JSON

Il JSON (JavaScript Object Notation) è un semplice formato per lo scambio di dati. Per le persone è facile da leggere, mentre per le macchine risulta facile da generare e analizzarne la sintassi. È ideale per lo scambio di dati perché è un formato di testo indipendente dal linguaggio di programmazione, ma che utilizza convenzioni per linguaggi C based, come C++, Java, Javascript, Python, ecc. è basato su due strutture:

* Un insieme di coppie chiave valore (come ad esempio un oggetto), che è racchiuso tra due parentesi graffe
* Un elenco ordinato di valori (come, ad esempio, un array o un vettore), che è racchiuso tra due parentesi quadre. Un valore può essere una stringa tra virgolette, un numero, un valore booleano true/false o null, un oggetto o un array.

Queste sono strutture di dati universali. Infatti, virtualmente tutti i linguaggi di programmazione moderni li supportano in entrambe le forme.

NO SQL

È un termine che fa rifermento a tipi di database non relazionali. Questi database archiviano dati in un formato diverso rispetto alle tabelle tradizionali. Infatti, al loro interno ospitano una struttura dati fatta da un documento JSON. Poiché questo design di database non prevede uno schema, offre una rapida scalabilità per gestire grandi set di dati e tipicamente non strutturati. Ad esempio, in un databse NoSQL, il record di un libro è solitamente memorizzato come documento JSON. Per ogni libro, l’ISBN, il titolo, il numero dell’edizione ecc., sono archiviati come attributi in un unico documento. In questo modello i dati sono ottimizzati per lo sviluppo intuitivo e la scalabilità orizzontale. La decisione di utilizzare un database SQL rispetto ad uno NoSQL è legata al contesto e varia a seconda del caso si utilizzo.

MongoDB

MongoDB è un database NoSQL senza schema, il che vuol dire che non è necessario specificare una struttura per le tabelle/database come avviene nei database NOSQL. Viene utilizzato e preferito da aziende come Forbes e Google per la sua velocità di esecuzione delle query.

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

Una **collezione** è un gruppo di documenti associati ad un database. Sono simili alle tabelle dei database relazionali, ma sono molto più flessibili. Infatti, non dipendono da uno schema e non è necessario che i documenti al loro interno abbiano lo stesso tipo di dati.

MongoDB ha un modello di dati documentale che memorizza i dati come documenti JSON. I **documenti** corrispondo agli oggetti del codice dell’applicazione, rendendo più semplice e intuitivo l’utilizzo da parte degli sviluppatori.

MONGOSH

Mongosh è la shell di MongoDB. È un ambiente di programmazione interattivo in cui possiamo interrogare il database, ma anche eseguire comandi di JavaScript. Ecco alcuni comandi principali:

|  |  |
| --- | --- |
| show dbs | Mostra la lista dei database |
| use nomeDB | Crea un database in maniera implicita |
| db.nomeCollection.insertOne({}) | Crea un documento vuoto |
| Db.nomeCollection.find() | Ritorna tutti i valori contenuti dentro la collection |

Ogni volta che viene creata una collection, viene creato ed associato ad essa un ObjectID, che univoco per ogni collection. Esso è un valore composto da 12 byte:

* 4 byte rappresentano il numero di secondi dall’inizio dell’epoca Unix. Per questo motivo è possibile risalire alla data di creazione di un documento senza salvare un campo aggiuntivo
* 3 byte sono identificatori della macchina
* 2 byte rappresentano l’ID del processo
* 3 byte sono contatori che partono da un numero casuale

|  |  |
| --- | --- |
| db.nomeCollection.insertOne({name: “Test”, description: “Lorem ipsum”}) | Inserisce un documento, in questo caso con due accoppiate chiave – valore |
| db.nomeCollection.find({name:”Test”}) | Legge il contenuto di un documento. Se non viene passato alcun filtro, la query restituirà tutto il contenuto della collection. Il risultato della query sarà in formato JSON |
| db.nomeCollection.updateOne({ name : "Test" }, { $set: {name: "Test updated", date:  ISODate("2018-05-29T09:08:01.488Z") }}) | Permette di modificare uno o più documenti inserendo, modificando o eliminando uno o più campi. Il metodo updateOne lavora soltanto su un documento, seguendo l’ordine dell’ID, anche se molteplici documenti soddisfano le condizioni del filtro. |
| db.nomeCollection.deleteOne({name: “Test updated”}) | Rimuove uno o più documenti da una collection |

BSON

Il BSON è la versione codificata in binario dei file JSON. Viene utilizzato anche al di fuori di MongoDB. Si tratta di un formato per strutture dati semplici e array associativi (come gli oggetti).

