Università degli studi di Messina
Dipartimento di Matematica e Informatica

Progetto Database 2

Realizzare un database basato su Hbase di un Hospital Information System (HIS)

Studenti: Castano Marco, Materia Gianluca

Matricole: 445811,446434

A.A 2015/2016

Professore: Antonio Celesti

Indice

1. Introduzione

Si richiede la realizzazione di un progetto che confronti le performance di diversi DBMS NoSql su uno schema preesistente e comune. Le performance andranno ad essere analizzate effettuando un set di stesse query su tutti i DBMS. Il dataset è stato concordato tra i vari gruppi e generato casualmente. Sono stati generati diversi da dataset da 10,100,1000,10000 record. Il lavoro è stato suddiviso in gruppi per ogni database che consistono in:

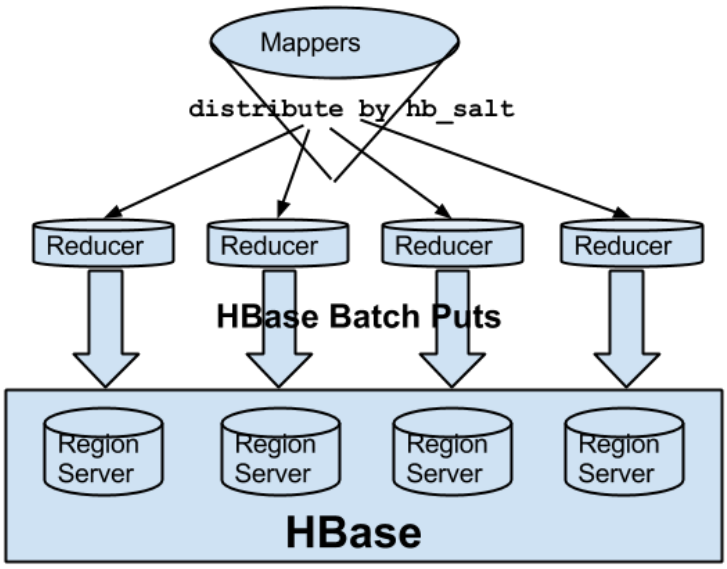
- Cassandra
- MongoDB
- Neo4J
- Hbase

Si richiede infine di andare a rappresentare i risultati per mezzo di istogrammi andando ad analizzare un intervallo di confidenza del 95%.

Il DBMS in questione è Hbase.

2. Scenario di riferimento

L'architettura di Hbase prevede dei region servers e master server. Le region server, che sono processi software chiamati demoni, vengono attivati per immagazzinare e recuperare dati in Hbase. Quando una tabella cresce oltre la configurazione di Hbase, questo automaticamente divide la tabella e distribuisce il carico su un'altro region server. Questo viene chiamato auto-sharding.
Ogni oggetto immagazzinato in una famiglia ha una cache di lettura chiamata BlockCache e una cache di scrittura chiamata MemCache.
Il design di Hbase è scorrere i dati nelle famiglie dal MemStore a un Hfile per ogni scorrimento. Ad intervalli regolari Hfiles sono combinato in un file più grande. Questo fenomeno prende il nome di compattazione.
Le compattazioni possono essere minori e maggiori.
Quelle minore combinano un numero di piccoli Hfiles in uno più grande e sono importanti perché senza di loro la lettura di una particolare riga potrebbe richiedere molte letture sul disco e performance lente.
Quelle maggiori cerca di combinare tutti gli Hfile in uno più grande e in più pulisce i record dopo che sono stati eliminati.
I master server monitorano i region servers nei cluster Hbase. Gestiscono le operazioni sui metadati, assegnano le region e gestiscono i problemi nelle regioni. Gestiscono inoltre il balancing. Ci dovrebbe sempre essere un backup master server nel caso in cui l'attuale master server ha avuto problemi.
Hbase cluster possono avere numerosi problemi di coordinazione tra i master Servers e region Servers , e i client possono avere compiti difficili, proprio per questo viene utilizzato Zookeeper che è un cluster distribuito di server che provvede la sincronizzazione dei servizi.


Mappers
distribute by hb_salt
Reducer
Reducer
Reducer
Reducer
HBase Batch Puts
Region Server
Region Server
Region Server
Region Server
HBase

3. Ambiente di sviluppo

L'applicazione è stata implementata utilizzando il linguaggio di programmazione Python e Hbase come DBMS.
L'editor utilizzato per la stesura del codice è PyCharm.
Sono state utilizzate le seguenti librerie:

- matpotlib: libreria grafica di python
- happybase: libreria per la configurazione di hbase
- json: libreria per la gestione dei file json

4. Architettura applicazione

L'architettura dell'applicazione è stata strutturata cercando di massimizzare la modularità e la riusabilità.

