

Progetto di Sistemi Distribuiti e Cloud Computing

Simone Squillace, 235129

*Dipartimento di Ingegneria Informatica, Modellistica, Elettronica e Sistemistica,
Università della Calabria, Via Pietro Bucci, 87036, Arcavacata di Rende, CS, Italia*

21 maggio 2022

1 INTRODUZIONE

Negli ultimi anni, con la crescita esponenziale nell'uso dei social media (recensioni, forum, discussioni, blog e social network), le persone e le aziende utilizzano sempre più le informazioni (opinioni e preferenze) pubblicate in questi mezzi per il loro processo decisionale. Tuttavia, il monitoraggio e la ricerca di opinioni sul Web da parte di un utente o azienda risulta essere un problema molto arduo a causa della proliferazione di migliaia di siti; in più ogni sito contiene un enorme volume di testo non sempre decifrabile in maniera ottimale (pensiamo ai lunghi messaggi di forum e blog). Risulta quindi necessario l'utilizzo di sistemi automatizzati di raccolta e analisi di informazioni, per superare limiti gestionali, al fine di giungere ad una metodologia di analisi il più semplice possibile.

Il lavoro svolto durante questo progetto mira a progettare e implementare un sistema distribuito per la visualizzazione su mappa di grandi moli di tweet geolocalizzati, con possibilità per l'utente di impostare delle query per il filtraggio dei dati (es. per tag, lingua o per parola contenuta).

2 ANALISI E SPECIFICA DEI REQUISITI

L'analisi dei requisiti è un'attività preliminare allo sviluppo (o alla modifica) di un sistema software, il cui scopo è quello di identificare e documentare i requisiti che il nuovo prodotto (o il prodotto modificato) deve offrire, ovvero i requisiti che devono essere necessariamente soddisfatti dal software al termine dello sviluppo. Si tratta una fase presente essenzialmente in tutti i modelli di ciclo di vita di un software, pur con diverse enfasi e diverse connotazioni: esistono diversi modelli di processo di produzione del software, l'analisi dei requisiti si interfaccia in maniera diversa con le altre fasi in base al modello scelto.

Questa costituisce la prima fase di un progetto per lo sviluppo di un software, viene effettuata a valle di uno studio di fattibilità per definire i costi ed i benefici della realizzazione del sistema. A tale fase partecipano: il cliente, specialisti di analisi di mercato, membri del team di sviluppo, detti stake holder (coloro che hanno un interesse nel sistema e che saranno responsabili della sua accettazione). Nel caso in cui i requisiti non fossero chiari, sono necessarie più interazioni tra cliente e sviluppatore. I requisiti dovrebbero essere espressi in una notazione che sia al contempo comprensibile dall'utente finale e rigorosa nella loro formalizzazione, alcune metodologie ritengono che in questa fase debbano essere progettati i test a cui sottoporre il sistema finale.

Il primo passo è l'analisi dei requisiti, trattati distinguendo quelli funzionali e non funzionali dell'applicazione, in seguito alla definizione degli stakeholder.

2.1 Stakeholders principali

Dalla traccia di progetto non emerge dettagliatamente chi sia l'utente finale, posso assumere che si tratti di un utente generico.

- **Utente generico:** colui che accede al servizio per usufruire di tutte le funzionalità e le caratteristiche disponibili che esso ha da offrire, quindi effettua ricerche dinamiche e arbitrarie sui tweets, con la possibilità di salvarle e organizzarle in diverse viste.

2.2 Analisi dei requisiti funzionali

Dalla traccia di progetto emerge che l'applicazione deve presentare alcune funzionalità obbligatorie:

- Visualizzazione su mappa di grandi moli di tweet geolocalizzati.
- Possibilità per l'utente di impostare delle query per il filtraggio dei dati (es. per tag, lingua o per parola contenuta).
- Possibilità di accedere ad una dashboard di default, con widget e query preimpostate.
- Per la realizzazione del sistema si dovrà utilizzare la dashboard interattiva "Kibana".
- I dati dovranno essere importati in batch da file e memorizzati su database NoSQL (es. Elasticsearch).
- Il sistema realizzato dovrà utilizzare le soluzioni di calcolo, storage e virtualizzazione messe a disposizione da Docker Engine.

In particolare, lo sviluppo del sistema distribuito prevede due principali focus riguardanti i dati, dai quali derivano differenti obiettivi, fondamentali da raggiungere::

- **Raccolta**, i tweet ottenuti dall'esterno devono essere filtrati e memorizzati, per consentirne la visualizzazione. Inoltre, trattandosi di informazioni conservate nel tempo, necessitano di essere raccolte in modo statico e permanente, oltre che sicuro.
- **Analisi**, tutti i dati memorizzati nel sistema devono poi essere analizzati in modo coerente per creare statistiche ed avere informazioni aggiuntive utili in un quadro generale più ampio; i tweet sono utili agli utenti generici per monitorare le attività all'interno della rete, mentre le query verso il database sono essenziali per creare viste personalizzate.

2.3 Analisi dei requisiti non funzionali

Di seguito è riportato l'elenco dei requisiti non funzionali del sistema:

- **Usabilità:** l'interfaccia utente del sistema deve essere implementata cercando di garantire la massima operabilità, un veloce apprendimento e una facile localizzazione dei comandi da utilizzare. Essa deve essere interamente messa a disposizione da *Kibana*, da cui l'utente generico può accedere a tutte le funzionalità offerte.
- **Visualizzazione:** l'applicazione deve essere in grado di permettere all'utente di cambiare il layout della dashboard, offrendo diversi modi per rappresentare i dati ottenuti dalle query.
- **Prestazioni:** l'applicazione deve garantire minimi tempi di latenza, sia per il caricamento dei dati, sia per l'esecuzione delle query. Come vedremo più avanti in dettaglio, ciò è stato ottenuto utilizzando un database NoSQL con lo stack ELK (Elasticsearch, Logstash e Kibana).
- **Temporalità:** l'applicazione deve essere consegnata almeno un giorno prima dalla data prevista.

