LAMA CLIMATE

Lambda Architecture for Monitoring and Analysis of Climate data

Progettazione e Realizzazione di un'Architettura Lambda per il monitoraggio e l'analisi di dati climatici

Progetto a cura del gruppo **Data-FaSt** (Farioli Davide, Stojani Marjo)

Indice Trattazione

1. Obiettivi del Progetto

La realizzazione del progetto fa riferimento al TOPIC 5: Architettura Lambda, pertanto ha i seguenti obiettivi:

- Creazione e sperimentazione di uno o più scenari applicativi del mondo dei Big Data sia per l'analisi batch che per l'analisi streaming (monitoraggio)
- Sperimentare tecnologie per Big Data come Kafka, database NoSQL come Cassandra e HBase

2. Scenario

Lo scenario scelto per la realizzazione del progetto fa riferimento a dati climatici.
In particolare abbiamo scelto di usufruire di misurazioni di dati climatici fornite da OpenWeather (https://openweathermap.org/).
OpenWeather colleziona e processa dati climatici da diverse sorgenti di misurazioni come satelliti, radar e una vasta rete di stazioni climatiche e consente ai propri clienti di ottenere tali dati tramite semplici API.
È possibile ottenere diversi tipi di dati, in particolare per il nostro progetto sono stati utilizzati i seguenti:

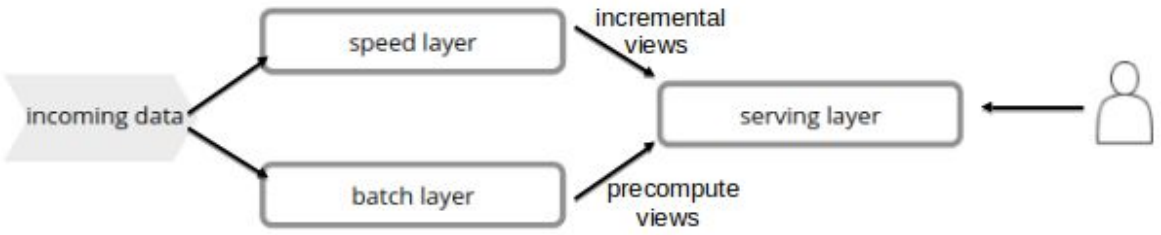
- Current Weather Data: dati climatici (near-)real-time di una particolare città o locazione, accessibili gratuitamente.
- Historical Weather Data: dati climatici storici di una particolare città o locazione (Max 40 anni), accessibili a pagamento.

La scelta della locazione è ricaduta sull'Antartide, luogo dove ad oggi il monitoraggio dei dati climatici è di fondamentale importanza.
Tuttavia il progetto è stato realizzato in modo tale da poter essere espandibile e consentirebbe di monitorare e analizzare dati provenienti da altre locazioni.

3. Architettura realizzata

Il progetto è stato realizzato con un'Architettura Lambda, come da obiettivo, per sfruttare il suo approccio ibrido che consente di processare ed analizzare grandi quantità di dati in modalità batch e al tempo stesso monitorare e processare dati in tempo reale. L'architettura lambda è fondamentalmente composta da 3 layers:

- Batch Layer: si occupa della gestione e analisi di dati storici, integrazione dei nuovi dati in arrivo con i dati storici e generazione di view precalcolate.
- Speed Layer: si occupa del monitoraggio, riceve dati a bassa latenza e li processa generando i risultati su view real time.
- Serving Layer: rende disponibili all'utente finale i risultati generati dal batch layer e dallo speed layer.



Architettura Lambda

3.1 Flusso dei dati

La sorgente dati in questione sono i servizi REST forniti da OpenWeather.
Per il data ingestion si è deciso di utilizzare una delle tecnologie più diffuse in questo ambito: Kafka, una piattaforma distribuita per lo streaming che consente la comunicazione asincrona tramite un sistema di publish-subscribe messaging su stream (flussi) di dati. Sfruttando le API di Kafka si implementano in modo opportuno Producer e Consumer sia lato speed sia lato batch, senza operare grandi trasformazioni di dati; in questo modo i dati provenienti dalla sorgente vengono distribuiti in entrambi i layer.

Facciamo ora una breve introduzione alla tecnologia Kafka e successivamente verranno presentati Batch layer, Speed layer, Serving layer e le relative tecnologie utilizzate, flussi di dati e implementazioni realizzate.

3.2 Kafka

Apache Kafka è una piattaforma open source di stream processing scritta in Java e Scala e sviluppata dall'Apache Software Foundation e fornisce tre capacità fondamentali:

- Publish-Subscribe su stream (flussi) di dati.
- Memorizzazione di stream di dati in modo duraturo e tollerante ai guasti.
- Elaborazione di stream di record, mentre occorrono.

Un cluster Kafka solitamente è composto da un insieme di nodi (chiamati broker) che eseguono Kafka; un cluster Kafka memorizza e distribuisce flussi di record (messaggi), organizzati in categorie chiamate topic (canali).

Tra le API fondamentali di Kafka, sono di interesse:

- Producer API: consente a un servizio o applicazione ("produttore") di pubblicare (publish) un flusso di record su uno o più topic
- Consumer API: consente a un servizio o applicazione ("consumatore") di abbonarsi (subscribe) a uno o più topic e di ricevere i corrispondenti flussi di record; è possibile definire anche gruppi di consumer.

I produttori e i consumatori agiscono come client di Kafka. La comunicazione tra Kafka e i suoi client avviene mediante un protocollo richiesta/risposta basato su TCP.

4. Batch Layer

In un'architettura lambda, il Batch Layer è lo strato applicativo che si occupa della gestione di grandi quantità di dati per poter effettuare analisi, integrare i nuovi dati in arrivo con i dati storici e generare view precompilate, utili per rendere disponibili i risultati delle analisi a bassa latenza.

4.1 Specifiche Hardware e Software

Per raggiungere gli obiettivi del batch layer è di fondamentale importanza scegliere un'adeguata architettura e adeguate tecnologie che, nonostante una computazione massiva su grandi quantità di dati, consentano di garantire requisiti di scalabilità e tolleranza ai guasti.
L'esecuzione dei Job del Batch Layer è stata svolta sia in macchina locale che su Cluster AWS.
I Job sono stati realizzati con il linguaggio di programmazione **Python.**
La tecnologia utilizzata nel progetto per effettuare il processamento dei dati è stata **Spark**, in quanto migliora le prestazioni in termini di efficienza con l'In-Memory Processing.
Per la memorizzazione strutturata dei dati si è utilizzato un NoSQL DBMS di tipo Column-family: **HBase**.
Come servizio di sincronizzazione per sistemi distribuiti è stato utilizzato **ZooKeeper**.
Tutto lo strato applicativo del batch layer è stato realizzato su **HDFS** (Hadoop Distributed File System).

