

Il Piccolo Libro di MongoDB

di Karl Seguin



Note sul Libro

Licenza

I contenuti del Piccolo Libro di MongoDB sono protetti da licenza Attribuzione - Non Commerciale 3.0. **Non dovresti aver pagato per questo libro**

Sei libero di copiare, distribuire, modificare o mostrare il libro. Tuttavia ti chiedo di attribuire sempre l'opera all'autore, Karl Seguin, e di non usarla per scopi commerciali.

Puoi consultare il testo integrale della licenza a questo indirizzo:

<http://creativecommons.org/licenses/by-nc/3.0/legalcode>

Informazioni sull'Autore

Karl Seguin è uno sviluppatore competente in diversi campi e tecnologie. E' esperto programmatore .NET e Ruby. Collabora saltuariamente a progetti OSS, è scrittore tecnico e, occasionalmente, speaker. Per quanto riguarda MongoDB è stato tra i principali autori della libreria C# per MongoDB, NoRM. Ha scritto il tutorial interattivo [moongly](#) nonché [Mongo Web Admin](#). Il suo servizio per sviluppatori di casual games, [mogade](#) gira con MongoDB.

Il suo blog è <http://openmymind.net>, e twitta come [@karlseguin](#)

Ringraziamenti

Un grazie speciale a [Perry Neal](#) per avermi prestato occhi, mente e passione. Mi hai dato un aiuto prezioso. Grazie.

Ultima Versione

L'ultima versione del sorgente di questo libro è disponibile qui:

<http://github.com/nicolaiarocci/the-little-mongodb-book>.

Le ultime versioni PDF, ePub e mobi sono invece reperibili qui:

<http://nicolaiarocci.com/il-piccolo-libro-di-mongodb-edizione-italiana/>.

Traduzione italiana

Traduzione a cura di [Nicola Iarocci](#) (<http://nicolaiarocci.com>). Collaboratori: Andrea Rabbaglietti, Michele Zonca.

Vi prego di segnalare ogni errore o imprecisione, così da migliorare nel tempo la qualità del testo.

Introduzione

Non è colpa mia se i capitoli sono brevi, MongoDB è davvero così facile da imparare.

Si dice spesso che la tecnologia avanza a velocità impressionante. E' vero che la lista di nuove tecnologie e tecniche da imparare è in continua crescita. Tuttavia sono convinto da tempo che le tecnologie fondamentali usate dai programmatori evolvono a un ritmo piuttosto lento. Una persona potrebbe passare anni senza imparare granché e tuttavia rimanere competente. Impressiona, piuttosto, la velocità con cui le tecnologie consolidate vengono rimpiazzate. Apparentemente da un giorno all'altro, tecnologie affermate sono messe in discussione da un repentino cambiamento di attenzione da parte dei programmatori.

Il fenomeno è evidente nell'affermazione delle tecnologie NoSQL a scapito dei ben consolidati database relazionali. Fino a ieri il web era guidato da pochi RDBMS, ed ecco che oggi quattro o cinque soluzioni NoSQL si sono già affermate come attendibili alternative.

Anche se sembra che queste transizioni avvengano nel corso di una notte, la realtà è che possono passare anni prima che una nuova tecnologia divenga pratica comune. L'entusiasmo iniziale è guidato da un gruppo relativamente piccolo di sviluppatori e aziende. I prodotti migliorano con l'esperienza e, quando ci si rende conto che una tecnologia è destinata a rimanere, altri cominciano a sperimentarla. Ciò è particolarmente vero nel caso NoSQL poiché spesso queste soluzioni non vengono progettate come alternative a modelli di storage più tradizionali, ma intendono piuttosto far fronte a nuove necessità.

Detto questo, prima di tutto dobbiamo capirci su cosa si intenda per NoSQL. E' un termine vago, che ha significati diversi a seconda di chi lo usa. Personalmente lo intendo in senso molto ampio, per far riferimento a un sistema che svolge un ruolo nel salvataggio dei dati. In altre parole per me NoSQL è la convinzione che lo strato di persistenza non è necessariamente responsabilità di un solo sistema. Laddove storicamente i fornitori di database relazionali hanno sempre tentato di posizionare i loro software come soluzione universale per qualunque problema, NoSQL tende a individuare piccole unità di responsabilità per ognuna delle quali scegliere lo strumento ideale. Quindi uno stack NoSQL potrebbe contemplare un database relazionale, MySQL per esempio, Redis per ricerche veloci e Hadoop per le elaborazioni dati intensive. In parole povere NoSQL è essere aperti e coscienti dell'esistenza di modelli e strumenti alternativi per la gestione dei dati.

Vi potreste domandare qual'è il ruolo ricoperto da MongoDB in tutto questo. In quanto database orientato ai documenti Mongo è una soluzione NoSQL piuttosto generalizzata, e in effetti andrebbe visto come una alternativa ai database relazionali. Come i database relazionali anche Mongo potrebbe trarre beneficio dall'abbinamento a soluzioni NoSQL più specializzate. MongoDB ha vantaggi e svantaggi che vedremo nei prossimi capitoli di questo libro.

Avrete notato che in questo libro useremo indifferentemente i termini MongoDB e Mongo.

Cominciare

Gran parte di questo libro è dedicata alle funzionalità di base di MongoDB. Per questo motivo ci affideremo alla shell di MongoDB. La shell è preziosa sia per imparare che come strumento di amministrazione, tuttavia il vostro codice applicativo farà uso senz'altro di uno dei driver MongoDB.

Questo ci porta alla prima cosa da conoscere di MongoDB: i driver. Mongo è dotato di un [buon numero di driver ufficiali](#) per i principali linguaggi di programmazione. Possiamo pensare ai driver allo stesso modo di quelli per database relazionali che probabilmente abbiamo usato in passato. La community di sviluppatori ha poi costruito, sulla base di questi driver, una serie di framework e librerie dedicate ai vari linguaggi. Per esempio [NoRM](#) è una libreria C# che implementa LINQ, mentre [MongoMapper](#) è una libreria Ruby compatibile con ActiveRecord. La scelta di programmare coi driver di base piuttosto che con le librerie di più alto livello è libera. Ne faccio cenno perché molte persone che si avvicinano a MongoDB rimangono confuse dall'esistenza di driver ufficiali e di librerie della community - in linea generale i primi si occupano di comunicazione e connettività di base con MongoDB, mentre le seconde implementano caratteristiche specifiche dei linguaggi/framework.