3 PROGETTAZIONE

L’architettura software è la struttura del sistema, costituita dalle parti del sistema, dalle relazioni tra le parti e dalle loro proprietà visibili. La struttura definisce, tra l’altro, la scomposizione del sistema in sottosistemi dotati di un’interfaccia e le interazioni tra essi, che avvengono attraverso le interfacce. Le proprietà visibili di un sottosistema definiscono le assunzioni che gli altri sottosistemi possono fare su di esso, come servizi forniti, prestazioni, uso di risorse condivise, trattamento di malfunzionamenti, ecc. La precisazione di considerare solo le proprietà visibili aiuta a chiarire la differenza tra progettazione architettonica e progettazione di dettaglio: solo in quest’ultima, infatti, ci si occupa degli aspetti “non visibili” dei sottosistemi, quali ad esempio strutture dati o algoritmi utilizzati per la loro realizzazione.

Tuttavia, come accennato nella sezione precedente, lo sviluppo software verte sull’utilizzo di tecnologie già esistenti, di cui non è necessario approfondire in dettaglio l’implementazione concreta.

3.1 Formato dei dati

Prima di entrare nel merito dello stack di sviluppo, è necessario descrivere il formato dei dati che l’applicazione deve gestire e, in particolare, quello del [dataset](#) messo a disposizione per lo sviluppo.

Il dataset è composto da file json, in cui ogni riga rappresenta un tweet, di cui di seguito è riportato un esempio, troncato per ovvie ragioni.

```
{
    "created_at": "Wed Sep 30 23:59:56 +0000 2020",
    "id": 1311455770983235584,
    "full_text": "RT @LindseyGrahamSC: Great news for South...",
    "entities": {
        "user_mentions": [
            {
                "screen_name": "LindseyGrahamSC",
                "name": "Lindsey Graham",
                "id": 432895323,
                ...
            }
        ],
        ...
    },
    "user": {
        "id": 1297872748279214081,
        "name": "Korvo",
        "screen_name": "faryadehazard",
        "location": "California, US",
        ...
    },
    "geo": null,
    "coordinates": null,
    ...
}
```

Dalle proprietà *geo* e *coordinates* emerge che il tweet d'esempio non è localizzato, pertanto non è utile per una rappresentazione geolocalizzata. Sfortunatamente, ciò accade per la gran parte dei tweets presenti nel dataset. Per cui, come suggerito dai docenti, la proprietà *coordinates* è stata valorizzata con parametri di longitudine e latitudine generati casualmente nell'area degli Stati Uniti, come riportato nell'esempio sottostante.

```
"coordinates": {
    "lat": 48.1030850548015,
    "lon": -88.56499398440278
}
```

Questo è stato implementato grazie alla scrittura di un semplice script in python, riportato di seguito, che fa uso della libreria *shapely* per caricare i poligoni che delimitano l'area degli Stati Uniti, della libreria *random* per generare numeri casuali e della libreria *json* per leggere gli omonimi file.

```
import random
import json
from shapely.geometry import Point, shape

def add_location(boundaries_file, data_file):
    geojson_file = json.loads(open(boundaries_file).read())
    input_file = open(data_file)
    output_file = open(data_file.replace(
        ".json", "") + "_USA.json", 'w')

    poly = shape(geojson_file['features'][238]['geometry'])
    min_x, min_y, max_x, max_y = poly.bounds

    for line in input_file.readlines():
        entry = json.loads(line)
        while True:
            random_point = Point(
                [random.uniform(min_x, max_x),
                 random.uniform(min_y, max_y)])
            if (random_point.within(poly)):
                entry['coordinates'] = {
                    'lat': random_point.y, 'lon': random_point.x}
                print(random_point)
                break
        output_file.write(json.dumps(entry) + "\n")

add_location('land-boundaries.geojson',
             'us-presidential-tweet-id-2020-10-01-04_hydrated.json')
```

Il file [land-boundaries.geojson](#) è un tipo di json particolare *geojson*, formattato in modo tale conservare dati geografici. In questo caso, contiene tutti i poligoni relativi ai confini geografici dei paesi. L'algoritmo considera il numero 238, che corrisponde agli USA e genera le coordinate geografiche al suo interno.

3.2 Lo stack software ELK

I framework scelti per svolgere l'analisi dei tweets sono Elasticsearch, Logstash e Kibana. Essi consentono di effettuare analisi di tipo descrittivo tramite il cosiddetto “log mining”, ossia l'operazione di estrazione di informazione e di conoscenza presenti nei file di log, o in questo caso dei tweet.