4.1.1 Macchina Locale

Specifiche Hardware**:**

- **Processor:** Intel Core i5 4th Gen 4310U
- **Memory:** 8GB
- **Storage:** 256GB (SSD)

Specifiche Software**:**

- **SO:** Linux Mint 19
- **Python:** 3.7.*

- **Java:** 1.8
- **Hadoop:** 3.2.1
- **Spark:** 2.3.4
- **HBase:** 2.2.5
- **ZooKeeper:** 3.6.1

4.1.2 Cluster AWS

Specifiche Hardware**:**

- **EMR:** 6.0.0
- **Istanza EMR:** m5.xlarge
- **Nodes:** 1 Master, 2 Core
- **Storage:** S3 bucket
- **Modalità di storage HBase:** HDFS

Specifiche Software**:**

- **Python:** 3.7.*
- **Java:** 1.8
- **Hadoop:** 3.2.1
- **Spark:** 2.4.4
- **HBase:** 2.2.3
- **ZooKeeper:** 3.4.14

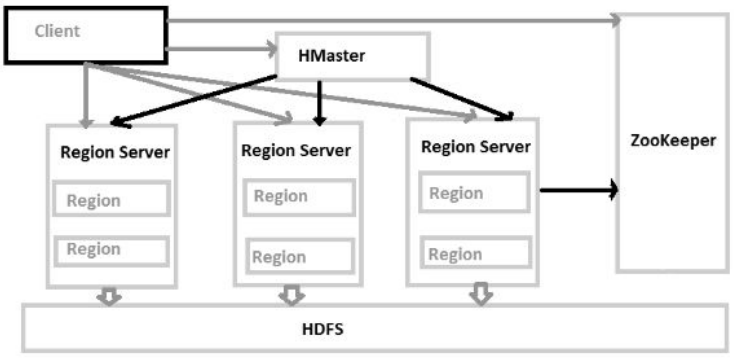
Facciamo ora una breve introduzione alla tecnologia HBase e successivamente verranno discussi i flussi di dati e le implementazioni realizzate nel Batch Layer.

4.1.3 HBase

Apache HBase è un database NoSQL distribuito open source di tipo column-family, eseguito su HDFS e fornisce capacità simili a quelle di BigTable per Hadoop.
HBase memorizza i dati per column-family, ovvero collezioni di colonne; i dati sono memorizzati in tabelle che sono ordinate in base alle righe RowId e sono organizzate per column-family; ogni colonna è una collezione di coppie chiave-valore.

HBase, analogamente agli altri componenti di Hadoop, ha un'architettura master-slave e consiste in tre componenti principali:

- HMaster
- HRegionServer
- HRegion



Architettura HBase

In un ambiente di cluster distribuiti, **HMaster** viene eseguito sul NameNode e svolge funzioni di gestione e monitoraggio delle istanze dei Region Server e funge da interfaccia per gestire tutti i cambiamenti di metadati.
Gli **HRegionServer** vengono eseguiti sui DataNodes e sono i componenti responsabili del hosting e della gestione delle regioni o dati presenti in un cluster distribuito; ogni HRegionServer può ricevere direttamente dal client richieste di lettura o scrittura e assegna tali richieste ad una specifica regione dove risiede la column family.
HRegion è il componente base di cluster HBase per la distribuzione di una tabella, contiene una porzione ordinata di righe della tabella. Quando una Region diventa troppo grande viene divisa in due Region più piccole.
All'interno di ciascuna Region è presente uno Store per ciascuna column family associata alla tabella. All'interno di uno Store troviamo:

- MemStore, che si occupa di conservare in RAM le modifiche (che verranno successivamente salvate su disco con un'operazione di flush)
- HFile che rappresentano i file su HDFS (ogni file è composto da 1 o più blocchi)

HBase è sviluppato volutamente come strumento in grado di garantire performance elevate ma solo su operazioni elementari; i comandi vengono inviati ai vari RegionServer che li eseguono in parallelo su tutte le Region associate ad un tabella.
In questo modo viene garantito un elevato livello di parallelismo e si minimizza lo spostamento dei dati ed il traffico di rete.

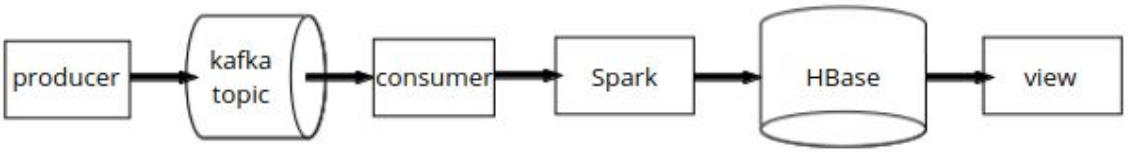
Per quanto riguarda l'interazione con Spark tuttavia presenta limiti in termini di aggiornamenti in quanto i connettori tra le due tecnologie sono un po' datati e relativi a versioni non troppo recenti, per questo motivo nel progetto lato Batch è stato effettuato un downgrade della versione utilizzata di Spark alla 2.3.4.

4.2 Architettura e flusso dei dati

Per la realizzazione del batch layer sono state previste due fasi principali:

- Fase 1: configurazione dei componenti e inizializzazione con popolamento del database con dati storici
- Fase 2: realizzazione di un'architettura che consenta in modo automatico la memorizzazione e l'elaborazione di flussi di dati

La fase 1 in pratica consiste nella memorizzazione su una tabella HBase dei dati storici, forniti tramite un file json, utilizzando un'applicazione Spark.
Per la realizzazione della fase 2 è stata utilizzata un'architettura di tipo "Pipes & Filters" che si basa su una pipeline di componenti ognuno dei quali svolge una determinata funzione sui dati e li passa al componente successivo, a partire dalla sorgente dati (Producer) fino all'ultimo componente (view) che fornisce i risultati ad altri strati applicativi esterni al batch layer.



Architettura e flusso dei dati nel batch layer

4.3 Dettagli implementativi

Fase 1: configurazione dei componenti e inizializzazione con popolamento del database con dati storici

La fase uno consiste in operazioni che sono state eseguite solamente una volta per scopi di inizializzazione del batch layer.

Per la realizzazione della fase 1 sono state eseguite due operazioni:

- creazione della tabella *'climate'* su HBase
- esecuzione del job Spark '*preprocess_historical_data.py*'

La creazione della tabella 'climate' è stata eseguita direttamente nella hbase shell tramite il comando:

create 'climate', 'weather', 'temperature', 'wind'

comando semplicissimo che crea una tabella chiamata *'climate'* e le relative column families (*'weather', 'temperature', 'wind'*). Non è necessario definire le colonne in quanto queste verranno definite nel job Spark.

Per l'esecuzione del job Spark '*preprocess_historical_data.py*' si esegue lo script '*preprocess_historical_data.sh*' che esegue il seguente comando:

```
$SPARK_HOME/bin/spark-submit \
   --packages com.hortonworks.shc:shc-core:1.1.0.3.1.5.90-1 \
   --repositories http://repo.hortonworks.com/content/groups/public/ \
   ../preprocess_historical_data.py
```

dove:

- *com.hortonworks.shc:shc-core:1.1.0.3.1.5.90-1* è il package del **connettore** che è stato utilizzato per consentire la connessione tra Spark e HBase

Il job Spark '*preprocess_historical_data.py*' legge più di 350.000 record di dati da un file json, effettua delle preliminari operazioni di pulizia dati e aggregazione per data (a livello giornaliero) calcolando le relative max/min/avg misure ed infine esegue la scrittura nella tabella HBase 'climate' nella seguente maniera:

```
DataFrame.write \
 .options(catalog=catalog,newTable="5") \
 .format("org.apache.spark.sql.execution.datasources.hbase") \
 .save()
```

dove:

- *catalog* è il catalogo definito nel codice per effettuare il mapping tra dati dello Spark Dataframe e la tabella del database.
- *newTable=5* è il numero di Regions che verranno create per salvare il dataframe nella tabella HBase
- *org.apache.spark.sql.execution.datasource.hbase* è una libreria del connettore utilizzato che definisce il formato tramite il quale scrivere il dataframe nella tabella HBase.

Fase 2: realizzazione di un'architettura che consenta in modo automatico la memorizzazione e l'elaborazione di flussi di dati

Per la realizzazione della fase 2 sono stati implementati diversi componenti che ora verranno descritti nel dettaglio.

