NoSQL
Domande di ripasso #1

1 Consistenza in DB distribuiti

Si fornisca l'enunciato e una dimostrazione intuitiva del *CAP Theorem*.

2 Consistenza in DB distribuiti

Si definisca la *linearizzabilità* nei DB distribuiti e si fornisca un esempio in cui tale proprietà non sussiste.

3 DB documentali

Un sistema informativo deve contenere dati relativi ad automobili e relative rilevazioni da parte di un sistema di monitoraggio mediante sensori (telecamere) posti in una cittadina. Di ogni automobile si devono rappresentare: numero di targa, marca, modello, cilindrata in cc e anno di immatricolazione. Di ogni rilevazione si devono rappresentare: data e ora, luogo (coordinate), e concentrazione di polveri fini PM10 in $\mu g/m^3$ al momento della rilevazione. Si stima che la base di dati dovrà contenere circa 1.000 auto e 5.000.000 di rilevazioni. Inoltre si prevede che saranno eseguite le seguenti interrogazioni con relative frequenze:

Q1: Data una cilindrata c, trovare tutte le posizioni delle rilevazioni relative ad auto con cilindrata inferiore a c (100 query al giorno).
Q2: Data una targa, trovare tutte le rilevazioni relative all'auto con detta targa (2.000 query al giorno).
Q3: Dato un anno a, trovare tutte le posizioni delle rilevazioni relative ad auto immatricolata in anni precedenti a (200 query al giorno).

Si illustri come memorizzare tali dati in MongoDB considerando le query e la loro frequenza; si formulino le query in questione in MongoDB (con un valore a piacere per cilindrata, targa e anno rispettivamente). Tutte le scelte effettuate dovranno essere concisamente giustificate.

4 Datalog

Si consideri un grafo orientato rappresentato dal predicato binario `e`; `e(a,b)`, per esempio, indica che c'è un arco da a a `b`. Si prenda ora la seguente query:

> Determinare tutti i nodi che hanno almeno due successori.

Si determini se detta query si può rappresentare in Datalog; in caso di risposta positiva, fornire la rappresentazione; in caso di risposta negativa, giustificare la risposta.

Soluzioni (tracce)

1 Consistenza in DB distribuiti

Consistency: *Ogni* operazione di lettura riflette l'aggiornamento/scrittura piú recente[1].

Availability: Ogni richiesta inviata ad un nodo che non è "spento" riceve una risposta (senza garanzia che tale risposta rifletta l'ultimio aggiornamento).

Partition-tolerance: Il sistema continua a funzionare (cioè a rispondere a richieste di lettura/scrittura) anche in caso di interruzioni delle comunicazioni che dividono i nodi in partizioni che non possono comunicare tra di loro.

Il *CAP Theorem* si può dimostrare (intuitivamente, giacché la dimostrazione formale pubblicata in letteratura è basata su specifiche assunzioni) in modo semplice considerando due nodi N_1, N_2 che contengono lo stesso valore V_0 per la variabile V (v. articolo). Ad un certo punto N_1 aggiorna il valore a V_1 e comunica l'aggiornamento (*update*) a N_2. In caso di interruzione della comunicazione (*partition*), N_2 rimane col valore V_0. In caso di messaggi sincroni, N_1 rimane bloccato in attesa della conferma da parte di N_2, che non arriva; quindi per garantire la *consistency* si considerano scrittura e invio del messaggio come operazione atomica, e pertanto si perde la *availability* per problemi di blocco/latenza. Se i messaggi sono asincroni, N_1 non può sapere se e quando M_2 riceve il messaggio, pertanto richieste che arrivano a N_2 dopo la scrittura su N_1 restituiranno il vecchio valore V_0, e la *consistency* sarà persa. Il problema sta nella *latenza* nella trasmissione del messaggio da N_1 a N_2, e soprattutto nel fatto che N_1 *non può avere controllo su quando letture del valore di* V *in* N_2 *avverranno*, cosa che in un sistema centralizzato si risolverebbe con un semplice *lock*, data la latenza minima; pertanto non si può garantire che successive letture su N_2 restituiscano il valore aggiornato V_1; qualsiasi misura tesa a garantire la *consistency* annullerà la *availability* o la *partition tolerance*.