Sono stati utilizzati 6 scripts Python per l'intero progetto :

- connection_db.py: utilizzato per la connessione al database tramite happybase.
- import_dataset.py: per andare ad effettuare i test sui tipi diversi di dataset composti da diversi elementi è stato utilizzato uno script che permette il facile import del file json.
- table_generator.py: utilizzato per poter generare le tabelle che compongono il database.
- table_pusher.py: utilizzato per poter popolare facilmente le tabelle che vengono generate da 'table_generator.py'.
- query_executor.py: utilizzato per eseguire le query ed effettuare il controllo sulle prestazioni che verranno mostrate tramite degli istogrammi.
- main.py: utilizzato per iniziare la simulazione.

5. Implementazione

Per quanto riguarda l'implementazione di seguito sono riportati gli elementi che compongono la progettazione e il modo in cui sono stati collegati fra di loro.

- connection_db:
  Per l'utilizzo di Hbase tramite Python è stata utilizzata la libreria happybase in grado di sincronizzare e permettere un utilizzo più efficiente e veloce di Hbase e della sua architettura.
  Tramite questo metodo è possibile connettersi al server specificando indirizzo e porta.

```
import happybase

def database_connection():
    connection = happybase.Connection('127.0.0.1', 9090)
    connection.open()
    return connection
```

- import_dataset:
  Per andare ad effettuare i test sui tipi diversi di dataset composti da diversi elementi è stato utilizzato il seguente metodo che dato il path permette il facile import del file json.

```
import json

def import_data_set(path):

    with open(path) as data_file:
        data_set = json.load(data_file)
    return data_set
```

- table_generator:
  tramite i seguenti metodi è possibile andare a creare, una volta instaurata la connessione, la tabella dei pazienti che conterrà al suo interno due famiglie : analysis, step_datas.

```
def generate_patient(connection):
    connection.create_table \
            (
            'patient',
            {
                'analysis': {},
                'step_datas':{}
            }
        )

def degenerate_patient(connection):
    connection.delete_table('patient', disable='true')
```

- table_pusher:
  tramite l'elemento table_pusher è stato possibile andare a popolare il database. Passando infatti il file json dei pazienti è possibile effettuare un parse dei dati inserendoli negli appositi campi che nel caso di Hbase vengono generati dinamicamente al momento dell'inserimento.

```
def push_patients(connection,dict_patients):

    patient_table = connection.table('patient', use_prefix='true')
    b = patient_table.batch()
    for patient in dict_patients:
        b.put('row-key'+str(patient['id']),
              {'analysis:id': str(patient['id']),
               'analysis:width': str(patient['width']),
               'analysis:lokomat_shank': str(patient['lokomat_shank']),
               'analysis:lokomat_thigh': str(patient['lokomat_thigh']),
               'analysis:lwalk_distance': str(patient['lwalk_distance']),
               'analysis:height': str(patient['height']),
               'analysis:version': str(patient['version']),
               'analysis:legtype': str(patient['legtype']),
               'analysis:lwalk_training_duration': str(patient['lwalk_training_duration']),
               'analysis:name': str(patient['name']),
               'analysis:l_shank': str(patient['l_shank']),
               'analysis:l_thigh': str(patient['l_thigh']),
               'analysis:lokomat_recorded': str(patient['lokomat_recorded'])
              })
        i=0
        for data in patient['step_datas']:

            for single_data in data:

                b.put('row-key' + str(patient['id']),
                      {
                          'step_datas:step_datas'+str(i): str(single_data)
                      })
                i = i + 1

    b.send()
```

- Query_executor:
  Il seguente script è stato utilizzato per effettuare le query scelte. Per ogni query viene utilizzata la connessione e la libreria 'Datetime' per poter ricavare i tempi d'interrogazione. Il metodo utilizzato per interrogare il database è 'scan()' a cui è possibile passare le colonne da dove verranno presi i dati, dei parametri per poter filtrare i dati. Le query che sono state scelte sono le seguenti:

  - ❖ get_info_patients: ricava tutti dati di tutti i pazienti
  - ❖ get_patient_with_name: ricava tutti i dati dei pazienti con uno specifico nome
  - ❖ get_misuration_patient: ricava tutti i dati dei pazienti che hanno un 'lwalk_training_duration' minore a 2400, una 'width' diversa da 0.80 e che hanno effettuato almeno 5 misurazioni.

