# Real-time context-aware recommendations to drivers based on the level of possible danger

Ciro Beneduce[1], Shaker Mahmud Khandaker[2]

[1]ciro.beneduce@studenti.unitn.it

[2]shaker.khandaker@studenti.unitn.it

***Abstract*— The project aims to implement a big data system that provides real-time context-aware recommendations to drivers based on the level of possible danger. A pipeline has been defined, starting from the data collection, passing through real-time data ingestion and ending with a working demo that, using the map of Trent, plots real-time fake accidents taking into account their severity.**

***Keywords*— Big Data Pipeline, Real-time architecture, Stream Processing, Apache Kafka, Apache Spark, Apache Cassandra**

## I. Introduction

### *Challenges*

Once we figured out how to design the pipeline, this project's most significant challenge was defining a proper danger that could fit a real-time data architecture. After considering several options, we opted for a solution in which various live incidents are simulated in a delimitated zone while simultaneously showing their respective danger levels. We were forced to fake the data to achieve this kind of result. Nevertheless, since we still wanted to operate with real data, we chose a hybrid approach: faking most data values but still using the position of accidents that actually occurred.

### *Data exploration*

We decided to restrict the operation of our project to the city of Trento. Therefore, we used the portal *"Open Data Trentino"* to find a dataset of the Trento accidents. Unfortunately, the only available dataset that took into account the position of the accidents was in the shapefile format and with no API available. Hence, we locally downloaded the dataset and converted it into the CSV format using an online platform. Lastly, using the script *mySQL_importer.py* , we stored the converted dataset in a MySQL database built on top of AWS.

## II. System Model

### A. *System architecture*

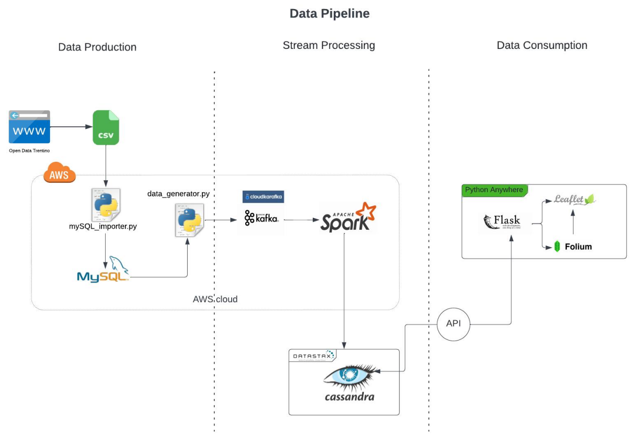



*Figure 1: Data pipeline*

### *Data production*

The next step was to retrieve the significant variables from the dataset. In particular, we were interested in the observations related to the longitude and the latitude of the accidents. This result was achieved using the 2 Python scripts: *mySQLconnection.py* and *accident_data_generator.py*. The first one connects to the database and, using a function, randomly outputs the needed data. The second one generates fake data and, importing the same function, implements those longitudes and the latitudes in an array with the fake data.

### *Stream Processing*

First, we correctly configured an environment for Kafka on top of CloudKafka.com, starting a Broker and creating a Topic. Then, we wrote a Kafka Producer in Python that serialized previously generated data as JSON and sent it to our Topic. Lastly, using the script *accident_kafka_consumer.py, we* subscribed to the Kafka Topic and received the data from it.

The second step of the stream processing was to connect Apache Spark to Kafka Topic in order to set up the structured streaming. Just as for Kafka, also with Spark we set up an environment in Amazon EC2. Afterwards, using the library *Pyspark,* we coded a script that defined a schema for the Kafka data, read it starting from the latest data available and applied on it the new schema. It was a crucial stage because we wanted to ensure that every single data had the same schema to organize and efficiently retrieve the data from the Cassandra table. Subsequently, the data stream was written in a DataStax Astra DB table previously created.

## B. Technologies

The architecture of this project is built on top of several cloud services to avoid the issues related to software installation in local machines, to be ready to use and to simplify the communication between the different technologies. In particular: MySQL and Spark are hosted on AWS, Kafka cluster is hosted CloudKarafka, the Cassandra database is built on top of DataStax Astra DB, and Flask, Leaflet JS and Folium are hosted on PythonAnywhere.