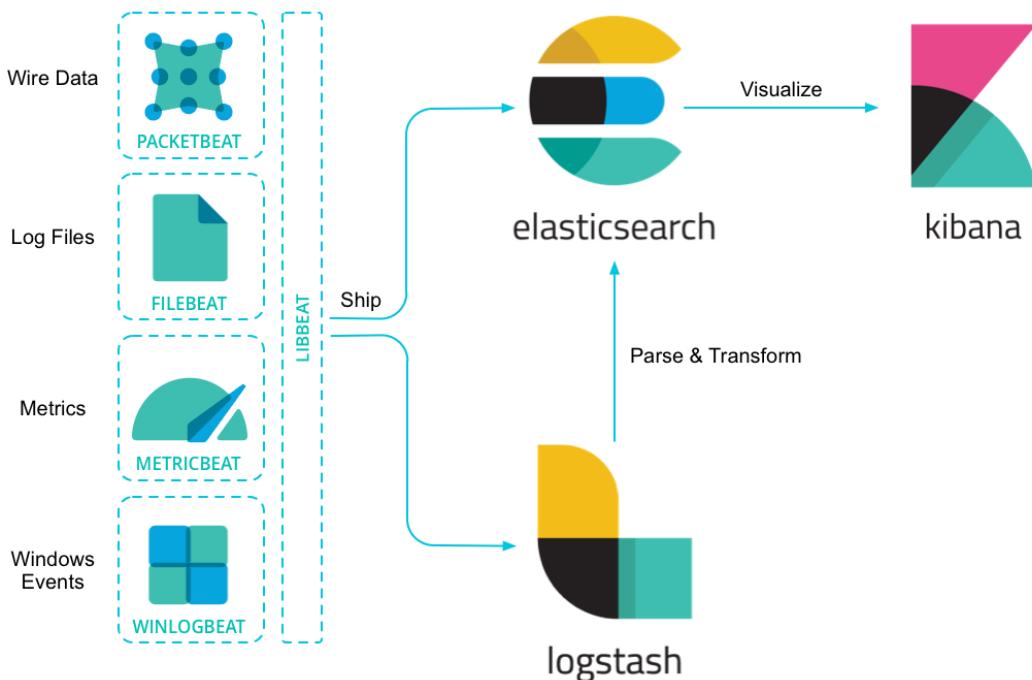


Figura 1. Raccolta dei dati da Logstash e Elasticsearch

Questi tre potenti strumenti sono, da qualche anno, molto utilizzati nel panorama aziendale internazionale sia per l'analisi real-time e sia per quella statica dei dati, in dettaglio distinguiamo:

- **Elasticsearch** è un database NoSQL basato su Lucene, con capacità Full Text e con supporto ad architetture distribuite. Tutte le funzionalità sono nativamente esposte tramite interfaccia RESTful (REpresentational State Transfer, un tipo di architettura software usata di norma su HTTP), mentre le informazioni sono gestite come documenti JSON. Elasticsearch è, inoltre, un motore di ricerca ed analisi open source altamente scalabile che consente di immagazzinare, ricercare ed analizzare grandi volumi di dati in tempo quasi reale. Spesso abbreviato come ES, Elasticsearch è generalmente collocato alla base di applicazioni che richiedono ricerche e requisiti complessi, per poterle sostenere.
- **Logstash** è un motore di raccolta dati open source con funzionalità di pipelining in tempo reale. Logstash può unificare dinamicamente i dati provenienti da fonti diverse e normalizzare i dati così definiti verso diverse destinazioni scelte ad hoc. Esso è utile anche per ripulire tutti i dati, per eseguire diverse analisi avanzate a valle, e per casi in cui è necessaria una corretta visualizzazione del dato. Mentre Logstash originariamente ha guidato l'innovazione nella raccolta di log, le sue capacità di

elaborazione e trattamento di dati si estendono ben oltre questo. Ogni tipo di evento può essere arricchito e trasformato con una vasta gamma di plugin di input, filtri ed output, con molti codec nativi che semplificano ulteriormente il processo di immagazzinamento. Logstash accelera le conoscenze dei dataset sfruttando un maggior volume e una maggiore varietà di dati.

- **Kibana** è una piattaforma di analisi e visualizzazione open source progettata per lavorare con Elasticsearch. Kibana è utilizzato principalmente per cercare, visualizzare e interagire con i dati memorizzati negli indici Elasticsearch. È possibile eseguire facilmente analisi di dati avanzate e visualizzare i dati in una varietà di tabelle, istogrammi e mappe.

La raccolta dei dati è affidata a Logstash tramite la definizione di una *pipeline*. La pipeline di elaborazione degli eventi Logstash ha tre fasi: input → filtri → output. Gli input generano eventi, i filtri li modificano e gli output li spediscono altrove. I filtri sono dispositivi di elaborazione intermedi nella pipeline Logstash, si possono combinare filtri con espressioni condizionali per eseguire un’azione su un evento se soddisfa determinati criteri. Input e output supportano i codec che consentono di codificare o decodificare i dati quando entrano o escono dalla pipeline senza dover utilizzare un filtro separato.

La pipeline costruita in questo caso è abbastanza semplice. Essa è stata definita nel file di configurazione di Logstash `logstash.conf`¹ nel seguente modo:

```

input {
    tcp {
        port => 5000
    }
}

filter {
    json {
        source => "message"
    }
}

output {
    elasticsearch {
        hosts => "elasticsearch:9200"
        user => "logstash_internal"
        password => "${LOGSTASH_INTERNAL_PASSWORD}"
        index => "tweets"
    }
}

```

Logstash accetta in input sulla porta 5000 del protocollo tcp file json, di cui viene filtrato il campo *message*, e li salva nell’index *tweets* di elasticsearch.

¹ <https://github.com/thatsimo/sdcc-project-2021/blob/main/docker-elk/logstash/pipeline/logstash.conf>

3.3 Architettura a "microservizi"

L'idea principale è quella di distinguere le funzioni di *data processing* (Logstash), *storage* (Elasticsearch) e *visualize* (Kibana) in servizi autonomi e, pertanto, si è scelto di utilizzare una particolare architettura, detta a **microservizi**: si tratta di un approccio per sviluppare e organizzare l'architettura dei software secondo cui quest'ultimi sono composti di servizi indipendenti di piccole dimensioni che comunicano tra loro tramite API ben definite. Questi servizi sono controllati da piccoli team autonomi.