Nel corso della lettura del libro vi invito a giocare con MongoDB, sia mettendo in pratica quel che propongo che sperimentando in proprio, rispondendo alle domande che senz'altro sorgeranno spontanee. E' facile cominciare a lavorare con MongoDB, quindi diamoci subito da fare cominciando dalla configurazione di quel che ci serve.

1. Andate alla [pagina di download ufficiale](#) e scaricate i file binari per il vostro sistema operativo (scegliete la versione stabile raccomandata). Ai fini dello sviluppo potete prelevare indifferentemente la versione a 32-bit o 64-bit.
2. Scompattate l'archivio (non importa la posizione) quindi andate alla cartella bin. Non eseguite nulla, ma sappiate che mongod è il processo server mentre mongo è la shell (il client) - questi sono i due eseguibili coi quali passeremo gran parte del nostro tempo.
3. Create un nuovo file di testo nella cartella bin e chiamatelo `mongodb.config`.
4. Aggiungete questa riga al vostro `mongodb.config`: `dbpath=PERCORSO_DOVE_SALVARE_IL_DATABASE`. Ad esempio, su Windows potreste scegliere `dbpath=c:\mongodb\data` mentre su Linux una scelta valida potrebbe essere `dbpath=/etc/mongodb/data`.
5. Assicuratevi che il `dbpath` scelto esista.
6. Lanciate `mongod` con l'opzione `--config /path/al/vostro/mongodb.config`.

Ad esempio un utente Windows potrebbe estrarre il file scaricato in `c:\mongodb\` e creare la cartella `c:\mongodb\data\`. In questo caso all'interno di `c:\mongodb\bin\mongodb.config` dovrà specificare `dbpath=c:\mongodb\data\`. A questo punto può lanciare `mongod` dalla linea di comando con `c:\mongodb\bin\mongod --config c:\mongodb\bin\mongodb.config`.

Naturalmente potete aggiungere la cartella bin al vostro PATH per rendere tutto più semplice. Tutto ciò è valido anche per gli utenti MacOSX e Linux, che probabilmente dovranno adattare i percorsi.

A questo punto dovreste trovarvi con MongoDB pronto e operativo. Se invece ottenete un messaggio di errore, leggete con attenzione l'output - il server è piuttosto bravo a spiegare cos'è andato storto.

Potete lanciare `mongo` (senza la *d*), che conatterà la shell al vostro server in esecuzione. Provate a digitare `db.version` (`)` per assicurarvi che tutto stia funzionando a dovere. Dovreste vedere il numero della versione installata.

Capitolo 1 - Le Basi

Cominciamo il nostro viaggio dai meccanismi base di MongoDB. Ovviamente sono fondamentali per capire MongoDB, ma a un livello più generale ci aiuteranno a rispondere alle nostre domande sul ruolo di MongoDB.

Per cominciare, ci sono sei semplici concetti che dobbiamo comprendere.

1. MongoDB implementa lo stesso concetto di 'database' al quale probabilmente siamo abituati (o schema, se venite dal mondo Oracle). All'interno di una istanza MongoDB potete avere zero o più database, ognuno dei quali agisce come un contenitore di alto livello per tutto il resto.
2. Un database può avere zero o più 'collezioni'. Una collezione ha molto in comune con le 'tabelle' tradizionali, tanto che potete considerarle la stessa cosa.
3. Le collezioni sono composte da zero o più 'documenti'. Di nuovo, potete pensare a un documento come a una 'riga' (record) di una tabella.
4. Un documento è a sua volta composto da uno o più 'campi', che come potete immaginare assomigliano alle 'colonne'.
5. Gli 'indici' in MongoDB funzionano in modo molto simile alle loro controparti RDBMS.
6. I 'cursori', a cui spesso viene data poca importanza, sono qualcosa di diverso dagli altri cinque concetti, e li ritengo abbastanza importanti da meritare attenzione. E' importante sapere che quando si chiedono dati a MongoDB questi restituisce un cursore col quale possiamo, per esempio, contare i documenti o spostarci avanti, senza che alcun dato venga effettivamente letto.

Riassumendo, MongoDB è fatto di database che contengono collezioni. Una collezione è una raccolta di documenti. Ogni documento è composto da campi. Le collezioni possono essere indicizzate, il che migliora le prestazioni di ricerche e ordinamenti. Infine, quando chiediamo dati a MongoDB otteniamo un cursore, la cui esecuzione è rinviata finché non si renderà necessaria.

Vi potreste domandare per quale ragione adottiamo una nuova terminologia (collezione invece di tabella, documento al posto di riga e campo piuttosto che colonna). Vogliamo solo complicare le cose? La verità è che questi nuovi concetti non sono identici alle loro controparti presenti nei database relazionali. La differenza più importante è che i database relazionali definiscono le colonne a livello tabella, mentre i database orientati ai documenti definiscono i campi a livello di documento. Ciò significa che ogni documento di una collezione può avere il suo set esclusivo di campi. Ne consegue che una collezione è un contenitore più semplice di una tabella, laddove un documento ha molte più informazioni di una riga.

Si tratta di un concetto importante da comprendere, ma non c'è da preoccuparsi se al momento non tutto è chiaro. Basteranno un paio di inserimenti per capire il vero significato di tutto questo. In definitiva una collezione non vincola il suo contenuto (è senza schema, o schema-less). I campi vengono tracciati per ogni singolo documento. Esploreremo vantaggi e svantaggi di tutto questo in uno dei prossimi capitoli.

Cominciamo a darci da fare. Se ancora non l'avete fatto eseguite pure il server mongod e la mongo shell. La shell esegue codice JavaScript. Ci sono alcuni comandi globali che potete lanciare, come `help` o `exit`. I comandi lanciati sul database attivo si eseguono nei confronti dell'oggetto `db`, come ad esempio `db.help()` o `db.stats()`. I comandi lanciati nei confronti una collezione specifica, cosa che si fa spesso, vanno eseguiti sull'oggetto `db.NOME_COLLEZIONE`, come per esempio `db.unicorns.help()` oppure `db.unicorns.count()`.

Provate a digitare `db.help()`. Otterrete una lista dei comandi che è possibile eseguire nei confronti dell'oggetto `db`.

Piccola nota a margine. Poiché questa è una shell JavaScript, se eseguite un metodo e omettete le parentesi `()` vedrete il contenuto del metodo piuttosto che ottenerne l'esecuzione. Ve lo ricordo affinché non rimaniate sorpresi la prima volta che vi capiterà di vedere una risposta che comincia con `function (...){`. Per esempio, se digitate `db.help` (senza le parentesi) quello che otterrete è la visualizzazione dell'implementazione interna del metodo `help`.