***MySQL***

MySQL is a relational database management system based on structured query language (SQL). Since our dataset is in CSV format, we have decided to store it in a relational database, which is better suited for this kind of tabular data. Therefore, we opted for MySQL, a high-performance but relatively simple database.

***Kafka***

Apache Kafka is an open-source streaming data platform designed to provide quick, scalable and fault-tolerant handling of real-time data feeds. It accepts streams of events written by data producers and stores the records chronologically in partitions across servers, also called brokers. Then, Kafka groups the records into topics, and consumers get their data by subscribing to the topics they want.

The main reason we opted for Kafka as the core of our architecture is that it can handle messages with extremely low latency and with a highly dependable, fault-tolerant, and scalable system in a lightweight distributed fashion.

***Spark***

Apache Spark is an in-memory distributed big data framework used for large-scale data processing. The main selling point of Spark is its speed, up to 100x faster in-memory than Hadoop MapReduce. That is thanks to the Resilient Distributed Datasets (RDDs), which consent to write on memory computations in a fault-tolerant way. By setting up the Spark and Kafka integration, we can ensure minimum data loss through Spark while saving all the received Kafka data synchronously for an easy recovery. Therefore, all these features considered, we chose Spark to enhance our pipeline in order to have a fast and less fragile architecture.

***Cassandra***

Apache Cassandra is a highly scalable, high-performance distributed No-SQL database. It suits our project best because it consents to store semi-structured data and handle large volumes of data distributed across multiple nodes. Furthermore, since every node in Cassandra can perform read and write operations, if a node goes down, there is not any downtime, and the availability for writes is 100% guaranteed.

***Flask***

Flask is a micro web framework which is written in Python. This microframework does not require particular tools or libraries. Also, as it is very small and lightweight, it is ideal for our project. We are using Flask for our Data Consumption Webapp

***Folium and Leaflet***

Folium is a powerful library which can be used for data visualization for several types of Leaflet maps. As it provides interactive maps, it is very handy for dashboard building. We want to plot our recommendations for dangers in a map, thus we considered Folium as an option.

# III. Implementation

We implemented 5 components for data production and streaming, shown in the figure below:

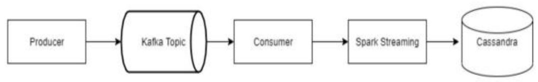
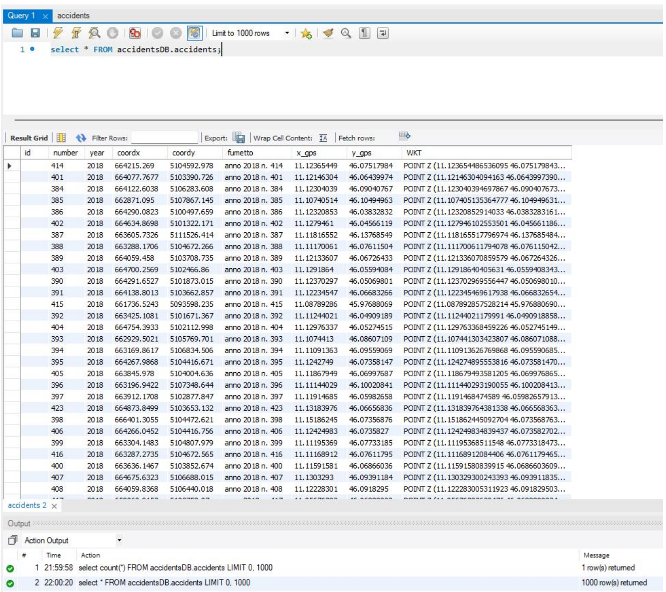



*Figure 2: Flow of the architecture*

These components are described in details.

### *Accident Data Producer*

To populate our database with real data, we downloaded Open Data Trentino historical dataset for accidents. This dataset consists of 16906 incident records. We are running an **AWS EC2 Ubuntu instance** where we deployed our Data Producer application scripts. We implemented "*mySQL_importer.py"* python script to import this csv file to AWS.




This dataset contains the latitude and longitude, level of accidental events from 2003 to 2019. We wrote a script *"accident_data_generator.py"* to generate fake data from the above dataset. "Kafka_accident_producer.py" script then send this fake data are sent to Kafka topic called "wbwkos2c-Accidents" as json.