- **Producer**

Il Producer è un componente che ha la funzionalità di produrre i dati climatici attuali per il batch layer in questo caso. Più precisamente, il Producer legge i 'Current Weather Data' (i dati climatici correnti) tramite le API pubbliche fornite da OpenWeather e le inoltre al KafKa Topic.

La lettura dei dati è eseguita tramite una semplice richiesta REST dove viene specificata la locazione di interesse (Antarctica); a questa richiesta consegue la response con uno stream di dati in formato json contenente le misurazioni correnti effettuate dalle stazioni localizzate sul posto di interesse.

L'inizializzazione dell'istanza Producer è semplice da realizzare grazie all'utilizzo della libreria *kafka-python*

```
producer = KafkaProducer(bootstrap_servers = ['localhost:9092'],
                         value_serializer = lambda v:
json.dumps(v).encode('utf-8'))
```

dove:

- bootstrap_servers = ['localhost:9092'] indica al Producer l'indirizzo e la porta tramite cui accedere al broker Kafka
- value serializer specifica come serializzare il valore del messaggio spedito al broker Kafka

mentre per l'effettivo invio del messaggio al Kafka Topic si utilizza:

```
producer.send('batch-climate', res)
```

dove ‘batch-climate’ è il topic di riferimento sul quale scrivere il messaggio.
il seguente Producer esegue le operazioni di richiesta a OpenWeather e invio del messaggio ogni 60 minuti.

- **Kafka Topic**

Prima di avviare il Producer, va creato il topic sul broker kafka tramite lo script ‘*create_kafka_topic.sh*’ che crea il topic ‘*batch-climate*’ e definisce indirizzo e porta del broker Kafka.

```
$KAFKA_HOME/bin/kafka-topics.sh --create --topic 'batch_climate'
--bootstrap-server localhost:9092
```

Il broker Kafka svolge la funzione di intermediario tra Producer e Consumer consentendo la comunicazione asincrona e un basso accoppiamento tra i componenti.

- **Consumer**

Il Consumer è il componente che ha la funzionalità di 'consumare’ i dati del Topic Kafka a cui esso è iscritto.
L’inizializzazione del consumer è semplice grazie all’utilizzo della libreria *kafka-python*, basta infatti definire il topic a cui è iscritto il consumer, indirizzo e porta di riferimento del broker Kafka, metodo di deserializzazione del valore compreso nel messaggio da leggere, modalità di lettura quando ci si connette al topic (es. earliest o latest).

```
consumer = KafkaConsumer('batch-climate',
                         bootstrap_servers=['localhost:9092'],
                         auto_offset_reset='earliest',
                         value_deserializer = lambda v:
json.loads(v.decode('utf-8')))
```

il messaggio letto dal consumer viene poi elaborato da Spark che effettua una pulizia dei dati e successivamente scrive il dataframe su una tabella HBase dove verranno depositati tutti gli stream di dati provenienti dal sistema publish-subscribe.
La tabella HBase in questione si chiama *‘batch_climate_streams’* ed è una tabella diversa da quella dedicata alla memorizzazione dei dati storici, poichè è dedicata solamente al deposito degli stream di dati.

- **Spark**

Il componente Spark è stato definito nell'architettura "Pipes & Filters" ma è tuttavia astratto nella nostra architettura, poichè rappresenta un insieme di funzionalità che possono essere eseguite in quel determinato livello della pipeline:

- scrittura degli stream di dati letti dal consumer
- lettura ed elaborazione degli stream di dati memorizzati su HBase per aggregarli e memorizzarli nella tabella principale contenente i dati storici.

Oltre a queste funzionalità potrebbero esserne eseguite altre, magari per sviluppi futuri, sempre riconosciute come funzionalità del componente Spark della nostra architettura, che componente tuttavia non è ma piuttosto è un insieme di altri sottocomponenti ognuno dei quali svolge una determinata funzionalità.

La lettura degli stream di dati dalla tabella *'batch_climate_streams'* viene eseguita dal Job '*daily_batch_stream_aggregation.py*' che appunto ogni 24 ore si attiva, legge da HBase i dati climatici registrati tramite il sistema Publish-Subscribe di Kafka, esegue l'aggregazione per data a livello giornaliero calcolando max/min/avg delle varie misure di interesse, infine scrive il dataframe contenente i dati aggregati nella tabella principale *'climate'* contenente i dati storici.

Lo scheduling per l'attivazione del Job in questione è stato effettuato tramite l'utilizzo di **CRON**, un time-based job scheduler per ambienti linux, che ogni 24 ore chiama lo script '*daily_batch_stream_aggregation.sh'* che esegue il Job.

- **HBase**

HBase è il componente che svolge la funzionalità della memorizzazione dei dati già precompilati e puliti dai componenti precedenti.

La scelta di HBase è stata supportata dal fatto che è un database NoSQL distribuito open source di tipo column-family, eseguito su HDFS; è dunque ideale per sistemi di Big Data in quanto offre elevate prestazioni e alta scalabilità, inoltre è un database column oriented il chè si addice al tipo di dati di interesse del nostro progetto in quanto fanno riferimento a diversi tipi di misure come Temperatura, dati del vento (velocità e direzione) e altri dati climatici come pressione, umidità ecc… Si è infatti pensato di creare diverse column family distinte per diverse tipologie di misure, in tale maniera nel caso servissero solamente dati relativi a una determinata misura (ad esempio solo le temperature massime/minime/medie), effettuando la query verranno inoltrati comandi solamente a una parte dei HRegionServer che li eseguono in parallelo su tutte le Region associate ad un tabella.

In questo modo viene garantito un elevato livello di parallelismo e si minimizza lo spostamento dei dati ed il traffico di rete.

- **View**

View è è il componente che svolge la funzionalità di creare effettivamente le view precompilate che poi verranno fornite al serving layer.
Sono stati creati due tipi di View:

- historical_measures: definiscono le massime e minime misure storiche registrate
- historical_trends: definiscono i trend degli ultimi tre anni su varie misure

Entrambe le view vengono generate sulla base di un'aggregazione basata su 'year' e 'season' in quanto ha senso confrontare le misure min/max/avg tra i vari anni sulla base della stessa stagione.
In particolare l'Antarctica ha solamente 2 stagioni:

- summer: da Ottobre (compreso) a Marzo (escluso)
- winter: da Marzo (compreso) ad Ottobre (escluso)

historical_measures

fornisce al serving layer un file json dove vengono registrate:

- Temperatura massima storica
- Temperatura minima storica
- Velocità vento massima storica
- Velocità vento minima storica
- Umidità massima storica
- Umidità minima storica

per ogni riga si riportano anche la relativa stagione e il relativo anno.

historical_trends

fornisce al serving layer un file json dove vengono registrati i trend degli ultimi tre anni per le seguenti misure:

- Temperatura massima
- Temperatura minima
- Temperatura media
- Velocità vento massima

- Velocità vento minima
- Velocità vento media
- Umidità massima
- Umidità minima
- Umidità media

per ogni riga si riportano anche la relativa stagione e il relativo anno.

Tali view sono state generate soprattutto per fare il confronto con i dati elaborati dallo speed layer in modo da effettuare un'attività di monitoraggio real time avendo a disposizione sia i dati attuali e al tempo stesso i dati storici che hanno portato a scenari di crisi.