Prima di tutto useremo il metodo globale `use` per cambiare il database attivo. Digitate `use learn`. Non importa che il database non esista ancora. Quando creeremo la prima collezione, allora verrà creato anche il database `learn`. Ora che abbiamo un database attivo possiamo eseguire comandi sul database stesso, come per esempio `db.getCollectionNames()`. Se lo fate ora dovreste ottenere un array vuoto `[]`. Poiché le collezioni sono schema-less non c'è necessità di crearle esplicitamente. Possiamo semplicemente inserire un documento nella nuova collezione. Per farlo usiamo il comando `insert`, passandogli direttamente il documento da inserire:

```
db.unicorns.insert({name: 'Aurora', gender: 'f', weight: 450})
```

La riga esegue il comando `insert` nei confronti della collezione `unicorns`, passando un singolo argomento. Per la serializzazione MongoDB usa internamente il formato JSON binario. Esternamente ciò significa che useremo parecchio JSON, come nel caso dei nostri parametri. Se ora eseguiamo `db.getCollectionNames()` otteniamo due collezioni: `unicorns` e `system.indexes`. `system.indexes` viene creata una volta per database, e contiene informazioni sugli indici del database.

Ora possiamo usare il comando `find` sulla collezione `unicorns` per ottenere una lista di documenti:

```
db.unicorns.find()
```

Notate che in aggiunta ai dati che avete indicato c'è un campo `_id`. Ogni documento deve avere un campo `_id` univoco. Potete generarlo da voi oppure lasciare che sia MongoDB a generare un `ObjectId` per voi. Probabilmente la maggior parte delle volte sarà sufficiente lasciarlo generare a MongoDB. Per impostazione predefinita il campo `_id` è indicizzato - il che spiega l'esistenza della collezione `system.indexes`. E' possibile consultare l'elenco degli indici:

```
db.system.indexes.find()
```

Ciò che otteniamo è il nome dell'indice, il database e la collezione ai quali appartiene e l'elenco dei campi inclusi nell'indice.

Torniamo alla nostra discussione sulle collezioni schema-less. Inseriamo un documento completamente diverso nella collezione `unicorns`:

```
db.unicorns.insert({name: 'Leto', gender: 'm', home: 'Arrakeen', worm: false})
```

Usiamo di nuovo `find` per vedere la lista dei documenti. Quando conosceremo qualcosa in più discuteremo questo interessante comportamento di MongoDB, ma giunti a questo punto dovreste cominciare a comprendere perché la terminologia tradizionale non è la più adeguata.

Padroneggiare i Selettori

Oltre ai sei concetti già visti c'è un aspetto pratico di MongoDB che è necessario comprendere a fondo prima di procedere con argomenti più avanzati: i selettori di query (query selectors). Un selettore di query in MongoDB assomiglia alla clausola where di un comando SQL. In quanto tale viene usato per trovare, contare, aggiornare e rimuovere documenti dalle collezioni. Un selettore è un oggetto JSON la cui forma più semplice è {}, che rintraccia tutti i documenti (**null** è altrettanto valido). Se volessimo trovare tutti gli unicorni femmina potremmo usare {gender: 'f'}.

Prima di addentrarci a fondo nei selettori prepariamo un po' di dati con cui giocare. Prima di tutto cancelliamo ciò che abbiamo inserito finora nella collezione unicorns: db.unicorns.remove() (poiché forniamo un selettore, rimuoveremo tutti i documenti). Ora digitiamo i comandi di inserimento che seguono, così da ottenere un po' di dati con cui lavorare (vi suggerisco di copiarli e incollarli da qui):

```
db.unicorns.insert({name: 'Horny', dob: new Date(1992,2,13,7,47), loves: ['carrot', 'papaya'], weight: 600, gender: 'm', vampires: 63});
db.unicorns.insert({name: 'Aurora', dob: new Date(1991, 0, 24, 13, 0), loves: ['carrot', 'grape'], weight: 450, gender: 'f', vampires: 43});
db.unicorns.insert({name: 'Unicrom', dob: new Date(1973, 1, 9, 22, 10), loves: ['energon', 'redbull'], weight: 984, gender: 'm', vampires: 182});
db.unicorns.insert({name: 'Roooooodles', dob: new Date(1979, 7, 18, 18, 44), loves: ['apple'], weight: 575, gender: 'm', vampires: 99});
db.unicorns.insert({name: 'Solnara', dob: new Date(1985, 6, 4, 2, 1), loves: ['apple', 'carrot', 'chocolate'], weight: 550, gender: 'f', vampires: 80});
db.unicorns.insert({name: 'Ayna', dob: new Date(1998, 2, 7, 8, 30), loves: ['strawberry', 'lemon'], weight: 733, gender: 'f', vampires: 40});
db.unicorns.insert({name: 'Kenny', dob: new Date(1997, 6, 1, 10, 42), loves: ['grape', 'lemon'], weight: 690, gender: 'm', vampires: 39});
db.unicorns.insert({name: 'Raleigh', dob: new Date(2005, 4, 3, 0, 57), loves: ['apple', 'sugar'], weight: 421, gender: 'm', vampires: 2});
db.unicorns.insert({name: 'Leia', dob: new Date(2001, 9, 8, 14, 53), loves: ['apple', 'watermelon'], weight: 601, gender: 'f', vampires: 33});
db.unicorns.insert({name: 'Pilot', dob: new Date(1997, 2, 1, 5, 3), loves: ['apple', 'watermelon'], weight: 650, gender: 'm', vampires: 54});
db.unicorns.insert({name: 'Nimue', dob: new Date(1999, 11, 20, 16, 15), loves: ['grape', 'carrot'], weight: 540, gender: 'f'});
db.unicorns.insert({name: 'Dunx', dob: new Date(1976, 6, 18, 18, 18), loves: ['grape', 'watermelon'], weight: 704, gender: 'm', vampires: 165});
```

Ora che abbiamo i dati possiamo fare pratica coi selettori. Usiamo {campo: valore} per trovare documenti il cui campo sia uguale a valore. Usiamo {campo1: valore1, campo2: valore2} per indicare l'operatore and. Usiamo gli operatori \$lt, \$lte, \$gt, \$gte e \$ne rispettivamente per minore di (less than), minore o uguale (less than or equal), maggiore di (greater than), maggiore o uguale (greater than or equal) e diverso da (not equal). Per esempio, per ottenere tutti gli unicorni maschi che pesano più di 700 libbre possiamo usare:

```
db.unicorns.find({gender: 'm', weight: {$gt: 700}})
```



```
//oppure (non è la scelta migliore, ma vale come esempio)
db.unicorns.find({gender: {$ne: 'f'}, weight: {$gte: 701}})
```

L'operatore `$exists` va usato per verificare la presenza o l'assenza di un campo, per esempio:

```
db.unicorns.find({vampires: {$exists: false}})
```

Dovrebbe restituire un singolo documento. Se vogliamo un OR invece di un AND usiamo l'operatore `$or` assegnandoli un array di valori sui quali vogliamo compiere l'OR:

```
db.unicorns.find({gender: 'f', $or: [{loves: 'apple'}, {loves: 'orange'}, {weight: {$lt : 500}}]})
```

Questa istruzione restituisce tutti gli unicorni femmina che amano le mele (apple), le arance (orange) oppure che pesano (weight) meno di 500 libbre.