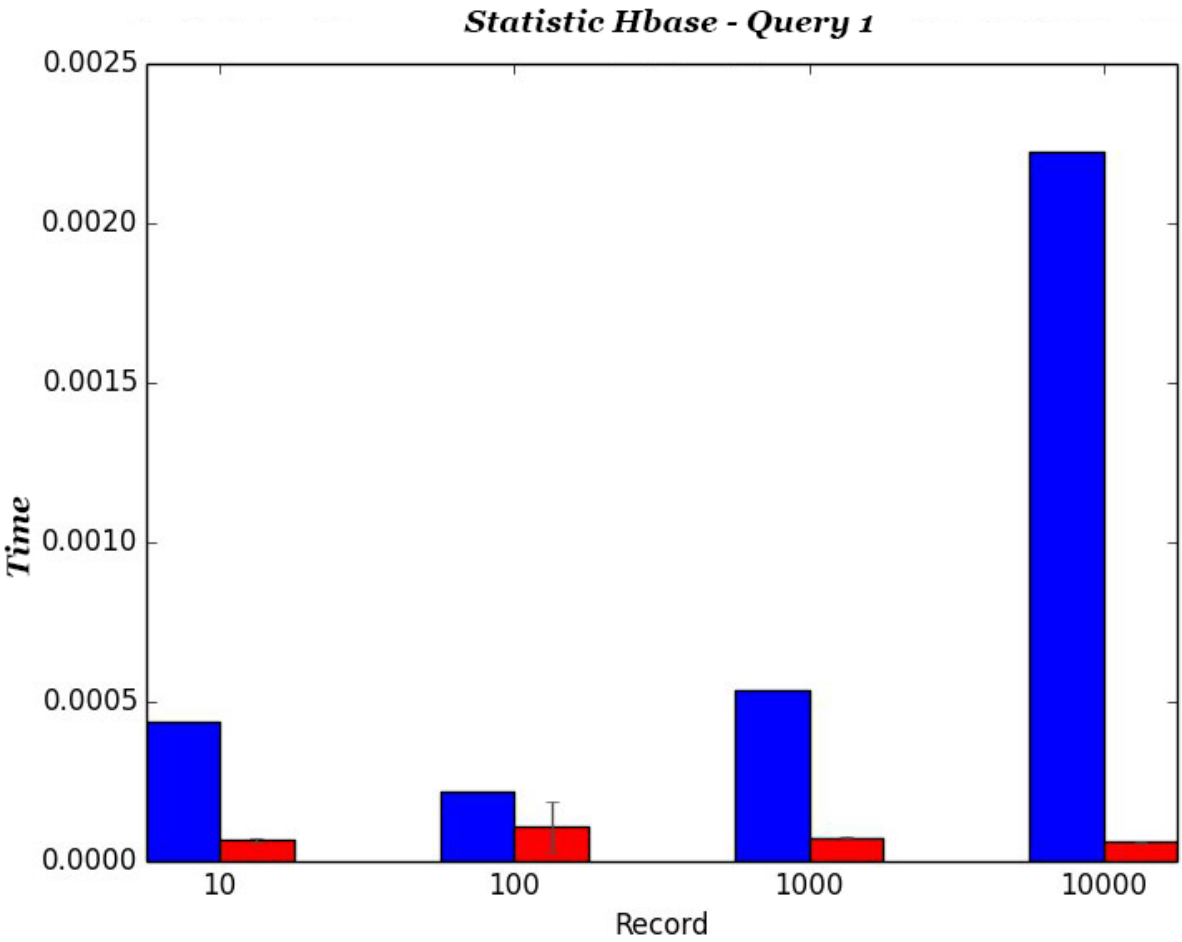
```
def get_info_patients(connection, dataset):
    time = datetime.datetime.now()
    patients_table = connection.table('patient')
    patients_table.scan(columns=
                        ['analysis:width',
                         'analysis:lokomat_shank',
                         'analysis:lokomat_thigh',
                         'analysis:lwalk_distance',
                         'analysis:height',
                         'analysis:id',
                         'analysis:legtype',
                         'analysis:lwalk_training_duration',
                         'analysis:name',
                         'analysis:l_shank',
                         'analysis:l_thigh',
                         'analysis:lokomat_recorded',
                         'analysis:step_datas'])
    save_time(str(datetime.datetime.now() - time),"get_info_patients","stat",dataset)
    return datetime.datetime.now() - time

def get_patient_with_name(connection,dataset):
    time = datetime.datetime.now()
    patients_table = connection.table('patient')
    patients_table.scan(filter="SingleColumnValueFilter ('analysis','name',=,'regexstring:^KRVYRSKX7')")
    save_time(str(datetime.datetime.now() - time),"get_patient_with_name","stat",dataset)
    return datetime.datetime.now() - time

def get_misuration__patient(connection,dataset):
    time = datetime.datetime.now()
    patients_table = connection.table('patient')
    patients_table.scan(filter="SingleColumnValueFilter ('analysis','lwalk_training_duration',<,2400) AND ('analysis','width',!=,0.80) AND ('analysis','step_datas',>,5)")
    save_time(str(datetime.datetime.now() - time),"get_misuration__patient","stat",dataset)
    return datetime.datetime.now() - time
```

