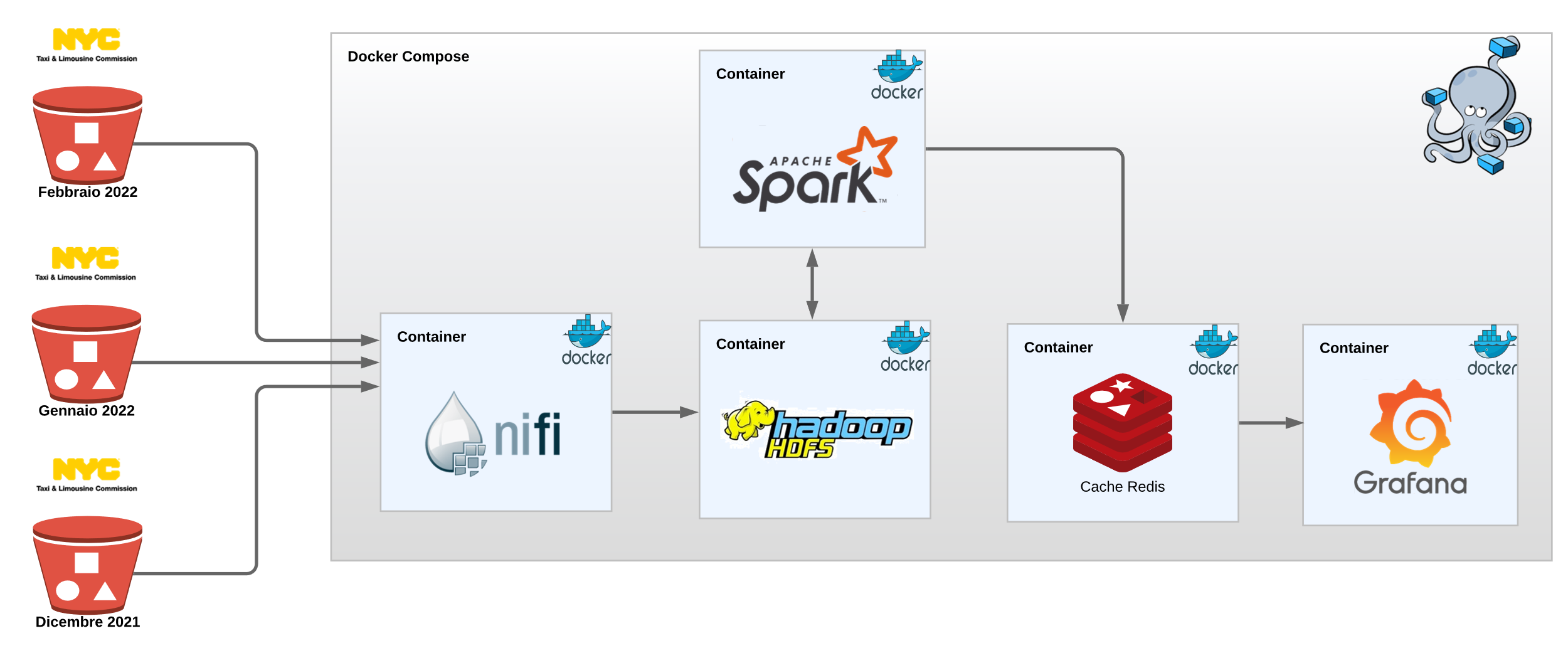
Sistemi e Architetture per Big Data

Progetto 1

# Introduzione

Il presente progetto ha lo scopo di eseguire batch-processing su dataset appartenenti alla NYC Taxi and Limousine Commission (TLC) relativi ai mesi di Dicembre 2021, Gennaio 2022 e Febbraio 2022. Di seguito si presentano l’architettura e le scelte progettuali e implementative delle tre queries richieste. Inoltre, si commentano le prestazioni per ciascuna query al variare delle configurazioni di deployment.

