# Introduzione

## Richiesta

Si richiede la realizzazione di un applicativo che simuli il funzionamento e le prestazioni di un database NoSql su uno schema preesistente. Il DBMS richiesto è nella fattispecie Cassandra. L'applicativo richiede di generare in modo casuale i dati che andranno in seguito a popolare il database iniziale da uno schema preesistente, con dataset di 10, 100, 1000, 10000 e 100000 elementi. Si richiede di analizzare le differenze in termini di costi ed efficienza con altri gruppi che stanno effettuando lo stesso progetto su database di tipo NoSql differenti (MongoDB, HBase, Neo4j). Si richede infine di plottare i risultati ottenuti e di confrontarli.

# Scenario di riferimento (inserire una figura che rappresenti lo scenario e descriva la problematica affrontata)

# Strumenti utilizzati

## DBMS: *DataStax DevCenter*

Il DBMS adoperato è DataStax DevCenter, scaricabile all'indirizzo <https://academy.datastax.com/downloads>: il tool permette di eseguire codice *CQL* per **Apache Cassandra**.

## IDE: *JetBrain PyCharm*

L'applicazione è stata implementata utilizzando il linguaggio Python 2.7, con IDE PyCharm di JetBrain nella versione Community (per gli studenti è possibile ottenere la versione *Professional* del prodotto con licenza di un anno utilizzando la mail della propria università, ad esempio codicefiscale@studenti.unime.it) scaricabile all'indirizzo <https://www.jetbrains.com/pycharm/download/#section=windows>. Il linguaggio è stato scelto a discapito di altri linguaggi perché permette di interfacciarsi agilmente con i vari DBMS assegnati nei vari progetti, permettendo dunque un risparmio di tempo non indifferente. Al fine della realizzazione dell'applicazione, è fondamentale scaricare il driver *cassandra-cluster*, attraverso un wizard guidato dallo stesso IDE.

## Stesura relazione: *Pandoc*

Per la stesura della relazione è stato utilizzato Pandoc, un strumento che permette di scrivere documenti in linguaggio markup e che converte successivamente gli stessi in , PDF e qualsiasi altro formato desiderato. Vi è inoltre la possibilità di utilizzare direttive direttamente all'interno del documento stesso. Per poter utilizzare Pandoc è necessario installare (consigliabile MikTex).

# Progettazione

Il progetto è stato realizzato suddividendolo in tre parti fondamentali

## Connessione al DBMS

Per realizzare la connessione al DBMS è necessario dapprima installare, attraverso procedura guidata dall'IDE, il driver che consente di interfacciare il DBMS al nostro codice, e nella fattispecie il driver utilizzato risulta essere *cassandra-cluster* . Successivamente, bisognerà selezionare il keyspace bersaglio sul quale operare.

## Gestione Dataset in JSON

La gestione dei vari dataset in JSON è facilitata in Python grazie all'utilizzo della libreria **json**. In questo caso è stato previsto l'utilizzo di una funzione che consente di selezionare il dataset adatto in base all'esigenza e un'altra che consente il caricamento dell'intero dataset così da poter effettuare le operazioni necessarie per lo svolgimento del progetto.

## Funzioni da implementare nel Main

Il Main ha la necessità di avere una funzione che consente lo svuotamento del DataBase e una che invece deve prevedere il caricamento di tutti i dati necessari nel DataBase, in base al Dataset selezionato. Infine, bisogna prevedere una funzione che tiene conto dei tempi di esecuzione di ciascuna query (che ricordiamo deve essere eseguita almeno 30 volte) per ogni Dataset.

# Implementazione

## Connessione al DBMS

Facendo riferimento a quanto scritto nel precedente capitolo, bisogna importare nell'IDE il driver *cassandra-cluster*. Fatto questo, bisogna dapprima importare *cassandra.cluster*, e successivamente è possibile accedere al proprio keyspace andando a specificarlo attraverso il seguente codice:

from cassandra.cluster import Cluster  
  
cluster = Cluster()  
session = cluster.connect('db2\_project')

Per facilitare e velocizzare le operazioni di querying, è stato implementato il seguente codice:

def update(query):  
 session.execute(query)  
  
def query(query):  
 return session.execute(query)

## Gestione del Dataset in JSON

I vari Dataset sono stati pensati in JSON, col seguente schema d'esempio:

[  
 {  
 "name": "string",  
 "id": "autoinc",  
 "width": "float",  
 "height": "float",  
 "l\_shank": "float",  
 "l\_thigh": "float",  
 "lokomat\_shank": "float",  
 "lokomat\_thigh": "float",  
 "lokomat\_recorded": "timestamp",  
 "version": "string",  
 "legtype": "string",  
 "lwalk\_training\_duration": "timestamp",  
 "lwalk\_distance": "float",  
 "step\_datas":  
 [  
 [0.0, 0.0, 0.0, 0.0, 0.0,   
 0.0, 0.0, 0.0, 0.0, 0.0,   
 0.0, 0.0, 0.0, 0.0, 0.0,  
 0.0, 0.0, 0.0, 0.0, 0.0,   
 0.0, 0.0, 0.0, 0.0, 0.0,   
 0.0, 0.0, 0.0, 0.0, 0.0,   
 0.0, 0.0],  
 [0.0, 0.0, 0.0, 0.0, 0.0,   
 0.0, 0.0, 0.0, 0.0, 0.0,   
 0.0, 0.0, 0.0, 0.0, 0.0,  
 0.0, 0.0, 0.0, 0.0, 0.0,   
 0.0, 0.0, 0.0, 0.0, 0.0,   
 0.0, 0.0, 0.0, 0.0, 0.0,   
 0.0, 0.0],  
 [0.0, 0.0, 0.0, 0.0, 0.0,   
 0.0, 0.0, 0.0, 0.0, 0.0,   
 0.0, 0.0, 0.0, 0.0, 0.0,  
 0.0, 0.0, 0.0, 0.0, 0.0,   
 0.0, 0.0, 0.0, 0.0, 0.0,   
 0.0, 0.0, 0.0, 0.0, 0.0,   
 0.0, 0.0]  
 ]  
 }  
]

In Python, la gestione dei Dataset avviene grazie alla libreria *json*, che bisogna importare. Successivamente, in base al Dataset che bisogna caricare, grazie alle funzioni open('json\_name.json') e json.load(json) gestiamo il caricamento dei Dataset JSON, come si evince dal seguente codice:

import json  
  
def settaData(scelta):  
 if(scelta == 0):  
 with open('ds10.json') as data:  
 record = json.load(data)  
 elif (scelta == 1):  
 with open('ds100.json') as data:  
 record = json.load(data)  
 elif (scelta == 2):  
 with open('ds1000.json') as data:  
 record = json.load(data)  
 elif (scelta == 3):  
 with open('ds10000.json') as data:  
 record = json.load(data)  
 elif (scelta == 4):  
 with open('ds100000.json') as data:  
 record = json.load(data)  
 return record  
  
def getAllData(scelta):  
 record = settaData(scelta)  
 all = record  
 return all

## Implementazione Main

Nel nostro Main importiamo dapprima tutti i file e le librerie necessari, quali ad esempio i file Connection e getJson, e le librerie *time*, utile per gestire e quantificare il tempo di ciascuna query, e *matplotlib.pyplot*, la quale deve essere prima inclusa nell'IDE come fatto in precedenza con *cassandra-cluster*. La funzione eliminaTuttiDati() consente di *"inizializzare"* e azzerare il DataBase:

import Connection  
import getJson  
import time  
import matplotlib.pyplot as plt  
  
  
def eliminaTuttiDati():  
 tab = ["patient"]  
 i = 0  
 while (i < len(tab)):  
 query = "truncate " + tab[i] + ";"  
 Connection.update(query)  
 i = i + 1

Per l'inserimento dei dati contenuti nel Dataset scelto all'interno del DataBase è stata creata la funzione inserisciAllData(scelta), dove appunto scelta indica quale Dataset caricare. Un fatto rilevante è il caricamento dei vari *step\_datas*, che sono array di array, all'interno del DataBase: è stato necessario dapprima definire un nuovo tipo che abbiamo chiamato *lista*, e successivamente impostare *step\_datas* come **list**.

CREATE TYPE lista  
(  
 step\_value list<text>  
);  
  
...  
step\_datas list<lista>  
...

Per la gestione di questo tipo di dato è stato necessario adoperare un ciclo for all'interno del quale comporre la stringa che verrà utilizzata come query, come si evince dal codice seguente. NB: bisogna valutare, attraverso un *if*, se il dato all'interno dell'array è un tipo *float* o *None*.