Le architetture dei microservizi permettono di scalare e sviluppare le applicazioni in modo più rapido e semplice, permettendo di promuovere l'innovazione e accelerare il time-to-market di nuove funzionalità. Questo perché, con un'architettura basata su microservizi, un'applicazione è realizzata da componenti indipendenti che eseguono ciascun processo applicativo come un servizio. Tali servizi comunicano attraverso un'interfaccia ben definita che utilizza API leggere. I servizi sono realizzati per le funzioni aziendali e ogni servizio esegue una sola funzione. Poiché eseguito in modo indipendente, ciascun servizio può essere aggiornato, distribuito e ridimensionato per rispondere alla richiesta di funzioni specifiche di un'applicazione.

Ciascun servizio nell'architettura basata su microservizi può essere sviluppato, distribuito, eseguito e ridimensionato senza influenzare il funzionamento degli altri componenti. I servizi non devono condividere alcun codice o implementazione con gli altri. Qualsiasi comunicazione tra i componenti individuali avviene attraverso API ben definite. Ciascun servizio è progettato per una serie di capacità e si concentra sulla risoluzione di un problema specifico. Se, nel tempo, gli sviluppatori aggiungono del codice aggiuntivo a un servizio rendendolo più complesso, il servizio può essere scomposto in servizi più piccoli.

La collaborazione è data dalla possibilità che, per portare a termine una richiesta, un servizio possa a sua volta inviare ulteriori richieste ad altri servizi. A seconda dei casi quindi, ogni microservizio può essere visto come consumatore o fornitore. L'architettura può essere vista come un'evoluzione della Service Oriented Architecture (SOA), della quale preserva la natura distribuita, rendendola però più decentralizzata e partizionata nella gestione della logica applicativa.

La tecnologia della containerizzazione delle applicazioni si presta particolarmente allo sviluppo di microservizi e sta prendendo piede sempre più velocemente nel mondo dell'informatica, tanto che molte aziende la considerano una prerogativa del loro processo di sviluppo. La pratica che garantisce la corretta comunicazione tra container di microservizi è detta *Container Orchestration*, che come vedremo più avanti sarà implementata con l'utilizzo di Docker Compose.

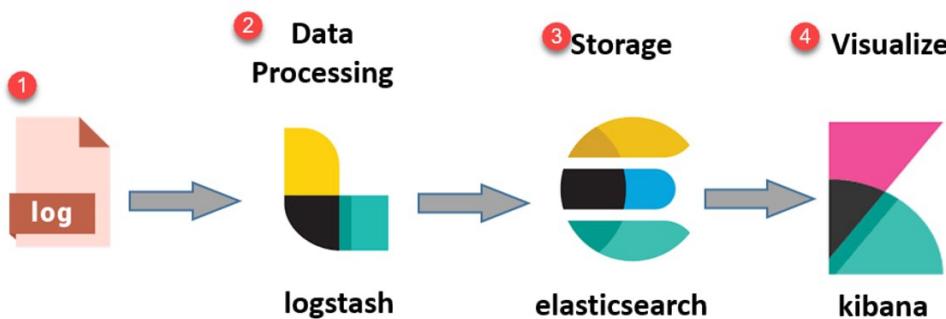


Figura 2. Classica pipeline di dati con ELK Stack

4 IMPLEMENTAZIONE

È necessario mettere in campo sistemi che permettano la gestione dei servizi, come la gestione delle configurazioni, e l'orchestrazione delle istanze in modo da automatizzare il ciclo di vita dei servizi, ad esempio tramite *Docker Compose*.

4.1 Docker e Docker Compose

Docker è una piattaforma software che permette di semplificare il deploy e lo sviluppo di un'applicazione integrando in un unico pacchetto, detto container, tutto il software e le dipendenze necessarie al corretto funzionamento del software.

Precedentemente, la soluzione più comune era rappresentata dall'impacchettamento del software in un'istanza di una macchina virtuale che però comporta alcuni problemi: innanzitutto è necessario gestire e deployare le macchine virtuali ed inoltre molte risorse vengono sprecate dal fatto che la virtual machine ospita un sistema operativo completo. Così come una macchina virtuale permette un'astrazione del livello fisico, un container docker permette l'astrazione del livello applicativo. Infatti, Docker utilizza le funzionalità del kernel Linux della macchina su cui è installato e grazie all'isolamento delle risorse del kernel, ogni container è isolato dagli altri, rendendo questo approccio sicuro e permettendo una gestione molto granulare di quelle che sono le risorse messe a disposizione per ogni container.

Un container in Docker viene creato a partire da un'immagine che ne rappresenta l'ambiente di esecuzione. La configurazione dell'immagine viene definita in un apposito file denominato *Dockerfile*, nel quale vengono aggiunte tutte le dipendenze con la versione desiderata. Nel caso una dipendenza del progetto venga aggiornata basta modificare il relativo Dockerfile e creare una nuova immagine. Alle immagini Docker inoltre è possibile aggiungere un tag identificativo. In questo modo si possono versionare le immagini relative a versioni di codice diverse e, ad esempio, controllare il comportamento tra la componente back-end e diverse versioni della componente front-end.

