**PROGETTO BIG DATA 22/23**

**POLIGLOTTISMO E WEB APP**

**CHIARA SACCONE 533601**

**MATTEO FOSCHI 535207**

**WEB APP E POLIGLOTTISMO**

La nostra Web Application è un'applicazione software che viene eseguita su un server web ed è accessibile tramite un browser. Nel nostro progetto la web app viene utilizzata come interfaccia da parte dell’utente per l’analisi dei dati, infatti tramite essa, l’utente seleziona su un menù a tendina un parametro scelto da lui riferito ad un volo (come ad esempio: scelto un orario dal campo “Partenza”, ad esempio “00:05”) automaticamente l’utente visualizzerà tutti i voli con l’orario di partenza selezionato.

Il termine "**poliglottismo**" in questo nostro progetto si riferisce a due diversi contesti:

1. All'abilità di una web app di essere scritta in più di un linguaggio di programmazione, infatti sono stati utilizzati:

* JavaScript, Html, CSS per la parte front-end interattiva
* Python per il back-end
* SQL per l'accesso al database Postgres
* CQL (Cassandra Query Language), simile a SQL ma ottimizzato per i database distribuiti e non relazionali.
* Per MongoDB, il linguaggio di query BSON (Binary JSON), che è un'estensione di JSON. Abbiamo eseguito query nei documenti JSON memorizzati in MongoDB utilizzando un insieme di operatori e metodi specifici.
* Per Neo4j, il linguaggio di query Cypher.

1. All'uso di diversi tipi di database o sistemi di gestione di dati (DBMS) all'interno dell’applicazione. Nel nostro caso il poliglottismo è il fulcro del nostro progetto data la presenza di:
   1. Dati strutturati e non strutturati: AI nostri dati sono state effettuate modifiche per riuscire a strutturarli in maniera tale che potessero interagire per estrarre nuove informazioni ad alto livello.
   2. Analisi: all’interno di database differenti sono stati salvati alcuni dati ridondanti, per essere utilizzati per eseguire dell’analisi.

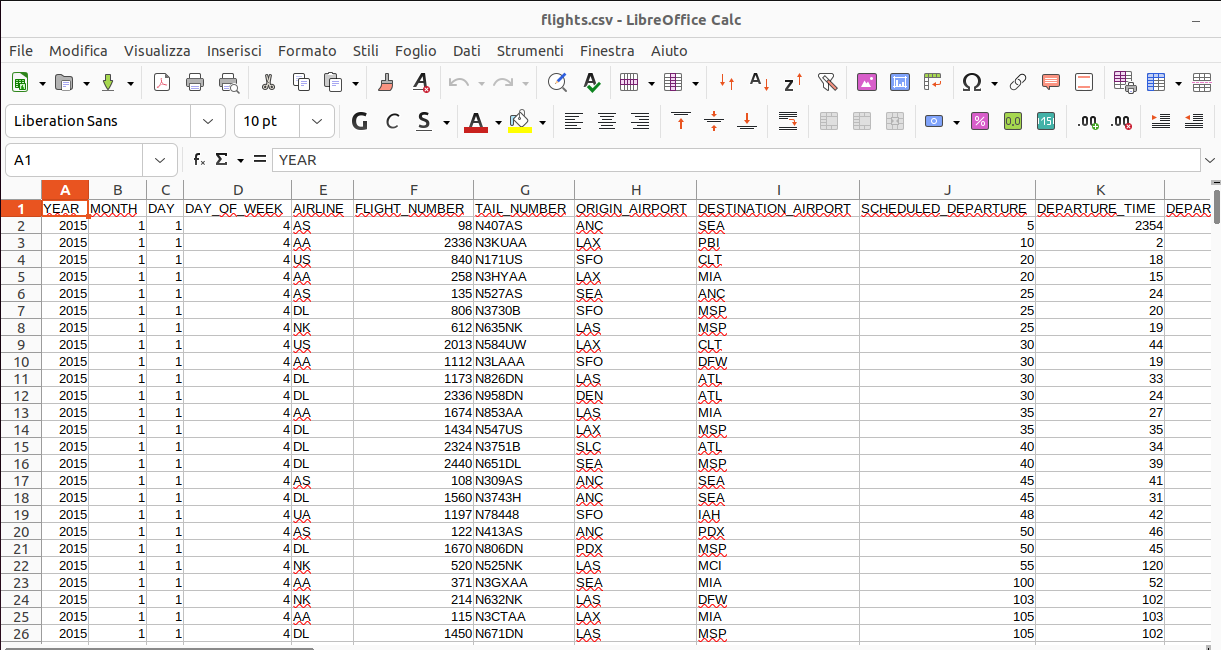
Postgres è stato utilizzato come database di riferimento, in quanto possiede il dataset principale, ovvero parte dei dati dei voli, ed è stato utilizzato per varie interrogazioni di appoggio, per far si che gli altri database come Mongo e Dynamo, che dovevano scambiarsi informazioni senza avere dati comuni, di interagire tra di loro senza intermediari. Come Postgres anche Cassandra ha avuto lo stesso ruolo.