### *Kafka*

The architecture is based on Kafka and we hosted it on **CloudKafka.com**. We created topic "wbwkos2c-Accidents" with the configuration: 5 partition, 3 replicas, 1048576 retention bytes and 86400000 retention ms. Both of our consumer and producer apps interact with this topic through API.

### *Consumer*

The consumer application subscribes to the kafka topic and consumes the messages which is implemented in "kafka_accident_consumer.py". We extended this script to "spark_stream_handler.py" to consume and stream the data.

### Accident Data Spark Streaming

This application is connected to the Kafka topic and Cassandra cluster. As mentioned above, Spark streaming app consumes Kafka topics through API and streams it to Cassandra. We configured the trigger for streaming with 5 seconds of processing time.

### Cassandra

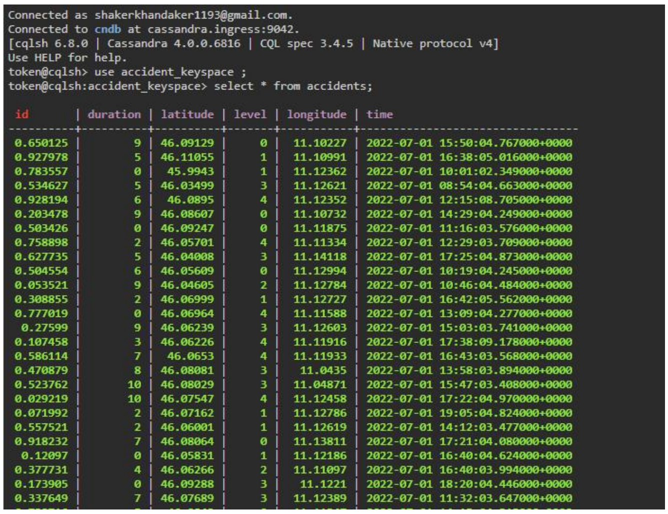
We hosted our Cassandra database on astra.datastax.com. We created a keyspace named "accident_keyspace" and a database "bdt_accidents".

```

CREATE TABLE accidents(
    id float PRIMARY KEY,
    latitude float,
    longitude float,
    level varint,
    duration varint,
    time timestamp
);

```

*Figure 3: Cassandra table creation*



```
Connected as shakerkhandaker1193@gmail.com.
Connected to cndb at cassandra.ingress:9042.
[cqlsh 6.8.0 | Cassandra 4.0.0.6816 | CQL spec 3.4.5 | Native protocol v4]
Use HELP for help.
token@cqlsh> use accident_keyspace ;
token@cqlsh:accident_keyspace> select * from accidents;

 id       | duration | latitude | level | longitude | time
----------+----------+----------+-------+-----------+---------------------------------
 0.650125 |        9 | 46.09129 |     0 |  11.10227 | 2022-07-01 15:50:04.767000+0000
 0.927978 |        5 | 46.11055 |     1 |  11.10991 | 2022-07-01 16:38:05.016000+0000
 0.783557 |        0 |  45.9943 |     1 |  11.12362 | 2022-07-01 10:01:02.349000+0000
 0.534627 |        5 | 46.03499 |     3 |  11.12621 | 2022-07-01 08:54:04.663000+0000
 0.928194 |        6 |  46.0895 |     4 |  11.12352 | 2022-07-01 12:15:08.705000+0000
 0.203478 |        9 | 46.08607 |     0 |  11.10732 | 2022-07-01 14:29:04.249000+0000
 0.503426 |        0 | 46.09247 |     0 |  11.11875 | 2022-07-01 11:16:03.576000+0000
 0.758898 |        2 | 46.05701 |     4 |  11.11334 | 2022-07-01 12:29:03.709000+0000
 0.627735 |        5 | 46.04008 |     3 |  11.14118 | 2022-07-01 17:25:04.873000+0000
 0.504554 |        6 | 46.05609 |     0 |  11.12994 | 2022-07-01 10:19:04.245000+0000
 0.053521 |        9 | 46.04605 |     2 |  11.12784 | 2022-07-01 10:46:04.484000+0000
 0.308855 |        2 | 46.06999 |     1 |  11.12727 | 2022-07-01 16:42:05.562000+0000
 0.777019 |        0 | 46.06964 |     4 |  11.11588 | 2022-07-01 13:09:04.277000+0000
  0.27599 |        9 | 46.06239 |     3 |  11.12603 | 2022-07-01 15:03:03.741000+0000
 0.107458 |        3 | 46.06226 |     4 |  11.11916 | 2022-07-01 17:38:09.178000+0000
 0.586114 |        7 |  46.0653 |     4 |  11.11933 | 2022-07-01 16:43:03.568000+0000
 0.470879 |        8 | 46.08081 |     3 |   11.0435 | 2022-07-01 13:58:03.894000+0000
 0.523762 |       10 | 46.08029 |     3 |  11.04871 | 2022-07-01 15:47:03.408000+0000
 0.029219 |       10 | 46.07547 |     4 |  11.12458 | 2022-07-01 17:22:04.970000+0000
 0.071992 |        2 | 46.07162 |     1 |  11.12786 | 2022-07-01 19:05:04.824000+0000
 0.557521 |        2 | 46.06001 |     1 |  11.12619 | 2022-07-01 14:12:03.477000+0000
 0.918232 |        7 | 46.08064 |     0 |  11.13811 | 2022-07-01 17:21:04.080000+0000
  0.12097 |        0 | 46.05831 |     1 |  11.12186 | 2022-07-01 16:40:04.624000+0000
 0.377731 |        4 | 46.06266 |     2 |  11.11097 | 2022-07-01 16:40:03.994000+0000
 0.173905 |        0 | 46.09288 |     3 |   11.1221 | 2022-07-01 18:20:04.446000+0000
 0.337649 |        7 | 46.07689 |     3 |  11.12389 | 2022-07-01 11:32:03.647000+0000
```

