RELAZIONE SECONDO PROGETTO BIG DATA

GRUPPO TAKOYAKI:

COSTA DANILO: 482492 - FRILLICI FILIPPO 460583

REPO GITHUB PROGETTO:
https://github.com/LSparkzwz/atac-roma-open-data-ingestion-system

REPO FRONT-END:

https://github.com/LSparkzwz/atacmonitor

SITO FRONT-END:

https://lsparkzwz.github.io/atacmonitor/

SITO DI RIFERIMENTO ORIGINALE DEL PROGETTO:

http://www.atacmonitor.com/

Obiettivi del progetto

Per il progetto abbiamo scelto il topic 12: AtacMonitor.

In particolare il nostro obiettivo è stato quello di ristrutturare ed estendere la parte di data ingestion e aggiornare e creare nuove analisi di dati.

Per arrivare alla decisione sulle attività a cui avremmo lavorato, abbiamo deciso di comune accordo di affrontare il progetto con l'idea di risolvere problemi reali, come se stessimo dialogando con un potenziale cliente. Abbiamo perciò fissato una call con l'Ingegner Marco Santoni, referente del nostro progetto, con l'idea di chiedere a lui stesso, in quanto creatore del sito, quali fossero i problemi effettivi, e pensare a delle soluzioni per migliorare il funzionamento di quest'ultimo.

Abbiamo individuato subito una criticità importante mentre ci veniva spiegato da dove il sito prendeva i propri dati, ovvero le API OpenData del sito https://romamobilita.it/it/tecnologie/open-data : le API attualmente usate dal sito AtacMonitor sono API che verranno deprecate nel corso del 2020.
Ci è parso quindi opportuno occuparci della parte di data ingestion, non solo come era nel progetto originale per estendere la raccolta dati alle quasi 9500 fermate di vari mezzi ATAC e TPL presenti su Roma e dintorni, dalle attuali circa 500 analizzate, ma anche per ristrutturare il progetto, effettivamente ricreando da zero tutte le funzioni per raccolta di dati nel nuovo formato offerto da ATAC.

Il nuovo formato aderisce alla convenzione GTFS, che sta per General Transit Feed Specification, ovvero uno standard Google per le API del trasporto pubblico, visibile a questo sito: https://developers.google.com/transit ;

l'obiettivo dichiarato è quello di creare uno standard aperto agli sviluppatori di applicazioni di terze parti (come lo è AtacMonitor), affinché possano creare applicazioni che consumino i dati in maniera interoperabile.

Le informazioni che arrivano con questo nuovo tipo di API, a differenza delle vecchie, si dividono in due parti: una parte di dati statici, giornalieri e una parte di dati in tempo reale, ogni 30 secondi.

I file in tempo reale danno informazioni su tutti gli attuali "viaggi", che hanno un identificatore univoco. Questi viaggi sono i percorsi effettuati da una determinata vettura, di una determinata linea, con informazioni su quale percorso viene effettuato, e quali saranno le prossime fermate.

I dati statici danno informazioni sulle fermate, sui tempi di arrivo in fermata, le corrispondenze tra identificativi univoci dei vari percorsi, vetture e linee, e altre info utili, come ad esempio la posizione delle fermate in coordinate, da cui abbiamo preso la posizione per vedere i quartieri.

Oltre all'importante fase di data ingestion, abbiamo anche concordato di fare analisi sui dati, sia riproducendo ciò che attualmente viene calcolato sul sito, ma utilizzando i nuovi dati, che trovando nuovi tipi di analisi.

Per la riproduzione di ciò che attualmente viene fatto sul sito, ci siamo sincerati di non avere sovrapposizioni con le analisi effettuate dall'altro gruppo che ha scelto questo topic, chiedendo al referente cosa stessero facendo.
Abbiamo quindi riprodotto la parte che calcola la media dei tempi di attesa per linea, estendendo a tutte le linee di Roma, e il tempo massimo di attesa attuale.

Per le analisi nuove, abbiamo pensato di fare analisi sempre sui tempi di attesa, ma diviso per i diversi quartieri di Roma. Per fare ciò abbiamo quindi dovuto trovare un modo per trovare i quartieri date delle coordinate, convenientemente fornite dalle API GTFS.

Il lavoro è contestualizzato per l'azienda di trasporti romana ATAC, ma con l'utilizzo delle nuove API si potrebbe estendere anche ad altri servizi di altri Paesi, a patto che restituiscano sempre informazioni seguendo lo standard GTFS.

Dal nostro punto di vista, gli obiettivi del progetto sono anche di imparare a lavorare su tecnologie reali usate nell'ambito Big Data, e di fare pratica su strumenti Amazon Web Services, molto utilizzati ad oggi in qualsiasi ambito, nonché di imparare a lavorare con linguaggi di programmazione a noi poco conosciuti, in questo caso Python.

È importante anche imparare la progettazione di un sistema, scegliendo quali tecnologie utilizzare, in quanto scelte sbagliate portano a risultati non soddisfacenti.

Abbiamo quindi prefissato come obiettivo anche quello di confrontare diverse tecnologie per scegliere quella più adatta alle nostre esigenze.

È stato anche importante imparare ad interfacciarsi con terzi nell'ambito di progetti, come fosse una prima esperienza lavorativa, chiedendo consigli a chi è più esperto di noi, e ascoltando i feedback sul nostro lavoro, aggiustando di volta in volta il nostro lavoro da neofiti nel campo Big Data.

Architettura e Tecnologie Utilizzate

L'architettura non è stata definita da subito durante il primo contatto con l'Ingegner Santoni, poiché ci ha lasciato molta flessibilità nella scelta di quali strumenti utilizzare, dandoci solo dei consigli su cosa lui reputava essere un buon punto di partenza, e spiegando cosa utilizzava AtacMonitor.
L'architettura attuale di AtacMonitor consiste nell'utilizzo di servizi AWS e non per prendere ed elaborare dei dati.
I dati offerti dalle API OpenData di ATAC venivano presi tramite due funzione di AWS Lambda e salvati direttamente, dopo una minima elaborazione, in un database MongoDB, in formato JSON, ed elaborati direttamente per poter andare a costruire tramite MongoDB Charts dei grafici per mostrare le informazioni desiderate.

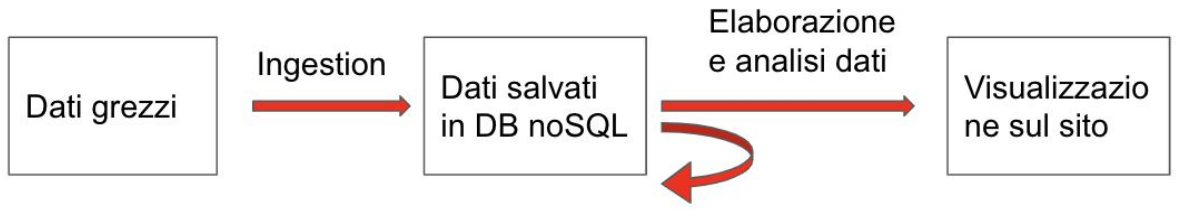
Le due funzioni servivano, una per prendere una lista di paline atac (vengono chiamati così i cartelli gialli o verdi che danno info sulle linee di quella fermata), che poi sarebbe stata passata ad una seconda funzione lambda per richiedere informazioni sui tempi di arrivo di tutti gli autobus alla singola fermata.

