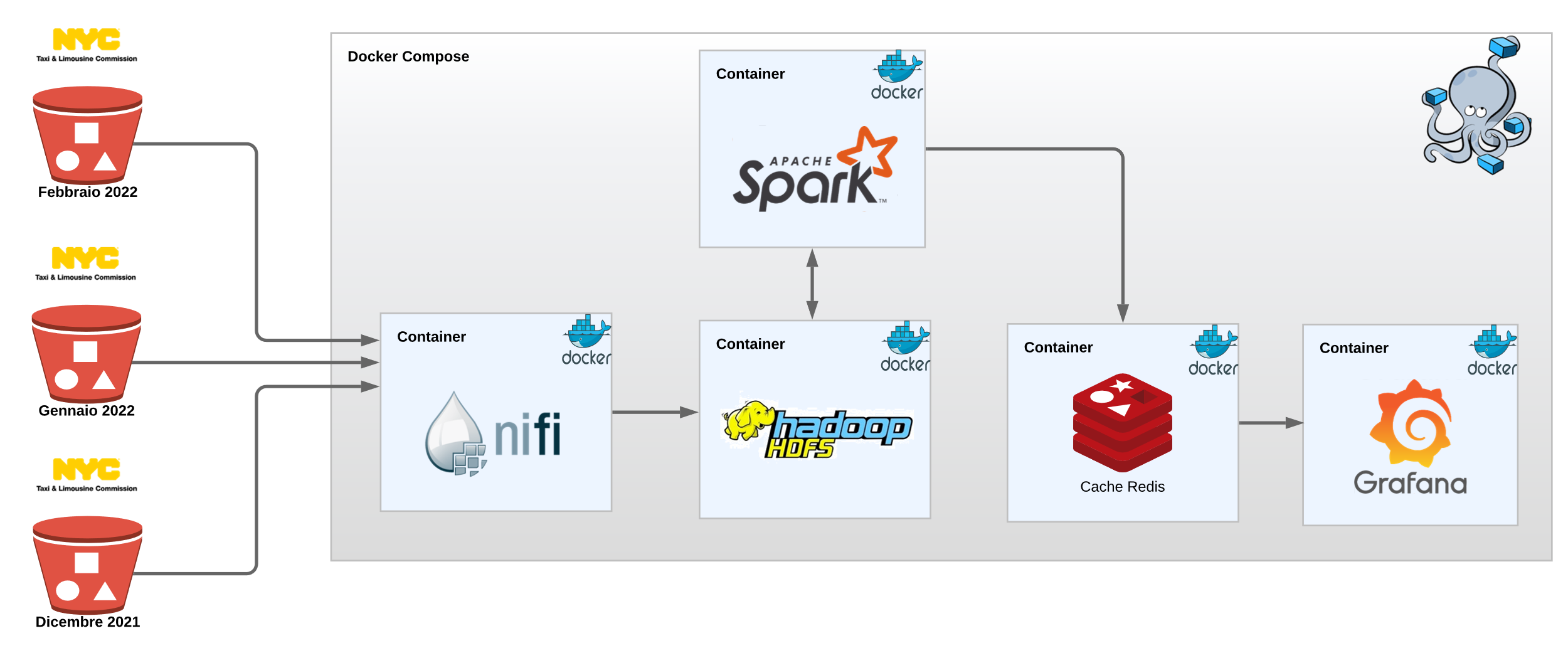
Sistemi e Architetture per Big Data

Progetto 1

# Introduzione

Questo progetto usa il framework Spark per eseguire batch-processing su molteplici dataset appartenenti alla NYC Taxi and Limousine Commission (TLC) relativi ai mesi di Dicembre 2021, Gennaio 2022 e Febbraio 2022. In questo report, sono presentate l’architettura del progetto, le scelte progettuali e implementative delle tre queries richieste. Inoltre, sono presentate in forma grafica gli andamenti delle prestazioni per ciascuna query all’aumentare del numero di Spark Workers.

# Architettura



## Data Acquisition e Pre-processing

Per la fase di acquisizione e pre-processamento dei dati, è stato utilizzato il framework **NiFi**, tramite il quale vengono recuperati i tre dataset relativi ai mesi di riferimento. In questa fase è stata effettuata anche una pulizia dei dati, eliminando eventuali record nulli e dati inconsistenti (e.g: date che riferiscono a mesi diversi rispetto a Gennaio 2022, Febbraio 2022, Dicembre 2021).