Nell'ultimo esempio succede qualcosa di interessante. Forse avrete notato che il campo `loves` è un array. MongoDB supporta gli array come oggetti di prima classe. Questa è una caratteristica incredibilmente utile. Una volta cominciato ad usarla ti domanderai come hai potuto vivere senza finora. Ciò che è ancor più interessante è quanto sia facile fare selezioni basate su un valore array: `{loves: 'watermelon'}` restituisce qualunque documento che abbia campi `'loves'` valorizzati a `'watermelon'`.

Sono disponibili più operatori di quelli che abbiamo visto finora. Il più flessibile è `$where`, il quale ci permette di passare codice JavaScript da eseguire sul server. Questi operatori sono discussi nella sezione [Advanced Queries](#) del sito di MongoDB. Quel che abbiamo visto fin qui è sufficiente per cominciare con MongoDB, ed è anche ciò che userete per la maggior parte del tempo.

Abbiamo visto come i selettori possano essere usati in abbinamento al comando `find`. Possono essere adoperati anche con `remove`, già incontrato brevemente, `count`, che ancora non abbiamo visto ma il cui significato potete intuire da soli, e col comando `update` a cui ci dedicheremo in seguito.

Il `ObjectId` che MongoDB ha generato per il nostro campo `_id` può a sua volta essere selezionato:

```
db.unicorns.find({_id: ObjectId("L'ObjectId")})
```

Riepilogo

Non abbiamo ancora conosciuto il comando `update` né abbiamo visto le cose più interessanti che possiamo ottenere con `find`. Tuttavia abbiamo fatto partire MongoDB, abbiamo dato una occhiata ai comandi `insert` e `remove` (su questi non c'è molto altro da aggiungere). Abbiamo introdotto `find` e scoperto che cosa sono i 'selettori' in MongoDB. Siamo partiti col piede giusto, impostando le basi per quel che deve ancora venire. Che ci crediate o no, a questo punto conoscete la maggior parte di quel che serve sapere per lavorare con MongoDB - è progettato davvero per essere facile e veloce da imparare e usare. Vi invito caldamente a giocare con la vostra copia locale prima di proseguire. Inserite documenti diversi, possibilmente in nuove collezioni, e prendete confidenza con i diversi selettori. Usate `find`, `count` e `remove`. Dopo pochi tentativi ciò che ora può sembrare poco chiaro finirà probabilmente per avere senso.

Capitolo 2 - Gli Aggiornamenti

Nel primo capitolo abbiamo introdotto tre delle quattro operazioni CRUD (create, read, update, delete). Questo capitolo è dedicato all'operazione di cui non abbiamo ancora parlato: update. Quest'ultima riserva qualche sorpresa e, per questo motivo, le dedichiamo un intero capitolo.

Update: Replace vs \$set

Nella sua forma più semplice update richiede due argomenti: il selettore da usare (where) e il valore del campo da aggiornare. Se Rooooooodles avesse guadagnato qualche chilo, potremmo fare:

```
db.unicorns.update({name: 'Roooooodles'}, {weight: 590})
```

(se nel frattempo avete cambiato la collezione unicorns e i dati originali sono compromessi, procedete con un remove di tutti i documenti quindi re-inserite i dati col codice visto nel capitolo 1).

Se questo fosse codice reale aggiorneremmo i dati in base al campo `_id`, ma poiché non possiamo sapere a priori quale `_id` verrà generato da MongoDB, continueremo a usare name. Ora, se andiamo a cercare il record che abbiamo aggiornato:

```
db.unicorns.find({name: 'Roooooodles'})
```

Abbiamo la prima sorpresa che update ci riserva. Non viene trovato alcun documento perché il secondo parametro che forniamo viene usato per **sostituire** l'originale. In altre parole la nostra update ha cercato il documento per name, quindi ha sostituito l'intero documento con il nuovo documento (il secondo parametro). Questo è molto diverso dal funzionamento del comando update nel mondo SQL. In alcune situazioni questo comportamento è molto comodo, e può essere sfruttato per ottenere aggiornamenti davvero dinamici. Tuttavia, quando tutto quel che vogliamo è cambiare il valore di uno o più campi è meglio usare il modificatore \$set di MongoDB:

```
db.unicorns.update({weight: 590}, {$set: {name: 'Roooooodles', dob: new Date(1979, 7, 18, 18, 44), loves: ['apple'], gender: 'm', vampires: 99}})
```

Questo ripristinerà i campi perduti. Non sovrascriverà il nuovo weight dato che non l'abbiamo indicato nel secondo argomento. Ora se eseguiamo:

```
db.unicorns.find({name: 'Roooooodles'})
```

Otteniamo il risultato che volevamo. Quindi il modo corretto di aggiornare il peso sarebbe stato:

```
db.unicorns.update({name: 'Roooooodles'}, {$set: {weight: 590}})
```

Modificatori di Aggiornamento

Oltre a \$set possiamo azionare altri modificatori che ci consentono di fare cose eleganti. Tutti questi modificatori di aggiornamento agiscono sui campi - non azzerano l'intero documento. Per esempio il modificatore \$inc consente di

aumentare o diminuire il valore di un campo. Supponiamo che a Pilot siano state assegnate un paio di uccisioni di vampiri di troppo; potremmo correggere l'errore eseguendo:

```
db.unicorns.update({name: 'Pilot'}, {$inc: {vampires: -2}})
```

Se Aurora sviluppasse improvvisamente una passione per i dolci, potremmo aggiungerli al suo array loves con il modificatore \$push:

```
db.unicorns.update({name: 'Aurora'}, {$push: {loves: 'sugar'}})
```

La sezione [Updating](#) del sito di MongoDB ha informazioni sugli altri modificatori di aggiornamento disponibili.