Questi JSON venivano salvati in MongoDB Atlas, un servizio che offre uno storage gratuito di 500 MB di MongoDB, limitando le informazioni disponibili a quel determinato quantitativo, comunque sufficiente per una buona analisi di un dataset limitato come quello preso in considerazione dalla versione preesistente del sito (500 stazioni invece delle 9500 totali).

Di seguito una figura esplicativa della vecchia architettura:

Noi abbiamo deciso di affrontare il problema con un approccio similare: dapprima abbiamo una fase di data ingestion, seguita dal salvataggio dei dati in un DB noSQL per l'elaborazione, l'analisi e successiva visualizzazione dei dati.

Una prima bozza dell'architettura è stata definita quindi come segue:



Ovviamente andando avanti con il lavoro e con l'esplorazione dei dati a nostra disposizione, ci siamo resi conto che l'architettura definita all'inizio aveva bisogno di qualche componente aggiuntivo.

Lavorando con Python, intanto ci siamo avvalsi di Google Colab per poter eseguire del codice in maniera veloce e facile, senza dover installare dipendenze o altro sulle nostre macchine. Questo ci è stato utile per la prima fase esplorativa del dataset, per vedere cosa ci veniva offerto dalle API GTFS, e cosa ci sarebbe servito, nonché per testare in maniera gratuita tutto il codice successivo.

Per la parte di Data Ingestion abbiamo deciso di mantenere fede a quello che era già stato fatto, per agevolare il nostro referente nell'utilizzo di tecnologie a lui già familiari, e abbiamo deciso di utilizzare AWS Lambda:

AWS Lambda consente di eseguire codice senza dover effettuare il provisioning né gestire server, pagando solo il tempo effettivo di elaborazione; una volta caricato il codice, Lambda si prende carico delle azioni necessarie per eseguirlo e ricalibrarne le risorse con la massima disponibilità. Si può configurare il codice in modo che venga attivato automaticamente da altri servizi AWS oppure richiamato direttamente da qualsiasi applicazione Web o mobile.

Questo garantisce facilità d'uso (non bisogna creare, mantenere, settare nessuna macchina virtuale) e flessibilità (la funzione può essere complicata quanto si vuole).

Inoltre i costi sono molto contenuti, in quanto abbiamo calcolato che per eseguire la nostra funzione di data ingestion per prendere i dati real-time ogni 3 minuti come già accade, il costo sarebbe stato intorno ai 2$ l'anno, in quanto abbiamo poco meno di 175.000 chiamate, con un costo è di 0.20$ per milione di chiamate (non frazionabile), all'anno più $0.000000208 per 100ms di esecuzione, e il tempo medio di esecuzione della chiamata di ingestion attualmente è di 5000 ms.

Si utilizza EventBridge per lo scheduling delle Lambda: infatti i dati sono in real-time e cambiano ogni 30 secondi, per questo tutte le funzioni per prendere nuovi dati vengono eseguite periodicamente (ogni 3 minuti) per assicurare che i dati siano sempre aggiornati

Il primo approccio pensato quindi è stato quello di prendere i dati, farci una minima elaborazione e salvarli su un DB noSQL Cassandra, per la persistenza degli stessi, dopodiché elaborarli per ricreare sia le analisi attualmente presenti sul sito, sia nuove analisi.

Come ci è stato fatto notare dall' Ing. Santoni, il nostro progetto ha molto valore per quanto riguarda la fase di data ingestion, quindi abbiamo voluto testare come anticipato nella nostra prima presentazione più soluzioni.

Usando Lambda di Amazon per prendere i dati, ci pareva opportuno continuare ad utilizzare tutta la suite di servizi offerti da loro per tutta la fase di Ingestion, per questo come DB Cassandra abbiamo utilizzato quello offerto da AWS, ovvero Keyspaces.

Amazon Keyspaces (per Apache Cassandra) è un servizio gestito Apache Cassandra scalabile ad alta disponibilità compatibile con il servizio di database.

Non bisogna eseguire il provisioning, applicare patch o gestire i server né ad installare, manutenere o utilizzare il software. Inoltre il servizio è serverless, quindi si pagano soltanto le risorse utilizzate e il servizio può ridimensionare automaticamente le tabelle in risposta al traffico delle applicazioni. È un servizio ottimo sia per quanto riguarda prestazioni che elasticità.

Infine per la visualizzazione dei dati nella parte front-end, avremmo fatto un fork sulla repository attuale di AtacMonitor, cercando un nuovo tool per creare grafici.

Avremmo mantenuto l'utilizzo di Github Pages, creando la pagina tramite un file di markup .md oppure html classico.

Anche qui, una prima bozza di architettura con i servizi effettivi era questa:

Per iniziare abbiamo creato una funzione che prendeva i dati da GTFS, e li metteva su Cassandra. Per fare ciò abbiamo dovuto importare in lambda delle librerie di Google Transit, tramite l'utilizzo di Layer:

un layer prende in input un file .zip contenente librerie e funzioni che poi possono essere utilizzate all'interno delle varie funzioni lambda, poiché un layer è riutilizzabile per più funzioni.

Un esempio di funzione da importare nel layer è quella per connettersi ad un'istanza di Cassandra su AWS Keyspace, illustrata di seguito:

```
from cassandra.cluster import Cluster
from ssl import SSLContext, PROTOCOL_TLSv1_2
from cassandra.auth import PlainTextAuthProvider
from cassandra.policies import RoundRobinPolicy
import os

def cassandra_session_init():
    cluster = os.environ['CASSANDRA_CLUSTER']
    username = os.environ['CASSANDRA_USERNAME']
    password = os.environ['CASSANDRA_PASSWORD']
    port = os.environ['CASSANDRA_PORT']
    ssl_context = SSLContext(PROTOCOL_TLSv1_2 )
    auth_provider = PlainTextAuthProvider(username=username, password=password)
    cluster = Cluster([cluster], load_balancing_policy=RoundRobinPolicy(),
        ssl_context=ssl_context, auth_provider=auth_provider, port=port)
    return cluster.connect()
```

Per mantenere il codice più universale possibile (e per mantenere la sicurezza sulle info sensibili) le credenziali sono salvate come variabili d'ambiente statiche in Lambda. Inoltre si possono anche decidere politiche di load balancing e altre caratteristiche del cluster lambda.

Per prendere i dati dal sito ATAC viene utilizzata questa funzione:

```
def update_feed():
    feed = gtfs_realtime_pb2.FeedMessage()
    trip_updates_feed_url = os.environ['TRIP_UPDATES_FEED_URL']
    response = urlopen(trip_updates_feed_url)
    feed.ParseFromString(response.read())
    return feed.entity
```

Questo ritorna un feed che verrà successivamente elaborato per ottenere informazioni sulle varie fermate, in maniera diversa rispetto al vecchio metodo, non solo perché cambia il formato della risposta (da un file JSON a un file .pb protobuffer), ma anche poiché rispetto alle vecchie API cambia il punto di vista; prima infatti si avevano gli autobus in arrivo sulla singola fermata, adesso invece vengono restituite le informazioni su tutti gli autobus in transito attualmente su Roma, con informazioni sul percorso e sulle prossime fermate che verranno effettuate.