def inserisciAllData(scelta):  
 people = getJson.getAllData(scelta)  
 step = ''  
 for x in people:  
 step = ''  
 query = "insert into patient (id, name, width, height, l\_shank, l\_thigh, " \  
 "lokomat\_shank, lokomat\_thigh, lokomat\_recorded, version, legtype, " \  
 "lwalk\_training\_duration, lwalk\_distance, step\_datas) " \  
 "values ("+ str(x['id'])+", '" + str(x['name']) + "', " + str(x['width']) \  
 + ", "+ str(x['height']) +", "+ str(x['l\_shank'])+", "+ str(x['l\_thigh']) \  
 + ", " + str(x['lokomat\_shank'])+", "+ str(x['lokomat\_thigh'])+", " \   
 + str(x['lokomat\_recorded'])+", '"+ str(x['version'])+"', '" \  
 + str(x['legtype'])+"', "+ str(x['lwalk\_training\_duration']) \  
 +", "+ str(x['lwalk\_distance'])+", "  
 step += "["  
 for s in x['step\_datas']:  
 step += "{step\_value: "  
 step += "["  
 for a in s:  
 if (a == None):  
 step += "'None', "  
 else:  
 step += "'"+str(a)+"', "  
 step = step[:-2]  
 step += "]}, "  
 if(len(x['step\_datas']) > 0):  
 step = step[:-2]  
 step += "]"  
 query += step  
 query += ");"  
  
 Connection.update(query)

L'ultima funzione da creare è stata quella che, in base al Dataset selezionato, ha permesso la misurazione del tempo necessario per effettuare ogni query. Nello specifico, è stata isolata (e registrata) la misurazione della prima query, e successivamente è stata presa in considerazione la misurazione media delle altre 30 query. I risultati sono stati plottati grazie alla libreria *matplotlib.pyplot* (abbreviata per comodità nel codice con *plt*), attraverso la funzione *plot*, utilizzando le due label dell'asse *x* e *y*, e la funzione *savefig* che ha permesso la creazione dell'immagine con estensione **png**.

def funzione(scelta):  
 tempi = []  
 for \_ in range(0,31):  
 tPrima = time.time()  
 risultato = Connection.query("select \* from patient;")  
 tTotale = time.time() - tPrima  
 tempi.append(tTotale)  
  
 plt.plot(tempi)  
 plt.ylabel("Query n " + str(1))  
 plt.xlabel(scelta)  
 plt.savefig('doc/' + str(scelta) + 'Query' + str(1) + '.png')  
 plt.clf()  
  
 print "ESECUZIONE QUERY 1"  
  
 tempi = []  
 for \_ in range(0,31):  
 tPrima = time.time()  
 risultato = Connection.query("SELECT \* FROM patient WHERE name = 'SIVV33W0' " \  
 "allow filtering;")  
 tTotale = time.time() - tPrima  
 tempi.append(tTotale)  
  
 plt.plot(tempi)  
 plt.ylabel("Query n " + str(2))  
 plt.xlabel(scelta)  
 plt.savefig('doc/' + str(scelta) + 'Query' + str(2) + '.png')  
 plt.clf()  
  
 print "ESECUZIONE QUERY 2"  
  
 tempi = []  
 x = []  
 for \_ in range(0,31):  
 tPrima = time.time()  
 risultato = Connection.query("select \* from patient where" \  
 "lwalk\_training\_duration > 5000 allow filtering;")  
 for a in risultato:  
 if(a[11] != None):  
 if((a[13]!=0.70338464) and (len(a[11])>4)):  
 x.append(a)  
 tTotale = time.time() - tPrima  
 tempi.append(tTotale)  
  
 plt.plot(tempi)  
 plt.ylabel("Query n " + str(3))  
 plt.xlabel(scelta)  
 plt.savefig('doc/' + str(scelta) + 'Query' + str(3) + '.png')  
 plt.clf()  
  
 print "ESECUZIONE QUERY 3"

Infine, è stato automatizzato il processo di selezione del Dataset attraverso il ciclo *while*, che itera le varie scelte e dunque i vari Dataset.

scelta = 0  
while (scelta < 4):  
 print scelta  
  
 print "elimino elementi dal DB"  
 eliminaTuttiDati()  
 print "inserisco gli elementi dal DB"  
 inserisciAllData(scelta)  
 print "inizio le query"  
  
 funzione(scelta)  
  
 scelta = scelta + 1

# Esperimenti (descrivere come sono stati condotti gli esperimenti e mostrare e commentare i grafici risultanti. I grafici dovranno considerare i valori medi e gli intervalli di confidenza al 95%(es. per gli istogrammi la parte superiore dovrà essere inclusa tra due barre che rappresentano gli intervalli di confidenza))

# Conclusioni

In conclusione possiamo affermare come, servendoci dei grafici per rafforzare la nostra tesi, nel nostro caso di utilizzo, che ricordiamo essere quello di un'applicazione che non utilizza Cluster bensì un solo elaboratore, la soluzione DBMS Cassandra appare sofferente rispetto qualsiasi altra scelta NoSql (vedi prestazioni assolutamente superiori dei vari **MongoDB** e **HBase**, senza disprezzare *Neo4J*), nonostante online si possa reperire molta documentazione, operazione che rende agevole il *setting* dell'ambiente di testing.