2 Consistenza in DB distribuiti

Si definisca la *linearizzabilità* nei DB distribuiti e si fornisca un esempio in cui tale proprietà non sussiste.

Si ha linerizzabilità quando sussiste la seg. proprietà:

[1] Si noti che la definizione di *consistency* in NoSQL for Mere Mortals (pag. 56) è piuttosto vaga; infatti la *consistency* in questo caso non si riferisce a vincoli di integrità quali quelli nei DB centralizzati, ma a inconsistenze tra copie diverse in DB distribuiti.

> *Se un'operazione B viene eseguita dopo che un'operazione A è stata completata con successo, l'operazione B deve vedere il sistema nello stesso stato in cui era al completamento di A, o in uno stato piú recente.*

Un esempio si trova alla trasparenza 23 di quelle della lezione 4; esempî analoghi possono essere facilmente costruiti.

3 DB documentali

Nei DB documentali è spesso molto utile procedere a *denormalizzare* i documenti con cui si rappresentano i dati. Nel caso (didattico) in questione, tutte le query date "navigano" entrambe i dati partendo dall'auto e poi andando ad estrarre le relative rilevazioni[2]; tuttavia, siccome esistono in media circa 5.000 rilevazioni per ogni automobile, non sarebbe utile denormalizzare includendo tutte le rilevazioni relative ad un'automobile nel documento che rappresenta l'automobile stessa. Peraltro, con 5.000 rilevazioni per ogni automobile non sarebbe nemmeno una buona scelta inserire tutti i *riferimenti* alle rilevazioni di un'automobile nel documento che rappresenta l'automobile stessa. Pertanto rappresentiamo le automobili con dei documenti specifici, e inseriamo nei documenti relativi alle rilevazioni le seguenti informazioni: *(1)* un riferimento all'automobile relativa (*parent referencing*); *(2)* la cilindrata di detta automobile; *(3)* l'anno di immatricolazione della medesima automobile. La denormalizzazione consiste nell'inserire *(2)* e *(3)* per ogni rilevazione; ciò ha un costo in termini di spazio ma consente di rispondere a Q1 e Q3 cercando solo tra le rilevazioni.

Creiamo pertanto una collezione `automobili` con documenti come il seguente:

```
{
  "_id": {
    "$oid": "6671f367af7d84179afd3bc2"
  },
  "targa": "AZ713FF",
  "marca": "FIAT",
  "modello": "Panda",
  "cilindrata": 1242
}
```

Inoltre creiamo una collezione `rilevazioni` con documenti come il seguente:

```
{
  "_id": {
    "$oid": "6671f592af7d84179afd3bc4"
  },
  "data": {
```

[2] Un caso "opposto" sarebbe una query che estrae, per esempio, le automobili di cui esistono rilevazioni in un'area specifica.

```
    "$date": "2024-06-16T17:54:41.020Z"
  },
  "luogo": [
    70.456633,
    -45.455522
  ],
  "idAuto": {
    "$oid": "6671f367af7d84179afd3bc2"
  },
  "PM10": 12.32,
  "cilindrataAuto": 1242,
  "annoAuto": 2021
}
```

A questo punto Q3 può essere formulata come segue:

```
db.rilevazioni.find(
  { "annoAuto": { "$lt": 2020 } },
  { "luogo": 1, "_id": 0 }
)
```

La formulazione di Q1 è analoga. Per quanto riguarda Q2, essa *deve essere divisa in due query* MongoDB: con la prima si estrae l'id dell'automobile con la targa data in ingresso, e con la seconda si cercano nella collezione `misurazioni` le misurazioni relative all'auto in questione, trovate con l'id estratto dalla prima query — attributo su cui si può definire un indice in `misurazioni` per migliorare le prestazioni rispetto a questa query.

Una spiegazione su come rappresentare relazioni uno-a-molti come quella in questione si può trovare ad esempio nella documentazione ufficiale di MongoDB.

4 Datalog

La query non può essere rappresentata in Datalog. Datalog non è in grado di determinare se due nodi sono distinti. L'insieme (congiunzione) di atomi `e(X,Y1), e(X,Y2)` si mappa su `e(a,b)`; in generale, non c'è modo di inferire atomi del IDB *solo* quando un nodo ha *almeno* due o piú successori immediati.