È stato quindi necessario capire come elaborare questi dati per ottenere informazioni sui tempi di arrivo, in quanto c'è una sostanziale differenza tra i vecchi dati e i nuovi.

Di seguito un esempio del confronto tra vecchi e nuovi dati:

Vecchi dati:


```
{"_id":
        {"$oid":"5ecd6b65e264ffc567f45c1a"},
        "nome_palina":"SABBADINO",
        "meb":false,
        "aria":false,
        "nessun_autobus":true,
        "moby":false,
        "id_palina":"77947",
        "pedana":false,
        "linea":"081",
        "utctimestamp":{"$date":"2020-05-26T19:17:56.982Z"}
}
{"_id":
        {"$oid":"5ecd6b6512398ed625220593"},
        "aria":1,
        "disabilitata":false,
        "tempo_attesa":10,
        "distanza_fermate":"11",
        "id_palina":"79642",
        "in_arrivo":0,
        "a_capolinea":0,
        "nome_palina":"PONTE MAMMOLO (MB)",
        "annuncio":"11 Ferm. (10')",
        "capolinea":"Ponte Mammolo (MB)",
        "moby":1,"prossima_partenza":"",
        "meb":1,
        "tempo_attesa_secondi":585,
        "carteggi":"",
        "banda":0,
        "id_veicolo":"9706",
        "id_percorso":"53478",
        "carteggi_dec":"",
        "destinazione":"Ponte Mammolo (MB)",
        "pedana":1,
        "linea":"343",
        "utctimestamp":{"$date":"2020-05-26T19:17:57.176Z"}
}
```

Nuovi dati:

```
entity {
  id: "4"
  trip_update {
    trip {
      trip_id: "VJ011606dada1937cb91cb6f23b37c66f824ac39ec"
      start_time: "07:12:00"
      start_date: "20200720"
      route_id: "124"
    }
    stop_time_update {
      stop_sequence: 16
      arrival {
        time: 1595223502
      }
      departure {
        time: 1595223502
      }
      stop_id: "ROME4316"
    }
    stop_time_update {
      stop_sequence: 17
      arrival {
        time: 1595223630
      }
      departure {
        time: 1595223630
      }
      stop_id: "ROME4317"
    }
```

Si nota subito la differenza, che ci ha obbligato ad un nuovo tipo di elaborazione.

Per ottenere i tempi di attesa, che prima venivano forniti da ATAC, dobbiamo effettuare noi il calcolo; abbiamo pensato alla struttura dati in questa maniera:

WAITING_TIMES(stop_id, route_id, waiting_time, current_timestamp)

dove stop_id è l'id di una fermata, route_id è l'id di una linea, waiting time il tempo di attesa a quella fermata, per quella linea, e il timestamp in cui è stata salvata l'informazione in caso servisse.

Quello sul tempo di attesa è stato il calcolo più difficile, non tanto matematicamente, in quanto intuitivamente basta sottrarre al timestamp dichiarato di arrivo quello attuale di quando vengono presi i dati, quanto concettualmente.

Il semplice calcolo descritto sopra restituisce dati imprecisi, tenendo conto del fatto che le informazioni restituite da ATAC molte volte sono sporche e non fedeli alla realtà;
Capita ad esempio che un autobus risulti a 30 secondi dalla fermata, e andando a riscaricare il feed qualche minuto dopo, risulti nuovamente a soli 30 secondi dalla stessa fermata dichiarata in precedenza, indicando sicuramente un dato corrotto.

Inoltre risulta molte volte che il tempo tra una fermata e l'altra sia dichiarato essere di qualche secondo, anche in condizioni di traffico, che seppur in questo periodo è molto minore del solito, è comunque presente e impedisce di percorrere alcuni tratti nel pochissimo tempo dichiarato.
Altra anomalia nei dati risulta essere che il tempo di arrivo a (arrival time) e di partenza da (departure time) una fermata coincidono: ciò ovviamente è fisicamente impossibile dato il tempo di discesa e salita passeggeri, di cui ovviamente non è tenuto conto.

Siamo quindi giunti alla conclusione, anche grazie a un colloquio con il nostro referente, di calcolare nel seguente modo il tempo di arrivo:

viene saltato il primo elemento del feed, che rappresenta la "prossima fermata", in quanto empiricamente abbiamo visto che la maggior parte delle volte, esso è troppo ottimistico (alle volte addirittura risulta negativo!). Quindi escludendo il primo elemento, alcuni percorsi senza fermate, e i feed con un solo elemento, calcoliamo come proposto il precedenza il tempo di attesa come:

Waiting_Time = timestamp(x) - timestamp(t) con x = 2...n dove n è il numero di fermate nel feed, e t è il tempo corrente.

Inoltre prima di scrivere i dati, si deve controllare che non ci sia un altro autobus della stessa linea più vicino ad una determinata fermata, poiché ricordiamo che il feed è relativo a tutti gli autobus in circolazione, e come è intuibile una determinata linea può avere più autobus che in quel momento stanno circolando, e che stanno arrivando ad una fermata. Dopo esserci sincerati quindi che i tempi di attesa siano relativi all'autobus più vicino ad una fermata, possiamo mandare tutto in scrittura.

Di seguito un estratto del codice per il calcolo del tempo di attesa per fermata:

```python
# To calculate waiting_time regarding a stop for a certain route for the current lambda function iteration
# we need to take into consideration the fact a bus route can have multiple buses at the same time
# Waiting_time of a stop for a route = arrival_time of the nearest bus of that route before that stop - current_time
# To achieve this we use a dictionary that stores the waiting time of the current nearest bus found
# Current nearest bus found is the one with the smallest waiting time
# Key = stop_id+route_id, since we want all the buses of the same route for that stop
# Value = [stop_id,route_id,waiting_time], to easily access every value for the insert
def get_stops(feed,current_time):
    stops = {}
    for entity in feed:
        trip_update = entity.trip_update
        trip = trip_update.trip
        route_id = int(trip.route_id)

        found_first_departure_time = False

        for stop_time_update in trip_update.stop_time_update:
            # we try to find the first valid stop_time_update with a valid departure_time
            # we also skip the first valid stop because its waiting_time is too optimistic
            # so we can calculate the waiting times of the successive ones

            if hasattr(stop_time_update, 'departure'):
                if not found_first_departure_time:
                    found_first_departure_time = True
                else:
                    stop_id = str(stop_time_update.stop_id)
                    arrival_time = stop_time_update.arrival.time
                    waiting_time = arrival_time - current_time

                    key = stop_id + str(route_id)
                    if key in stops:
                        if waiting_time < stops[key][2]:
                            stops[key][2] = waiting_time
                    else:
                        stops[key] = [stop_id, route_id, waiting_time]

    return stops
```

Siccome le informazioni sono tante e arrivano con alta frequenza, è stato predisposto un batch insert dentro cassandra per risparmiare sul tempo di esecuzione della lambda. Purtroppo è limitato a 30 statements per batch per ora.