La diffesrenza tra il JSON e il BSON è che il BSON occupa meno spazio di archiviazione ed è più veloce, ma in alcuni casi è meno efficiente a causa delle lunghezze fisse imposte e degli indici di array espliciti. Il JSON supporta solo dati supportati da JavaScript, mentre il BSON supporta anche altri dati (ad esempio il formato data). Un documento BSON è limitato ad una dimensione massima di 16 megabyte ed a un massimo di 100 livelli di nesting. Questi limiti sono punti fondamentali per la creazione della struttura del database.

|  |  |
| --- | --- |
| db.nomeCollection.updateMany({filtro}, {$set:{documento}}) | Esegue la stessa operazione dell’updateOne, con la differenza che modifica tutti i documenti che soddisfano la condizione della query di filtro |
| db.nomeColelction.replaceOne({filtro,{nuovoDocumento}}) | È un comando simile all’update, con la differenza che il documento non viene aggiornato in base ai parametri indicati, ma viene sovrascritto |

Gli operatori

Gli operatori sono simboli speciali che aiutano i compilatori a eseguire operazioni matematiche e logiche. MongoDB offre diversi operatori per interagire con il database, che vengono indicati con il simbolo **$.** Un esempio di alcuni operatori logici è il seguente:

|  |  |
| --- | --- |
| $and | Seleziona i documenti dove tutte le condizioni delle espressioni sono soddisfatte |
| $or | Seleziona i documenti in cui almeno una della due espressioni è vera |

Find e cursori

Come visto prima, il find restituisce un set di dati in base ai criteri da noi espressi. Una cosa da notare è che questo comando non restituisce tutti i dati ma un oggetto chiamato **cursore.** Questo ci permette di non elaborare i dati in una sola volta, ma di suddividere il carico in operazioni più piccole. Per avere il risultato come unico array basta utilizzare il comando **toArray.**

Projection

A volte non è sempre necessario estrarre tutto il documento. Tramite la clausola Projection possiamo infatti selezionare solo i campi del documento che ci interessano. La sintassi è la seguente:

**db.nomeCollection.find({filtro}, {campo\_da\_estrarre:1, campo\_da\_omettere:0})**

I campi da non estrarre vanno specificati perchè, ad esempio, l'ObjectID di default viene mostrato, quindi se non voglio visualizzarlo devo impostare il suo valore a 0.

Schemas e data modelling

MongoDB è schema-less, quindi non supporta gli schemi. Ma nella realtà della pratica può capitare i doverli usare. Come posso quindi decidere come strutturare un database secondo le mie esigenze?

Immagine che contiene testo, schermata, Carattere, numero

Descrizione generata automaticamente

MONGODB è orientato ai documenti e quindi agli oggetti! L'obiettivo è quello di salvare i propri dati nel formato che ti servirà nella tua applicazione.

Le relazioni

|  |  |
| --- | --- |
| Uno a uno nestata | Uno studente ha una media di voti e la media dei suoi voti appartiene solo a quello studente |
| Uno a uno relazionata | Uno studente ha una macchina e una macchina appartiene ad uno studente |
| Uno a molti nestata | Uno studente ha un set di voti (array) e quell’array appartiene solo a quello studente |
| Uno a molti relazionata | Uno studente ha una classe e una classe ha molti studenti |
| Molti a molti nestata | Un utente può ordinare molti prodotti sono ordinati da molti utenti |
| Uno a molti relazionata | Un libro ha molti autori e un autore ha scritto molti libri |

Gli indici

Un indice in MongoDB è una struttura in cui vengono conservate porzioni di dati di una collezione. Per gli indici vengono sfruttate delle strutture di dati efficienti e veloci da scansionare in modo da rendere le query più performanti. Per ogni collezione si possono creare diversi indici in base alle query da eseguire. Essi saranno poi usati internamente o come struttura di supporto per la ricerca efficiente dei dati, riducendo così il numero di documenti da esaminare.