# Architettura



La nostra applicazione spark si divide in tre fasi: **pre-processamento** con NiFi, **processamento** con Spark o Spark SQL e **post-processamento** (Serving Layer) con Redis. I dataset iniziali (ottenuti da dei bucket AWS S3 di appartenenza di TLC) vengono recuperati tramite NiFi e sia all’inizio che alla fine del pre-processamento vengono salvati su HDFS. Al termine del processamento i file di output sono salvati su HDFS in formato CSV e su un Hash Redis per poter essere recuperati velocemente dal framework di visualizzazione Grafana. Tutti i framework sono stati eseguiti su container Docker orchestrati su una singola macchina tramite docker-compose.

## Data Acquisition e Pre-processing

Per la fase di *data-acquisition* e *data-injestion* è stato utilizzato il framework **NiFi**. In questa fase si effettua una pulizia dei dati e l’unificazione dei diversi dataset in un unico dataset ottimizzato per la query richiesta.

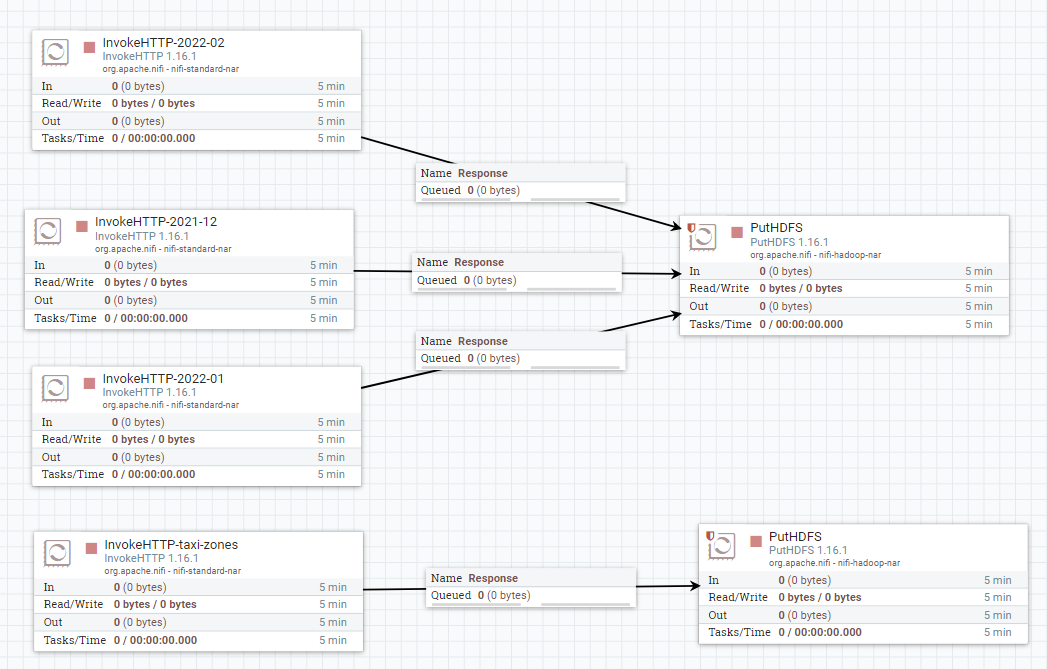


Figura - Template NiFi per scaricare i dataset e salvarli su HDFS

Il template *DownloadToHDFS* utilizza il processor InvokeHTTP per scaricare i datasets (compreso il file che specifica le zone di NYC) e li salva all’interno del cluster HDFS con PutHDFS, in modo che vengano recuperati più velocemente le volte successive in caso di scarsa banda di rete.