Upserts

Una delle sorprese più piacevoli che update ci riserva è senz'altro il supporto per gli upsert. Se upsert trova il documento cercato lo aggiorna, altrimenti lo crea. Gli upsert sono utili in diverse situazioni, ve ne renderete conto non appena vi ci imbatterete. Per attivare gli upsert impostiamo un terzo parametro a `true`.

Un esempio banale è quello di contatore di visite ad un sito web. Se volessimo gestire un contatore in tempo reale dovremmo verificare l'esistenza del record per la pagina attuale, quindi decidere per l'inserimento o l'aggiornamento. Poiché omettiamo il terzo parametro (oppure se lo impostiamo a false) l'esecuzione del comando seguente non ottiene risultati:

```
db.hits.update({page: 'unicorns'}, {$inc: {hits: 1}});  
db.hits.find();
```

Tuttavia attivando gli upsert il risultato cambia:

```
db.hits.update({page: 'unicorns'}, {$inc: {hits: 1}}, true);  
db.hits.find();
```

Poiché non esistono documenti col campo page equivalente a unicorns, viene inserito un nuovo documento. Se eseguiamo lo stesso comando una seconda volta, il documento esistente viene aggiornato, e il suo campo hits aumentato a due.

```
db.hits.update({page: 'unicorns'}, {$inc: {hits: 1}}, true);  
db.hits.find();
```

Aggiornamenti Multipli

L'ultima sorpresa che update ci riserva è il fatto che, per default, aggiorna un solo documento. Stando agli esempi visti finora questo comportamento sembrerebbe logico. Tuttavia se eseguite qualcosa di questo genere:

```
db.unicorns.update({}, {$set: {vaccinated: true }});  
db.unicorns.find({vaccinated: true});
```

Probabilmente vi aspettereste di trovare tutti i vostri preziosi unicorni vaccinati. Per ottenere il comportamento desiderato è necessario mettere a true un quarto parametro:

```
db.unicorns.update({}, {$set: {vaccinated: true }}, false, true);  
db.unicorns.find({vaccinated: true});
```

Riepilogo

Questo capitolo conclude la nostra introduzione alle operazioni CRUD che è possibile eseguire su una collezione. Abbiamo visto in dettaglio il comando update scoprendo tre comportamenti interessanti. Primo, a differenza di una update SQL, la update in MongoDB sostituisce un documento. Per questo motivo il modificatore \$set risulta piuttosto utile. Secondo, update supporta gli upsert (aggiornamento oppure inserimento) in modo piuttosto intuitivo, ciò che lo rende particolarmente utile quando viene abbinato al modificatore \$inc. Infine, per default update aggiorna solo il primo documento trovato.

Tenete sempre presente che stiamo usando MongoDB dal punto di vista della sua shell. Il driver e la libreria adottata potrebbero alterare questi comportamenti predefiniti, o esporre una API differente. Il driver Ruby, per esempio, unisce gli ultimi due parametri in una singola hash: `{:upsert => false, :multi => false}`.

Capitolo 3 - Padroneggiare il metodo Find

Nel capitolo 1 abbiamo dato una veloce occhiata al comando `find`. Su `find` c'è altro da sapere; la sola comprensione dei selettori non è sufficiente. Abbiamo già detto che `find` restituisce un cursore. E' giunta l'ora di andare a fondo e capire cosa ciò significa esattamente.

Selettori di Campo

Prima di passare ai cursori è necessario sapere che `find` accetta un secondo parametro opzionale. Si tratta dell'elenco dei campi che vogliamo recuperare. Per esempio possiamo chiedere i nomi di tutti gli unicorni con questo comando:

```
db.unicorns.find(null, {name: 1});
```

Per default il campo `_id` viene restituito sempre. Possiamo escluderlo in modo esplicito con `{name: 1, _id: 0}`.

Ad eccezione del campo `_id`, non è possibile mescolare inclusioni ed esclusioni. A ben vedere ciò ha senso, di solito vogliamo escludere oppure includere uno o più campi esplicitamente.

Ordinamenti

Abbiamo ripetuto più volte che `find` restituisce un cursore la cui esecuzione è ritardata finché questa non si rende veramente necessaria. Tuttavia avrete senz'altro notato che nella shell `find` viene eseguito immediatamente. Questo è un comportamento peculiare della shell. Possiamo osservare il vero comportamento dei cursori quando usiamo uno dei metodi che è possibile concatenare a `find`. Il primo che prendiamo in esame è `sort`. `sort` funziona in maniera simile al selettore di campo che abbiamo visto nella sezione precedente. Elenchiamo i campi da ordinare, usando 1 per ottenere un ordinamento crescente e -1 per un ordinamento decrescente. Per esempio:

```
//gli unicorni più pesanti per primi:  
db.unicorns.find().sort({weight: -1})  
  
//per nome, quindi per numero di vampiri uccisi:  
db.unicorns.find().sort({name: 1, vampires: -1})
```

Come succede nei database relazionali, anche MongoDB è in grado di ricorrere a un indice per eseguire un ordinamento. Approfondiremo gli indici più avanti, tuttavia è utile sapere che in assenza di un indice MongoDB impone un limite alla dimensione dell'ordinamento. Ciò significa che il tentativo di ordinare un set dati molto grande e sprovvisto di indice genererà un errore. Alcuni ritengono che questa sia una limitazione. In realtà vorrei davvero che più database fossero in grado di rifiutare le query non ottimizzate (non ho intenzione di trasformare ogni svantaggio di MongoDB in un vantaggio, ma ho visto fin troppi database scarsamente ottimizzati per non sapere che un controllo più stretto sarebbe quanto mai necessario).

Paginazione

La paginazione dei risultati può essere ottenuta con i metodi cursore `limit` e `skip`. Per ottenere solo il secondo e il terzo unicorno più pesante potremmo digitare:

```
db.unicorns.find().sort({weight: -1}).limit(2).skip(1)
```

Usare `limit` in combinazione con `sort` è un buon sistema per non incappare in problemi quando si fanno ordinamenti su campi non indicizzati.

Conteggi

La shell consente l'esecuzione di `count` direttamente sulla collezione:

```
db.unicorns.count({vampires: {$gt: 50}})
```

In realtà `count` è a sua volta un metodo cursore, la shell in questo caso implementa una scorciatoia. Per i driver che non implementano questa scorciatoia dovremo usare la sintassi completa (che funziona anche nella shell):

```
db.unicorns.find({vampires: {$gt: 50}}).count()
```

Riepilogo

Usare `find` e i cursori è piuttosto semplice. Ci sono alcuni comandi aggiuntivi che vedremo nei capitoli successivi, o che servono solo in casi rari ma, giunti a questo punto, dovrete cominciare a sentirvi a vostro agio nell'uso della shell di Mongo che nella comprensione dei principi fondamentali di MongoDB.