5. Speed Layer

In un'architettura lambda, lo Speed Layer ha l'obiettivo di compensare l'inevitabile latenza introdotta dal Batch Layer in modo da consentire l'analisi degli ultimi dati registrati dal sistema, anche se questi possono rappresentare un quadro incompleto dell'intera elaborazione.

Questi risultati vengono resi disponibili a valle di opportune elaborazioni svolte *near-real-time* dallo strato, ovvero elaborando uno stream di dati in intervalli di tempo non troppo grandi. Questi ultimi sono tarati in base al tipo di evento analizzato, generalmente nell'ordine dei minuti.

Un compito fondamentale dello Speed Layer è la gestione degli aggiornamenti a bassa latenza aggregando però continuamente i dati in ingresso al sistema, in modo da offrire analisi simili a quelle che offrirebbe il batch layer. Per soddisfare questo requisito lo Speed Layer fa affidamento sulla computazione incrementale dei dati d'ingresso che si contrappone alla computazione massiva effettuata in un'unica esecuzione su tutto il dataset caratteristica del batch layer.

Il risultato della computazione incrementale viene salvato continuamente in opportune *viste* (che sono dunque sempre aggiornate) e rese fruibili agli utenti attraverso il Serving Layer. Queste viste devono essere salvate in un database che supporti:

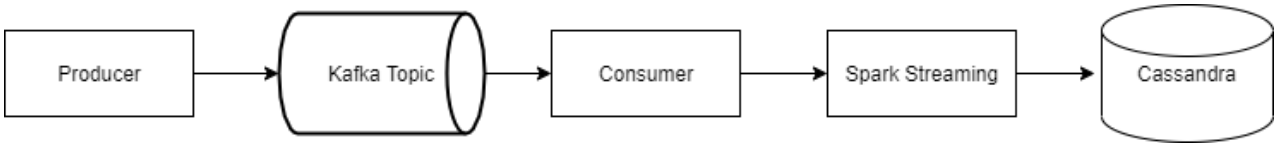
- letture randomiche, in modo da rispondere velocemente alle query degli utenti
- scritture randomiche, per modificare le viste con bassa latenza
- scalabilità, in modo da scalare con l'ammontare di dati che vengono salvati e con il relativo numero di letture/scritture richieste dal sistema.
- resistenza ai guasti, per garantire che la vista ottenuta possa continuare ad essere disponibile anche al fallimento di un nodo su cui sono salvati i dati.

Queste sono caratteristiche tipiche dei database NoSql, per cui è stato naturale selezionare uno dei tanti prodotti disponibili di questo tipo al fine di soddisfare questi requisiti.

L'obiettivo dello Speed Layer di questo progetto è di generare, per diversi intervalli di tempo, l'analisi delle tendenze (trends) dei valori di temperatura, umidità e velocità del vento dell'Antartide. Seguono i dettagli sull'implementazione dello strato.

.5.1 Dettagli Implementativi

Lo Speed layer di *Lama Climate* è composta da 5 componenti, mostrati nella figura seguente:



Flusso dello Speed Layer nella Climate Architecture

Questa sotto-architettura è di tipo “Pipe & Filters” con un flusso di dati dalla sorgente (Producer) che viene spinto verso il pozzo (Cassandra).
Vengono ora descritti in dettaglio i vari componenti dell’architettura.

Producer

Il producer è un processo che ha il compito di leggere i dati climatici dalle API pubbliche di Open Weather al fine di creare l’input per l’intera architettura lambda. Questo si concretizza in uno script python che è configurabile attraverso 2 parametri: è possibile infatti specificare il nome della località da cui si vogliono leggere i dati e l’intervallo di tempo fra una lettura e l’altra. I risultati delle letture sono inviate nel topic (canale) di Kafka denominato “climate” da cui attingono ai dati sia i consumers del batch layer che dello speed layer.
Il dato che viene inviato è semplicemente il file json restituito dalle API di Open Weather con i vari dati climatici attuali della località scelta.

Kafka

L’intera architettura è basata su Kafka al fine di permettere la lettura dei dati climatici sia dallo Speed Layer che dal Batch Layer. Questo infatti è possibile mediante un canale di Kafka, propriamente chiamato Topic, denominato “climate”. Il canale è stato creato con 4 partizioni in modo da rendere parallele (ed efficienti) sia le scritture che le letture di messaggi nel canale. Sia il consumatore che il produttore si interfacciano con il canale Kafka mediante le API offerte dall’omonima libreria per Python 3, installabile facilmente dal Python Package Index (pip) con l’apposito comando: `pip install kafka-python.`

Consumer

Il consumer dello Speed Layer è stato reso necessario da alcuni problemi di incompatibilità con la versione utilizzata di SparkStreaming (3.0) e Python. Nel periodo di costruzione del

progetto, l'implementazione di Spark in Python (PySpark) non offriva la possibilità di accedere ad un flusso di messaggi da un topic di Kafka direttamente con le API offerte. Per questo motivo è stato necessario creare un "connettore" tra lo stream di Kafka e il componente di Spark Streaming. Questo connettore ha i seguenti compiti:

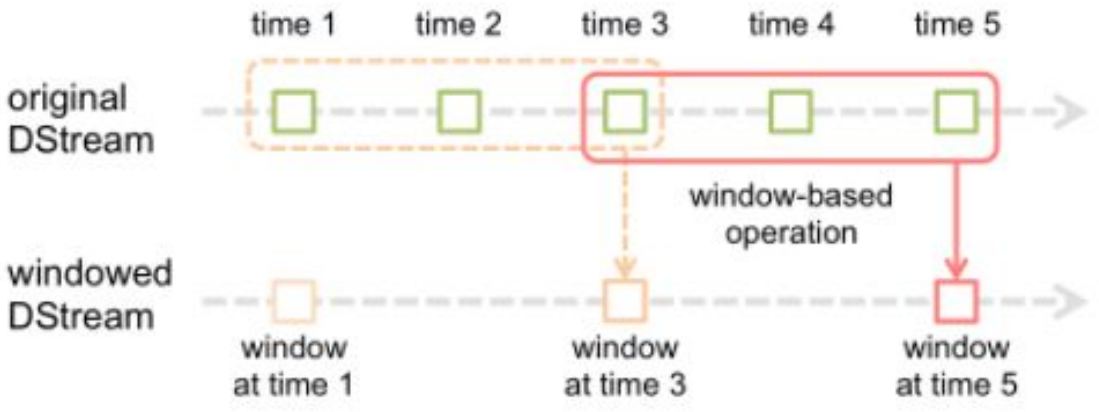
- consumare i messaggi provenienti dallo stream di Kafka
- effettuare delle piccole elaborazioni per la pulizia dei messaggi ottenuti
- inoltrare lo stream dei dati puliti al componente di processamento e analisi effettivo.

Dato che l'unica opzione disponibile per PySpark Streaming era la lettura di un flusso proveniente da un socket TCP, il consumer crea un server in ascolto su una porta della stessa macchina in cui viene eseguita l'elaborazione dello stream. Nel momento in cui PySpark accede alla porta per la richiesta del flusso, il consumer spinge i dati raccolti verso il componente di elaborazione dello stream, come se effettivamente la connessione tra Kafka e Pyspark fosse trasparente.
Va sottolineato che questo consumer non sarebbe stato affatto necessario se si fosse scelto l'uso di Spark mediante le API di Java o Scala, in quanto permettono di "incapsulare" il componente direttamente nell'analisi dello stream.