I *template\_query1*, *template\_query2*, *template\_query3* sono molto simili ed implementano il preprocessamento vero e proprio in modo specifico per ogni query. Inizialmente recuperano i tre dataset salvati su HDFS dal precedente template con il processor *GetHDFS.* Dopodiché si occupano di **pulire i dati da eventuali incongruenze** e **valori nulli** e di selezionare esclusivamente le informazioni necessarie (e.g: colonne utili allo svolgimento della singola query). Questa fase è stata realizzata tramite un processore *QueryRecord*, che permette di filtrare i dati tramite query SQL. I dati in output dalla fase di pre-processamento mantengono il formato originale Parquet, in modo da mantenere ridotte le dimensioni dei dataset da trasferire allo strato di processamento e permettere un’esecuzione più veloce delle query. I tre Flow Files risultanti dei **dataset vengono infine uniti** con un processore *MergeRecord* a formare un Flow File unico che viene salvato su HDFS nella directory */home/dataset-batch*.

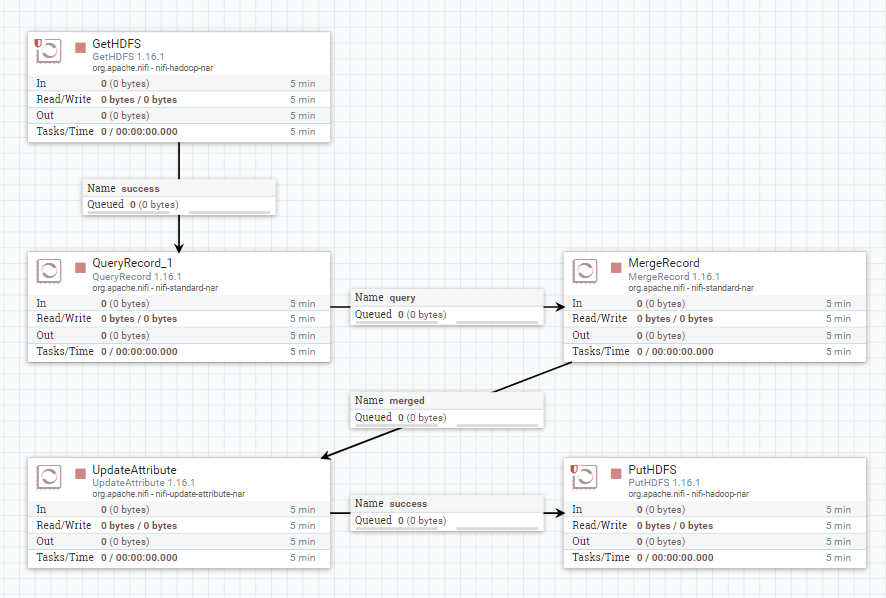


Figura – Il template NiFi per il pre-processamento della query1

### Automatizzazione NiFi

Tutta l’esecuzione dell’acquisizione dei dati e del pre-processamento degli stessi è stata ***completamente automatizzata*** nel codice dell’applicazione grazie alle **API REST** messe a disposizione da NiFi. Di default, il pre-processamento viene eseguito solo se non è disponibile il file HDFS da dare in input alla query desiderata, ma è stata implementata anche un’opzione per forzare il preprocessamento.

L’automatizzazione di NiFi comprende *caricamento e istanziazione* dei Nifi templates, *l’avvio* dei Controller Service e dell’intero Processor Group. Durante il pre-processamento, si controlla periodicamente se i file da dare in input alle query sono disponibili su HDFS e, quando lo sono, il pre-processamento ha termine. Dopodiché tutti i componenti di Nifi (templates, controller service, processors, processor group) vengono *disabilitati* e *rimossi*, per poter ripetere lo stesso procedimento le volte successive.

## Processing

La fase di processamento dei dati è stata effettuata utilizzando **Spark** come framework di batch-processing. È stato scelto questo framework in alternativa ad Hadoop MapReduce, poiché Spark fa un uso più efficiente della memoria e fornisce delle primitive più flessibili e developer-friendly.

Per migliorare la leggibilità del codice, si è preferito l’uso di classi Java piuttosto che delle tuple Scala, in particolare i dati in input sono inseriti all’interno di una classe Query#Bean, le somme parziali e i conteggi nelle classi Query#Calc mentre i dati in output sono inseriti all’interno delle classi Query#Result.

