density (counts) of the different types of vehicles to the various time aggregates; these will be efficiently retrieved and passed to the data viewer to allow users to visually explore traffic history across time, focusing on the desired area, time granularity and vehicle type(s). We released the code for data aggregation and data visualization open source[3] [4]. The next sections will give more details on the whole process: reverse geocoding (Section 5.2), stream data processing and aggregation (Section 5.3) and visualization (Section 5.4).

5.2 Reverse geocoding

Reverse geocoding is the process of converting a coordinate pair to an address. There are many APIs that allow you to do this, both Online Web APIs (such as Nominatim, Google API, Yahoo, and OpenStreetMap) and Offine APIs (such as PostGis). The problem with

[3] https://github.com/fBarbanti/class-real-time-aggregator.git
[4] https://github.com/fBarbanti/class-traffic-monitor.git

both types of APIs is that they are slow and unsuitable for large amounts of incoming data. Moreover, standard reverse geocoders typically work by returning the name of the city (and state) where the input coordinates lay. This is obviously not feasible for our setting: in particular, we have to: a) operate with a large amount of incoming data; b) have a finer-grain answer identifying the specific routes (and, inside a given route, the specific sections) where the input points are located (spatial aggregation of the data). Therefore, it was necessary to implement an ad-hoc reverse geocoder. First of all, we created all the required map data, then we built the actual reverse geocoder to work on it.
Map data creation. We started from the geographic data available from OpenStreetMap (OSM), a collaborative project aimed at creating free-content world maps through the contribution of a community of users from all over the world. The OSM data model is represented by elements, which consist of:

- *Nodes*: they represent specific points defined by latitude and longitude. They are necessary to define a path;
- *Paths*: they are an ordered connection of at least 2 and a maximum of 2000 nodes describing a linear feature (such as a road).

Each element can have several properties, called Tags, which consist of key-value pairs, and serve to specify the characteristics of the elements. For instance, a "highway" key is used to identify highways. Other examples of tags are "lanes" (number of lanes), "bridge" (boolean, true if the path that contains the point "passes under" a bridge), "lit" (boolean, true if the path is illuminated). The data about the needed region has been exported and manipulated using QGIS, a well-known open-source geographic information system. In particular:

- we exported all routes having a "highway" key;
- we defined the section length and subdivided routes into 50 meters long sections. Through a QGIS feature called "Points along geometry", it was possible to divide each line into equidistant points, and show the points on the map (see Figure 7);
- the newly created points inherit all the attributes of the line element on which they were created: this means that each point in a street has the same id as the street itself. To uniquely identify each point in a street, a "section_id " attribute has been added that contains a unique integer for each route, so each point is identified by the pair (route_id, section_id);
- by means of the "Add Attributes of Geometry" tool, we added latitude and longitude to the created points.

Ad-hoc offline reverse geocoder. We implemented our reverse geocoder in Python and SciPy library. It uses a K-D tree to perform efficient processing of the input and return the nearest neighbor point available in the map, which is the closest point to the input coordinates. The K-D tree is a multi-dimensional binary search tree in which each node indexes a point in k dimensions. Each inner node can be thought to generate a hyperplane that divides the search space into two parts, called semi spaces. The data structure of the K-D tree is based on a recursive division of space into rectangular regions called boxes. As splitting rule, we chose sliding midpoint, where the splitting dimension is the longest size of the current box, and the splitting value is the midpoint of this side of the box.

Figure 7: Portion of created map, with equidistant points delimiting route sections (section length set at 50 meters).

property	value
route_name	'South Canaletto National Road'
highway	secondary
lanes	1
bridge	0
lit	0
route_id	w622677478
section_id	19

Figure 8: Example of reverse geocoding output for input point coordinates (44.655410767, 10.934341431). The output corresponds to section 19 of The South Canaletto National Road, where the input point is located.

Search in the K-D tree exploits the properties of the tree to quickly prune large portions of search space. In this way, the obtained reverse geocoder can cope well with the large amount of incoming data (see also the experimental tests Section 6.2). An example of input/output is shown in Figure 8.