Nel caso dell'applicativo che vogliamo eseguire fosse composto da diversi container, e più in generale in un contesto di produzione, Docker mette a disposizione il tool Docker Compose. All'interno di un file di configurazione, solitamente chiamato `docker-compose.yml`², è possibile definire tutti i parametri dei vari container che compongono la nostra applicazione (chiamati servizi) e gestire tutti i container con un solo comando. Tramite Docker Compose è infatti possibile avviare, fermare e rebuildare i servizi, monitorare il loro stato ed eseguire comandi su uno specifico servizio.

In particolare, sono stati realizzati 4 servizi, ognuno con il relativo Dockerfile:

- **setup**, questo servizio esegue uno script una tantum che inizializza gli utenti `logstash_internal` e `kibana_system` all'interno di Elasticsearch con i valori delle password definite nel file delle variabili d'ambiente `.env`³. Questa attività viene eseguita solo durante l'avvio iniziale dello stack. Su tutte le esecuzioni successive, il servizio ritorna semplicemente immediatamente, senza eseguire qualsiasi modifica agli utenti esistenti.
- **elasticsearch**, questo servizio è responsabile di avviare e gestire il container di Elasticsearch, il database NoSQL utilizzato per conservare i dati ed eseguire le query.
- **logstash**, questo servizio è responsabile di avviare e gestire il container di Logstash, il server di ingestion e filtraggio dei dati.

² <https://github.com/thatsimo/sdcc-project-2021/blob/main/docker-elk/docker-compose.yml>

³ <https://github.com/thatsimo/sdcc-project-2021/blob/main/docker-elk/.env>

- **kibana**, questo servizio è responsabile di avviare e gestire il container di Kibana, il front-end dell'applicazione.

Questi servizi comunicano tra loro grazie alla creazione di una rete condivisa da tutti i container del compose. Le reti permettono la comunicazione tra container aggirando le proprietà di isolamento degli stessi. Di default ogni container viene collegato alla rete docker0 (denominata “bridge”) attraverso la quale è possibile comunicare con un container solo effettuandone il link esplicito. In alternativa, come è implementato nel presente docker-compose file, con l'aggiunta della rete bridge `elk`, è sufficiente aggiungere i container ad una nuova rete che utilizza il network driver bridge per permettere la comunicazione attraverso il nome del container di interesse. Conoscere il funzionamento e le modalità di comunicazione tra container è di fondamentale importanza in quanto l'obiettivo è la completa containerizzazione dello stack ELK e le varie componenti devono poter comunicare tra loro. Un'altra prerogativa importante per la corretta comunicazione di un'applicazione *multi-container* è il *port mapping*: le porte esposte da un container possono essere rese pubbliche e mappate sulle porte del sistema host nel quale è ospitato Docker Engine per permettere l'utilizzo delle funzionalità offerte all'esterno.

Pertanto, sono state messe a disposizione le seguenti porte, che lo stack ELK espone di default:

- 5601: Kibana, punto di accesso per la dashboard.
- 9600: Logstash monitoring API
- 9300: Elasticsearch TCP transport
- 5000: Logstash TCP input, tramite cui è possibile inoltrare i file a Logstash, ad esempio, utilizzando la seguente direttiva

```
$ cat example.json | nc -q0 localhost 5000
```

- 9200: Elasticsearch HTTP, essenziale per comunicare direttamente con Elasticsearch tramite REST API. In questo progetto è stato fondamentale definire il mapping, riportato di seguito, del campo *coordinates* come tipo `geo_point`, ciò permette al front-end di rappresentare correttamente i tweets geolocalizzati.

```
curl -X PUT "localhost:9200/_index_template/tweets-location"
-H 'Content-Type: application/json' -d'
{
  "template": {
    "mappings": {
      "properties": {
        "coordinates": {
          "type": "geo_point"
        }
      }
    }
  },
  "index_patterns": [
    "tweets"
  ]
}'
```

5 CONCLUSIONI

In conclusione, di seguito è mostrata una rapida anteprima della dashboard predefinita in esecuzione.