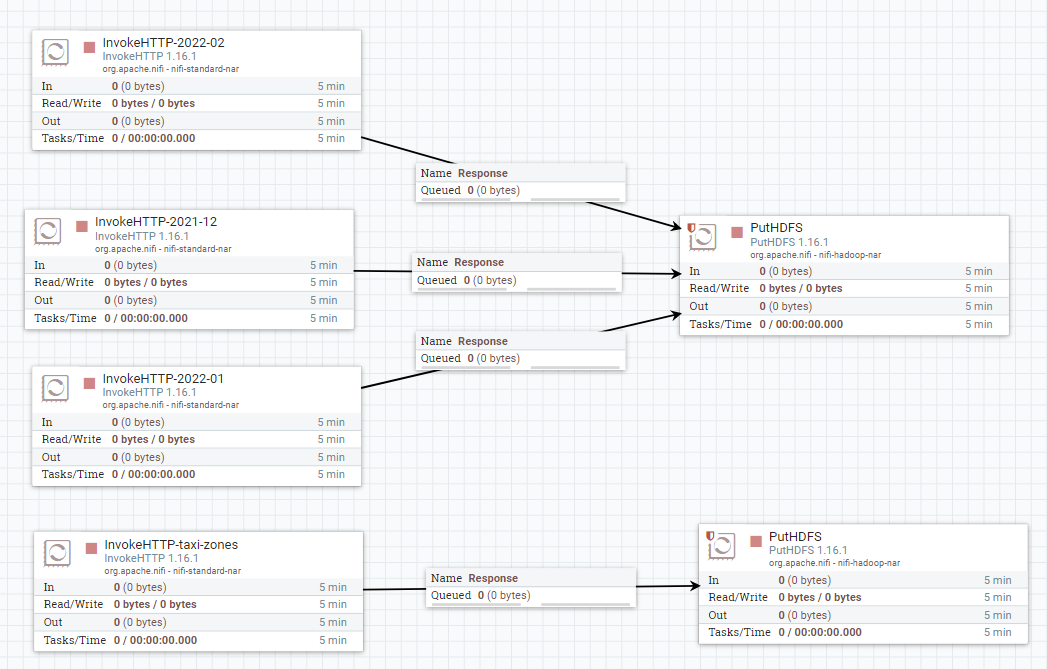


Figura 1 - Template per download dei dataset

Tutta l’esecuzione dell’acquisizione dei dati e del pre-processamento degli stessi è stata ***completamente automatizzata*** usando le API REST messe a disposizione da NiFi, compreso il caricamento dei Nifi templates e l’avvio dei Controller Service e dell’intero Processor Group. Quando i file da dare in input alle query sono disponibili su HDFS, il pre-processamento ha termine.

Il template *DownloadToHDFS* si occupa di scaricare i datasets (compreso il file che specifica le zone di NY) e di salvarli all’interno del cluster HDFS, in modo che vengano recuperati più velocemente le volte successive in caso di scarsa banda di rete.

Altri templates che sono stati creati si occupano di pulire i dati da eventuali incongruenze e valori nulli e di selezionare esclusivamente le informazioni necessarie (e.g: feature utili allo svolgimento della singola query). La fase di selezione delle informazioni è stata realizzata tramite un processore di tipo QueryResult, 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 ridurre le dimensioni dei file batch da trasferire allo strato di processamento e permettere un’esecuzione più veloce di ciascuna query. I tre Flow Files risultanti vengono infine uniti a formare un Flow File unico per poi essere salvato all’interno dell’HDFS nella directory */home/dataset-batch*.