* 1. Scalabilità: Questa applicazione richiede una grande scalabilità, quindi le diverse tecnologie di database sono state utilizzate anche al fine di gestire carichi di lavoro diversi. Il database relazionale è stato utilizzato anche per l’intermediazione tra i vari database NoSQL distribuiti per la memorizzazione e la ricerca dei dati ad alta velocità. Nel caso di Neo4j, l’utilizzazione della teoria dei grafi come visualizzazione in modo compatto e istantaneo dei risultati delle query.

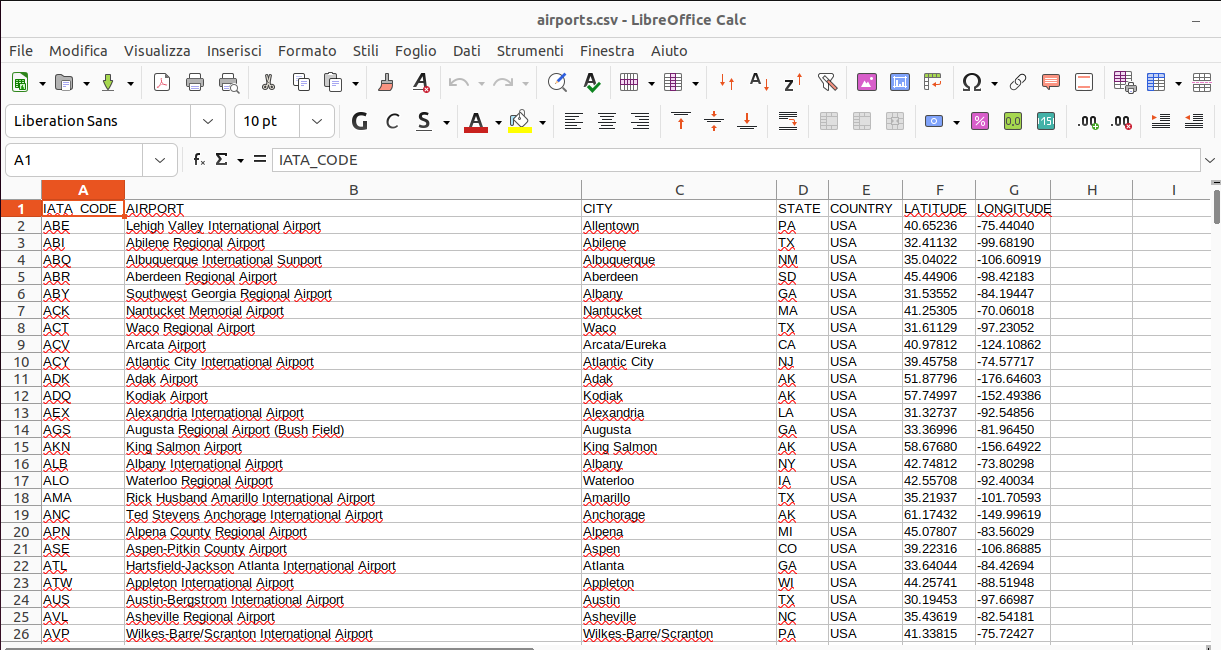
**IL DATASET**

L’intero dataset è stato preso su Kaggle ed ha il nome “2015 Flight Delays and Cancellations “[www.kaggle.com/datasets/usdot/flight-delays](http://www.kaggle.com/datasets/usdot/flight-delays); Con 592.43 MB abbiamo 3 diversi dataset:

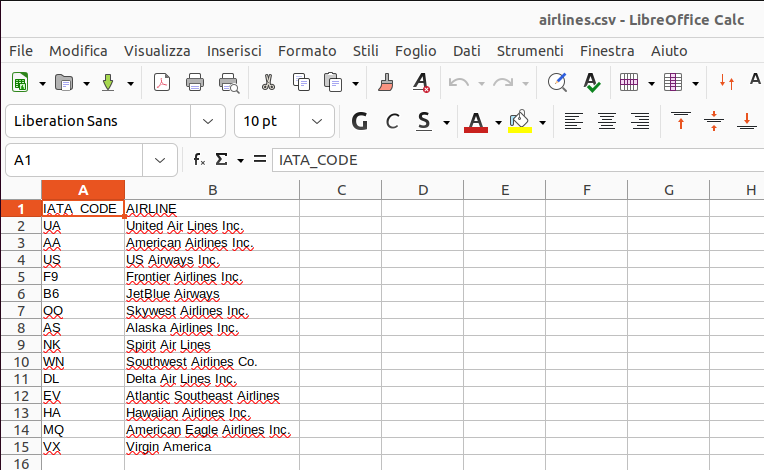
1. Flights.csv è il dataset che contiene le informazioni relativi ad alcuni voli effettuati in negli Stati Uniti nel 2015, il quale ha più di 5 milioni di ennuple e i 31 seguenti campi: (anno, mese, giorno, giorno\_settimana, compagnia, volo\_id, aeromobile, origine, destinazione, partenza\_prevista, partenza, decollo, distanza, atterraggio, arrivo\_previsto, arrivo, ritardo\_partenza, transito\_pista\_decollo, durata\_prevista, durata, tempo\_volo, distanza, transito\_pista\_atterraggio, ritardo\_arrivo, deviazione, cancellato, motivo\_cancellazione, ritardo\_malfunzionamento, ritardo\_sicurezza, ritardo\_compagnia, ritardo\_aereo, ritardo\_maltempo)