Spark Streaming

Questo componente rappresenta il cuore dello Speed Layer. La sua funzione è di raccogliere i messaggi contenenti le informazioni climatiche e di generare le statistiche di questi su diversi periodi di tempo.
La prima operazione che viene effettuata da questo componente è la connessione al cluster di Cassandra, su cui vengono salvati i risultati delle elaborazioni in modo da essere fruibili in seguito dal Serving Layer.
Successivamente, il componente si connette alla porta su cui è in ascolto il server locale del consumer, per aggirare i limiti di PySpark evidenziati nel componente precedentemente trattato. Il flusso d'ingresso ottenuto viene trattato come un "DStream" (Discretized Stream), cioè l'astrazione alla base di Spark Streaming che rappresenta un flusso di dati continuo, implementato attraverso una serie continua di RDD (Resilient Distributed Dataset), cioè gli elementi di base su cui opera lo strumento d'elaborazione.
Questo stream di dati è in seguito analizzato mediante una caratteristica di Spark Streaming, cioè le "*Window operations*", che consentono l'elaborazione incrementale richiesta dallo Speed Layer.
Le *Window operations* sono un comodo strumento offerto da Spark Streaming che consentono di applicare una trasformazione di Spark su una finestra temporale scorrevole

dei dati d'ingresso. Quando la finestra scorre sui dati ottenuti dal DStream si combinano i dati appartenenti ai rispettivi RDD dello stream che sono all'interno della finestra. Questo consente di ottenere un unico RDD associato alla finestra su cui è possibile effettuare le operazioni di aggregazione e di analisi sui dati. E' possibile specificare, per ogni finestra, la lunghezza della finestra, cioè quanto analizzare dallo stream (1 minuto, 5 minuti, 10 minuti, mezz'ora..) e lo scorrere della finestra, cioè ogni quanto ripetere l'operazione. E' necessario che questi valori siano dei multipli del valore definito per il DStream, che rappresenta ogni quanto vengono presi i dati dallo stream e passati all'elaborazione.



Analisi streaming attraverso Window Operations

(fonte: https://spark.apache.org/docs/latest/streaming-programming-guide.html)

Per questo progetto i valori che sono stati usati sono:

- 10 secondi per il DStream
- una finestra da un minuto (prevalentemente usata per testare)
- una finestra da 5 minuti
- una finestra da 10 minuti
- una finestra da 30 minuti
- una finestra da un'ora.

Tutte le finestre sono configurate per essere rielaborate ad ogni minuto, al fine di ottenere un'analisi near real-time.

I risultati dei trend ottenuti per ogni finestra vengono salvati all'interno del database di Cassandra, con cui è stata inizialmente creata la sessione.

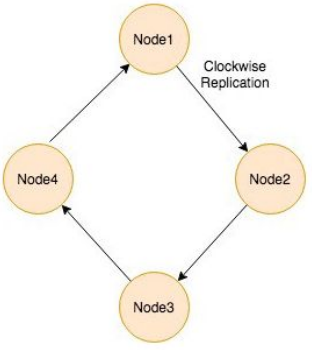
Al fine di poter interoperare tra Spark Streaming e Cassandra è stato necessario combinare l'uso di Spark Streaming con Spark Sql e l'utilizzo della libreria offerta dalla compagnia Datastax (quest'ultima è necessaria per connettere Spark e Cassandra).

Cassandra

Cassandra è un database noSql “column oriented”, cioè con l’organizzazione dei dati su colonne.
E’ stato scelto per conservare la grande mole di dati prodotti dallo SpeedLayer ma anche per le prestazioni e la scalabilità che offre. Cassandra infatti nasce per essere un database distribuito, dunque scalabile, con la possibilità di conservare ampie quantità di dati.

Una caratteristica fondamentale di questo database è che permette di specificare il TTL (“time to live”) per i valori inseriti all’interno di una column-family (cioè di una tabella). Al fine di conservare i dati per i tempi di elaborazione del Batch Layer, è stato scelto di impostare una durata di 24 ore per ogni record. Questo consente anche di tenere sotto controllo la dimensione del contenuto del database.

Per consentire la scalabilità, Cassandra viene instanziato attraverso un cluster, composto da nodi. Ogni nodo rappresenta una macchina (fisica o virtuale) in cui i dati sono salvati. Un insieme di nodi che si occupano degli stessi dati è chiamato datacenter. Per questo progetto è stato utilizzato un datacenter composto da 2 nodi.
Su un cluster deve essere definita una strategia di replicazione, configurabile all’inizializzazione del cluster, che decide su quali nodi i dati vanno ridondati. Cassandra mette a disposizione due tipologie di strategie: Network Topology Strategy e Simple Strategy. Per questo progetto è stata scelta la seconda perchè è indicata quando si ha un solo datacenter. Questa strategia pone la prima replica nel nodo selezionato dal “partitioner”, cioè il componente che determina come i dati sono distribuiti nel cluster. Dopo questa decisione, i restanti dati sono replicati in senso orario fra gli altri nodi del cluster.



Simple Strategy in un datacenter di Cassandra
(fonte: https://towardsdatascience.com/getting-started-with-apache-cassandra-and-python-81e00ccf17c9)

L’organizzazione dei dati all’interno del database è molto semplice. Tutte le operazioni di definizione dei dati sono state eseguite utilizzando “Cassandra Query Language”, un linguaggio molto simile all’SQL che consente di interagire facilmente con il cluster di Cassandra.
Come prima operazione è stata definito il keyspace “climate”, attraverso la seguente query:

```
CREATE KEYSPACE IF NOT EXISTS climate
WITH REPLICATION =
{ 'class' : 'SimpleStrategy', 'replication_factor' : 2 }
```

si noti come venga definita la strategia di replicazione e il numero di nodi su cui replicare.
A seguito sono state create le 2 column-family che mantengono i dati d’interesse raccolti. Il primo è la column family chiamata “antarctica” che ha il compito di conservare ogni record acquisito dallo Speed Layer, al fine di favorire l’interrogazione di questi dati in maniera libera:

```
CREATE TABLE IF NOT EXISTS antarctica
(timestamp int, temperature float, wind float, humidity float PRIMARY KEY(timestamp))
```

gli attributi che definiti per ogni valore sono:

- **timestamp**: istante di lettura del record (su cui viene definita una primary key in modo da identificare univocamente il record ed evitare duplicati)
- **temperature**: temperatura rilevata (in Kelvin)
- **wind**: velocità del vento in metri/secondo
- **humidity**: percentuale d’umidità
- **weather**: clima attuale dell’Antartide

La seconda column-family denominata “trends_view” invece rappresenta i risultati dell’elaborazione temporale svolta dallo Speed Layer:

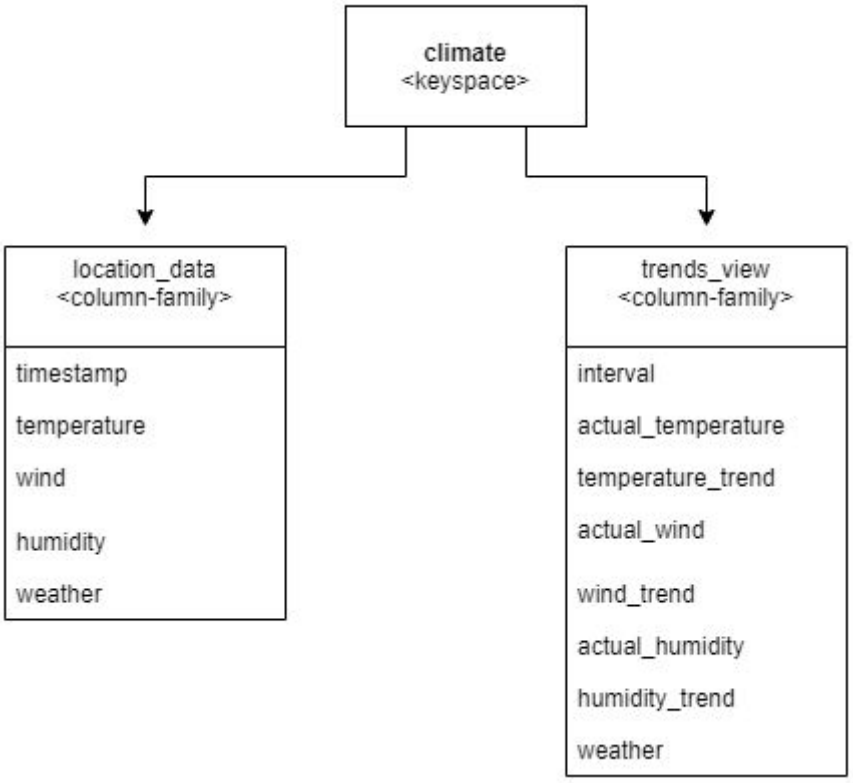
```
CREATE TABLE IF NOT EXISTS trends_view
(interval text, temperature_trend float, wind_trend float, humidity_trend float, weather text, PRIMARY KEY(interval))
```

Gli attributi definiti sono:

- **interval**: intervallo della finestra temporale su cui sono effettuate le analisi temporali (su cui è definita la chiave primaria. Ogni record dunque è univoco e viene aggiornato in caso di nuovi valori) Gli intervalli disponibili sono 60 (un minuto), 300 (cinque minuti), 1800 (mezz’ora), 3600 (un’ora)
- **temperature_trend :** incremento percentuale di temperatura durante l’intervallo di riferimento
- **wind_trend:** incremento percentuale della velocità del vento durante l’intervallo di riferimento

- **humidity_trend:** incremento percentuale dell'umidità durante l'intervallo di riferimento
- **weather** : clima attuale dell'Antartide

Durante un aggiornamento di questa column-family, è stato deciso di includere anche i valori della temperatura, della velocità vento e della percentuale d'umidità dell'istante in cui viene effettuato il calcolo del trend, al fine di migliorare le informazioni rese disponibili all'utente.
Segue uno schema riassuntivo della struttura del database:



Struttura di dati all'interno di Cassandra

Ambiente di esecuzione

Il sistema è stato realizzato su una macchina virtuale con sistema operativo Ubuntu 19.10, a 64 bit. La macchina ha a disposizione 16 GB di memoria RAM, un processore quadcore Intel i5-7600 e uno storage di 30 GB.
Sulla macchina è stato installato Hadoop (versione 3.2.1) per l'uso di Spark (versione 3.0).
Al fine di mantenere l'ambiente d'esecuzione il più pulito possibile, è stato scelto l'uso di Docker per definire due contenitori, contenenti il cluster di Cassandra e il canale di comunicazione di Kafka.
La versione di Python installata e utilizzata è la 3.7.5.

Al fine di avviare facilmente il progetto, è sono stati creati due file: il primo è il docker-compose.yml che consente di instanziare i due contenitori docker descritti in precedenza semplicemente invocando il comando “docker-compose up” mentre il secondo è uno script shell denominato “analysis.sh” per l’esecuzione in automatico degli script relativi allo speed layer (inizializzazione del producer, del consumer e del processo di spark streaming.)

In seguito, l’ambiente è stato replicato su **Amazon Web Services**. Questo ha richiesto l’uso di due macchine virtuali:

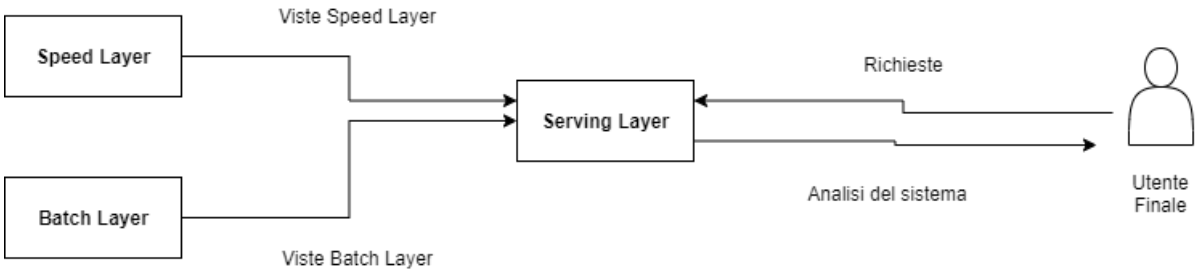
- Per avviare lo script producer, i contenitori docker e il serving layer è stata usata una macchina ec2 **m5ad.large**, su cui è stato installato Ubuntu, Docker e Python.
- Per l’elaborazione dei dati attraverso PySpark è stata usata una istanza elastic map reduce **emr-6.1.0** (l’unica che attualmente offre hadoop 3.2.1 e Spark versione 3.0.0 installati), con un cluster composto da 3 nodi.

Dopo alcune sperimentazioni, è stato registrato che il costo per la prima macchina è di circa *0,20 euro/ora* mentre per la seconda è di *8,24 euro/ora*.

6. Serving Layer

Il Serving Layer di un'architettura lambda si occupa di raccogliere le analisi prodotte dal Batch e dallo Speed Layer per poterle esporre all'utenza del sistema ad una latenza molto bassa. Generalmente, questo strato è implementato attraverso un database che ha il compito di gestire le viste materializzate degli altri due strati, cioè i risultati delle elaborazioni sui dati. E' importante sottolineare che questo layer ha unicamente questo compito, per cui l'aggiornamento dei dati è a carico degli altri componenti del sistema.

Le viste relative alle analisi prodotte dal Batch Layer rappresenta l'*insight* su tutti i dati disponibili dal sistema eccetto gli ultimi analizzati. Questo comporta che i risultati forniti sono sempre obsoleti a causa dell'alta latenza che caratterizza i processi di analisi batch. Ciò però non rappresenta un problema grazie allo Speed Layer, che è responsabile di fornire tutti i dati che non sono ancora disponibili all'interno dello storage del Batch.



Schema generale del Serving Layer

Al fine di garantire una bassa latenza nell'erogare i risultati e di evitare fallimenti, il Serving Layer viene distribuito su vari nodi per garantire la scalabilità del sistema.
Le varie view disponibili in questo strato vengono indicizzate per ottimizzare due metriche di performance: la latenza (quanto tempo è richiesto per rispondere ad una query) e il throughput (cioè il numero di query che possono essere erogate in un fissato intervallo). Un altro aspetto importante è che queste view sono denormalizzate rispetto al master dataset contenuto nel Batch Layer. Ciò consente di evitare query con aggregazioni complesse senza però pagare il prezzo della ridondanza, dato che questi risultati vengono aggiornati unicamente sull'ultimo strato della catena. In caso ci siano errori nel calcolo di una di queste view è possibile correggerli facilmente effettuando nuovamente una computazione del master dataset, mentre lo speed layer aggiorna automaticamente i vari errori.
Segue ora una descrizione dell'implementazione del Serving Layer di questo progetto.

.6.1 Dettagli Implementativi

Il Serving Layer di questo progetto è stato realizzato mediante il micro-framework "*Flask*" di Python. Questa libreria mette a disposizione un web server e delle API per la definizione di servizi.