Capitolo 4 - Modellazione dei Dati

Cambiamo marcia e passiamo a un argomento più astratto che riguarda MongoDB. Spiegare qualche nuovo termine e nuove sintassi è tutto sommato un compito banale; parlare della modellazione dei dati applicata a un nuovo paradigma quale è NoSQL è tutt'altra cosa. In realtà in fatto di modellazione dati applicata a queste nuove tecnologie tutti noi siamo ancora impegnati nel tentativo di scoprire cosa funziona e cosa no. Possiamo discuterne, ma in ultima analisi dovrete far pratica e imparare lavorando sul vero codice.

In confronto alla gran parte delle soluzioni NoSQL, i database orientati ai documenti sono probabilmente i meno differenti dai database relazionali. Le differenze sono sottili, ma questo non significa che non siano importanti.

Niente Join

La prima e fondamentale differenza alla quale dovrete abituarvi è l'assenza, in MongoDB, delle join. Non conosco la ragione precisa per cui almeno qualche tipo di join non sia supportato in MongoDB ma so che, in linea generale, le join sono considerate poco scalabili. Una volta che si comincia a suddividere orizzontalmente i dati si finirà prima o poi per lanciare le join lato client (l'application server). Al di là delle spiegazioni rimane il fatto che i dati *sono* relazionali, e che MongoDB non supporta le join.

Per quel che sappiamo finora, sopravvivere in un mondo senza join significa eseguirle via codice nella nostra applicazione. In pratica dobbiamo lanciare una seconda query per trovare (find) i dati coerenti alla nostra ricerca. Impostare la ricerca non è diverso dal dichiarare una chiave esterna in un database relazionale. Lasciamo da parte i meravigliosi unicorni e passiamo agli impiegati (employees). La prima cosa che facciamo è creare un impiegato (al fine di costruire esempi coerenti userò un `_id` esplicito)

```
db.employees.insert({_id: ObjectId("4d85c7039ab0fd70a117d730"), name: 'Leto'})
```

Ora aggiungiamo un paio di impiegati e impostiamo Leto come loro manager:

```
db.employees.insert({_id: ObjectId("4d85c7039ab0fd70a117d731"), name: 'Duncan', manager:
  : ObjectId("4d85c7039ab0fd70a117d730")});
db.employees.insert({_id: ObjectId("4d85c7039ab0fd70a117d732"), name: 'Moneo', manager:
  ObjectId("4d85c7039ab0fd70a117d730")});
```

(vale la pena ripetere che `_id` può essere un qualunque valore univoco. Poiché in una applicazione vera useremmo probabilmente un `ObjectId`, lo usiamo anche nel nostro esempio)

Naturalmente per trovare tutti gli impiegati di Leto è sufficiente eseguire:

```
db.employees.find({manager: ObjectId("4d85c7039ab0fd70a117d730")})
```

Niente di speciale. La maggior parte delle volte e nel caso peggiore, l'assenza di join richiederà semplicemente l'esecuzione di una query in più (e probabilmente sarà eseguita su campi indicizzati).

Array e Documenti Incorporati

L'assenza di join non significa che MongoDB non abbia un paio di assi nella manica. Ricordate quando abbiamo detto che MongoDB supporta gli array come oggetti di prima classe del documento? Scopriamo che ciò è incredibilmente utile quando abbiamo a che fare con relazioni uno-a-molti oppure molti-a-molti. Per esempio nel caso che un impiegato possa avere due manager, potremmo memorizzarli facilmente in un array:

```
db.employees.insert({_id: ObjectId("4d85c7039ab0fd70a117d733"), name: 'Siona', manager:
  [ObjectId("4d85c7039ab0fd70a117d730"), ObjectId("4d85c7039ab0fd70a117d732")] })
```

E' interessante notare che per alcuni documenti manager può essere un valore scalare, mentre per altri può essere un array. La nostra query find originale funzionerà in entrambi i casi:

```
db.employees.find({manager: ObjectId("4d85c7039ab0fd70a117d730")})
```

Scoprirete presto che gli array di valori sono molto più convenienti che non le join molti-a-molti tra più tabelle.

Oltre agli array Mongo supporta i documenti incorporati. Provate a inserire un documento che a sua volta incorpori un altro documento, come per esempio:

```
db.employees.insert({_id: ObjectId("4d85c7039ab0fd70a117d734"), name: 'Ghanima', family
  : {mother: 'Chani', father: 'Paul', brother: ObjectId("4d85c7039ab0fd70a117d730")} })
```

Nel caso ve lo stiate chiedendo, i documenti incorporati possono essere cercati usando una notazione-a-punto:

```
db.employees.find({'family.mother': 'Chani'})
```

Tratteremo brevemente il ruolo dei documenti incorporati e l'uso che se ne dovrebbe fare.

DBRef

MongoDB supporta un aggeggio noto chiamato DBRef, che altro non è che una convenzione supportata da molti driver. Quando un driver incontra un DBRef può richiamare automaticamente il documento referenziato. Un DBRef include il nome della collezione e l'id del documento a cui fa riferimento. Di solito si usa in un caso specifico: quando documenti dalla stessa collezione possono far riferimento a documenti che appartengono a collezioni diverse l'una dall'altra. Per esempio il DBRef di documento1 potrebbe puntare a un documento contenuto in managers, mentre quello di documento2 potrebbe puntare a un documento in employees.

Denormalizzazione

Un'altra alternativa alle join consiste nel denormalizzare i dati. In passato la denormalizzazione è sempre stata riservata alle ottimizzazioni della performance, oppure ci si ricorreva quando era necessario creare degli snapshot dei dati (come nel caso dei log di revisione). Tuttavia con la popolarità crescente dei NoSQL, molti dei quali non hanno join, la denormalizzazione come parte integrante della modellazione dei dati si sta facendo sempre più frequente. Ciò non significa che è necessario duplicare ogni informazione in ogni documento. Tuttavia, piuttosto che lasciare che la paura

di duplicare dati vi guidi nel design, provate a modellare i dati basandovi su quale informazione appartiene a quale documento.