1. Airports.csv è il dataset che contiene le informazioni sugli aeroporti, con 322 righe e 7 colonne con i seguenti campi: (iata\_code, aeroporto, città, stato, paese, latitudine, longitudine)



1. Airlines.csv è il dataset che contiene le informazioni sulle compagnie di volo, con 14 righe e 2 colonne con i seguenti campi: (compagnia\_id, compagnia)



**ARCHITETTURA GENERALE**

Immagine che contiene testo, schermata, diagramma, linea

Descrizione generata automaticamente

La nostra è una architettura Web Application basata su servizi e container Docker.

Qui di seguito forniremo una descrizione più dettagliata di ciascun componente e della loro interazione:

**Frontend:** Il frontend è la parte dell'applicazione con cui l'utente interagisce direttamente tramite un browser. È responsabile dell'interfaccia utente, della presentazione dei dati e delle interazioni utente. Ed è scritta nei linguaggi HTML, CSS e JavaScript.

**Backend:** Il backend è la parte server dell'applicazione. Gestisce la richiesta dell’utente, che dopo esser stata tradotta in una query, comunica con i database per recuperare e memorizzare dati, infatti, tramite i metodi del backend, i nostri database riescono a comunicare tra loro e a scambiarsi risultati per comporre il risultato finale della richiesta dell’utente in modo chiaro e strutturato.

È composto da più servizi che forniscono funzionalità specifiche.

**Servizi:** I servizi sono componenti software autonomi che costituiscono il backend. Ogni servizio esegue una funzione specifica dell'applicazione e comunica con gli altri tramite API. In questa architettura, ci sono servizi che gestiscono l'interazione con ciascun database: MongoDB, PostgreSQL, DynamoDB, Cassandra e Neo4j.

**Database:**Ogni database è selezionato per rispondere alle esigenze specifiche di modellazione dei dati e di accesso ai dati nell'applicazione. Questo approccio poliglotta consente di sfruttare al meglio le caratteristiche di ciascun database per ottimizzare le prestazioni e la flessibilità dell'applicazione.

**MongoDB:** Un database NoSQL orientato ai documenti utilizzato per archiviare dati semi-strutturati o non strutturati.

**PostgreSQL:** Un sistema di gestione di database relazionali (RDBMS) utilizzato per dati strutturati con relazioni complesse.

**DynamoDB:** Un database NoSQL chiave-valore gestito da AWS, per applicazioni altamente scalabili.

**Cassandra:** Un database NoSQL distribuito e altamente scalabile utilizzato per gestire grandi volumi di dati su cluster di macchine.

**Neo4j:** Un database di tipo grafo utilizzato per rappresentare relazioni complesse tra i dati. Viene utilizzato per la visualizzazione dei risultati delle query basate su grafi.

**Docker:** Tutti i componenti dell'applicazione (frontend, backend, database) vengono containerizzati utilizzando Docker. Abbiamo infatti distribuito ogni database su un container Docker diverso così da essere unità isolate che possono essere facilmente distribuite sui diversi ambienti.

L'interazione tra questi componenti avviene attraverso API e comunicazioni tramite HTTP. Il backend gestisce le richieste del frontend, interagisce con i database per l'accesso ai dati e fornisce i risultati a Neo4j per la visualizzazione dei dati basati su grafi.

L'uso di servizi e containerizzazione (Docker) consente di scalare e gestire facilmente i componenti dell'applicazione in base alle esigenze di carico e alle esigenze di sviluppo. Inoltre, Neo4j svolge un ruolo cruciale nella visualizzazione dei dati complessi in modo intuitivo attraverso rappresentazioni grafiche, consentendo una migliore comprensione delle relazioni tra i dati.

L'intera architettura è progettata per fornire un'applicazione scalabile, resiliente e ad alte prestazioni.

**LE NOSTRE TECNOLOGIE**

Abbiamo utilizzato come database Postgres, CassandraDB, MongoDB e DynamoDB.

Neo4j è stato utilizzato per la visualizzazione dei risultati delle query.

Introdurremo brevemente la struttura dei database con i quali abbiamo lavorato per la prima volta grazie a questo progetto.