Un esempio del codice per il salvataggio in Cassandra è questo:

```python
def insert_stops(stops, current_time, cassandra_session):
    cassandra_table = os.environ['CASSANDRA_TABLE']
    cassandra_row = os.environ['CASSANDRA_ROW']
    cassandra_prep_values = os.environ['CASSANDRA_PREP_VALUES']

    statement = "INSERT INTO " + cassandra_table + \
        " " + cassandra_row + " VALUES " + cassandra_prep_values
    insert_stop = cassandra_session.prepare(statement)
    batch = BatchStatement(batch_type=BatchType.UNLOGGED,
                           consistency_level=ConsistencyLevel.LOCAL_QUORUM)
    # can only put 30 statements in a batch
    batch_counter = 0

    current_time = current_time * 1000
    for key in stops:
        stop = stops[key]
        stop_id = stop[0]
        route_id = stop[1]
        waiting_time = stop[2]

        batch.add(insert_stop, (stop_id, route_id, waiting_time, current_time))
        batch_counter = batch_counter + 1

        # can only put 30 statements in a batch
        if batch_counter == 30:
            cassandra_session.execute(batch)
            # create a new batch
            batch = BatchStatement(
                batch_type=BatchType.UNLOGGED, consistency_level=ConsistencyLevel.LOCAL_QUORUM)
            batch_counter = 0

    # execute remaining batch operations
    if batch_counter > 0:
        cassandra_session.execute(batch)
```

Alla fine la tabella che ne viene fuori è così composta:

WAITING_TIMES(stop_id, route_id, waiting_time, current_timestamp)

Per ottenere quali sono i corrispettivi reali degli id sopracitati, bisogna incrociare questi id con i dati proveniente da alcuni file statici forniti da ATAC a latere rispetto ai file in real-time.

Anche per quelli è stata predisposta una funzione lambda che prende il zip dove sono salvati i dati online, li apre e li salva su Cassandra; ad esempio per vedere il corrispettivo del route_id a quale linea effettiva (quelle che si leggono sulle vetture) appartiene, è stato fatto così:

```
def lambda_handler(event, context):
    #  INIZIALIZZAZIONE CASSANDRA OMESSA
    response = requests.get('https://romamobilita.it/sites/default/files/rome_static_gtfs.zip').content
    zipfile = ZipFile(BytesIO(response))
    stmt = session.prepare("INSERT INTO static_files.lines_correspondence (route_id, actual_line_name) VALUES (?, ?)")
    execution_profile = session.execution_profile_clone_update(session.get_execution_profile(EXEC_PROFILE_DEFAULT))
    execution_profile.consistency_level = ConsistencyLevel.LOCAL_QUORUM
    with zipfile.open('routes.txt', 'r') as myfile:
        #print(myfile.read())
        content = myfile.readlines()
        for elem in content:
            elemList=elem.decode().split(sep=',')
            elemList[2] = elemList[2].replace('"','')
            session.execute(stmt, (elemList[0],elemList[2]), execution_profile=execution_profile)
```

La struttura è la seguente:
STOPS(route_id, actual_line_name)

Otteniamo quindi i tempi medi di attesa come previsto, salvati in Cassandra.

Oltre alla replicazione delle informazioni già presenti sul sito di AtacMonitor, per validare la nuove data ingestion, abbiamo pensato anche ad un'analisi dei tempi di attesa divisa per quartieri.

Per fare questo dunque ci serviva di ottenere, da ogni fermata, in quale quartiere fosse ubicata, e fortunatamente nei file statici è presente la coordinata di ogni fermata.

In prima battuta la scelta è ricaduta sulle API Google Maps per prendere informazioni sul quartiere date le coordinate. Purtroppo quello delle API di Google Maps è un servizio a pagamento, e pur fornendo del credito gratuito, per l'iscrizione richiede una carta di credito, che nessuno di noi due possiede, non accettando carte di debito (a differenza di Amazon).
Abbiamo quindi dovuto cercare quale fosse un servizio gratuito che offrisse la stessa funzione, detta di "Reverse Geocoding", ovvero da coordinate, risalire a delle informazioni su un determinato luogo.

Per questo ci è venuto in aiuto il progetto OpenStreetMap, ovvero una mappa del mondo Open Source. OpenStreetMap mette a disposizione un motore di ricerca simile a quello di Google Maps chiamato Nominatim, visibile a questo indirizzo:

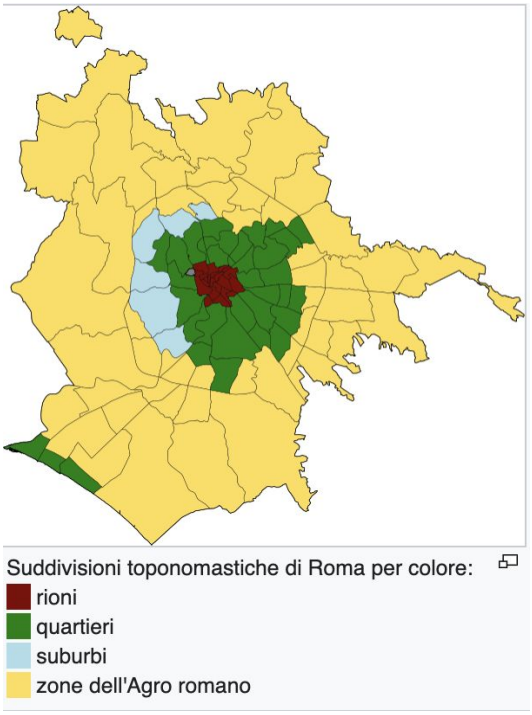
https://nominatim.openstreetmap.org/

e offre anche delle API in maniera simile a quelle offerte da Google per il Geocoding, ma in maniera totalmente gratuita e senza bisogno di registrazione.
Essendo un servizio gratuito, ovviamente sono poste delle limitazioni, sul numero di chiamate giornaliere, e sulla frequenza della chiamate, limitate a 1 ogni secondo.

Usare AWS Lambda per questo tipo di operazione risulta sconveniente, in quanto si paga per la durata dell'esecuzione, e con 1 chiamata al secondo (più tempo di computazione) per 9500 stazioni presenti nel file statico, avremmo dovuto pagare per circa 3 ore di esecuzione, per produrre un semplice file csv con le info sui quartieri.
Abbiamo dunque scelto di usare Google Colab per questa operazione, che consiste nel prendere il file statico relativo alle info sulle fermate, fare una chiamata alle API OpenStreet Map per ottenere il quartiere, e salvare tutto in un CSV e successivamente "ingestarlo" in Cassandra.

Le API sono abbastanza precise, seguendo il modello toponomastico ufficiale di Roma

(cfr. https://it.wikipedia.org/wiki/Suddivisioni_di_Roma#Suddivisione_toponomastica)



E ritornano appunto Quartieri, Suburbi e Municipi in cui è divisa Roma;

in alcuni casi ci sono delle omonimie, pur essendo zone diverse, ad esempio Suburbio e Quartiere Ostiense condividono lo stesso nome ma sono in posizioni differenti.