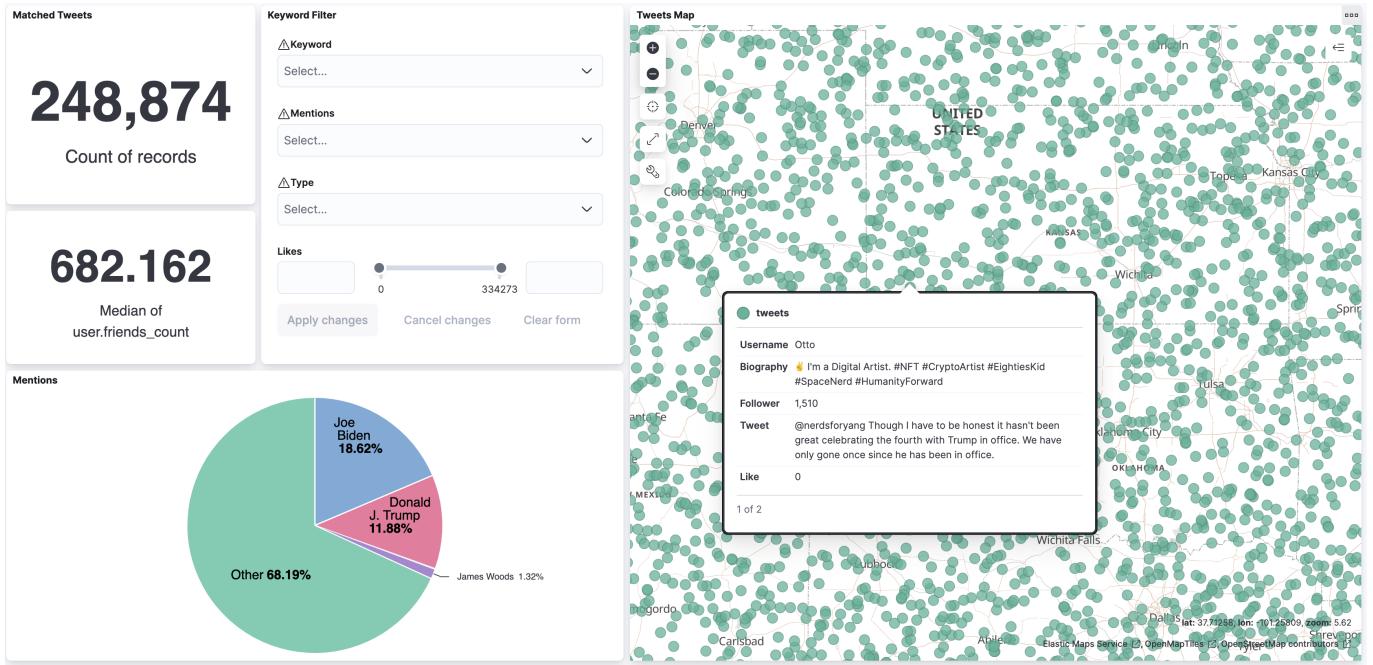


Figura 3. Vista predefinita

La dashboard predefinita dispone di una mappa, che permette di localizzare i tweets geolocalizzati, di cui sono mostrati le proprietà salienti come, ad esempio, il nome utente, numero di "mi piace" ricevuti, testo del tweet stesso, e di una serie di controlli per filtrare i risultati, oltre che a diverse metriche, come il numero di risultati trovati.

Infatti, è possibile filtrare per *keyword*, campo che identifica gli hashtag contenuti nel tweet, scegliendo da una lista ottenuta dai tweets analizzati, per *mention*, per *likes*, selezionando il numero di "mi piace" con uno slider o inserendo un range nel campo di testo associato, per *type*, utile a distinguere i tweet che contengono contenuti multimediali.

Inoltre, l'utente può, se necessario, effettuare query specifiche nel linguaggio KQL. Il Kibana Query Language (KQL) è una semplice sintassi per filtrare i dati di Elasticsearch utilizzando la ricerca di testo libera o la ricerca basata sui campi. KQL viene utilizzato solo per filtrare i dati e non ha alcun ruolo nell'ordinamento o nell'aggregazione dei dati. Pertanto, sono state predisposte due query, che individuano i tweet che contengono "Trump" o "Biden" nel proprio testo.

Come in accordo all'analisi dei requisiti, l'utente è libero di aggiungere nuove query personalizzate, visualizzazioni, tra quelle che Kibana mette a disposizione, e nuove dashboard create ad hoc da zero.