Per tutte e tre le queries, al termine del processamento, i risultati sono stati scritti nuovamente su HDFS in formato csv, utilizzando delle classi CSVQuery#, e inoltre vengono salvati in una cache Redis tramite Jedis (libreria per Java di Redis), in modo da avere una lettura più veloce degli output sul framework di visualizzazione Grafana.

### Query1

Immagine che contiene testo, cielo

Descrizione generata automaticamenteLa prima query richiede di calcolare la percentuale media dell’importo della mancia rispetto al costo della corsa esclusi i pedaggi.

Dalla SparkSession si legge il parquet e si deserializza ogni record in una classe Query1Bean con tutti i dati necessari alla query. Dopodiché si usa una **mapToPair** per convertire il Query1Bean in una tupla (YearMonth, Query1Calc). La classe Query1Calc permette di contare il numero di viaggi (usando l’espediente degli “1” visto anche nel WordCount) e i rapporti Successivamente si esegue una **reduceByKey** con il nostro metodo Query1Calc::sumWith per sommare a due a due i conteggi e i rapporti, in modo da ottenere il numero totale di corse e la somma dei rapporti per ogni mese.

Infine, tramite un task di **map**si calcolano le medie dei rapporti per ogni mese e si salvano nelle classi Query1Result. Poi si esegue una azione **collect**, percollezionare tutti i risultati in una lista e materializzare le trasformazioni precedenti.

Infine si converte la lista di Query1Result in una lista di CSVQuery1 per salvare i risultati in un CSV su HDFS, formattandoli. La stessa lista di CSVQuery1 viene poi salvata anche in un hash Redis, per poter essere recuperata velocemente da Grafana.

La stessa query è stata implementata in Spark SQL con una semplice query SQL:

Immagine che contiene testo

Descrizione generata automaticamente

### Query2

Immagine che contiene testo, cielo, screenshot

Descrizione generata automaticamenteLa seconda query richiede di calcolare per ogni ora la distribuzione in percentuale del numero di corse rispetto alle zone di partenza e inoltre di calcolare la mancia media, la sua deviazione standard e il metodo di pagamento più diffuso.

Anche nel caso di questa query, si legge il dataset parquet unificato e si deserializza ogni record in un Query2Bean con timestamp e zona di partenza, ammontare della mancia e tipo di pagamento.

Dopodiché si esegue una **mapToPair** in una tupla (“yyyy-MM-dd-hh”, Query2Calc) e anche qui si usano gli “1” per contare il numero di corse. La Query2Calc contiene il conteggio (inizialmente 1), l’ammontare della mancia della singola corsa, il quadrato della mancia, e due array: uno con le *occorrenze dei tipi di pagamento* e l’altro con le *occorrenze delle partenze* da ciascuna zona. Inizialmente i due array conterranno tutti 0 tranne un 1 in corrispondenza del tipo di pagamento utilizzato nella corsa e della zona di partenza della corsa.

Successivamente si usa una **reduceByKey** con il metodo Query2Calc::sumWith in cui si sommano i campi conteggio, mancia e quadrato della mancia e si aggiornano gli array in modo che alla fine si avrà la distribuzione delle occorrenze sia dei tipi di pagamento che delle partenze da ogni zona, per ogni ora di ciascun giorno presente nel dataset.

Infine con la **map** si passa da un PairRDD di (“giorno-ora”, Query2Calc) a un RDD di Query2Result, che contiene i risultati desiderati. In particolare le deviazioni standard sono state calcolate sfruttando il conteggio, la somma e la somma dei quadrati delle mance, con la formula:

In modo simile alla query 1, dopo una **collect**, si usa un CSVQuery2 per salvare i dati formattati su HDFS in formato csv e su un altro hash Redis.