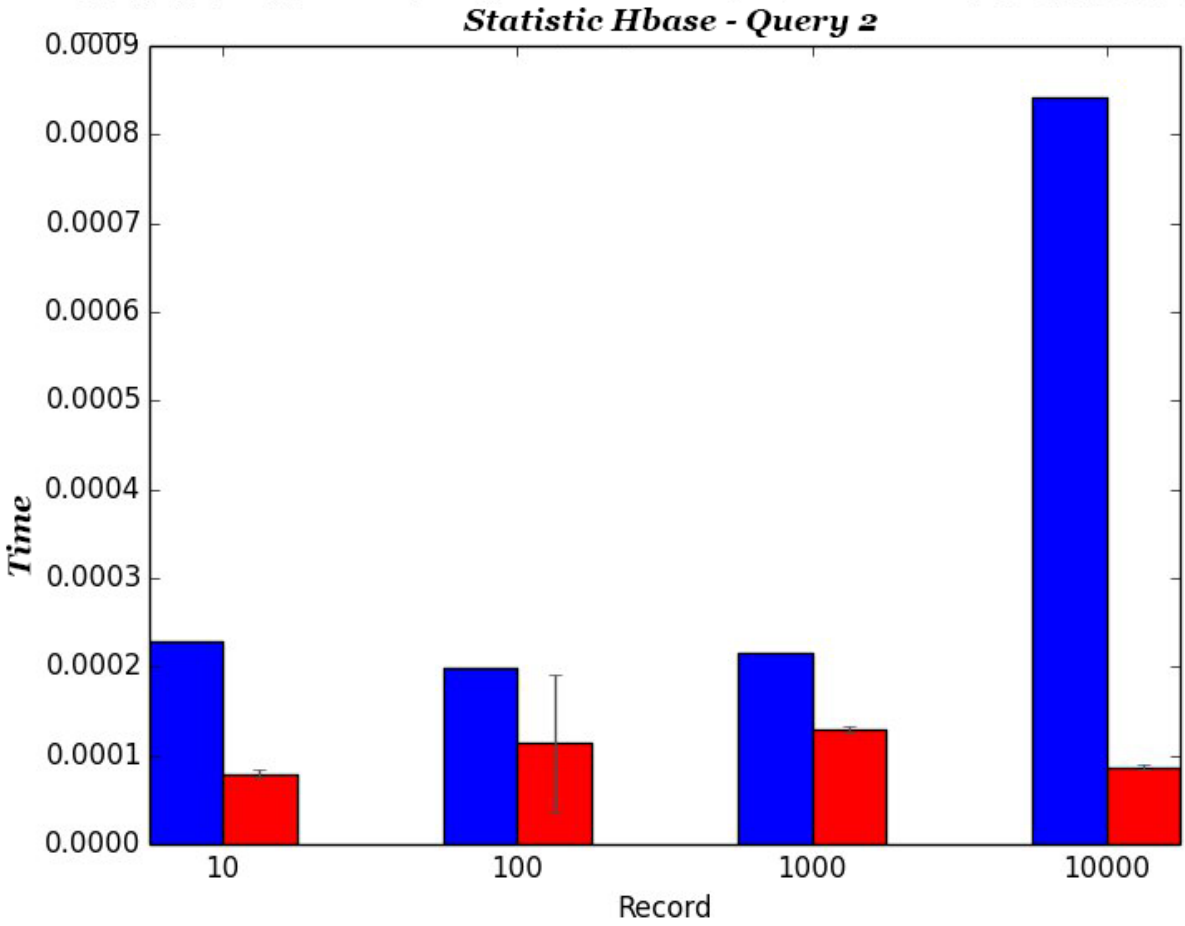
- Main

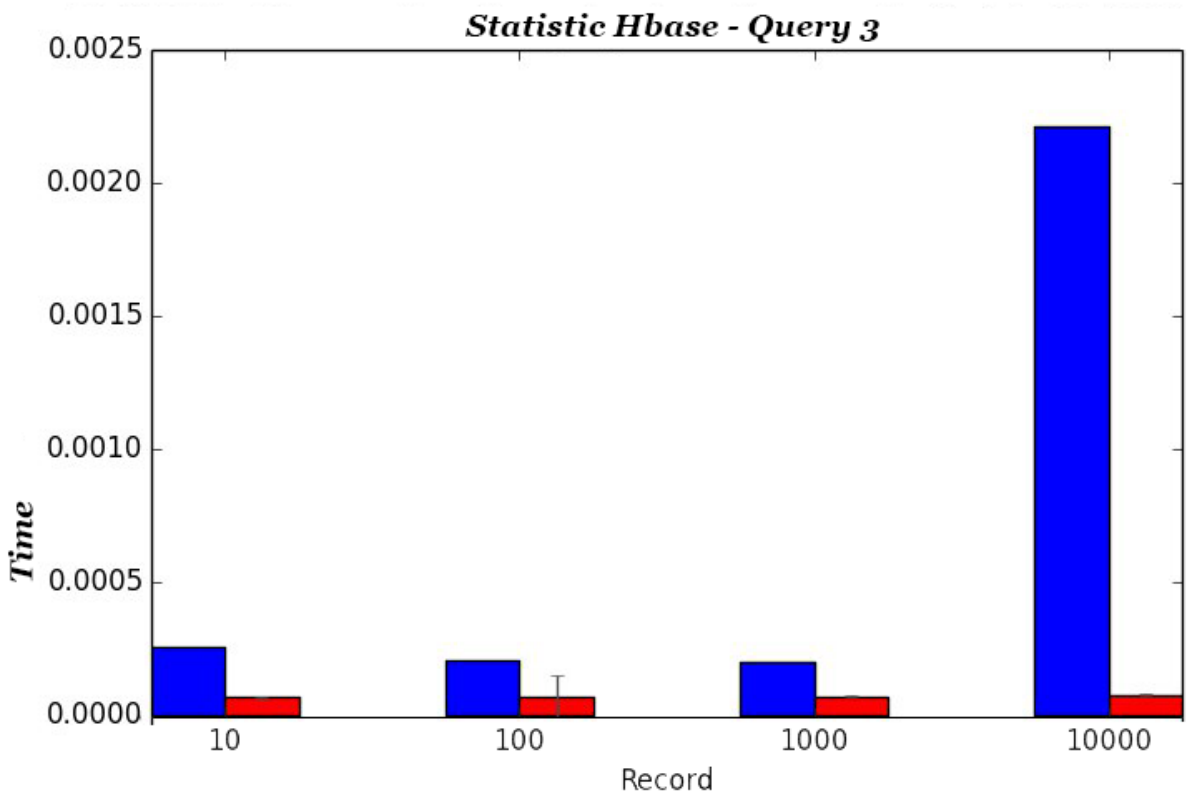
```
dsr =['ds10.json','ds100.json','ds1000.json','ds10000.json']
aqr=[query_executor.get_info_patients,query_executor.get_patient_with_name,query_executor.get_misuration__patient]
connection = connection_db.database_connection()
all_media = []
first_time = []
array_all_time = []
time = datetime.datetime.now()

for q in aqr:

    for ds in dsr:
        data_set = import_dataset.import_data_set(ds)
        table_generator.degenerate_patient(connection)
        table_generator.generate_patient(connection)
        table_pusher.push_patients(connection, data_set)

        media = 0
        all_time = []
        for _ in range(0,31):
            single_time = q(connection, str(ds))
            t = single_time.total_seconds()
            if(_ == 0):
                first_time.append(t)
            else:
                media = media + t
                all_time.append(t)
        array_all_time.append(all_time)
        all_media.append(media/30)


    query_executor.create_graph(first_time,all_media,array_all_time,q)
    print("completed" + str(q))
    del all_media[:]
    del first_time[:]

print("TEMPO TOTALE")
print(datetime.datetime.now() - time)
```

6. Esperimenti

I risultati ottenuti dalla simulazione sono stati studiati attraverso degli istogrammi. Ogni query viene eseguita 31 volte al fine di poter studiare il fenomeno di caching. Per ogni query troviamo un istogramma che rappresenta per ogni dataset il tempo di esecuzione della prima volta e delle restanti 30.