Un utente interessato all'analisi dei dati climatici raccolti ha due modalità di fruizione delle viste prodotte dai due Layer:

- Tramite l'accesso al web server può visualizzare l'interfaccia grafica in cui i le viste materializzate sono presentate in tabelle
- Tramite l'invocazione dei servizi definiti è possibile accedere ai file json contenenti le viste. Questo consente di rendere il progetto interoperabile con sistemi esterni.

I servizi REST definiti sono elencati di seguito:

- **/climate/historical/measures** : restituisce le viste materializzate dal Batch Layer, come mostrato nella seguente immagine:

Antarctica Climate Historical Measures

Current Season: winter

Measure	Value	Measure's related year
Max Temperature	-35.3	in the winter of **2007**
Min Temperature	-67.61	in the winter of **1982**
Max Wind Speed	4.6250448208220754E18	in the winter of **1992**
Min Wind Speed	4.5978149315750866E18	in the winter of **2004**
Max Humidity	82.0	in the winter of **2006**
Min Humidity	24.0	in the winter of **2017**

- **/climate/historical/trends** : restituisce le viste materializzate circa analisi temporali effettuate sui dati storici, che includono anche i dati recentemente importati nel sistema

Antarctica Climate Historical Trends - Current Season: winter

Season	Year	Measure	Value	Trend
winter	2017	Max Temperature	-27.85	0.0
winter	2018	Max Temperature	-29.64	-6.43
winter	2019	Max Temperature	-33.37	-12.58
winter	2020	Max Temperature	-25.09	24.81
winter	2017	Min Temperature	-64.2	0.0
winter	2018	Min Temperature	-61.16	4.74
winter	2019	Min Temperature	-65.1	-6.44
winter	2020	Min Temperature	-54.54	16.22
winter	2017	AVG Temperature	-48.09	0.0
winter	2018	AVG Temperature	-45.61	5.16
winter	2019	AVG Temperature	-47.47	-4.08
winter	2020	AVG Temperature	-43.37	8.64

- **/climate/test** : mostra gli ultimi dati raccolti dallo Speed Layer, sotto forma di JSON


```
{"humidity":5.480000019073486,"location":"'Antarctica'","temperature":222.92999267578125,"time":1601052933,"weather":"'Clouds'","wind":52.0},
{"humidity":5.480000019073486,"location":"'Antarctica'","temperature":222.92999267578125,"time":1601052938,"weather":"'Clouds'","wind":52.0},
{"humidity":5.480000019073486,"location":"'Antarctica'","temperature":222.92999267578125,"time":1601052948,"weather":"'Clouds'","wind":52.0},
{"humidity":5.849999904632568,"location":"'Antarctica'","temperature":221.97999572753906,"time":1601066429,"weather":"'Clouds'","wind":46.0},
{"humidity":5.849999904632568,"location":"'Antarctica'","temperature":221.97999572753906,"time":1601066435,"weather":"'Clouds'","wind":46.0},
{"humidity":5.849999904632568,"location":"'Antarctica'","temperature":221.97999572753906,"time":1601066439,"weather":"'Clouds'","wind":46.0},
{"humidity":5.849999904632568,"location":"'Antarctica'","temperature":221.97999572753906,"time":1601066444,"weather":"'Clouds'","wind":46.0},
{"humidity":5.849999904632568,"location":"'Antarctica'","temperature":221.97999572753906,"time":1601066449,"weather":"'Clouds'","wind":46.0},
```


- **/climate/view** : mostra le view calcolate dallo Speed Layer. Questi dati vengono presentati all'utente attraverso una tabella:

Antarctica Climate Trends: 2020-09-26 19:18:34.293290

Location	Interval	Actual Temperature	Temperature Trend	Actual Wind	Wind Trend	Actual Humidity	Humidity Trend	Weather
('Arctic',)	30 m	295.1499938964844	0.0 %	0.45 m/s	0.0 %	79.0 %	0.0 %	'Clear'
('Arctic',)	6 h	295.1499938964844	0.0 %	0.45 m/s	0.0 %	79.0 %	0.0 %	'Clear'
('Arctic',)	5 m	295.1499938964844	0.0 %	0.45 m/s	0.0 %	79.0 %	0.0 %	'Clear'
('Arctic',)	1 h	293.32000732421875	0.25 %	4.24 m/s	5.21 %	76.0 %	-1.0 %	'Clear'
('Arctic',)	1 m	295.1499938964844	0.0 %	0.45 m/s	0.0 %	79.0 %	0.0 %	'Clear'
('Arctic',)	10 m	295.1499938964844	0.0 %	0.45 m/s	0.0 %	79.0 %	0.0 %	'Clear'
('Antarctica',)	30 m	225.6999969482422	0.0 %	5.42 m/s	0.0 %	62.0 %	0.0 %	'Clear'
('Antarctica',)	6 h	225.6999969482422	0.0 %	5.42 m/s	0.0 %	62.0 %	0.0 %	'Clear'
('Antarctica',)	5 m	225.6999969482422	0.0 %	5.42 m/s	0.0 %	62.0 %	0.0 %	'Clear'
('Antarctica',)	1 h	221.97999572753906	0.0 %	5.85 m/s	0.0 %	46.0 %	0.0 %	'Clouds'
('Antarctica',)	1 m	225.6999969482422	0.0 %	5.42 m/s	0.0 %	62.0 %	0.0 %	'Clear'
('Antarctica',)	10 m	225.6999969482422	0.0 %	5.42 m/s	0.0 %	62.0 %	0.0 %	'Clear'

- **/climate/view/json** : restituisce il file JSON delle analisi temporali dello Speed Layer


```
"[{\"interval\": \"30 m\", \"actual_temperature\": 295.1499938964844, \"temperature_trend\": 0.0, \"actual_wind\": 0.45, \"wind_trend\": 0.0, \"actual_humidity\": 79.0, \"humidity_trend\": 0.0, \"location\": [\"Arctic\"], \"weather\": \"Clear\"}, {\"interval\": \"6 h\", \"actual_temperature\":
295.1499938964844, \"temperature_trend\": 0.0, \"actual_wind\": 0.45, \"wind_trend\": 0.0, \"actual_humidity\": 79.0, \"humidity_trend\": 0.0, \"location\": [\"Arctic\"], \"weather\": \"Clear\"}, {\"interval\": \"5 m\", \"actual_temperature\": 295.1499938964844, \"temperature_trend\": 0.0, \"actual_wind\":
0.45, \"wind_trend\": 0.0, \"actual_humidity\": 79.0, \"humidity_trend\": 0.0, \"location\": [\"Arctic\"], \"weather\": \"Clear\"}, {\"interval\": \"1 h\", \"actual_temperature\": 293.32000732421875, \"temperature_trend\": 0.25, \"actual_wind\": 4.24, \"wind_trend\": 5.21, \"actual_humidity\": 76.0,
\"humidity_trend\": -1.0, \"location\": [\"Arctic\"], \"weather\": \"Clear\"}, {\"interval\": \"1 m\", \"actual_temperature\": 295.1499938964844, \"temperature_trend\": 0.0, \"actual_wind\": 0.45, \"wind_trend\": 0.0, \"actual_humidity\": 79.0, \"humidity_trend\": 0.0, \"location\": [\"Arctic\"], \"weather\":
\"Clear\"}, {\"interval\": \"10 m\", \"actual_temperature\": 295.1499938964844, \"temperature_trend\": 0.0, \"actual_wind\": 0.45, \"wind_trend\": 0.0, \"actual_humidity\": 79.0, \"humidity_trend\": 0.0, \"location\": [\"Arctic\"], \"weather\": \"Clear\"}, {\"interval\": \"30 m\", \"actual_temperature\":
225.6999969482422, \"temperature_trend\": 0.0, \"actual_wind\": 5.42, \"wind_trend\": 0.0, \"actual_humidity\": 62.0, \"humidity_trend\": 0.0, \"location\": [\"Antarctica\"], \"weather\": \"Clear\"}, {\"interval\": \"6 h\", \"actual_temperature\": 225.6999969482422, \"temperature_trend\": 0.0,
\"actual_wind\": 5.42, \"wind_trend\": 0.0, \"actual_humidity\": 62.0, \"humidity_trend\": 0.0, \"location\": [\"Antarctica\"], \"weather\": \"Clear\"}, {\"interval\": \"5 m\", \"actual_temperature\": 225.6999969482422, \"temperature_trend\": 0.0, \"actual_wind\": 5.42, \"wind_trend\": 0.0, \"actual_humidity\":
62.0, \"humidity_trend\": 0.0, \"location\": [\"Antarctica\"], \"weather\": \"Clear\"}, {\"interval\": \"1 h\", \"actual_temperature\": 221.97999572753906, \"temperature_trend\": 0.0, \"actual_wind\": 5.85, \"wind_trend\": 0.0, \"actual_humidity\": 46.0, \"humidity_trend\": 0.0, \"location\": [\"Antarctica\"],
\"weather\": \"Clouds\"}, {\"interval\": \"1 m\", \"actual_temperature\": 225.6999969482422, \"temperature_trend\": 0.0, \"actual_wind\": 5.42, \"wind_trend\": 0.0, \"actual_humidity\": 62.0, \"humidity_trend\": 0.0, \"location\": [\"Antarctica\"], \"weather\": \"Clear\"}, {\"interval\": \"10 m\",
\"actual_temperature\": 225.6999969482422, \"temperature_trend\": 0.0, \"actual_wind\": 5.42, \"wind_trend\": 0.0, \"actual_humidity\": 62.0, \"humidity_trend\": 0.0, \"location\": [\"Antarctica\"], \"weather\": \"Clear\"}]"
```