5.3 Stream data processing and aggregation

Due to the high frequency and volume of incoming data, we chose to perform data stream real-time processing, aggregation and storing employing the Big Data platform Spark. Spark has become popular and widely used in industry due to its properties of flexibility, speed, and ease of use. With respect to other popular frameworks such as Hadoop, by choosing Spark we maintain tolerance and scalability characteristics and, through its advanced in-memory processing and stream processing features, we target very high levels of performance.

Full input data storing. We leveraged the structured streaming API to store the whole incoming data stream within a Streaming Dataframe, so to facilitate later processing.

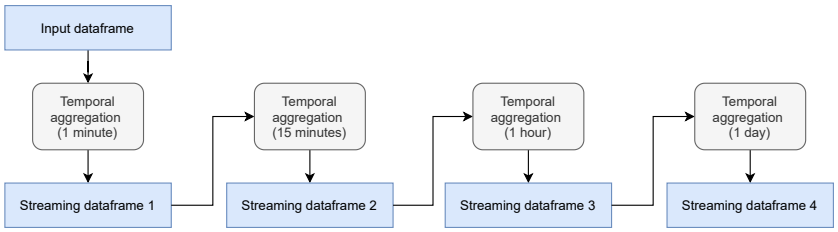


Figure 9: Schema of hierarchical aggregation processing: from left to right, 1 minute, 15 minutes, 1 hour and 1 day aggregations.

Structured Streaming is a scalable and fault-tolerant stream processing engine built on Spark SQL; we chose it since it allows real-time manipulation of the incoming data stream using the optimizations introduced with Spark SQL, guaranteeing fault tolerance and data consistency and the ability to query tables as if they were static data (Spark itself is responsible for updating the results table in the presence of new data). More specifically, the data is stored in a "Parquet" file format, which is an open-source and columnar binary file format introduced into the Hadoop ecosystem. This choice enables less memory consumption than row-based files (such as CSV and JSON) and increased performance for queries that process large volumes of data.

(Hierarchical) data aggregation. Achieving hierarchical aggregation requires to efficiently generate and store the flow of aggregated data in the 4 desired time granularities. To do this, we designed a hierarchical aggregation pipeline, which allows us to calculate a new aggregate starting from a finer granularity one: first, the aggregate with the finest granularity (1 minute) is generated, then this is used to generate the next coarser granularity aggregate (15 minutes), and so on. This required solving some technical challenges due to the current limitations of the Spark Streaming implementation: as reported in Spark's official documentation, multiple streaming aggregations (i.e., a chain of aggregations on a streaming DF) are currently not supported on a single dataframe. To overcome this, we slightly modified the conceptual pipeline: after the computation of each new granularity, the newly obtained results are dynamically stored in a new streaming dataframe, which is then read to perform the subsequent aggregation, and so on (see Figure 9).

Figure 10 shows an example of input data (top part) and its processing after reverse geocoding / spatial aggregation (center part) and temporal aggregation for granularity one minute (bottom). In particular, to give a glimpse of the Spark implementation behind the first step (spatial aggregation), this is the code that is run in order to compute the "count" and "average_speef" aggregate attributes after having grouped the data of the starting dataframe "df" on "timestamp", "route_name" and "section_id":

```
query = df
     .withWatermark("timestamp", "2 minutes")
     .groupBy("timestamp", "route_name", "section_id")
     .agg(F.count("*").alias("count"),
        F.avg("speed").alias("average_speed"))
```

After setting a two-minute watermark, the records with the same timestamp, street name and section of the street are grouped together. Next, the aggregate columns are created and populated with the number of grouped records and the average of the objects'

Input: original streaming DataFrame

timestamp	vehicle_type	yaw	speed	route_ name	highway	lanes	bridge	lit	route_id	section_id
1542626100013	5	70.00	40	South Canaletto National Road	secondary	1	0	0	w622677478	20
1542626100013	6	70.00	32	South Canaletto National Road	secondary	1	0	0	w622677478	20
1542626100013	6	70.00	34	South Canaletto National Road	secondary	1	0	0	w622677478	19
1542626100044	6	70.00	30	South Canaletto National Road	secondary	1	0	0	w622677478	19
1542626100044	6	70.00	32	South Canaletto National Road	secondary	1	0	0	w622677478	19