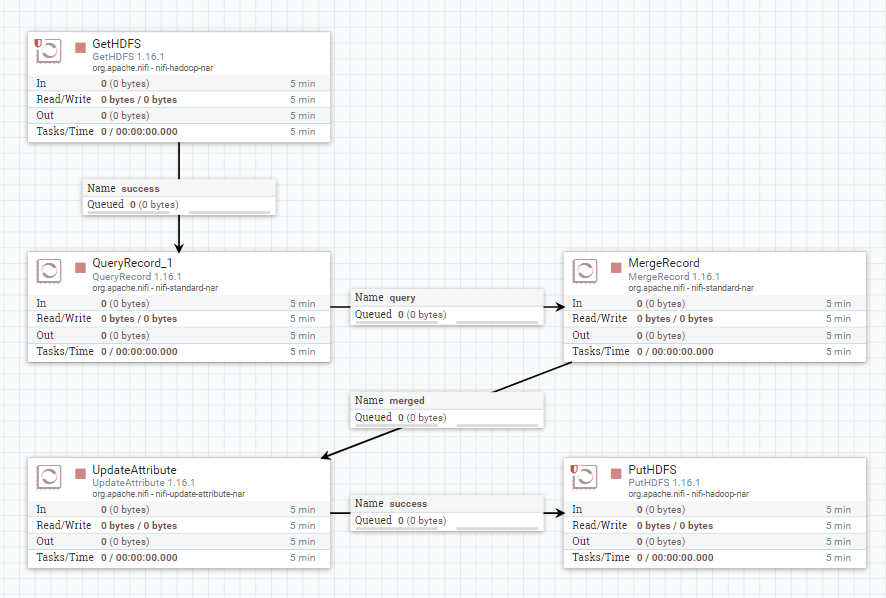


Figura - Template preprocessamento query

## Processing

La fase di processamento dei dati è stata effettuata utilizzando il framework di processamento batch di **Spark**. 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 di tipo bean piuttosto che delle tuple Java, in particolare i dati in input sono inseriti all’interno di una classe Query1Bean, mentre i dati in output sono inseriti all’interno di una classe Query1Result.

Per tutte e tre le queries, i risultati sono scritti nuovamente su HDFS e inoltre salvati in una cache Redis tramite Jedis (libreria per Java di Redis), in modo di 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.

In questo caso era necessario effettuare diverse computazioni di aggregazione. Per questo motivo, si è scelto di utilizzare l’espediente degli “1” usato anche in WordCount per contare il numero di corse per mese-anno e inoltre tutti i calcoli di somma relativi al conteggio delle corse e alla somma dei contributi della percentuale di mancia sono stati completati in un unico task di *reduceByKey*.

Tramite un task di *map* invece si riportano i dati ottenuti alla fine del task di reduce nel tipo Query1Result, che è stato utilizzato come bean per la scrittura comoda del csv in output finale.

### 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, per calcolare i valori di media e deviazione standard e per contare il numero di corse per metodo di pagamento è stato utilizzato l’espediente degli “1” associato tramite un task di *map* ad ogni coppia giorno-ora, di modo che nel successivo task di *reduceByKey* (con chiave giorno-ora) si possano sommare i diversi contributi di 1 e ottenere in questo modo il numero totale di corse in un’ ora, oltre ai contributi all’interno di un array con tanti elementi quante sono le diverse zone di destinazione, di modo da ottenere il numero di corse relative per ogni zona relativa ad una certa ora del giorno specifico. Lo stesso ragionamento è stato inoltre applicato contestualmente per il calcolo del metodo di pagamento più diffuso, con la differenza che dall’array che rappresenta la distribuzione dei metodi di pagamento per giorno-ora è stato estratto il metodo di pagamento che corrisponde al valore massimo nell’ultimo task di map.

### 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 il valore nominare recuperato dal dataset “*taxi + \_zone\_lookup.csv”*.

Per risolvere questa query sono stati necessari un task di *reduceByKey* per aggregare e calcolare le somme che servono al calcolo della media e deviazione standard della tariffa pagata e del numero di passeggeri e inoltre per calcolare le distribuzioni delle varie zone di destinazione per giorno.

Successivamente, si procede raggruppando tutti i risultati ottenuti per ogni giorno tramite un task di *groupByKey*, in modo tale da poter selezionare i primi 5 risultati maggiori per ogni giorno, utilizzando una *flatMap* che permette di ottenere i dati trasformati ottenendo un RDD risultante con un numero maggiore di istanze.

## Post-Processing e Visualizzazione

I dati in output dalla fase di processamento sono stati salvati su HDFS in modo da rimanere persistenti e inoltre sono stati inseriti su una cache **Redis** in modo da essere recuperabili con minore latenza dallo strato di visualizzazione dei dati. Per tutte le query è stato scelto di utilizzare un HashSet. Per la query 2 sono stati salvati i dati in delle HashSet con chiave il giorno e l’ora e con campi che sono le informazioni restanti (mancia media e deviazione standard, metodo di pagamento più popolare e le varie percentuali di distribuzione delle corse per ogni zona di partenza). Per la query 3 invece i dati sono stati salvati HashSet con chiave il giorno e le restanti informazioni come campi (media dei passeggeri, media e deviazione standard della tariffa media per ogni zona di arrivo nella classifica). Il salvataggio dei dati su Redis è stato effettuato utilizzando la libreria Jedis.