Questo layer consente anche di realizzare un sistema di monitoraggio delle temperature dell'Antartide: è infatti possibile incrociare i risultati ottenuti dalle elaborazioni dei due layer per controllare se i valori attualmente registrati sono in linea con quelli disponibili nei dati storici. E' possibile infatti sviluppare un sistema di controllo sugli ultimi dati raccolti in modo da notificare un utente (via e-mail o semplicemente attraverso la web app) in caso i valori siano anomali.

.7 Approfondimento: Architettura Kappa

Come approfondimento di questo progetto, si è deciso di personalizzare lo Speed Layer al fine di ottenere un'architettura Kappa. Questo passaggio è stato deciso anche per la disponibilità limitata dei dati climatici storici in nostro possesso (attualmente solo dell'Antartide).

L'obiettivo per questa fase è stato quello di espandere il sistema all'analisi near-real-time di più zone. Nella pratica si è scelto di monitorare anche l'Artide.

Un'architettura Kappa si concentra solamente sull'analisi dei dati come flusso. L'idea è quella di gestire il flusso dei dati da Kafka rielaborando continuamente i risultati prodotti. L'architettura Kappa, al contrario della Lambda, non dispone di un Batch Layer, quindi è ideale per il monitoraggio dei dati, e mira a semplificare il sistema ad un'unica base di codice (solo quella dello Speed Layer).
E' stato possibile estrapolare l'architettura Kappa in quanto il flusso d'input è facilmente manipolabile in modo da ottenere l'analisi su più luoghi.

Le modifiche apportate al sistema in realtà non sono state molte: è stato necessario creare più istanze di producer Kafka, ognuna associata ad un luogo differente. Questi producono dati che vengono inviati all'interno del topic Kafka, dichiarando per ogni record, il luogo associato. Nello streaming processor, è stato necessario ridurre i valori all'interno delle finestre temporali mediante il campo con il nominativo del luogo, inserito in testa in ogni record processato dal sistema.

Per quanto riguarda Cassandra, la modifica necessaria è stata quella di dover modificare le column family in modo da avere una primary key composta da due valori: nei trend temporali, ogni record è identificato univocamente dalla coppia "*intervallo temporale - luogo*".

I risultati ottenuti da questa nuova architettura sono facilmente esplorabili mediante il Serving Layer già realizzato.

Va sottolineato che questa nuova architettura non è confinata all'analisi di 2 luoghi ma è possibile espandere il monitoraggio dell'analisi temporale dei valori di temperatura, velocità del vento e umidità a svariati luoghi. L'unica modifica necessariamente richiesta è quella di inserire nuovi producer relativi ai nuovi luoghi. E' possibile dunque affermare che questo

nuovo sistema è altamente scalabile e flessibile rispetto a quello precedentemente realizzato, con la lacuna però della mancanza delle analisi su periodi superiori alle 24 ore (non disponendo del Batch Layer).

Conclusioni e Sviluppi futuri

In conclusione, è possibile affermare che il progetto ha soddisfatto pienamente gli obiettivi preposti: sono state sperimentare nuove tecnologie come i database noSQL Cassandra e HBase e si è potuto seguire passo passo l'implementazione dell'intera architettura Lambda, dalla raccolta dei dati fino alla presentazione dei risultati all'utente. Il sistema è stato sviluppato sia in locale che sugli Amazon Web Services. E' stato impiegato anche l'uso di Docker.

Le più grandi difficoltà sono emerse soprattutto nel cercare i connettori adatti ai sistemi noSQL impiegati, in quanto questi non sono pienamente compatibili con la versione Python di Spark. Nonostante ciò sono state trovate delle soluzioni originali sia per il Batch Layer che per lo Speed Layer.

L'architettura realizzata rappresenta un ottimo esempio di sistema di monitoraggio: è infatti possibile analizzare i dati attuali in relazione a quelli prodotti dall'analisi storica. Questo però non è vincolato al solo luogo analizzato (Antartide) ma è facilmente espandibile a più zone, come mostrato nell'approfondimento dell'architettura Kappa. Se per ogni zona fosse disponibile il relativo dataset storico, sarebbe possibile monitorare con efficacia varie zone, cioè con la possibilità di incrociare i dati attuali con quelli storici.

Come ulteriori sviluppi futuri si propone un'analisi più approfondita dei dati storici, attraverso tecniche di Machine Learning. In questo modo è infatti possibile scoprire valori fuori dalla norma, indice di periodi particolari, e poter salvare questi risultati per poter confrontare i dati attualmente registrati nel sistema. In questo modo è possibile estendere il monitoraggio attuale con un confronto a scenari di crisi climatiche.

Bibliografia

- Big Data: Principles and best practices of scalable realtime data systems, Marz N., Warren J. (2015)

Sitografia

- https://openweathermap.org/ (Open Weather)
- http://lambda-architecture.net/ (Architettura Lambda)
- https://kafka.apache.org/quickstart (Kafka)
- https://spark.apache.org/docs/latest/sql-programming-guide.html (Spark SQL)
- https://diogoalexandrefranco.github.io/interacting-with-hbase-from-pyspark/
- https://hbase.apache.org/book.html (HBase)
- https://spark.apache.org/docs/latest/streaming-programming-guide.html
- https://towardsdatascience.com/getting-started-with-apache-cassandra-and-python-81e00ccf17c9 (Cassandra & Docker)
- https://www.talend.com/it/blog/2017/08/28/lambda-kappa-real-time-big-data-architectures/ (Architettura Kappa)
- https://hazelcast.com/glossary/kappa-architecture/ (Architettura Kappa)