Output after spatial aggregation

timestamp	vehicle_type	route_ name	section_id	count	average_speed
1542626100013	5	South Canaletto National Road	20	1	40
1542626100013	6	South Canaletto National Road	19	1	34
1542626100013	6	South Canaletto National Road	20	1	32
1542626100044	6	South Canaletto National Road	19	2	31

Output after temporal aggregation:

Window	vehicle_type	route_name	section_id	average_count	average_speed
Mon Nov 19 2018 12:15:00 - 12:16:00	6	South Canaletto National Road	19	1.5	32.5
Mon Nov 19 2018 12:15:00 - 12:16:00	6	South Canaletto National Road	20	1	32
Mon Nov 19 2018 12:15:00 - 12:16:00	5	South Canaletto National Road	20	1	40

Figure 10: Example of data aggregation processing: starting from the input (top), spatial (center) and temporal (bottom) aggregations are applied.

speeds. Watermarking is used in order to allow the system to remove intermediate states from memory after the specified period of time, thus optimizing performance.

5.4 Data visualization

The goal of this phase is to implement a module for the visualization of aggregates obtained in the previous phases. In fact, we wanted to give the user the possibility to monitor traffic to the various time aggregates, with the possibility of differentiating traffic according to the type of vehicle. The viewer is implemented in Python using the PySpark (for connection to the Spark data node) and Folium (for displaying geospatial data on an interactive map) libraries.

The interface of the current preliminary prototype of the viewer is shown in Figure 11 After the user has selected the filter options (date, reference object type, and time granularity), the map is displayed, and the traffic situation is shown. The user is also able, by scrolling through the provided slider, to monitor traffic during specific hours of the selected day (also based on the selected granularity).

The visualization is created by reading the previously aggregated data available in Spark (see the previous section). The first operation is to read the map data containing information about the routes and their sections. Next, the interactive map is created: the Spark context is created for reading the parquet files, the data is read and the traffic in each section is represented as colored dots. The colors of the dots are determined based on the density of objects in the section. In particular, the traffic evaluation criterion, that determines when/how it is flowing, derives directly from the traffic flow data made available by the Emilia-Romagna Region:

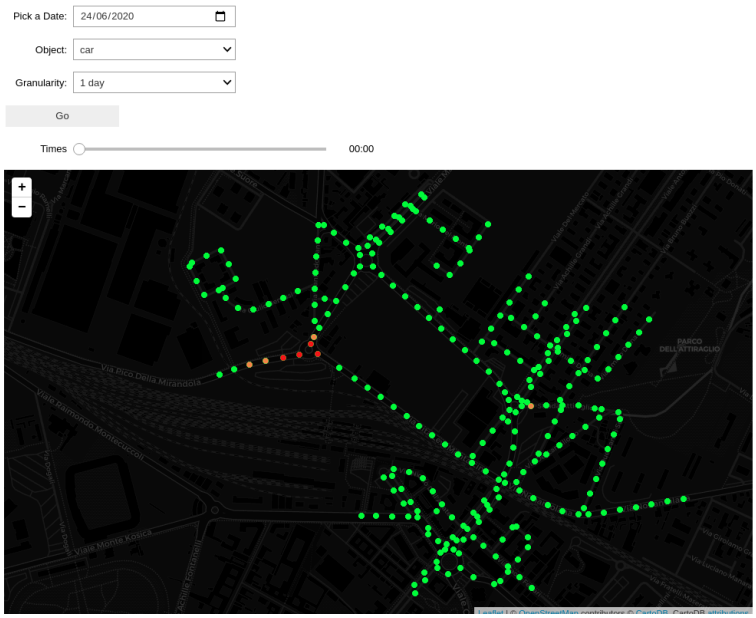


Figure 11: Traffic Viewer interface, including map window (bottom) and parameter selectors (top).

- $<$ 30: traffic is free flowing (Green);
- $>=$ 30 and $<=$ 50: traffic is free flowing with some slowdown (Orange);
- $>$ 50: traffic is heavy (Red).

6 Experimental evaluation

In this section we report on the experiments we performed for both edge (Section 6.1) and cloud processing (Section 6.2).