*Figure 4: Cassandra table*

***Serving Application***

This application is implemented with the Flask framework using the Folium python library. At first, we get the real-time accident records from Cassandra and parse it for creating markers on the Leaflet maps. We are also generating random cars as markers in the map. We implemented 2 APIs, one for collecting current data and another for recent past events.

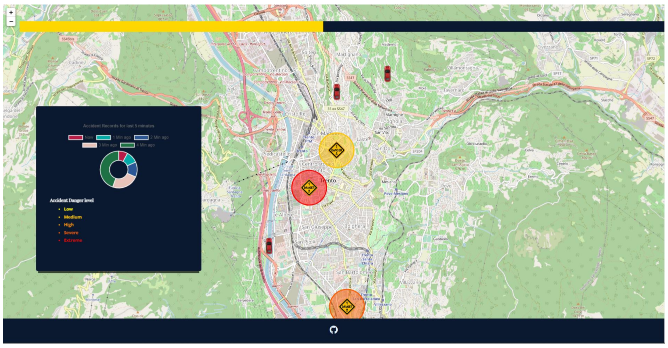


*Figure 5: web app UI*

We are using 5 different colors based on the level of danger due to the accidents. This webpage is refreshed every 10 seconds to update recent data. We are using a Donut Chart to visualize the accident counts for the last 5 minutes.

## IV. Results

The final output of our project is a working web platform that localizes live fake accidents on the map of Trent, taking into account their severity and ranking them on a scale from low to extreme. Moreover, the simulation randomly generates the position of possible drivers/users on the map, who are therefore aware of the real-time dangers in the surrounding context. This demo refreshes every 10 seconds, allowing us to check the work done by the whole pipeline effectively.

To see concretely how our Web platform works, just connect to the link: BDT2022-Group12 (bdt2022group12.pythonanywhere.com)

## V. Conclusions

In this report, we presented our work on a big data architecture able to provide real-time context-aware recommendations to drivers on the level of possible danger.

We are fully aware that the project is not flawless, and some improvements are possible. One of the main limitations is the lack of real GPS data of the drivers that could bring our web platform closer to a real-world scenario. Furthermore, considering multiple sources of danger, more direct and detailed recommendations could be provided.

Nevertheless, all things considered, we are satisfied with the results achieved and believe that our project represents a valid solution for the given assignment.

### References


1. https://www.javatpoint.com/apache-spark-interview-questions
2. https://cloud.google.com/learn/what-is-apache-spark
3. https://hevodata.com/learn/apache-cassandra-vs-mongodb/#availability
4. https://github.com/CloudKarafka/python-kafka-example
5. Elgendy, Nada & Elragal, Ahmed. (2014). Big Data Analytics: A Literature Review Paper. Lecture Notes in Computer Science. 8557. 214-227. 10.1007/978-3-319-08976-8_16.