***Postgres***

Abbiamo deciso di dare a Postgres il dataset flight.csv escludendo i campi che esprimevano intervalli di tempo; quindi, abbiamo tenuto: (anno, mese, giorno, giorno\_settimana, compagnia, volo\_id, aeromobile, origine, destinazione, partenza\_prevista, partenza, decollo, distanza, atterraggio, arrivo\_previsto, arrivo, deviazione, cancellato, motivo\_cancellazione, ritardo\_malfunzionamento, ritardo\_sicurezza, ritardo\_compagnia, ritardo\_aereo, ritardo\_maltempo) senza esplicitare alcuna chiave primaria, in quanto il codice del volo (volo\_id) non è univoco come parametro in sé per sé.

***Cassandra***

Cassandra è un database distribuito e altamente scalabile che fa parte della famiglia degli NoSQL.

Utilizza un modello di dati chiave-valore con colonne denominate e tabelle organizzate in famiglie di colonne.

Abbiamo definito il keyspace (l'equivalente di un database in Cassandra), il quale contiene tutte le tabelle e i dati relativi a un'applicazione, che abbiamo chiamato “progettobig”.

Ci è stato utile lavorare con Cassandra in quanto fino ad ora avevamo interagito solo con Postgres con il semplice database relazionale, mentre in Cassandra vi è una struttura diversa.

Le tabelle sono distribuite orizzontalmente su nodi. Ogni riga in una tabella ha una chiave primaria unica che identifica in modo univoco quella riga. Le chiavi sono fondamentali in Cassandra e vengono utilizzate per accedere ai dati e la chiave primaria identifica univocamente una riga in una tabella. Noi abbiamo utilizzato una chiave costituita da più colonne, quindi una chiave primaria composta.

Le colonne contengono i dati effettivi. Una riga in Cassandra può avere diverse colonne, ma non è richiesto che tutte le righe abbiano le stesse colonne.

Le righe in una tabella sono suddivise in partizioni basate sulla chiave primaria. Ogni partizione risiede su un nodo e può essere distribuita su molti nodi.

Le query in Cassandra sono ottimizzate per l'accesso alle partizioni utilizzando la chiave primaria; infatti, poiché la chiave composta nel nostro caso è (volo\_id, destinazione, compagnia) le nostre query vertono esclusivamente su quei campi in quanto sappiamo che le query che coinvolgono colonne al di fuori della chiave primaria possono essere inefficienti e molto spesso neanche accettate.

Abbiamo deciso di dare a Cassandra il dataset flight.csv escludendo, al contrario di Postgres, i campi che esprimevano orari; quindi abbiamo tenuto: (anno, mese, giorno, giorno\_settimana, compagnia, volo\_id, aeromobile, origine, destinazione, ritardo\_partenza, transito\_pista\_decollo, durata\_prevista, durata, tempo\_volo, distanza, transito\_pista\_atterraggio, ritardo\_arrivo, deviazione, cancellato, motivo\_cancellazione, ritardo\_malfunzionamento, ritardo\_sicurezza, ritardo\_compagnia, ritardo\_aereo, ritardo\_maltempo) con chiave primaria composta impostata su (volo\_id, destinazione, compagnia)

***Dynamo***

DynamoDB è uno dei database NoSQL di tipo chiave-valore. Puoi memorizzare e recuperare dati utilizzando una chiave primaria composta da una o due colonne: una chiave di partizione e, se necessario, una chiave di ordinamento (chiave di clustering). Su Dynamo è stato caricato il dataset relativo alle compagnie di volo Airlines.csv con chiave primaria a una colonna “compagnia\_id” e attributo il nome della compagnia “nome”.

Abbiamo poi memorizzato i dati semi-strutturati come JSON dopo ogni query e passato poi il file json al frontend.

***Mongo***

MongoDB è un database NoSQL orientato ai documenti. Memorizza i dati in formato BSON (Binary JSON) e offre una grande flessibilità nella struttura dei dati.

Su Mongo è stato caricato il dataset relativo agli aeroporti Airports.csv, con i seguenti attributi (iata\_code, aeroporto, città, stato, paese, latitudine, longitudine) e chiave primaria impostata su “iata\_code”.

***Neo4j***

Neo4j è un database di tipo grafo che si basa sulla rappresentazione dei dati tramite nodi, relazioni e proprietà.

In questo progetto non è stato utilizzato come analisi dei dati basata su grafi, come suo uso più comune, ma come strumento di visualizzazione per rappresentare graficamente i risultati delle query fatte su dati provenienti dagli altri database.

Facendo ciò abbiamo quindi creato rappresentazioni grafiche delle relazioni e dei collegamenti tra i dati risultanti in formato Json che Neo4j ha preso in input. Questo approccio ha reso più comprensibili e interpretabili i risultati delle nostre analisi, riuscendo a interconnettere anche dati provenienti da diversi file in input.