Come framework di visualizzazione è stato utilizzato **Grafana**. Sono stati visualizzati i dati completi della query 1 in forma tabellare, mentre per le query 2 e 3 sono stati visualizzati solo alcune informazioni chiave. In particolare, per la seconda query è mostrata la distribuzione in percentuale del numero di viaggi per ogni luogo di partenza e il valore numerico relativo al tipo di pagamento più popolare. Cambiando il valore alle variabili di 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.

Immagine che contiene testo, monitor, interni, screenshot

Descrizione generata automaticamente

Figura - Dashboard Grafana completa

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

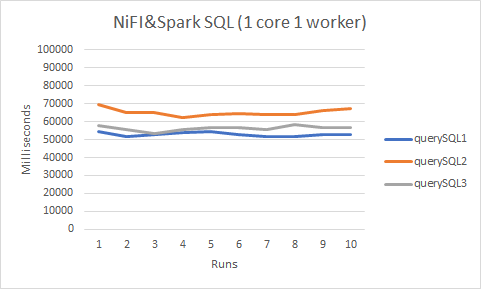
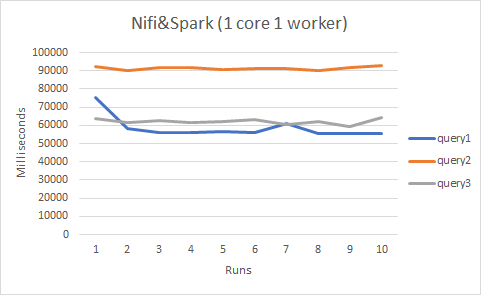
<http://localhost:8000/d/2J8ln097k/sabd-project-1?orgId=1>

# 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.

## Incluso Pre-processamento

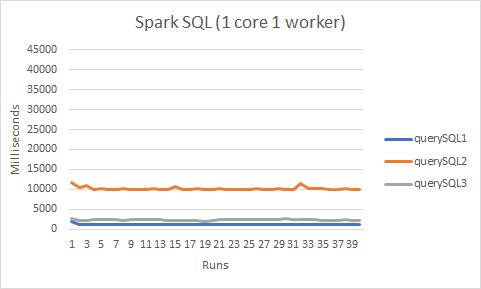
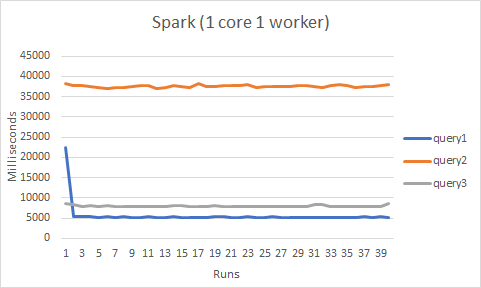
Sono state effettuate 10 esecuzioni di ogni query. Prima di ogni esecuzione è stata forzata la fase di pre-processamento.



È possibile notare come mediamente sulle varie esecuzioni il tempo di esecuzione si attesta tra i 50 e i 70 secondi nel caso delle query 1 e 3 sia con SparkSQL e sia 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.

## Escluso Pre-processamento

Sono state effettuate 40 esecuzioni di ogni query. All’inizio della prima esecuzione di ogni query viene eseguita la fase di pre-processamento solo se necessario (e comunque non inclusa nella misurazione finale).



È possibile notare come l’esecuzione del pre-processamento con NiFi è non trascurabile in termini di latenza. Ciò è dovuto all’inserimento di NiFi all’ interno della pipeline di esecuzione, poiché è stato necessario aggiungere dei piccoli tempi di attesa di qualche secondo per assicurarsi che la fase di preprocessamento sia terminata. Infatti, la latenza complessiva di processamento mediamente si attesta intorno ai 5-10 secondi con Spark e <5 secondi con SparkSQL per le query 1 e query 3; e intorno ai 35-40 secondi con Spark e 10-12 secondi con SparkSQL.