MongoDB, in assenza di indici, esegue sempre una **COLLECTIONSCAN**, ovvero va a verificare l’intera collection, documento per documento. Se però una collection è molto grande, si può andare ad incappare in problemi di performance, ed è proprio per questo che vengono utilizzati gli indici. Questi però non rimpiazzano la collectiom, ma sono una aggiunta. In poche parole, quindi, l’indice è una lista ordinata di valori relativi ai campi che indichiamo. Ognuno di questi oggetti ha un “puntatore” che mette in rifermento la lista dell’indice con il relativo documento nella collection. Questo permette a MongoDB di effettuare un **Index Scan**, utilizzando la lista appena creata per poi puntare al documento interessato.

**NB: Una volta creati, gli indici vengono mantenuti da MongoDB, che li aggiorna ad ogni Insert. Questo va ad appesantire il processo e il database stesso, quindi non bisogna abusarne.**

|  |  |
| --- | --- |
| db.nomeCollection.createIndex({nomeCampo: 1/-1 (ASC/DESC)}) | Permette di creare un indice e di specificare l’ordine ascendente (1) o discendente (-1) |
| db.nomeCollectiom.explain().find() | Permette di creare un json che contiene info sulla query appena lanciata. Serve per testare le performance delle query stesse. |

Un altro utilizzo degli indici riguarda il sorting. MongoDB per eseguire questa operazione carica in memoria tutti i documenti che ha trovato con il comando find. Il problema è che MongoDB ha un limite di 32 megabytes. Su db piccoli questo non rappresenta un problema ma su DB grandi con milioni di documenti si. In questo gli indici vengono in nostro aiuto. Le liste che andiamo a creare con gli indici sono ordinate e contengono pochi dati. MongoDB può quindi usarle per ordinare il risultato delle nostre find. Questo a patto che il campo su cui andiamo a fare il sort sia presente nell’indic.

All’interno di una collection è già presente un indice di default, che rappresenta l’ID dei singoli campi. Viene creato alla creazione della collection con vincolo di unicità.

Con il comando createIndex oltre ai campi interessati possiamo passare un ulteriore json di opzioni. Ad esempio per creare un vincolo di unicità la sintassi sarà la seguente:

*db.nomeCollection.createIndex(*

*{nomeCampo: 1/-1 },*

*{unique: true}*

*)*

Come abbiamo visto nell’esempio gli indici sono retroattivi, le loro condizioni devono essere rispettate già al momento della loro creazione.

Può capitare di avere contesti in cui non serve sempre verificare tutto il contenuto di una collection e del suo indice. In questo caso si utilizza un’opzione del comando createIndex, chiamato **partialFilterExpression**.

Esistono anche contesti in cui non è necessario mantenere i dati in un DB per tempi prolungati. File di log, dati di sessione, sono esempi di dati che una volta utilizzati possono essere eliminati. MongoDB permette di rendere questa operazione automatica tramite l’opzione **expireAfterSeconds (va usato solo su campi data):**

db.nomeCollection.createIndex(

{createdAt: 1},

{expireAfterSeconds: 1})

Una volta trascorso il numero di secondi indicati MongoDB automaticamente elimina il documento.

È possibile creare indici su campi di tipo array. Questi indici vengono definiti multichiave. Ad ogni elemento dell’array corrisponderà un singolo elemento dell’indice. Questo implica che gli indici multi chiave sono potenzialmente molto più grandi rispetto a quelli standard.

È anche possibile creare indice di testo, che vengono utilizzati per stoccare le parole contenute in uno o più campi. L’indice di testo (text indexes) è di fatto un indice multichiave ma con delle sue regole. Il testo (in base alla lingua) viene spezzettato in singole parole che vengono salvate nel relativo oggetto.

|  |  |
| --- | --- |
| db.nomeCollection.createIndex(  {nomeCampo: “text” }) | Qui al posto del 1/-1 si passa una keyword ‘text’. |

Una collection può avere un solo indice di testo. Questo non significa che possiamo inserire un solo campo, possiamo infatti passare più campi all’indice. Questo ci permette di effettuare ricerche testuali. Possiamo andare più a fondo e verificare ilpunteggio di ricerca. Ogni ricerca avrà infatti un valore numerico che rappresenta quanto il campo sia vicino alle parole cercate. Con la query qui indicata potete verificare il “risultato” di questo punteggio e ordinare secondo questo valore.

Come detto, inizialmente può esistere un unico indice di testo per ogni collection. Questo può avere più campi, e ogni campo può avere un “peso” nella ricerca. Per esempio potremmo volere che il nome di un prodotto valga più della sua .descrizione.