Per esempio immaginate di essere al lavoro su un forum. Il modo tradizionale di associare uno specifico user a un post è per via di una colonna `userId` nella tabella `posts`. Una alternativa possibili è quella di memorizzare semplicemente sia il nome (`name`) che il `userId` in ogni post. Potreste usare addirittura un documento incorporato, come: `user: { id: ObjectId('Something'), name: 'Leto' }`. E' vero, se consentite il cambio del nome degli utenti allora dovrete aggiornare ogni documento (il che significa una query aggiuntiva).

Per alcuni di noi adattarsi a questo tipo di approccio non sarà una passeggiata. In molti casi non avrà effettivamente senso. Tuttavia non abbiate timore di sperimentarlo. Non solo è adattabile a diverse circostanze, ma addirittura potrebbe risultare la cosa giusta da fare.

Quale Scegliere?

Gli array di id sono sempre una strategia utile quando abbiamo a che fare con scenari uno-a-molti o molti-a-molti. E' probabilmente il caso di ammettere che i `DBRef` non sono usati di frequente, ma se siete senz'altro liberi giocarci un pò. I nuovi sviluppatori si domandano spesso cosa sia meglio tra documenti incorporati e riferimenti manuali.

Prima di tutto sappiate che al momento i singoli documenti hanno un limite a 16 megabyte. Sapere che c'è un limite alla dimensione dei documenti, benché piuttosto ampio, aiuta a farsi una idea di come bisognerebbe usarli. Al momento pare che gran parte dei programmatori ricorra pesantemente ai riferimenti diretti per la maggioranza delle relazioni. I documenti incorporati sono molto usati, ma per blocchi di dati relativamente piccoli, che si vogliono sempre richiamare col documento principale. Un esempio reale che ho usato in passato è il salvataggio di un documento `accounts` per ogni utente, qualcosa tipo:

```
db.users.insert({name: 'leto', email: 'leto@dune.gov', account: {allowed_gholas: 5,
    spice_ration: 10}})
```

Questo non significa che dovrete sottovalutare la potenza dei documenti incorporati, o derubarli a utility di importanza secondaria. Avere un modello dati mappato direttamente sugli oggetti rende tutto molto semplice e spesso elimina la necessità di join, il che è particolarmente vero visto che MongoDB permette ricerche e indicizzazioni sui campi di un documento incorporato.

Poche o Tante Collezioni

Dato che le collezioni non impongono alcun schema è ovviamente possibile concepire un sistema con una sola collezione contenente oggetti di ogni tipo. Per quanto ho visto io la maggior parte dei sistemi MongoDB è disposto in maniera simile a quella che troviamo in un sistema relazionale. In altre parole se in un database relazione ci vorrebbe una tabella, allora è probabile in MongoDB che ci voglia una collezione (in questo caso le tabelle per relazioni molti-a-molti sono una importante eccezione).

La faccenda si fa ancor più interessante prendendo in considerazione i documenti incorporati. L'esempio più usato è il blog. Dovremmo avere una collezione `posts` e una collezione `comments`, oppure dovremmo far sì che ogni post abbia una array di `comments` incorporati? Lasciando da parte il limite dei 16MB (tutto l'Amleto è meno di 200KB, quanto è

famoso il vostro blog?) la maggior parte degli sviluppatori preferiscono separare le cose. E' semplicemente più limpido ed esplicito.

Non ci sono regole precise (a parte la faccenda dai 16MB). Giocate con i diversi approcci e capirete presto cosa ha senso e cosa non funziona nel vostro caso.

Riepilogo

In questo capitolo il nostro scopo era fornire delle linee guida utili alla modellazione dati in MongoDB. Un punto di partenza, se volete. La modellazione in un sistema orientato ai documenti è cosa diversa, ma non troppo, da quella nel mondo relazionale. C'è un pò più di flessibilità e un vincolo in più ma, per essere un nuovo sistema, le cose sembrano aggiustarsi piuttosto bene. L'unico modo di sbagliare è non provarci nemmeno.

Capitolo 5 - Quando Scegliere MongoDB

Giunti a questo punto dovremmo conoscere Mongo abbastanza bene da intuirne il ruolo che può svolgere nel nostro sistema. Ci sono talmente tante nuove tecnologie in competizione tra loro, così tante possibilità che è facile lasciarsi intimidire.

Per quanto mi riguarda la lezione più importante, che non ha nulla a che vedere con MongoDB, è che non siamo più costretti ad affidarci a un'unica soluzione per la gestione dei nostri dati. Non c'è alcun dubbio sul fatto che l'adozione di un'unica soluzione offra ovvi vantaggi, e che per molti progetti, probabilmente la maggior parte, questo sia l'approccio migliore. L'idea non è che bisogna per forza usare più tecnologie, ma piuttosto che è possibile farlo. Solo voi potete sapere se l'affiancare nuove tecnologie al vostro progetto può portare più vantaggi o svantaggi.

Detto questo, sono fiducioso che ciò che abbiamo visto sinora abbia messo in luce MongoDB come soluzione generale. Abbiamo già detto un paio di volte che i database orientati ai documenti hanno molto in comune con quelli relazionali. Allora, piuttosto che girarci attorno diciamolo chiaramente: MongoDB dovrebbe essere considerato una alternativa diretta ai database relazionali. Se Lucene è un database relazionale con indicizzazione full-text e Redis è un archivio persistente di coppie chiave-valore, allora MongoDB è deposito centralizzato per i nostri dati.

Notate che non ho definito MongoDB una *sostituzione* dei database relazionali, ma piuttosto una *alternativa*. E' uno strumento in grado di fare gran parte delle cose che fanno gli altri strumenti, alcune le fa meglio, altre peggio. Approfondiamo un pò il discorso.

Schema-less

Un aspetto molto propagandato dei database orientati ai documenti è l'assenza di schema, il che li rende molto più flessibili delle tabelle dei tradizionali database relazionali. Io concordo che schema-less sia una bella caratteristica, ma non per la ragione che la maggior parte della gente pensa.

Quando pensiamo di strutture senza schema immaginiamo di archiviare dati eterogenei. Ci sono domini e set di dati che possono essere davvero difficili da modellare con i database relazionali, ma si tratta di casi limite. Schema-less è bello, ma la gran parte dei dati finirà per essere altamente strutturata. E' vero che avere dati eterogenei è comodo, specialmente quando introduciamo novità, cosa che in realtà potremmo ottenere con una banale colonna nullabile in un database relazionale.