Statistic Hbase - Query 2
Time
0.0009
0.0008
0.0007
0.0006
0.0005
0.0004
0.0003
0.0002
0.0001
0.0000
10
100
1000
10000
Record

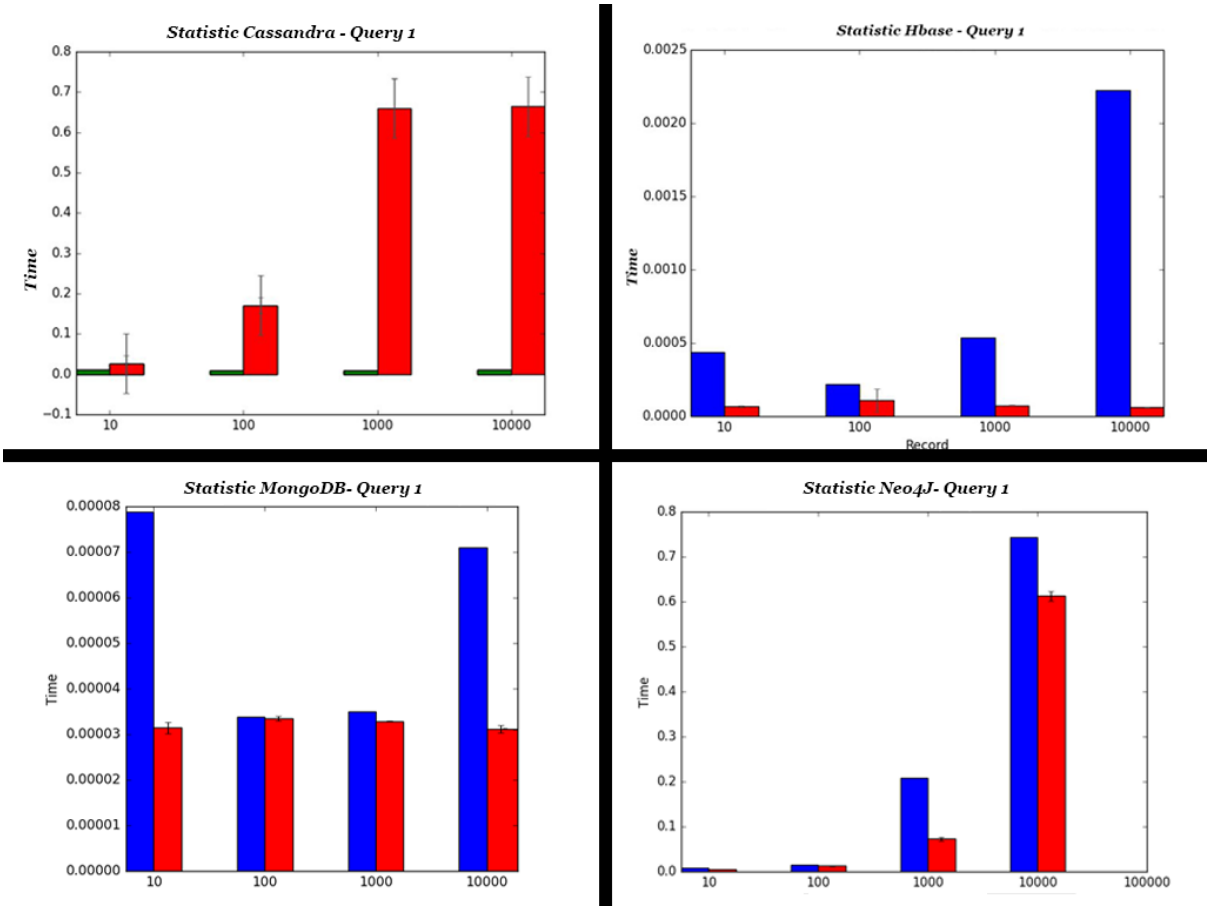


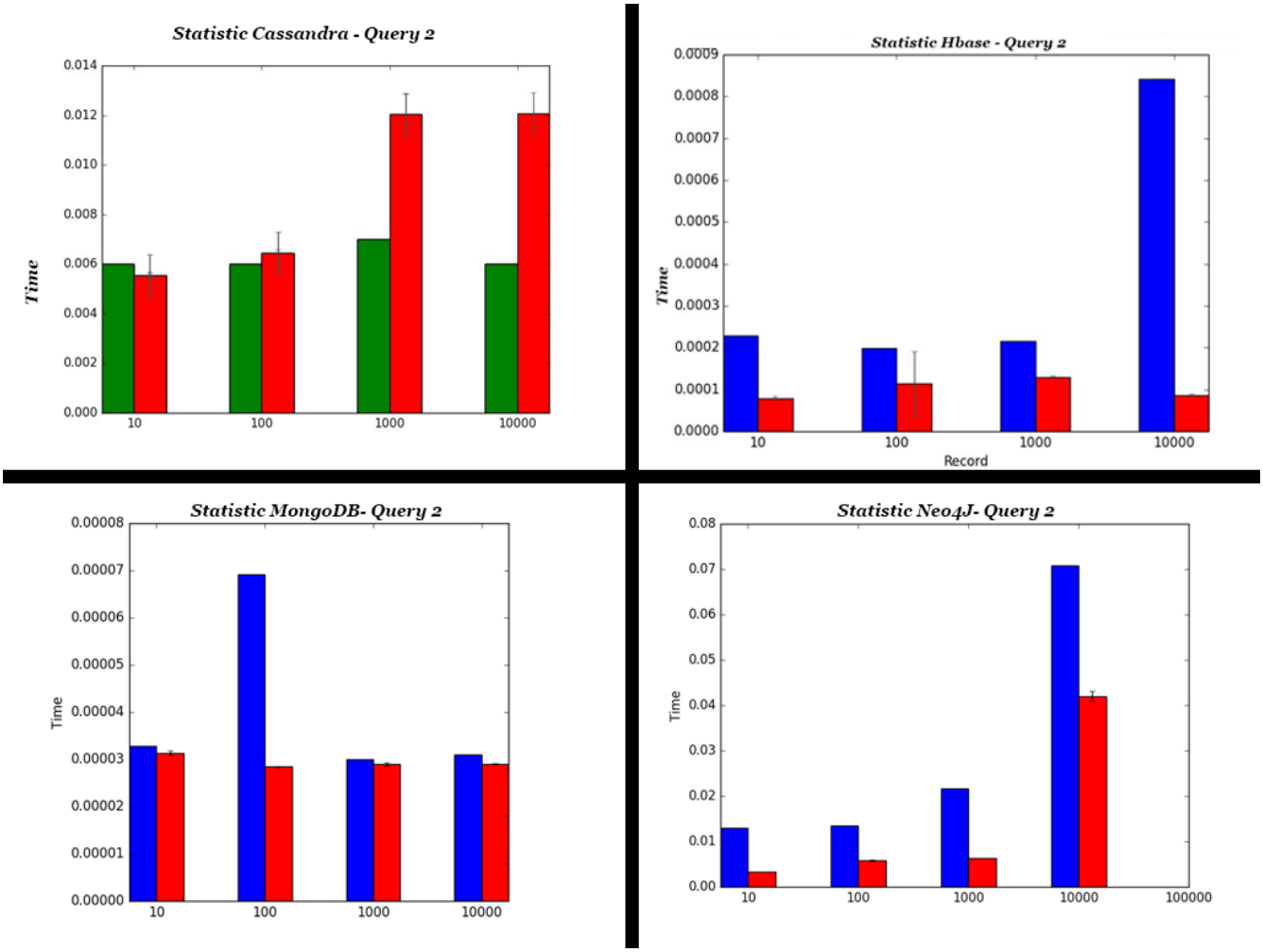
Come possiamo vedere dai grafici precedenti Hbase utilizza molto la cache per andare a salvare i risultati precedentemente ottenuti. Dal grafico risulta che i tempi d'interrogazioni sono molto brevi e grazie al sistema di caching la media dei tempi delle ultime 30 query è notevolmente minore rispetto alla prima.