6.1 Edge processing experiments

The experiments on the proposed edge processing techniques (Section 4) have been carried on an Intel i9-9900KF (@3.60GHz) coupled with an NVIDIA RTX 2080Ti. Two streams were considered: (i) stream 1 with an image resolution of 1920x1080 and rate of 25 FPS, (ii) stream 2 with image resolution 3072x1728 and rate of 30 FPS. A better idea of the timing of the five threads and their data exchange is depicted in Figure 12.

Figure 13 reports the minimum, average and maximum execution times of subtask *str1|2* (Figure 4) over 5k frames, split in three further phases: (i)frame acquisition, form the RTSP stream, (ii) frame resize to a smaller resolution (i.e., 960x540) and frame copy to a shared buffer. From the chart, two main observations can be made. The first

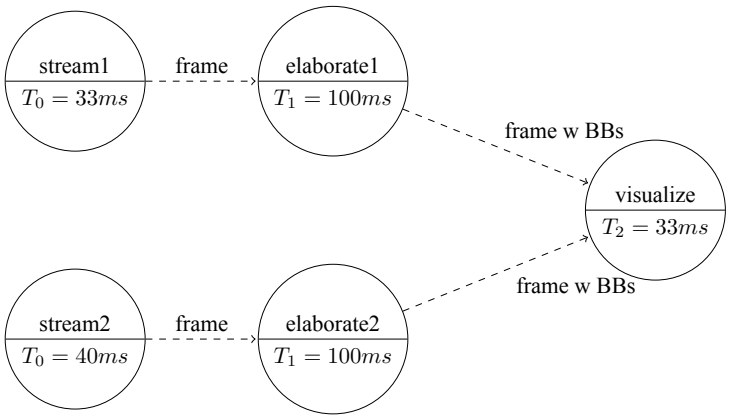


Figure 12: `class-edge` *timing and data exchange details of the threads of class-edge. Elaborate stands for the undistort-detect-track-send flow.*

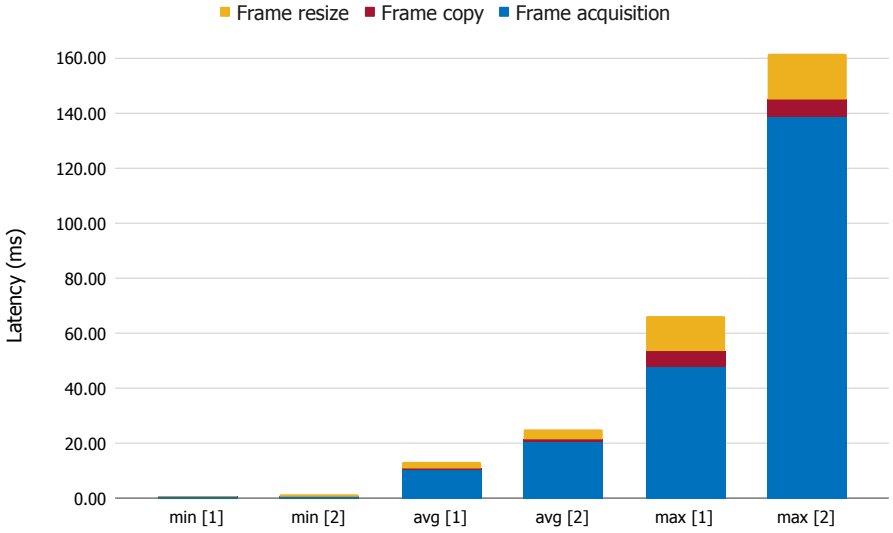


Figure 13: Minimum, average and maximum latencies of subtask str1|2 of Figure 4, split in three further phases: frame acquisition, frame resize and frame copy.

is that the most expensive part is the capture of the stream, which is affected by the original resolution and it's higher for stream 2. The second is that, even if in average *str1|2* take between 15 and 20 milliseconds to execute, the same operation can take up to 160 milliseconds. Given these results, we decided to set 1920x1080 as an upper bound resolution for the RTSP stream, changing the setting directly on the cameras, so that the maximum execution is always less than 100 milliseconds.