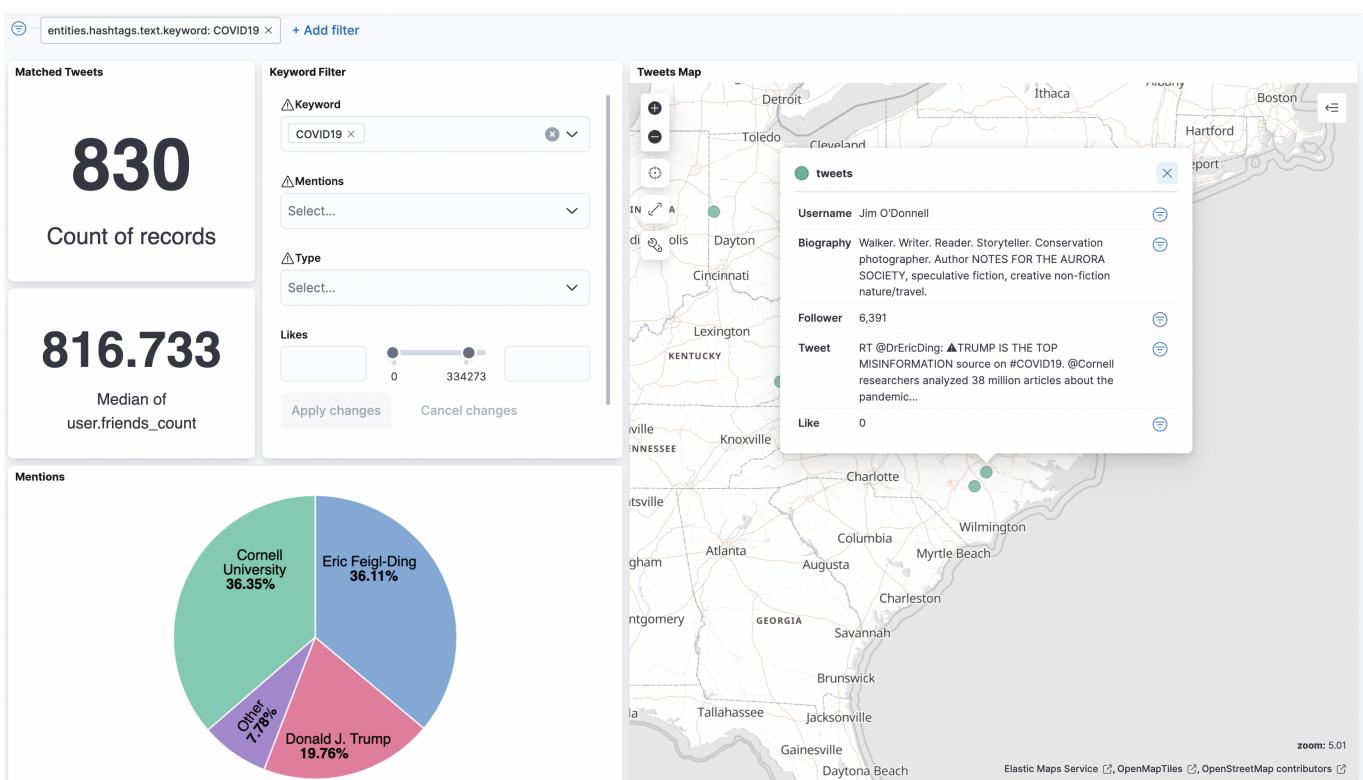


Figura 4. Vista filtrata per keyword

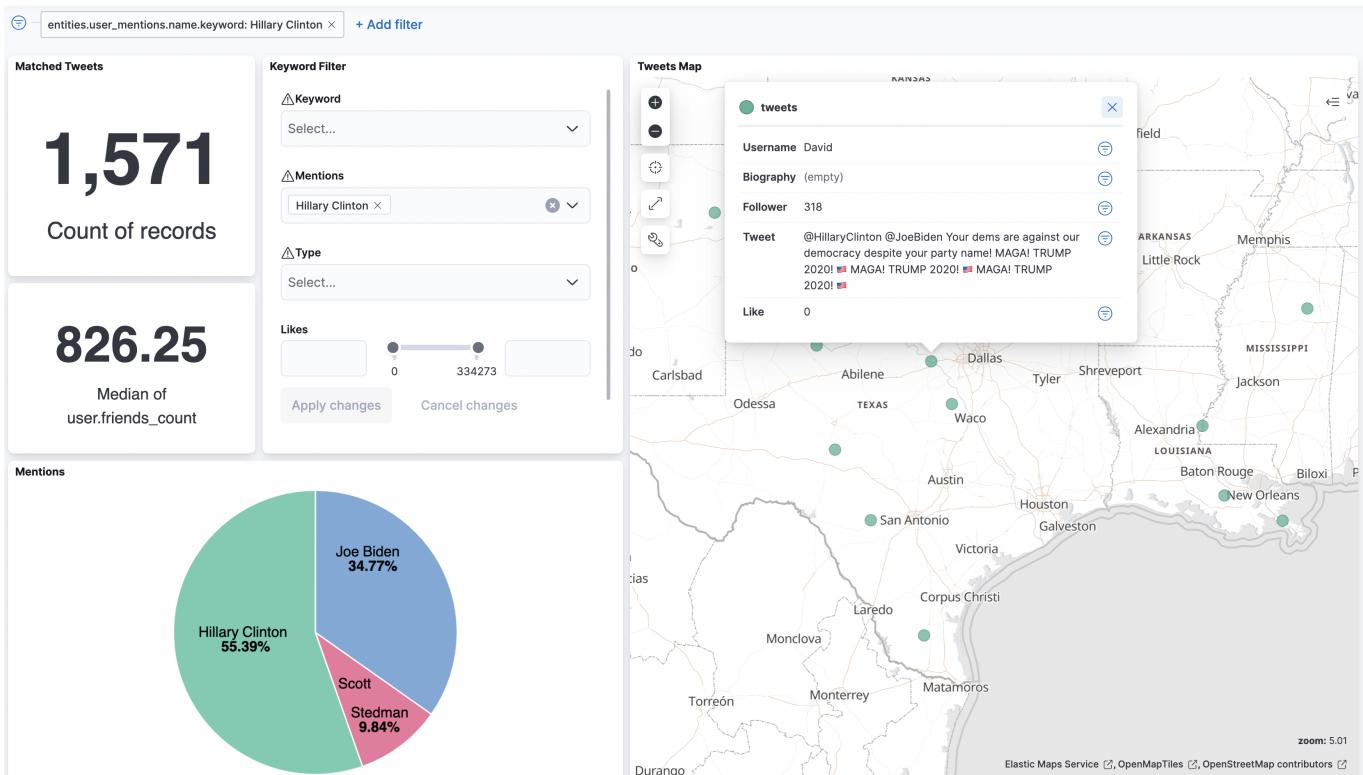


Figura 5. Vista filtrata per mention