Inoltre il livello di dettaglio alle volte non è sufficiente, ad esempio non ritorna correttamente le informazioni sui Rioni, che sono inclusi in quello che è denominato "Municipio I".

Da notare che vengono anche etichettate con “Municipio” (che è una macro area) zone di Roma di cui non conosce l’esatto luogo, ad esempio alcune aree vengono etichettate con “Municipio IX”, altre correttamente con “Quartiere XXXII Europa”: quest’ultimo è parte del Municipio IX.

Quindi la precisione delle informazioni è garantita, il livello di dettaglio che possiede però alle volte è insufficiente.

Queste informazioni sulle diverse ubicazioni in zone delle fermate sono salvate poi in Cassandra secondo lo schema:
NEIGHBORHOODS(stop_id, stop_code, stop_name, latitude, longitude, neighborhood)

dove stop_id è l’id che identifica la fermata nelle API real time, stop_code è il codice che si trova scritto fisicamente sulle fermate, stop_name il nome della fermata, latitudine e longitudine, e il quartiere (o municipio, o suburbio).

Di seguito il codice utilizzato per ottenere i quartieri, eseguito su Colab.

```
#to get neighborhoods from coordinates
def getNeighborhoodFromCoordinates():
  response = requests.get('https://romamobilita.it/sites/default/files/rome_static_gtfs.zip').content
  zipfile = ZipFile(BytesIO(response))

  url = "http://nominatim.openstreetmap.org/reverse" #free map service

  querystring = {"accept-language":"it","addressdetails":"1","zoom":"16","namedetails":"0","limit":"5","format":"json"}

#read stop.txt, calls the openstreetmap api obn every coordinate, and saves to csv
  with zipfile.open('stops.txt', 'r') as myfile:
      content = myfile.readlines()
      with open('quartieri.csv', 'w+',  newline = '\n') as csvFile:
          csvWriter = csv.writer(csvFile, delimiter = ',')
          count = 0
          for elem in content:
              if(count >= 0 and count < 10000): #if-else to limit computation to a certain number of elements because of cap
                  elemList=elem.decode().split(sep=',')
                  elemList[2] = elemList[2].replace('"','')
                  querystring.update({"lon":elemList[5],"lat":elemList[4]})
                  response = requests.request("GET", url, params=querystring)
                  data = response.json()
                  if(data):
                      indirizzoCompleto = data.get("address")
                  if(indirizzoCompleto):
                      quartiere = indirizzoCompleto.get("suburb")
                      neigh = indirizzoCompleto.get("neighbourhood")
                      #some coordinates have suburb, some don't
                      if(quartiere):
                        csvWriter.writerow([count,elemList[0],elemList[1],elemList[2],elemList[4],elemList[5],quartiere])
                      elif(neighborhood):
                        csvWriter.writerow([count,elemList[0],elemList[1],elemList[2],elemList[4],elemList[5],neigh])
                      else:
                        csvWriter.writerow([count,elemList[0],elemList[1],elemList[2],elemList[4],elemList[5],'Non trovato'])
                  count+=1
                  print(count)
                  print(quartiere)
                  time.sleep(1) # capped to one second between one call and another
              elif(count < 0):
                  count+=1
              else:
                  print("fatto")
                  break
      files.download('quartieri.csv')
```

Ottenute quindi le informazioni su:

Tempo di attesa per fermata

Fermate

Quartieri di Roma

salvati su DB NoSQL, dovevamo procedere all'analisi vera e propria, riproducendo quello che viene attualmente mostrato sulla pagina di AtacMonitor, ovvero tempo medio d'attesa per linea.

Purtroppo qui ci siamo resi conto che Cassandra su AWS Keyspace non rispondeva alle nostre esigenze.

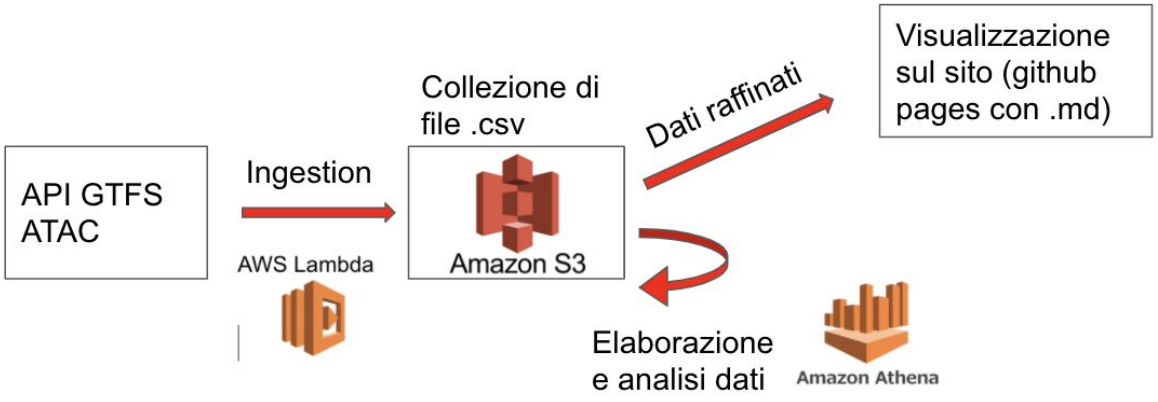
AWS Keyspaces è infatti un servizio giovane (inaugurato a fine aprile di quest'anno, 2020), quindi molte funzioni già disponibili su CQL (Cassandra Query Language) nell'attuale versione di Cassandra (3.11) non sono state implementate dai driver che connettono Keyspaces a Cassandra, dal team di sviluppatori Amazon.

Ci siamo quindi ritrovati ad avere un DB dalle potenzialità enormi, ma che non rispondeva a comandi base; CQL è molto simile a SQL, utilizza molti degli stessi comandi, quindi abbiamo creato delle query che utilizzano funzioni di aggregazione (SUM, AVG, GROUP BY) che non sono attualmente disponibili per l'utilizzo.

Sicuramente sono tutte funzioni che verranno inserite in futuro, ma attualmente abbiamo dovuto modificare questa implementazione con una più adatta ai nostri bisogni.

Essendo il confronto di diverse tecnologie uno degli obiettivi dichiarati del nostro progetto, questo non è stato un problema, in quanto avevamo già in programma di utilizzare altri due strumenti, ovvero S3 e Amazon Athena.

La nuova architettura pensata per l'ingestion, l'analisi e la visualizzazione dei dati quindi è stata pensata come segue:



La prima fase di Data Ingestion rimane pressoché invariata, ma i dati vengono salvati direttamente su Amazon S3:

Amazon Simple Storage Service (S3) è un servizio di storage che offre scalabilità, disponibilità dei dati, sicurezza e prestazioni all'avanguardia nel settore. Si utilizza per archiviare e proteggere una qualsiasi quantità di dati per una vasta gamma di casi d'uso, nel nostro caso ci è utile per fare appunto analisi di big data. La configurazione e la gestione sono semplici, e la durabilità dei dati è garantita;

Non trattando dati sensibili o importanti, e in quantità non troppo eccessiva, per risparmiare abbiamo optato per uno storage con le impostazioni di default (single-region, senza replicazione) che garantisce il costo minore possibile, anche quella caratteristica molto gradita essendo questo un progetto personale dell'Ing. Santoni.