**Poiché queste tecnologie sono state utilizzate come database per il nostro lavoro, specifichiamo che per verificare la correttezza dei dati dopo l’inserimento nelle tabelle, le varie creazioni degli ambienti per ogni database, le verifiche delle chiavi come metodo univoco di riconoscimento della ennupla ecc, abbiamo svolto delle query direttamente da tutti i container web utilizzati come interfacce connesse sulla stessa rete dei nostri database, ognuno con il linguaggio specifico: CQL per Cassandra DB, SQL per Postgres, Query Language per Mongo DB (cioè abbiamo eseguito le query utilizzando il linguaggio basato su documenti che utilizza operatori e metodi specifici di MongoDB, stessa cosa per DynamoDB che non ha linguaggio proprio ma utilizziamo i suoi metodi specifici tramite la istanza “connessioneDynamo”), e Cypher per Neo4j .**

**DOCKER CONTAINER**

Ovviamente, l'uso di diversi database ha introdotto complessità nella gestione dei dati ed ha richiesto una pianificazione accurata per garantire l'integrazione e l'efficienza complessiva del sistema.

Per tale ragione, ci siamo avvalsi di Docker come piattaforma di containerizzazione in quanto ci serviva sviluppare, distribuire ed eseguire l’applicazione all'interno di container isolati. Abbiamo importato nel container Docker tutto il necessario per eseguire la web app, tra cui i vari DBMS, il codice, le librerie e le dipendenze.

Le componenti di Docker che abbiamo utilizzato sono:

**1. Docker Container:**

Un container Docker è un'istanza eseguibile di un'immagine Docker. Un'immagine Docker è uno snapshot leggero e standalone di un ambiente che include tutto il necessario per eseguire un'applicazione, mentre il container è l'istanza in esecuzione di quella immagine. I container sono isolati tra loro e dal sistema host, infatti nella parte iniziale del progetto, che riguardava il caricamento del dataset in ogni database, abbiamo eseguito i vari container singolarmente senza influire l'uno sull'altro così da essere facilmente avviati, arrestati o spostati.

**2. Docker Compose:**

Abbiamo utilizzato Docker Compose perché è uno strumento che semplifica la definizione e la gestione di applicazioni multi-container. Permette di descrivere un'applicazione composta da più servizi (ognuno in un proprio container) all'interno di un file YAML (“docker-compose.yml”), dove abbiamo specificato le dipendenze tra di essi, le variabili d'ambiente, le porte esposte e altro ancora. Lo abbiamo usato soprattutto per comodità in quanto ci consente di avviare e arrestare l'intera applicazione con un solo comando. (Con un solo comando abbiamo utilizzato anche il `docker build` per generare un'immagine Docker basata sulle istruzioni nel dockerfile)

**##############comando**

**3. Dockerfile:**

Il Dockerfile è un file di testo che contiene una serie di istruzioni per la creazione di un'immagine Docker personalizzata. Le istruzioni nel Dockerfile specificano come configurare l'immagine, quali pacchetti installare, quali file copiare all'interno dell'immagine e come eseguire comandi durante la creazione dell'immagine.

**4. Immagini Docker**

Per ogni container abbiamo eseguito una immagine Docker. Questo processo è noto come "commit" e consente di catturare lo stato attuale di un container in un'immagine Docker. Con la creazione dell’immagine si ha:

1. Persistenza dei dati: I dati all'interno dei database sono presenti anche quando il container viene arrestato o eliminato. Questo è utile per mantenere la consistenza dei dati tra esecuzioni del container.

2. Riproducibilità: Creando un'immagine Docker a partire da un container, replicavamo esattamente lo stesso ambiente ogni volta che rilanciamo la web app.

3. Facilità di distribuzione: Le immagini Docker sono facilmente distribuibili e possono essere condivise tra me e il mio collega. Avendo configurato i container con le impostazioni specifiche necessarie per il progetto, è più efficiente creare un'immagine da condividere così da non ripetere manualmente la configurazione su ogni server ogni volta.

4. Backup e ripristino: sono utili per eseguire il backup dello stato dell'applicazione e i dati del database. Questo rende più semplice il ripristino dell'applicazione in caso di guasto o perdita dei dati (ce ne sono state parecchie, soprattutto di guasti, durante la progettazione).