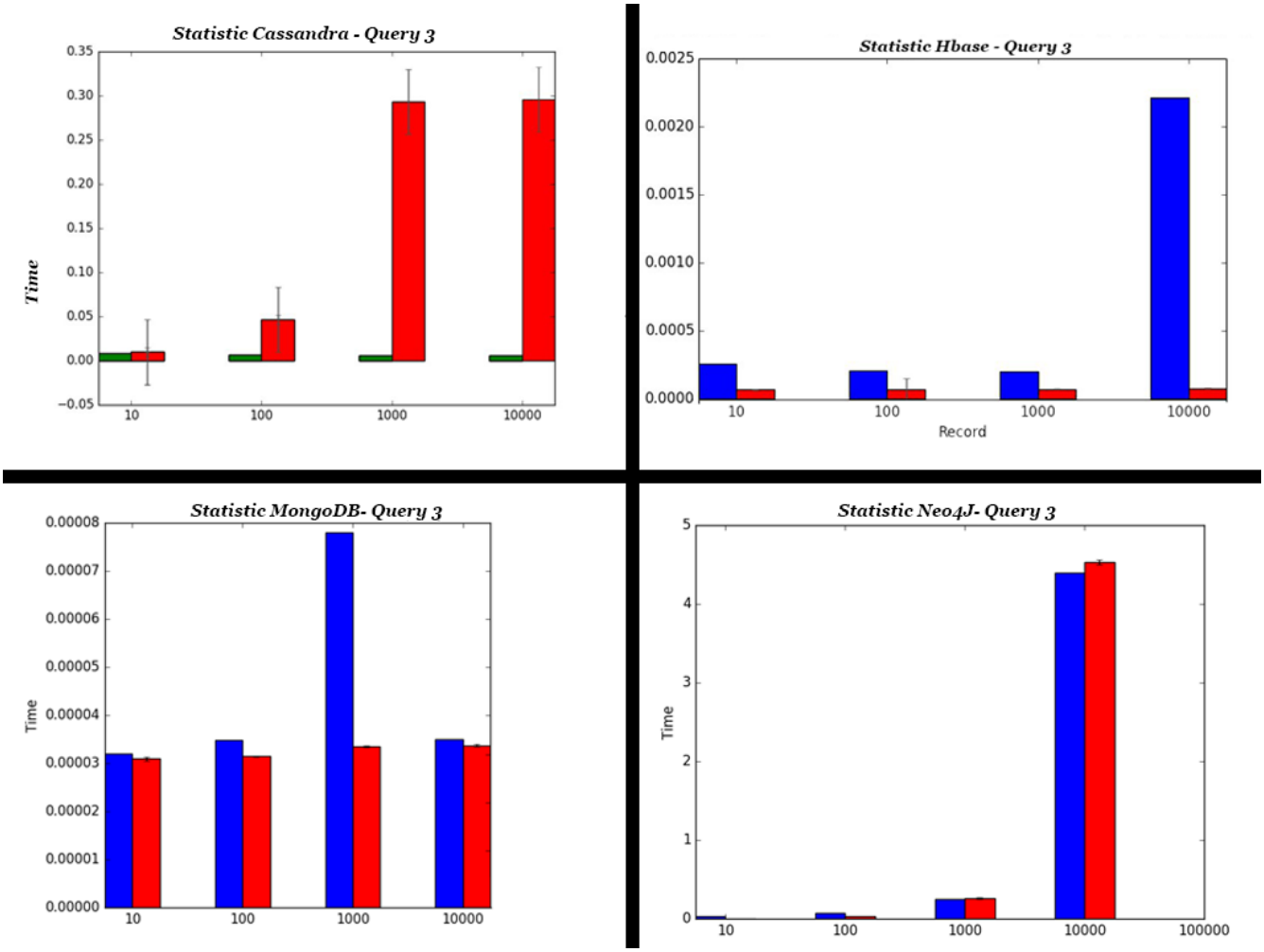
7. Conclusioni

In conclusione possiamo affermare dopo aver confrontato i vari grafici che riportano i tempi di esecuzione delle diverse query eseguite su tutti e quattro i tipi di DBMS noSql che Hbase è più lento di MongoDB ma più veloce di Cassandra e Neo4J.
Possiamo infatti notare che le performance sono di un ordine di grandezza inferiore al primo ma superiori di un ordine di grandezza rispetto a Cassandra e di due ordini di grandezza rispetto a Neo4J. Possiamo giustificare la lentezza rispetto a MongoDB causato probabilmente dall'interazione tra i vari componenti dell'architettura Hbase che vengono gestiti tramite ZooKeeper.

Di seguito vengono riportati i grafici a confronto delle tre query.




Statistic Cassandra - Query 2
Time
0.014
0.012
0.010
0.008
0.006
0.004
0.002
0.000
10
100
1000
10000
Statistic Hbase - Query 2
Time
0.0009
0.0008
0.0007
0.0006
0.0005
0.0004
0.0003
0.0002
0.0001
0.0000
10
100
1000
10000
Record
Statistic MongoDB- Query 2
Time
0.00008
0.00007
0.00006
0.00005
0.00004
0.00003
0.00002
0.00001
0.00000
10
100
1000
10000
Statistic Neo4J- Query 2
Time
0.08
0.07
0.06
0.05
0.04
0.03
0.02
0.01
0.00
10
100
1000
10000
100000


Statistic Cassandra - Query 3
Time
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
−0.05
10
100
1000
10000
Statistic Hbase - Query 3
0.0025
0.0020
0.0015
0.0010
0.0005
0.0000
10
100
1000
10000
Record
Statistic MongoDB- Query 3
Time
0.00008
0.00007
0.00006
0.00005
0.00004
0.00003
0.00002
0.00001
0.00000
10
100
1000
10000
Statistic Neo4J- Query 3
Time
5
4
3
2
1
0
10
100
1000
10000
100000