A similar chart for the undistort-detect-track-send flow is reported in Figure 14. Additionally to the already described phases, the copy of the frame from the shared buffer and the optional viewer feeding have been profiled. From this chart, we can evince

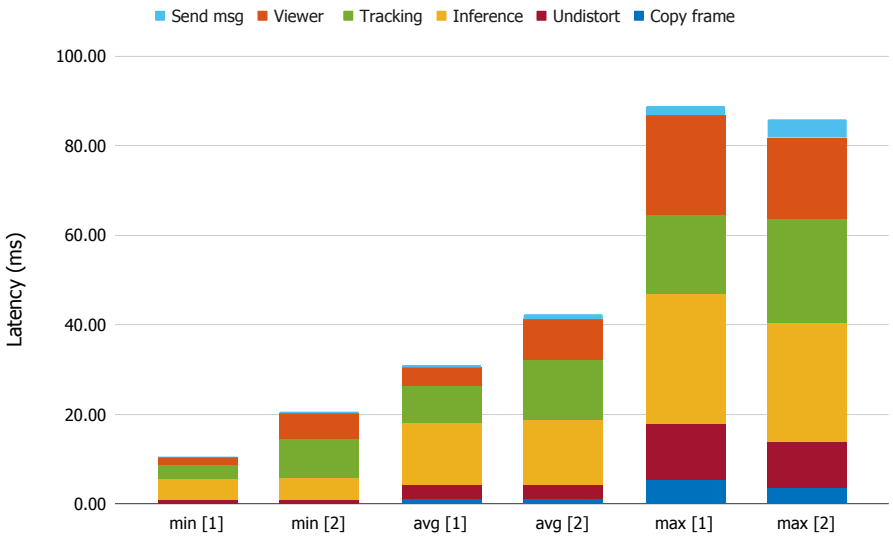


Figure 14: Minimum, average and maximum latencies of undistort-detect-track-send flow.

that still there are some differences in the streams, not related to the resolution but on the scene itself instead. Stream 1 comes from a camera that points on a roundabout: it's a dynamic scenario and even though there are many objects, the trackers don't last long. On the other hand, stream 2 comes from a camera that points to a parking lot: it's a static view, there are many objects and their trackers are always alive, keeping their information (with a limited history). For this reason, differences in terms of latency can be found in the tracking and viewer phases, which are proportional to the stored trajectories, while the other phases have similar duration. In any case, the total time of the flow is always less than 100 milliseconds.

6.2 Cloud processing tests

In this section, we report on the efficiency tests we performed on the cloud processing part of our prototype. In particular, we tested the efficiency of the reverse geocoder and the overall cloud data processing. The tests were executed on a standard PC system having an AMD Ryzen 5 3500U CPU and 8 GB RAM.

Reverse geocoder efficiency test. The ad-hoc reverse geocoder's performance has been evaluated to have a preliminary idea of its impact on the overall performance. In this test, we first randomly generated 100 points with coordinates belonging to the MASA; then, we issued 100 queries and monitored the response times. This whole process has been repeated for 10 runs. The average results are the following: the ad-hoc geocoder was able to complete the test in 9.79 ms, showing the satisfying performances that will be needed for real-time processing of the data. To give a comparative idea of a standard online reverse geocoder performance (Nominatim), we performed the same test on it. The performance of Nominatim was of 50.007 seconds, therefore more than 5000 times the one of our offline implementation.

Table 1: Application performance: for each processing phase (left), required percentage of CPU, memory and execution time.

Processing phase	CPU (%)	Memory (MB)	Time
Spatial aggr & rev. geocoding	78.4	1430.20	8 min 12 sec
1 min granul temporal aggr	47.2	985.51	40 sec
15 min granul temporal aggr	41.6	780.69	32 sec
1 hour granul temporal aggr	43.4	528.59	21 sec
Complete processing	73.6	1901.26	14 min 29 sec

Overall cloud data processing test. In this test we evaluated the overall cloud data processing performance in terms of memory consumption and CPU and processing time. We considered a data stream containing the input data of one hour of typical traffic and monitored the memory and CPU consumption of all the pipeline operations, including individual time aggregations. Table 1 summarizes the performance results, while Figure 15 presents a graph of the entire processing CPU and memory usage.