Ora, invece di mettere i dati dentro Cassandra (AWS Keyspaces), essi sono salvati direttamente in S3 come file .csv, i quali sono poi analizzati con AWS Athena per ottenere un aggregato dei dati che a sua volta viene nuovamente salvato (in un altro Bucket per evitare sovrapposizioni) in S3, e utilizzato per la visualizzazione dei dati.

AWS Athena è utilizzato per l’analisi in quanto è un servizio di query interattivo che semplifica l'analisi dei dati in Amazon S3 con espressioni SQL standard. Inoltre è serverless, quindi anche qui non è necessario gestire alcuna infrastruttura e si paga solo in base al tempo di query.

Le query sono eseguite utilizzando SQL standard, sempre tramite AWS Lambda, e i risultati sono disponibili in pochi secondi. L’attività di ETL è quindi ridotta all’osso (solo il calcolo dei tempi di attesa), per tutto il resto si fanno direttamente delle query SQL sui dati salvati in .csv.

È molto comodo perchè, pur utilizzando lo stesso motore di Hive, e quindi fornendo la stessa reattività e capacità di questo strumento visto nel corso, ci ha permesso di non dover installare o configurare nulla su server;

abbiamo direttamente query sui dati, che sono comunque strutturati, essendo salvati in file .csv, e delle tables.

La struttura dei dati è rimasta quella descritta in precedenza, meno il timestamp che non serve più.

Ora viene creata una tabella così definita:

```
CREATE EXTERNAL TABLE stops_feed (
  stop_id string,
  route_id int,
  waiting_time bigint
)
PARTITIONED BY (date date, hour int)
ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.OpenCSVSerde'
WITH SERDEPROPERTIES ("separatorChar" = ",")
STORED AS INPUTFORMAT 'org.apache.hadoop.mapred.TextInputFormat'
OUTPUTFORMAT
'org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat'
LOCATION 's3://stops-feed/'
TBLPROPERTIES ('has_encrypted_data'='false')
```

Notare l'utilizzo di una partizione per data e ora, che ci permette di effettuare delle queries su tutti i vari csv ad esempio in un certo giorno, o per fascia oraria.

Un esempio di utilizzo è il seguente:

```
ALTER TABLE stops_feed ADD
  PARTITION (date = '2020-07-22', hour = 17) LOCATION
's3://stops-feed/date=2020-07-22/hour=17/'
```

per aggiungere la partizione desiderata.

Quindi i vari feed sono inseriti divisi per ora e per data, in cartelle dentro S3 che corrispondono effettivamente alle partizioni, dividendo i dati in base al giorno e all'ora in cui sono stati scaricati.

Si crea dunque una sola volta la tabella, e ogni volta che arriva un nuovo feed si fa una query simile a quest'ultima per aggiungere dati relativi a un certo giorno e a un certo orario.

I dati salvati nelle varie cartelle e sottocartelle, corrispondenti alle partizioni, sono presi ed aggregati, con la query seguente:

```
WITH w AS (
SELECT stop_id, route_id, sum(waiting_time) as waiting_time, count(*) as counter
        FROM "stops_feed"."stops_feed"
        GROUP BY stop_id, route_id
),

n AS (
SELECT * FROM "stops_feed"."locations"
),

r AS (
SELECT * FROM "stops_feed"."routes"
),

t AS (
SELECT n.location, w.route_id, sum(w.waiting_time)/sum(w.counter) as
waiting_time
FROM w LEFT OUTER JOIN n
ON w.stop_id = n.stop_id
GROUP BY n.location, w.route_id
)

SELECT t.location, r.route_name, t.waiting_time
FROM t LEFT OUTER JOIN r
ON t.route_id = r.route_id;
```

Con questo si ottiene infine il risultato desiderato, ovvero la media dei tempi di attesa divisi per zona di Roma, divise per linea.

WAITING_INFO (location, line, avg_waiting_time)

Ovviamente una linea di solito passa per più zone di Roma, quindi comparirà in più punti.

Dettaglio importante, le query su Athena sono veloci, ma non immediate: essendo le query asincrone, avremmo dovuto aspettare di avere il risultato di ognuna di esse, per poi salvarlo e mostrarlo, e questo avrebbe fatto perdere tempo (e conseguentemente soldi, dovendo aspettare il tempo di computazione).
Questo aspetto è stato risolto grazie al fatto di poter attivare con un trigger una funzione lambda all’accadere di eventi in S3.

Per questo abbiamo una lambda function che fa le query, e un’altra separata per l’elaborazione di essi per la visualizzazione, che si attiva solo quando effettivamente ho il risultato della query salvato in S3.

Costando solo 1$ per milione di eventi, si risparmia sul costo di dover aspettare l’esecuzione della query nella lambda function.

Infine quindi gli output delle computazioni da utilizzare nella parte di visualizzazione front-end sono così:

Il primo relativo ai tempi medi di attesa per linea:

764	340.6896551724138
72	503.0
446	934.4375
119	651.625
27	652.56
310	462.3225806451613
723	986.4
543	419.42
628	610.9859154929577
213	558.8222222222222
Roma Lido	642.5
70	957.9024390243902
33	553.8666666666667

I dati sono disordinati e i tempi, espressi in secondi, non sono arrotondati, poiché questi dati non sono fatti per essere visualizzati così come sono, bensì visualizzati su un grafico, dove verranno successivamente ordinati, in ordine crescente di tempo di attesa, arrotondato.

Stesso vale per i tempi di attesa per quartiere:

Quartiere XXIX Ponte Mammolo	873
Quartiere XI Portuense	1240
Quartiere XIII Aurelio	803
Quartiere III Pinciano	3435
Suburbio VIII Gianicolense	2811
Suburbio I Tor di Quinto	936
Zona XXVII Torrino	1022
Quartiere XXXI Giuliano-Dalmata	2234
Quartiere XXII Collatino	1361
Zona II Castel Giubileo	769
Quartiere X Ostiense	1285
Quartiere VII Prenestino-Labicano	2743
Quartiere VIII Tuscolano	1585
Quartiere XXVIII Monte Sacro Alto	1325

Per comodità sono stati aggregati i tempi per tutte le linee di un determinato quartiere, resta il fatto che è tutto predisposto per analisi più approfondite al bisogno.