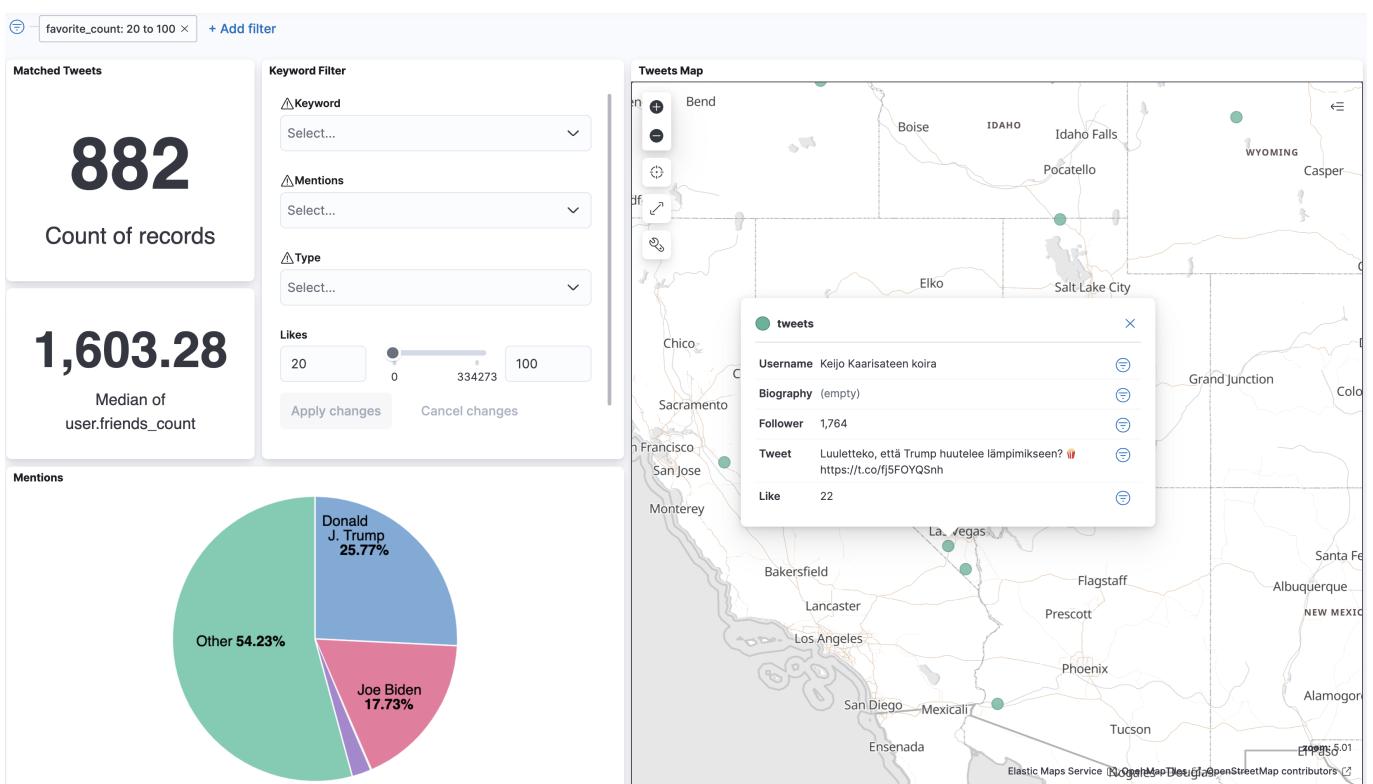


Figura 6. Vista filtrata per *like*

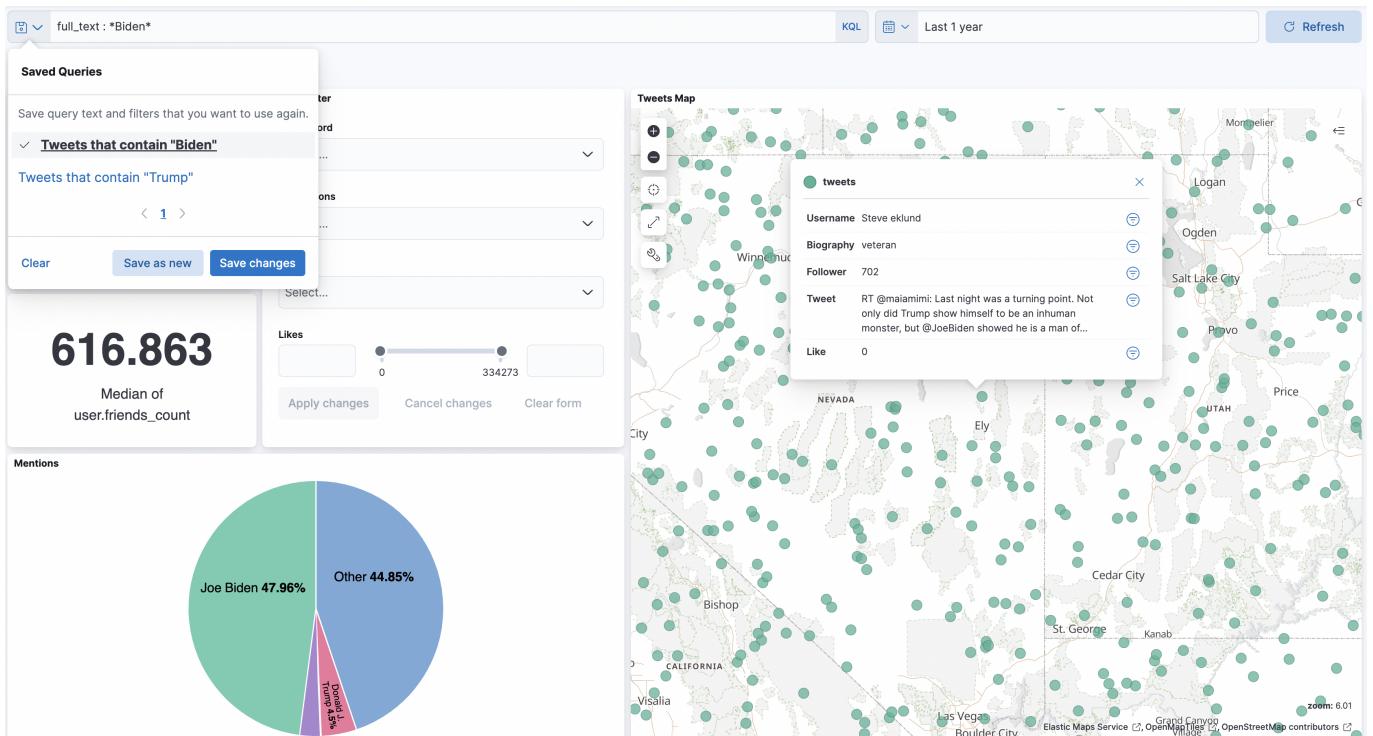


Figura 7. Vista filtrata per *query*