**############SCREEN DEI CONTAINER E IMMAGINI CHE SI VENGONO A CREARE OGNI VOLTA**

**LA NOSTRA WEB APP: BACKEND**

**Metodi:**

I metodi che sono stati implementati nel back-end riguardano:

1. Connessioni: connessioneNeo() e per le altre abbiamo definito variabili che inglobano la connessione, si trovano tutti nel file python principale dell’applicazione *app.py*.
2. Connessioni test: abbiamo lasciato nella parte finale di app.py dei metodi di connessione che utilizzavamo all’inizio della progettazione per testare le connessioni.
3. Caricamento: utilizzati per i caricamenti dei vari dataset sui database; abbiamo creato un file python per ogni caricamento su database, essi si trovano su una cartella \flaskapp\caricamentiDB e abbiamo: *caricamentoMongo.py, caricamentoNeo.py, caricamentoPos.py, caricamentoDy.py, caricamentoCassandra.py*
4. Query singole NoSql: utilizzati per connettersi ed effettuare query singolarmente su ogni database in base al parametro ricevuto in input: *selectCassandra(volo), selectDynamo(compagniaid), selectMongo(iatacode*)
5. Query singole Postgres: utilizzati per connettersi ed effettuare query singolarmente su Postgres in quanto è possibile effettuarli su attributi non chiave e utilizzati per il riempimento delle liste di appoggio dei risultati parziali: *selectpostgrescompagniaid(compagniaid), selectpostgres(partenzaPrevista), selectpostgresdestinazione(destinazione), selectpostgresvoloid(voloid);*
6. Query a cascata: utilizzati per effettuare query a cascata. Si parte sul browser selezionando arbitrariamente un parametro a scelta caratterizzante il dataset all’interno del database specificato, e da lì verranno effettuate le selezioni sugli altri database che mostreranno a cascata i loro risultati: *selectdynamocascata(compagniaid), selectmongocascata(iatacode), selectcassandracascata(idvolo), selectpostgrescascata(partenzaprevista).*

**Connessioni**

Per consentire ai container Docker di connettersi a un back-end tramite un file Python, abbiamo:

1. Creato il Dockerfile dove abbiamo incluso tutte le dipendenze di Python e le librerie necessarie per eseguire il tuo file Python.

2.Creato l’immagine Docker

3. Eseguito i container mappando tutte le porte di ogni container nel file Python *app.py* così da connettersi con il back-end tramite una determinata porta.

4. Nel file Python app.py, abbiamo utilizzato i vari metodi di connessioni e le librerie. **Per non essere dipendenti dagli indirizzi IP da dover cambiare ogni volta che facevamo girare il programma ognuno sul proprio pc, abbiamo messo il nome del container così da rendere indipendente la connessione da indirizzi statici.**

**Le connessioni sono state fatte nel back-end tutte all’inizio di app.py così da poter utilizzare l’istanza di connessione per ogni metodo.**

Infine, con l’esecuzione dei container, il file Python app.py viene eseguito ed è in grado di connettersi al backend.

I codici sottostanti rappresentano le connessioni ai diversi tipi di database all'interno di Docker. Queste connessioni utilizzano librerie specifiche per ciascun database e le porte il nome dei servizi e le porte sono configurate nel Docker Compose:

1. Connessione a Cassandra:

*connessione=Cluster(['cassandraDb'], port=9042).connect()*

Con il driver di Python per Cassandra, `cassandra-driver`, stabiliamo una connessione al database Cassandra, creando un oggetto `Cluster` e connettendoci al nostro cluster "cassandraDb" sulla porta 9042.

1. Connessione Postgres:

*connessione = psycopg2.connect(host="localhost", port="5432", user="postgres", password="password", database="postgres")*

Utilizziamo la libreria `psycopg2` per connetterci a PostgresDB. "postgres" è il nome del servizio del container Docker che esegue PostgresDB. La porta 5432 è la porta predefinita e abbiamo inserito anche le credenziali.

1. Connessione a MongoDB:

*connessioneMongo= pymongo.MongoClient("mongodb://mongoDb:27017/")*

Utilizziamo la libreria `pymongo` per connetterci a MongoDB. "mongoDb" è il nome del servizio del container Docker che esegue MongoDB. La porta 27017 è la porta predefinita di MongoDB.

1. Connessione a DynamoDB:

*conn=boto3.resource('dynamodb',endpoint\_url='http://dynamoDbGUI:8000',region\_name='us-east-1')*

Utilizziamo la libreria `boto3` per connetterci a DynamoDB. Configuriamo la connessione per l'endpoint in base alla tua configurazione specifica di DynamoDB in Docker, e la regione "us-east-1", regione default di AWS.

1. Connessione Neo4j :

*connessioneNeo=Graph("bolt://neo4jDbGUI:7687")*

Abbiamo creato un'istanza dell'oggetto Graph specificando l'URL del server Neo4j corrispondente al nostro container Docker.

**Caricamenti database**

Ogni database è stato caricato in backend dai file csv presi su Kaggle utilizzando per ogni database l'oggetto `connessione` precedentemente stabilito all’inizio di app.py.

Caricamento Mongo: La funzione `caricamentoMongo` carica dati dal file CSV './airports.csv'` in una collezione MongoDB chiamata *aeroporti* utilizzando la libreria `pymongo`.