Per assegnare un “peso” (weight) ad ogni campo la sintassi è la seguente:

*db.nomeCollection.createIndex(*

*{campo1: “text” ,*

*campo2: “text”,*

*...},*

*{weights : {campo1: 1, campo2: 10}}← opzioni*

*)*

In questo modo il campo2 avrà una valenza maggiore in caso di corrispondenza. Di default la creazione di indici può essere un’operazione molto lunga in caso di DB di grandi dimensioni.

Questo pone la collection in uno stato di “lock”, che ne impedisce l’utilizzo fino ad operazione conclusa.

Atomicità

Quando inseriamo un documento (InsertOne()) questo elemento verrà salvato nella sua interezza oppure (in caso di errore) questo non verrà salvato neppure in parte. Questo ci indica che l’operazione di insertOne ha una sua atomicità.

“Un'operazione atomica, in informatica, consiste in un'operazione di esecuzione indivisibile dal punto di vista logico.”

O l’operazione ha successo o non ha successo in questo caso. Poniamo l’esempio di 5 documenti da inserire. Se arrivati al numero 3 questo va in errore cosa accade? In questo esempio i documenti inseriti sono soltanto 2 (in caso di ordered: true).Il numero 3 viene escluso in quanto ha generato errore. Questo ci fa notare che l’atomicità del comando è contestualizzata al documento e non all’intera operazione.

Per garantire l’atomicità su più operazioni si ricorre alle **transaction**. Questa operazione ci garantisce che tutti i comandi che lanciamo vengano eseguiti nel loro insieme o che falliscano nel loro insieme. Una transazione atomica è costituita dai seguenti passaggi:

1. Start transiction
2. Operazioni di eseguire
3. Direttive di convalida o disfacimento dell’intera transazione
4. End transaction

Durante questa operazione si apre una connessione del nostro database. Il codice all’interno può effettuare tutte le operazioni sui dati ma il database verrà effettivamente aggiornato al termine dell’intera transaction.

Aggregate

Quando si ha a che fare con una gran mole di dati una delle funzionalità più importanti è l’aggregazione, che permette di avere una visione globale dei dati e di ricavarne statistiche.

Questo comando (**aggregate()**) accetta come parametro una pipeline di operatori, cioè un elenco ordinato di operatori di aggregazione. L’effetto è che la collezione dei documenti passa tra i vari stadi della pipeline. Per avere le migliori performance è quindi importante mettere all’inizio della pipeline le operazioni di filtraggio in modo da far passare meno elementi possibili attraverso il resto della pipeline.

|  |  |
| --- | --- |
| db.person.aggregate( [  { **$match**: { gender: "female" } }  ]) | ci permette di effettuare un filtro sulla base dati, esattamente come nel find() |
| db.person.aggregate( [  { **$group**: { \_id: { gender: “$gender”},  totalPerson: {$sum: 1}  } }  ]) | separa i documenti in gruppi usando una “group key” come discriminante. L’output è un singolo documento per ciascuna chiave unica. |
| db.person.aggregate( [  { **$project**: { \_id: 0,  gender: 1,  code:”TD”,  fullname: {$concat: [“$name.first”, “ ”, “$name.last”]}  } }  ]) | passa i documenti allo stage successivo della pipeline con i campi  richiesti. Questi possono essere campi presi dal documento, campi nuovi o computati. |
| db.friends.aggregate( [  { $group: { \_id: “$age”, allHobbies: **{$push**: “$hobbies”}  } }  ]) | permette di aggiungere un valore ad un array.  Questo, nell’aggregate, può essere usato per unire il contenuto array nestati. |
| db.friends.aggregate( [  { **$unwind:** “$hobbies”}  ]) | permette di “esplodere” il contenuto di un array. |
| db.friends.aggregate( [  { $unwind: “$hobbies”},  { $group: { \_id: “$age”,  allHobbies: {**$addToSet**: “$hobbies”}  } }  ]) | permette di aggiungere un valore ad un array a meno che questo  non sia già presente. |
| db.person.aggregate( [  { **$bucket**: {  groupBy: "$dob.age",  boundaries:[0,18,30,50,80],  output:{  averageAge: {$avg: "$dob.age"},  names:{$push:"$name.first"}  }}]) | permette di dividere il contenuto in gruppi, impostando “confini” e  l’output desiderato. |
| db.person.aggregate( [  { $bucketAuto: {  groupBy: "$dob.age",  buckets:5,  output:{  averageAge: {$avg: "$dob.age"},  names:{$push:"$name.first"}  }}]) | lascia a Mongo la creazione dei limiti.  Mongo cercherà di creare la distribuzione più uniforme. |