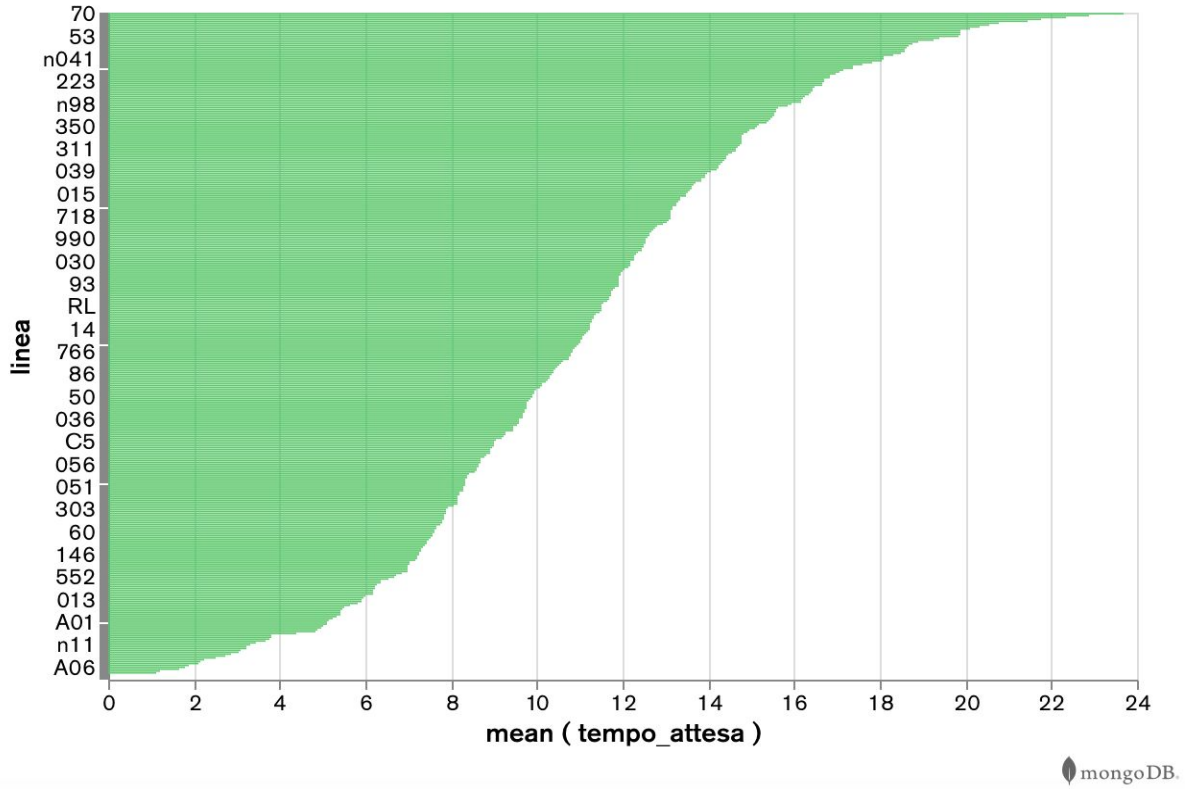
L'output della computazione viene poi utilizzato per la visualizzazione dati nella parte front-end.

Il sito attualmente è fatto molto semplicemente con Github Pages, abbiamo mantenuto quindi l'attuale architettura per il front-end in quanto lo scope del corso è focalizzato sulla parte back-end (ingestion e analisi) dei dati, anche noi ci siamo focalizzati più su quella, anche se ovviamente non abbiamo trascurato questa parte comunque importante.

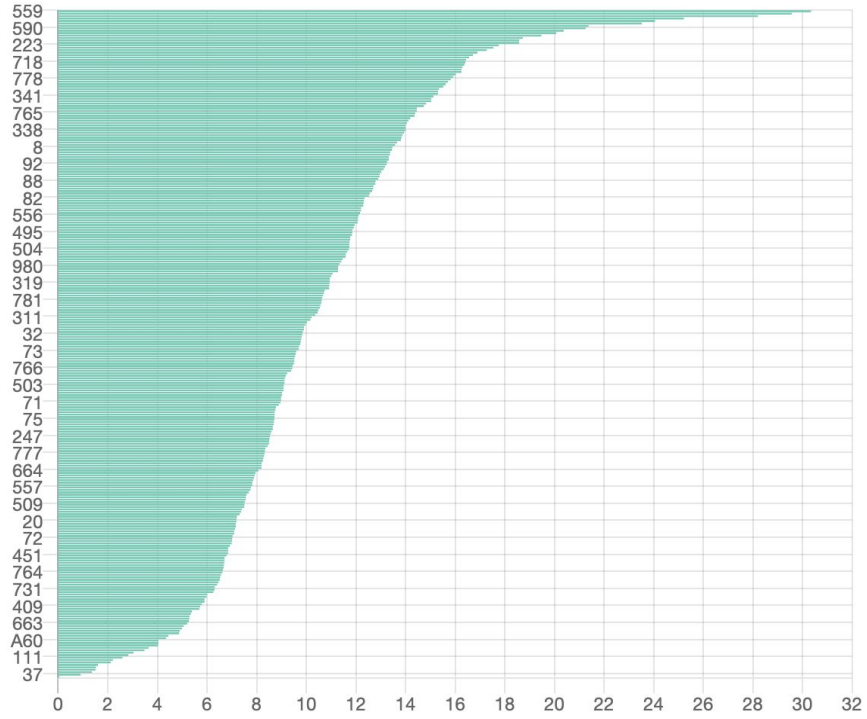
Github Pages permette di hostare un sito direttamente su github, scrivendo in markdown, un linguaggio di markup open-source, utilizzato ad esempio nei readme di github, oppure in html.

Seguendo quindi ciò che è attualmente presente sul sito, mostriamo due grafici:

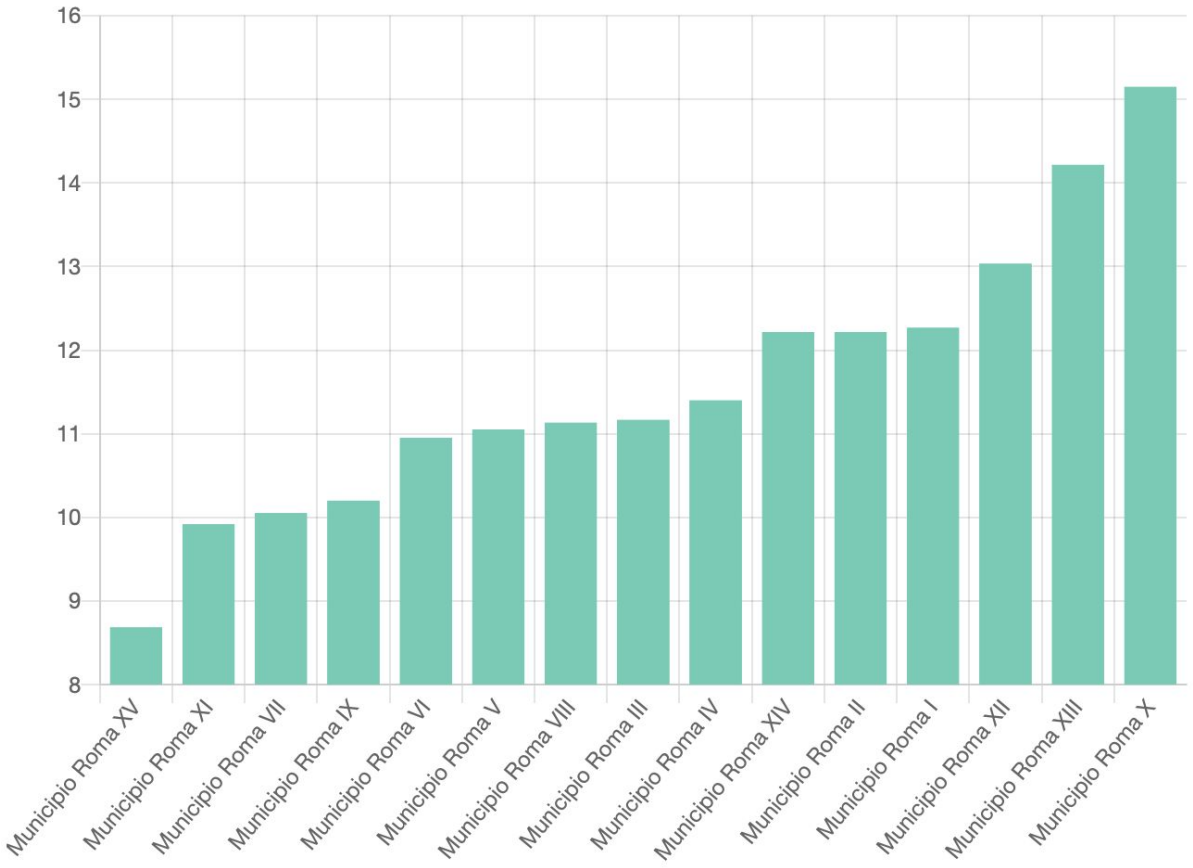
uno che mostra i tempi medi di attesa per linea, e di seguito c'è un confronto tra l'originale, fatto con MongoDB charts, attualmente presente sul sito:



e il nostro, creato con la libreria Chart.js (https://www.chartjs.org/):



Inoltre abbiamo anche il grafico dei tempi medi di attesa diviso per zone di Roma,che è un’analisi nuova e non attualmente presente sul sito, di seguito un esempio sui Municipi:



E infine, il tempo massimo di attesa tra tutte le linee, anch'esso dato presente sul sito originale.

Risultati

Siamo riusciti a rifare da zero tutta l'ingestion dei dati del sito AtacMonitor, e questo era l'obiettivo principale del progetto, che abbiamo portato a termine.

Il tutto è stato fatto su nostri account personali, quindi ci vorrà del tempo prima che l'Ing. Santoni implementi tutte le funzioni create da noi sul proprio account AWS.

Abbiamo avuto cura però di testare tutto quanto, quindi siamo sicuri che tutto sia funzionante e pronto per essere operativo.
Abbiamo utilizzato anche strumenti di cui il nostro referente non ha familiarità, e per i quali è nostra cura fornire una piccola guida all'utilizzo: ciò esula dagli obiettivi del progetto, ma reputiamo sia necessario per la buona riuscita finale di esso; avendo collaborato e lavorato anche noi al progetto AtacMonitor, è nostro auspicio che esso funzioni al meglio.

Anche l'obiettivo prefissato del confronto tra diversi approcci e diverse tecnologie attualmente presenti sul mercato è stato raggiunto:

infatti abbiamo iniziato a utilizzare alcuni degli strumenti principali messi a disposizione dal colosso Amazon per quanto riguarda la gestione di dati, e crediamo questo ci dia una marcia in più in un eventuale futuro lavorativo in questo ambito, essendo AWS tra i servizi cloud più utilizzati al mondo; abbiamo approfittato del fatto che non eravamo legati all'utilizzo di una particolare tecnologia per provarne di diverse, cosa che spesso non è possibile se si lavora per un'azienda, che magari utilizza un solo strumento, senza possibilità di cambiare se non effettuando un costoso refactor, non solo in termini di denaro, ma anche di tempo.

Quello dell'analisi è un obiettivo secondario ma propedeutico alla validazione: infatti riuscendo a replicare quello che è già stato fatto, riusciamo concretamente a dimostrare che il tutto funziona, mentre inventando nuove analisi, miglioriamo ciò che già esiste in modo che il progetto cresca.

Ovviamente le analisi sono abbastanza semplici, e inoltre dovevamo stare attenti a non sovrapporci con le analisi effettuate dall'altro gruppo che ha scelto come noi questo topic.

Ad esempio una delle analisi che avremmo voluto effettuare, visto che i dati forniti da ATAC come detto prima sono abbastanza imprecisi, era di vedere tempi arrivo stimati vs. tempi arrivo effettivi, ma ci è stato detto che questo tipo di analisi veniva già fatta.

Abbiamo optato quindi per queste analisi semplici, ma utili ed efficaci, sui tempi di attesa medi per linea.

Osservazioni conclusive

Come prima cosa vorremmo ringraziare l'Ing. Marco Santoni per il suo supporto, in quanto si è dimostrato fin da subito gentile e disponibile. Non abbiamo ovviamente voluto abusare della sua disponibilità, ma ci ha sicuramente aiutato a fare chiarezza su alcuni dubbi che abbiamo avuto durante lo svolgimento del progetto.

In generale il progetto ci ha fatto capire come sia importante, in un contesto reale, definire chiaramente un'architettura, anche provando e testando quando si parla di tecnologie nuove e giovani, come sono appunto i noSQL, per evitare problemi tipici di esse, sfruttando al meglio solo i vantaggi che esse offrono.

Infine quindi per questo progetto, pur partendo con l'idea di utilizzare strumenti più recenti, abbiamo virato verso l'utilizzo di strumenti più classici, ma che appunto essendo presenti da anni sono consolidati e sempre affidabili.

È stato però utile districarsi tra le tante soluzioni, cercando ogni volta la più adatta, e non fermandosi solo a quelle più conosciute, potendo sperimentare e, perchè no, alle volte sbagliare, correggendosi per portare a termine il progetto in maniera ottimale.

In generale l'architettura da noi creata è coerente con quelle viste durante il corso, quindi abbiamo avuto anche modo di applicare concretamente i concetti visti solo in teoria.

Purtroppo abbiamo incontrato delle difficoltà nell'utilizzo del servizio AWS Keyspaces con account universitario, sia personale che della Classroom Big Data.

Inoltre anche i servizi abilitati, avevano delle limitazioni sull'utilizzo (ad esempio era impossibile cambiare alcuni parametri di computazione, creare dei "ruoli" per l' esecuzione e altro)

Abbiamo anche chiesto se fosse possibile abilitare determinati servizi, ma purtroppo non è stato possibile. Quello che abbiamo dovuto fare quindi è stato creare un account personale;

fortunatamente AWS fornisce un anno di servizi gratuiti (ovviamente con alcune limitazioni sulla quantità di dati che sia possibile elaborare), e quindi per l'utilizzo di servizi quali AWS Keyspaces, AWS Lambda e AWS Athena non c'è stato problema. L'iscrizione inoltre è possibile anche con una carta di debito, quindi con virtualmente 0 rischi di "andare in rosso".

Speriamo che in futuro venga esteso da Amazon l'utilizzo a tutti i servizi, e non solo quelli più basilari, poiché sono molto comodi e validi, e soprattutto in continua evoluzione, come d'altronde tutte le tecnologie in ambito software;

Avremo cura anche di esporre questo nostro feedback raccontando dell'esperienza ad Amazon stessa, affinché possiamo aiutare a migliorare il servizio.