As expected, the initial processing phase, responsible for reverse geocoding the incoming data stream and calculating vehicle density for each moment of time (spatial aggregation), is the most demanding. Most of the resources used by the application are required at this aggregation stage; then, in the next phases (temporal aggregation), processing, memory, and CPU time decrease significantly. At the conclusion of this analysis, we see that the prototype can quickly process the incoming data stream, completing one hour of typical input processing in less than 15 minutes of processing (with 70% of CPU and about 2GB of memory consumption). This shows that thanks to the optimized data managing techniques, even a very standard PC hardware is more than adequate for full real-time processing of the generated data.

7 Conclusions

In this paper we described the software foundations of our innovative framework, detailing the overall architecture and then focusing on its two main parts for edge computing and cloud data management. We considered the real use-case of the Modena Automotive Smart Area (MASA) and specifically analyzed the issues of obstacle detection and tracking and a traffic density monitoring application with hierarchical data aggregation features, also discussing the efficiency aspect with a number of tests. Differently from many state of the art proposals, which focus primarily on one of the two complementary issues of edge and cloud computing, the presented framework exploits both in order to manage a variety of heterogeneous sensors and offer data processing features for both real-time and historical data management. The discussed techniques and architecture is devised so to constitute the groundwork enabling many further services, from smart parking to air pollution estimation. Nevertheless, the proposed architecture has some limitations. The innovation of installing smart edge devices capable of processing videos in real-time and covering a vast area comes with a considerable economical effort. The described area covers only a couple of square kilometers in Modena, and extending it would be a fair expense for the municipality. On the other hand, the cost of the cloud

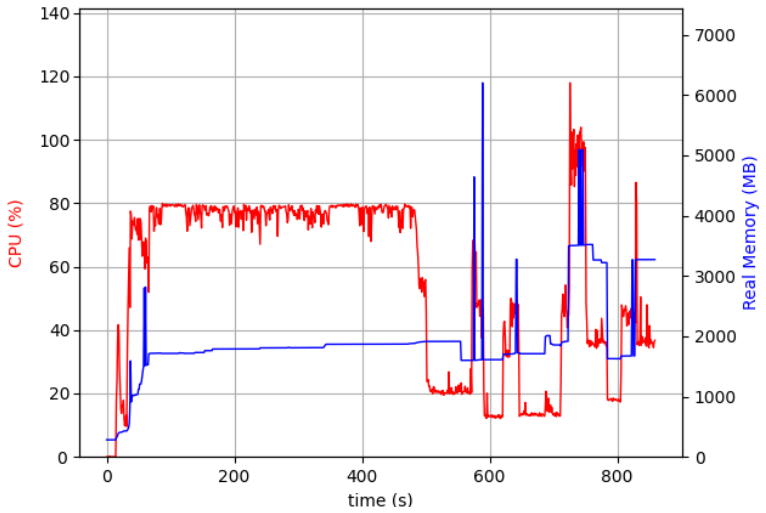


Figure 15: Overall cloud data processing CPU (red) and memory (blue) performance test

infrastructure can be reduced, since only the metadata has to be processed with no need of elaborating the full video streams, therefore requiring less computational capability on the server side.

In future work, we plan to focus on supporting and testing further innovative services, also devising and integrating machine learning techniques for enabling real-time forecast in different smart city contexts (e.g., traffic density as in preliminary work [Martoglia and Savoia, 2021], free parkings, etc.). The data analytics methods could be easily upgraded to exploit state of the art approaches to the different tasks, such as Bloom filters for data deduplication, segmentation networks for better road user detection, visual feature trackers for enhanced reliability and so on. Another line of research would be applying the digital twin multilevel approach to the system, to simulate at different levels of granularity the digital copy of a vehicle, a crossroad or a road block in order to test different scenarios and the possible behaviour of the road users when facing issues like accidents, unexpected traffic conditions or closed roads.

Acknowledgements

The authors would like to thank Francesco Barbanti, who contributed to the development of a preliminary version of the cloud data management techniques during his bachelor internship.

References

[cla, 2020] (2020). Edge and Cloud Computation: A Highly Distributed Software for Big Data Analytics (CLASS). https://class-project.eu/. [Online; accessed October 2020].

[Apa, 2021] (2021). Apache Flink. http://flink.apache.org.

[hba, 2021] (2021). Apache HBase. http://hbase.apache.org.