1. Crea una connessione MongoDB utilizzando l'oggetto `connessione` precedentemente stabilito.

2. Specifica il nome del database e il nome della collezione a cui desideri caricare i dati.

3. Verifica se il database specificato esiste. Se esiste, ottiene un riferimento al database; altrimenti, lo crea.

4. Verifica se la collezione specificata esiste all'interno del database. Se esiste, ottiene un riferimento alla collezione; altrimenti, la crea.

5. Legge i dati da un file CSV utilizzando la libreria `pandas` e li converte in un dizionario di record JSON.

6. Inserisce i dati nella collezione MongoDB utilizzando il metodo `insert\_many`.

Caricamento Dynamo: La funzione `caricamentoDynamo` in primis ottiene la lista delle tabelle esistenti in DynamoDB utilizzando il metodo `dynamodb.tables.all()`, estrae i nomi delle tabelle dalla lista di oggetti Table e verifica se la tabella *CompagnieAeree* che stiamo creando esiste già.

Il codice di creazione della tabella definisce gli attributi chiave (`KeySchema`) e gli attributi (`AttributeDefinitions`) utilizzati per la creazione della tabella. Inoltre, specifica la capacità (`ProvisionedThroughput`) per la lettura e la scrittura.

Il codice aggiunto sembra essere il completamento della funzione `caricamentoDynamo` per l'inserimento effettivo dei dati nella tabella DynamoDB. Ecco cosa fa il codice aggiunto:

Dopo aver letto il file ‘airlines.csv` utilizzando la libreria `pandas` e lo carica in un DataFrame per poi iterare attraverso le righe del DataFrame e creare un dizionario `item` per ogni riga. Questo dizionario contiene gli attributi `IATA\_CODE` e `AIRLINE` dove iata\_code è chiave primaria e poi con il metodo put inserisce l’elemento nella tabella.

**QUERY E BACK-END**

I metodi utilizzati nel back-end sono volti a effettuare query in tutti i database tramite i loro connettori, per poi restituire gli output delle query in formato JSON da passare agli altri database per eseguire le selezioni sui dati in input e arrivare a un risultato finale. Il risultato finale sarà poi visualizzato tramite grafo con Neo4j.

Metodi query a cascata

Possiamo partire da qualsiasi database e, selezionato un parametro, manderà direttamente le sue ennuple relative a quel dato agli altri database a cascata, che faranno le loro selezioni con i loro parametri specifici come fosse un join con il parametro in input.

I parametri specifici di ogni database che sono gli attributi chiave delle loro rispettive selezioni, sono:

* partenzaPrevista o volo\_id o destinazione per Postgres
* (volo\_id, destinazione, compagnia\_id) per Cassandra
* compagnia\_id per Dynamo
* destinazione per Mongo

Esempio metodo a cascata: **Dynamo Cascata**

Metodo che effettua una query sul database DynamoDB e successivamente effettua query su un database SQL (PostgresDB), database NoSQL (Cassandra, MongoDB) e un database graph (Neo4j), per poi restituire i risultati in formato JSON.

1. La route `/selectdynamocascata/<compagniaid>` accetta richieste HTTP GET con un parametro `<compagniaid>` nell'URL.

2. Estraggo il parametro ` compagniaid ` dalla richiesta HTTP.

3. Apro la connessione al database a DynamoDB utilizzando `connessioneDynamo`.

4. Eseguo una query su DynamoDB per cercare elementi nella tabella "compagnieAeree" in cui il campo "compagnia\_id" è uguale a `param`.

5. Memorizzo gli elementi risultanti dalla query nella variabile `items`.

6. Inizializzo un dizionario `output` per memorizzare i risultati futuri provenienti dagli altri (PostgreSQL, MongoDB, Cassandra).

7. Creo la lista di appoggio chiamata `appoggio` che utilizzerò per memorizzare i risultati di Postgres i quali mi restituiranno le sue ennuple con i campi (volo\_id, destinazione, compagnia\_id) così da fare da tramite e passare i suoi risultati per le selezioni in Cassandra e Mongo, che senza non avrebbero avuto i dati necessari per poter svolgere autonomamente la query con il solo dato (compagnia\_id) passati da Dynamo.

8. Itero su `items`, che contiene i risultati della query DynamoDB, e hai effettuato una query su PostgreSQL utilizzando la funzione `selectpostgrescompagniaid`. I risultati sono stati aggiunti a `output["resultPostgres"]` e in`appoggio`.

9. Itero su `appoggio` (che contiene i risultati da PostgreSQL) per effettuare la query su Cassandra per ciascun elemento utilizzando `selectcassandra`. I risultati sono stati aggiunti a `output["resultCassandra"]`.

10. Itero nuovamente su `appoggio` e per effettuare la query su MongoDB utilizzando `selectmongo`. I risultati sono stati aggiunti a `output["resultMongo"]`.

11. Utilizzo Neo4j tramite una chiamata a `caricamentoNeo4j.neo4jElaborazione` memorizzando i risultati in `resultN4j`.

12. Restituisco i risultati completi in formato JSON utilizzando `jsonify(output)`.

I metodi utilizzati all’interno di selectDynamoCascata sono specifici per ogni database e si occupano di realizzare separatamente la query sul singolo Database.