La stessa query è stata implementata con Spark SQL sfruttando diverse query SQL:

Immagine che contiene testo

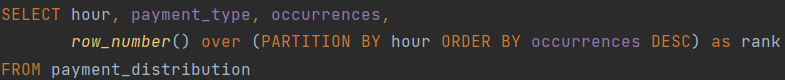
Descrizione generata automaticamente Immagine che contiene testo

Descrizione generata automaticamente

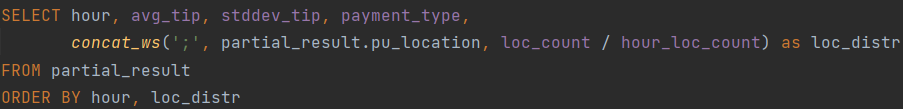
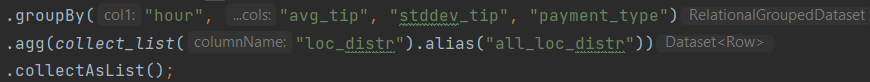
Immagine che contiene testo

Descrizione generata automaticamente Immagine che contiene testo

Descrizione generata automaticamente

Immagine che contiene testo

Descrizione generata automaticamente

### Immagine che contiene testo, cielo Descrizione generata automaticamenteQuery3

Per la terza query si richiedeva di identificare le 5 zone di destinazione più popolari (in ordine decrescente) e di indicare per ciascuna di esse il numero medio di passeggeri, la media e la deviazione standard della tariffa pagata e di rappresentare le zone di destinazione tramite la denominazione recuperata dal dataset “*taxi+\_zone\_lookup.csv”*.

Ancora una volta abbiamo letto il parquet e deserializzato in un Query3Bean che contiene timestamp di arrivo, numero di passeggeri, id della zona di arrivo e ammontare della tariffa. L’RDD di Query3Bean viene quindi mappato con **mapToPair** in un JavaPairRDD di (DayLocationKey, Query3Calc).

Anche in questo caso ogni Query3Calc inizialmente contiene il conteggio (1), il numero di passeggeri, la tariffa e il quadrato della tariffa.

Per calcolare la media e la deviazione standard della tariffa pagata e la media dei passeggeri è stato utilizzato nuovamente un task di ***reduceByKey*** con il metodo Query3Calc::sumWith per aggregare e calcolare le somme che servono al calcolo della media e deviazione standard. Dopodiché si esegue una **map** per spostare la destinazione dalla chiave al valore, ottenendo così un PairRDD con (giorno, (destinazione, Query3Calc)).

Per calcolare le occorrenze delle partenze dalle varie zone di destinazione per ogni giorno, si procede raggruppando tutti i risultati ottenuti per ogni giorno tramite un task di **groupByKey**, che produce un JavaPairRDD con il giorno e un Iterable di tuple (destinazione, Query3Calc). Dopodiché con una **flatMap** si produce un nuovo PairRDD con più istanze con (giorno, (destinazione-in-top-5, Query3Calc)) in modo tale da poter selezionare le prime 5 zone di destinazione per ogni giorno e mantenere anche i corrispettivi dati.

Immagine che contiene testo, screenshot, monitor, schermo

Descrizione generata automaticamenteInfine si usa una **map** per convertire il tutto in un Query3Result e si salva su HDFS in formato CSV e su Redis formattando i dati con la classe CSVQuery3. Su Spark SQL, sono state implementate le query seguenti nell’immagine.

## Post-Processing e Visualizzazione

La cache **Redis** è stata usata per recuperare con minore latenza i dati dal framework di visualizzazione **Grafana**. In ogni caso sono stati salvati i dati in delle HashSet con diverse chiavi a seconda della query (anno/mese, anno-mese-giorno:ora, anno-mese-giorno) e con i campi che hanno le informazioni restanti.