[sam, 2021] (2021). Apache Samza. http://samza.apache.org.

[spa, 2021] (2021). Apache Spark. http://spark.apache.org.

[Abadi et al., 2016] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. (2016). Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pages 265–283.

[Akhter et al., 2019] Akhter, F., Khadivizand, S., Siddiquei, H. R., Alahi, M. E. E., and Mukhopadhyay, S. (2019). Iot enabled intelligent sensor node for smart city: pedestrian counting and ambient monitoring. *Sensors*, 19(15):3374.

[Armbrust et al., 2015] Armbrust, M., Xin, R. S., Lian, C., Huai, Y., Liu, D., Bradley, J. K., Meng, X., Kaftan, T., Franklin, M. J., Ghodsi, A., and Zaharia, M. (2015). Spark sql: Relational data processing in spark. In *Proc. of ACM SIGMOD*, pages 1383–1394. ACM.

[Barthélemy et al., 2019] Barthélemy, J., Verstaevel, N., Forehead, H., and Perez, P. (2019). Edge-computing video analytics for real-time traffic monitoring in a smart city. *Sensors*, 19(9):2048.

[Bewley et al., 2016] Bewley, A., Ge, Z., Ott, L., Ramos, F., and Upcroft, B. (2016). Simple online and realtime tracking. In *2016 IEEE international conference on image processing (ICIP)*, pages 3464–3468. IEEE.

[Biem et al., 2010] Biem, A., Bouillet, E., Feng, H., Ranganathan, A., Riabov, A., Verscheure, O., Koutsopoulos, H., Rahmani, M., and Güç, B. (2010). Real-time traffic information management using stream computing. *IEEE Data Eng. Bull.*, 33(2):64–68.

[Bochkovskiy et al., 2020] Bochkovskiy, A., Wang, C.-Y., and Liao, H.-Y. M. (2020). Yolov4: Optimal speed and accuracy of object detection. *arXiv preprint arXiv:2004.10934*.

[Carafoli et al., 2012] Carafoli, L., Mandreoli, F., Martoglia, R., and Penzo, W. (2012). Evaluation of data reduction techniques for vehicle to infrastructure communication saving purposes. In *Proceedings of the 16th International Database Engineering and Applications Symposium, August 2012 (IDEAS 2012)*, pages 61–70, Prague, Czech Republic.

[Carafoli et al., 2013] Carafoli, L., Mandreoli, F., Martoglia, R., and Penzo, W. (2013). A framework for its data management in a smart city scenario. In *Proceedings of the 2nd International Conference on Smart Grids and Green IT Systems, May 2013 (SmartGreens 2013)*, Aachen, Germany.

[Carafoli et al., 2016] Carafoli, L., Mandreoli, F., Martoglia, R., and Penzo, W. (2016). A data management middleware for its services in smart cities. *Journal of Universal Computer Science*, 22(2):228–246.

[Carbone et al., 2015] Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., and Tzoumas, K. (2015). Apache flink™: Stream and batch processing in a single engine. *IEEE Data Eng. Bull.*, 38(4):28–38.

[Chen et al., 2015] Chen, X., Fang, H., Lin, T.-Y., Vedantam, R., Gupta, S., Dollár, P., and Zitnick, C. L. (2015). Microsoft coco captions: Data collection and evaluation server. *arXiv preprint arXiv:1504.00325*.

[De Gennaro et al., 2016] De Gennaro, M., Paffumi, E., and Martini, G. (2016). Big data for supporting low-carbon road transport policies in europe: Applications, challenges and opportunities. *Big Data Research*, 6:11–25.

[Dean and Ghemawat, 2008] Dean, J. and Ghemawat, S. (2008). Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113.

[Gulli and Pal, 2017] Gulli, A. and Pal, S. (2017). *Deep learning with Keras*. Packt Publishing Ltd.

[Honarvar and Sami, 2019] Honarvar, A. and Sami, A. (2019). Towards sustainable smart city by particulate matter prediction using urban big data, excluding expensive air pollution infrastructures. *Big Data Research*, 17:56–65.

[Kalman, 1960] Kalman, R. E. (1960). A new approach to linear filtering and prediction problems.