Ecco illustrati i metodi di appoggio:

Metodo: **selectpostgrescompagniaid**

Metodo che esegue una query su PostgreSQL per ottenere i risultati in base al parametro `compagniaid` ricevuto in input.

1. Apro la connessione al database PostgreSQL utilizzando i dettagli di connessione: host, porta, nome utente, password e nome del database.

2. Definisco il cursore per eseguire la query sul database.

3. Eseguo la query SQL sulla tabella "volitimes" per selezionare le righe in cui il campo "compagnia" corrisponde al valore di `compagniaid` passato come parametro.

4. Memorizzo nella variabile `results’ i risultati della query utilizzando `cursor.fetchall()`.

5. Restituito i risultati come un elenco di tuple.

Metodo: **selectcassandra**

Metodo che esegue una query su CassandraDB per ottenere i risultati in base ai parametri ricevuti in input `volo\_id, compagniaid, destinazione` racchiusi in una tupla fornita da Postgres di nome “volo”.

1.Estraggo i componenti della tupla `volo` (volo\_id, compagnia, destinazione) utilizzando la sua posizione nell'elenco.

2. Apro una sessione Cassandra utilizzando `connessioneCassandra`.

3. per Specifico il keyspace tramite `USE ProgettoBig` per entrare nel keyspace che contiene la tabella con il dataset dei voli.

4. Eseguo la query Cassandra sulla tabella "voliInt" per selezionare le righe in cui i campi "volo\_id", "compagnia" e "destinazione" corrispondono ai valori estratti dai parametri.

5. Hai Ottengo i risultati della query in forma di oggetto `rows` e lo converto in formato JSON tramite `json\_util.dumps(rows)`, che restituisco in output.

Metodo: **selectmongo**

Metodo che esegue una query su MongoDB per ottenere i risultati in base al parametro ricevuto in input fornito da Postgres con il nome “iata\_code” (cioè l’id della`destinazione`, ovvero il nome dell’aeroporto)

1. Estraggo il parametro `iatacode`.

2. Stabilisco la connessione a MongoDB utilizzando `connessioneMongo`.

3. Per arrivare ai dati di interesse, vado sul dataset (database) chiamato "aeroporti" e la collezione (tabella) chiamata "aeroporto" all'interno di MongoDB.

4. Definisco la query MongoDB che seleziona i documenti in cui il campo "IATA\_CODE" corrisponde al valore di `param`.

5. Eseguo la query sulla collezione "aeroporto" utilizzando `tabella.find(myquery)` e memorizzo i risultati in `result`.

6. Utilizzo `json\_util.dumps(result)` per convertire l'oggetto `result` in formato JSON.

**File json frontend**

I risultati delle query vengono sempre restituiti dai metodi in formato JSON, in quanto l’utilizzo dei JSON per immagazzinare i risultati dai database è diventato uno standard per le applicazioni web o servizi, per avere dati strutturati a client.

La necessità di convertirli in questo formato si ha perchè per il recupero dei dati, interroghiamo database diversi, e queste interrogazioni vengono effettuate in linguaggi di query specifici per ciascun database (ad esempio, SQL per database relazionali come PostgreSQL o linguaggi di query NoSQL per database non relazionali come MongoDB).

Una volta eseguite le query, si ottengono risultati sotto forma di dati. Questi dati sono rappresentati come record, oggetti, elenchi o tabelle, a seconda del database e della query effettuata. Oltretutto, poiché nel caso specifico della nostra applicazione i risultati vengono passati da un database all’altro, i dati devono essere scritti in un formato leggibile così da poter essere elaborati dagli altri database in modo da estrarre i dati da poter utilizzare per le proprie interrogazioni, mandarli come risposta http e successivamente visualizzarli in modo strutturato nell'interfaccia utente dell'applicazione.

**Neo4j visualizzazione**

---intro

1. In input ricevo il parametro ‘output’, ovvero i dati elaborati dalla selezione nei database, la stringa ‘strMetodo’ che mi individua da quale db parte la ricerca e il parametro di connessione, connessioneNeo, per connettermi al database di neo4j.

2. Mi connetto a neo4j

3. Ripulisco il database

4. Mi dispongo il json in una lista

5. Definisco le liste dove salvo i nodi creati in neo4j per le connessioni

6. Definisco delle liste dove salvo l'informazione di ogni db relativa al proprio nodo della lista precedente

7. Seleziono il db di partenza dell'analisi (se inizio da postgres avrò dati di mongo,dynamo e cassandra)

8. Scorro i dati dei 3 db e creo i nodi relativi evitando duplicati

9. Salvo tutti i nodi

10. Creo archi di connessione sui nodi

11. Recupero il grafo generato in neo4j

**IL NOSTRO PROGETTO ISTRUZIONI**