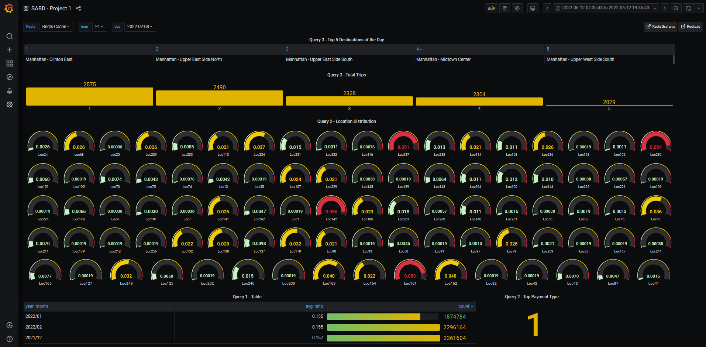


Figura 3 - Dashboard Grafana completa

Per la query1 sono stati visualizzati tutti i dati in forma tabellare, mentre per le query 2 e 3 sono state visualizzate solo alcune informazioni chiave. In particolare, per la seconda si mostra la distribuzione in percentuale del numero di viaggi per ogni luogo di partenza e il valore numerico relativo al tipo di pagamento più popolare. Scegliendo dalla choice box il valore alle variabili Grafana *day* e *hour* è possibile cambiare i dati visualizzati. Per quanto riguarda la terza query invece, sono stati visualizzate le informazioni relative alla classifica delle 5 destinazioni più popolari per il giorno selezionato con il relativo numero di corse.

Dopo aver eseguito le tre queries, è possibile visualizzare i risultati su Grafana sulla pagina:

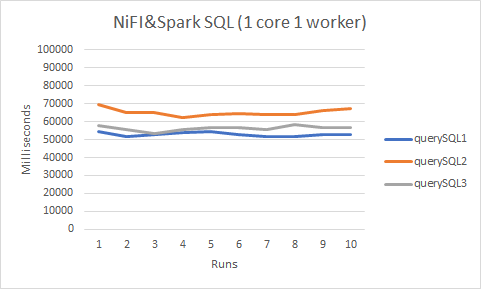
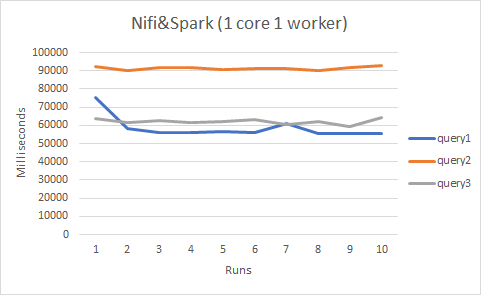
<http://localhost:8000/d/2J8ln097k/sabd-project-1?orgId=1>

# Valutazione delle Performance

Per valutare le performance sono state prese in considerazione diverse configurazioni di deployment in termini di numero di Spark workers. Inoltre, nella configurazione con 1 solo nodo worker sono state considerate due varianti di pipeline: con e senza fase di pre-processamento.

## 1 worker incluso pre-processamento

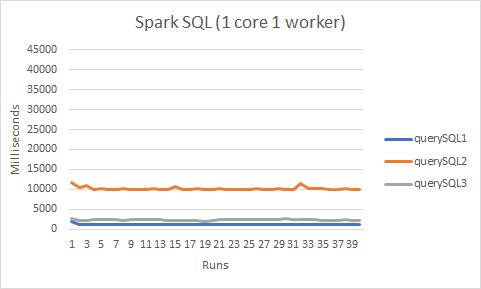
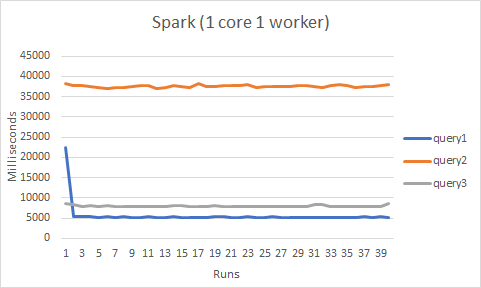
Sono state effettuate 10 esecuzioni di ogni query (riutilizzando le sparksession). Prima di ogni esecuzione è stata **forzata** la fase di pre-processamento.



È possibile notare come mediamente il tempo di esecuzione si attesta tra i 50 e i 70 secondi nel caso delle query 1 e 3 sia con SparkSQL che con Spark. Nel caso della query 2 invece è evidente come i tempi medi per eseguire la query risolta usando gli RDD sia molto meno performante rispetto all’uso dei Datasets. Questo è giustificato dal fatto che nel primo caso Spark non esegue alcuna ottimizzazione sui dati, e pertanto se le operazioni da eseguire sono complesse, allora si avrà conseguentemente un tempo di processamento maggiore.