[Kim et al., 2020] Kim, S., Choi, M., Lee, S., Park, H., and Park, S. (2020). Intelligent management system with energy data block in smart city. In *2020 IEEE International Conference on Consumer Electronics (ICCE), Las Vegas, NV, USA, January 4-6, 2020*, pages 1–3. IEEE.

[Martoglia and Savoia, 2021] Martoglia, R. and Savoia, G. (2021). Towards Multi-Model Big Data Road Traffic Forecast at Different Time Aggregations and Forecast Horizons. In *Proc. of 8th EAI International Conference on Mobility, IoT and Smart Cities (Mobility IoT 2021)*. EAI.

[Meng et al., 2016] Meng, X., Bradley, J., Yavuz, B., Sparks, E., Venkataraman, S., Liu, D., Freeman, J., Tsai, D., Amde, M., Owen, S., et al. (2016). Mllib: Machine learning in apache spark. *The Journal of Machine Learning Research*, 17(1):1235–1241.

[Muthanna et al., 2020] Muthanna, M. S. A., Lyachek, Y. T., Musaeed, A. M. O., Esmail, Y. A. H., and Adam, A. B. (2020). Smart system of a real-time pedestrian detection for smart city. In *2020 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus)*, pages 45–50. IEEE.

[Neilson et al., 2019] Neilson, A., Indratmo, Daniel, B., and Tjandra, S. (2019). Systematic review of the literature on big data in the transportation domain: Concepts and applications. *Big Data Research*, 17:35–44.

[NVIDIA, 2021] NVIDIA (2021). cudnn, release: 8.1.

[NVIDIA et al., 2020] NVIDIA, Vingelmann, P., and Fitzek, F. H. (2020). Cuda, release: 10.2.89.

[Özal et al., 2011] Özal, A., Ranganathan, A., and Tatbul, N. (2011). Real-time route planning with stream processing systems: a case study for the city of lucerne. In *Proceedings of the 2nd ACM SIGSPATIAL International Workshop on GeoStreaming*, IWGS '11, pages 21–28.

[Paszke et al., 2017] Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., and Lerer, A. (2017). Automatic differentiation in pytorch.

[Pedregosa et al., 2011] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., et al. (2011). Scikit-learn: Machine learning in python. *the Journal of machine Learning research*, 12:2825–2830.

[Redmon and Farhadi, 2018] Redmon, J. and Farhadi, A. (2018). Yolov3: An incremental improvement. *arXiv preprint arXiv:1804.02767*.

[Serrano et al., 2016] Serrano, D., Baldassarre, T., and Stroulia, E. (2016). Real-time traffic-based routing, based on open data and open-source software. In *2016 IEEE 3rd World Forum on Internet of Things (WF-IoT)*, pages 661–665.

[Vanholder, 2016] Vanholder, H. (2016). Efficient inference with tensorrt.

[Verucchi et al., 2020a] Verucchi, M., Bartoli, L., Bagni, F., Gatti, F., Burgio, P., and Bertogna, M. (2020a). Real-time clustering and lidar-camera fusion on embedded platforms for self-driving cars. In *2020 Fourth IEEE International Conference on Robotic Computing (IRC)*, pages 398–405. IEEE.

[Verucchi et al., 2020b] Verucchi, M., Brilli, G., Sapienza, D., Verasani, M., Arena, M., Gatti, F., Capotondi, A., Cavicchioli, R., Bertogna, M., and Solieri, M. (2020b). A systematic assessment of embedded neural networks for object detection. In *2020 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, volume 1, pages 937–944. IEEE.

[White, 2012] White, T. (2012). *Hadoop: The definitive guide*. " O'Reilly Media, Inc.".

[Wu et al., 2017] Wu, D., Sakr, S., and Zhu, L. (2017). *Big Data Programming Models*.

[Zaharia et al., 2013] Zaharia, M., Das, T., Li, H., Hunter, T., Shenker, S., and Stoica, I. (2013). Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 423–438. ACM.

[Zaharia et al., 2016] Zaharia, M., Xin, R. S., Wendell, P., Das, T., Armbrust, M., Dave, A., Meng, X., Rosen, J., Venkataraman, S., Franklin, M. J., et al. (2016). Apache spark: a unified engine for big data processing. *Communications of the ACM*, 59(11):56–65.