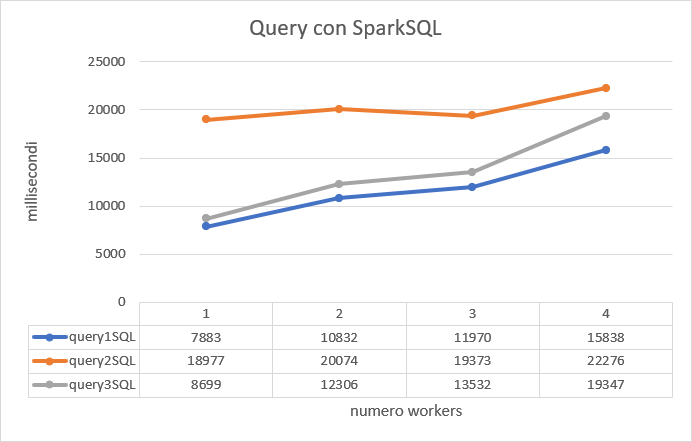
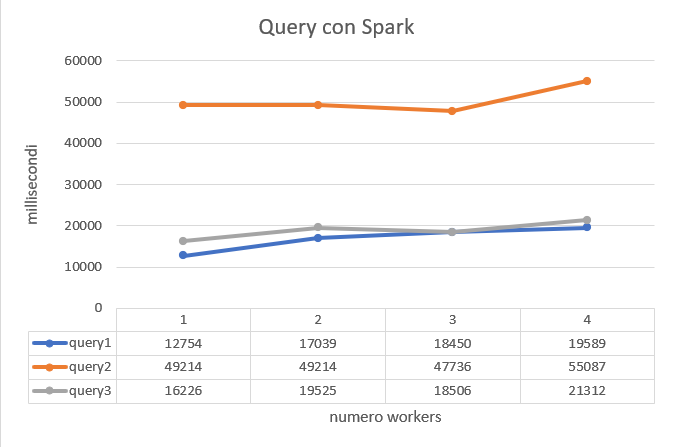
## 1 worker escluso pre-processamento

Sono state effettuate 40 esecuzioni di ogni query (ciascuna riutilizza la sparksession, quindi la prima volta si hanno tempi maggiori rispetto alle successive). All’inizio della prima esecuzione di ogni query viene eseguita la fase di pre-processamento solo se necessario (e comunque non viene inclusa nella misurazione finale).



Confrontando questi risultati con i precedenti, si nota che l’esecuzione del pre-processamento con NiFi è non trascurabile in termini di latenza. Infatti nell’implementazione dell’automatizzazione NiFi è stato necessario aggiungere dei piccoli tempi di attesa di qualche secondo per assicurarsi che i vari componenti di NiFi si attivino/disattivino e per verificare se la fase di preprocessamento sia terminata. Infatti, la latenza complessiva di processamento mediamente si attesta intorno ai 5-10 secondi con Spark e meno di 5 secondi con SparkSQL per le query 1 e 3; e circa 35-40 secondi con Spark e 10-12 secondi con SparkSQL per la query2.

## 1-4 worker escluso pre-processamento



E’ possibile notare che aumentando il numero di worker c’è una tendenza ad un aumento della latenza, perché l’overhead di comunicazione tra i container è più alto rispetto al tempo di processamento. In ogni caso le query con Spark SQL in media sono molto più performanti, grazie al Catalyst Optimizer di Spark.

Tutte le misurazioni sono state effettuate su una macchina con le seguenti caratteristiche:

* Processore: Intel(R) Core(TM) i7-9750H CPU @2.60GHz (6 core, 12 thread)
* RAM: 16,0 GB (12 disponibili per Docker)
* Sistema Operativo: Windows 10 (Docker usa WSL2)