Per quanto mi riguarda il vero vantaggio del design schema-less è l'assenza di un setup iniziale e la ridotta frizione con la programmazione orientata agli oggetti, cosa particolarmente utile se adoperiamo un linguaggio statico. Ho usato MongoDB sia in C# che in Ruby e la differenza è impressionante. Il dinamismo di Ruby e le sue rinomate implementazioni ActiveRecord riducono già sensibilmente il problema della discordanza tra oggetti e database. Ciò non significa che MongoDB sia una scelta superflua per Ruby, al contrario. Credo che per i programmatori Ruby MongoDB sia un miglioramento, mentre quelli C# o Java si riverlerà un cambiamento fondamentale nel modo di interagire coi dati.

Vedetela dal punto di vista di uno sviluppatore di driver. Vuoi salvare un oggetto? Serializzalo in JSON (tecnicamente si tratta di BSON, ma poco cambia) e invialo a MongoDB. Non c'è mappatura delle proprietà, o dei tipi di dato. Questa immediatezza si riflette direttamente su di noi, gli sviluppatori finali.

Scritture

Un'area in cui MongoDB può svolgere un ruolo peculiare è il logging. Ci sono due fattori che rendono le scritture in MongoDB piuttosto veloci. Primo, possiamo dare un comando di scrittura e vederlo ritornare senza aspettare che sia effettivamente avvenuta. Secondo, con l'introduzione del journaling nella versione 1.8 e i miglioramenti fatti nella 2.0 possiamo controllare il comportamento delle write per quanto riguarda la durabilità dei dati. Queste impostazioni, in aggiunta all'indicazione di quanti server devono ricevere i dati prima che una scrittura sia considerata sicura, sono configurabili per ogni write, il che ci garantisce un ottimo controllo sulle performance.

In aggiunta alle performance, il log è uno di quei tipi di dato che può avvantaggiarsi delle collezioni senza schema. Infine, MongoDB è dotato delle [collezioni limitate](#). Finora tutte quelle create erano collezioni normali. Possiamo creare una collezione limitata passando il parametro `capped` al comando `db.createCollection`:

```
//limitiamo la nostra collezione alla dimensione massima di 1 megabyte
db.createCollection('logs', {capped: true, size: 1048576})
```

Quando la dimensione della nostra collezione raggiungerà il limite di 1MB i vecchi documenti verranno cancellati automaticamente. E' anche possibile usare `max` per impostare un limite sul numero dei documenti piuttosto che sulla dimensione della collezione. Le collezioni limitate hanno alcune caratteristiche interessanti. Per esempio, è possibile aggiornare un documento ma non aumentarne le dimensioni. Inoltre l'ordine di inserimento è preservato, così non è necessario aggiungere un indice ulteriore per fare ordinamenti sulla data di creazione.

Questo è un buon momento per dirvi che per sapere se la vostra write è andata bene basta farle seguire il comando `db.getLastError()`. La maggior parte dei driver supportati implementa questa opzione come una *safe write*, per esempio specificando `{:safe => true}` come secondo parametro della `insert`.

Affidabilità

Fino alla versione 1.8 MongoDB non era molto affidabile su server singolo. Ciò significa che un crash sul server avrebbe condotto probabilmente a una perdita di dati. La soluzione è sempre stata quella di eseguire MongoDB in configurazione multi-server (Mongo supporta la replication). Una delle più importanti novità introdotte con la versione 1.8 è il journaling. Per attivarlo basta aggiungere una nuova linea con `journal=true` al file `mongodb.config`, quello che abbiamo creato quando abbiamo configurato MongoDB la prima volta (è necessario riavviare il server se vogliamo attivare subito il journaling). Probabilmente vale sempre la pena di attivare il journaling (sarà attivo di default in una delle prossime versioni di MongoDB). A volte tuttavia potrebbe valer la pena di sacrificare il journaling in cambio dell'aumento di prestazioni (alcuni tipi di applicazioni possono accettare il rischio di perdita dati).

In passato si è discusso molto della mancanza in MongoDB del supporto per la durabilità sul singolo server. E' probabile che queste discussioni salteranno fuori su Google ancora per parecchio tempo, ma sappiate che si tratta, semplicemente, di informazioni non obsolete.

Ricerca Full-Text

Speriamo che la ricerca full text arrivi con una dei prossimi aggiornamenti di MongoDB. Grazie al supporto per gli array è piuttosto facile implementare una ricerca full text di base. Per ottenere qualcosa di più potente ci dovremo rivolgere

a soluzioni tipo Lucene/Solr. Naturalmente, questo vale anche per molti database relazionali.

Transazioni

MongoDB non supporta le transazioni. Offre due alternative, una delle quali è ottima, ma di uso limitato, mentre l'altra è macchinosa, ma flessibile.

La prima corrisponde alle sue molte operazioni atomiche. Sono eccellenti, fintanto che riescono a risolvere il nostro problema. Abbiamo già visto alcune delle più semplici, come `$inc` e `$set`. Sono disponibili anche comandi come `findAndModify`, che aggiorna o cancella un documento e lo restituisce atomicamente.

La seconda soluzione, da usare quando le operazioni atomiche non sono sufficienti, consiste nel ripiegare su una commit in due fasi. Una commit in due fasi è l'equivalente, per le transazioni, della dereferenziazione per le join. Si tratta di implementare la soluzione nel proprio codice, indipendentemente dalla base dati. In realtà le commit a due fasi sono piuttosto diffuse nel mondo dei database relazionali, nel quale vengono usate per implementare transazioni multi-database. Il sito di MongoDB [propone un esempio](#) di uno scenario tipico, un trasferimento di fondi. L'idea di fondo è che lo stato della transazione venga archiviato col documento stesso, e che si proceda manualmente alle varie fasi init-pending-commit/rollback.

Il supporto di MongoDB per documenti nidificati e design schema-less rende meno impegnative le commit in due fasi, ma senz'altro non si tratta di una procedura comoda, specialmente se siamo alle prime armi.

Elaborazione Dati

MongoDB si affida a MapReduce per la gran parte dei compiti di elaborazione. E' dotato di alcune capacità di [aggregazione di base](#), ma per qualcosa di serio dovreste senz'altro ricorrere a MapReduce. Nel prossimo capitolo vedremo MapReduce in dettaglio. Per il momento possiamo limitarci a considerarlo una tecnica diversa e molto potente per eseguire dei group by (stiamo semplificando). Uno dei punti di forza di MapReduce è che quando servono elaborazioni su grandi quantità di dati è possibile lanciarlo in parallelo. Tuttavia l'implementazione di MongoDB si affida a JavaScript, che è single-threaded. Dunque? Per elaborare una grande mole di dati ci sarà bisogno di rivolgersi a qualcos'altro, come Hadoop. Per fortuna i due sistemi si complementano a vicenda, ed esiste un [adapter MongoDB per Hadoop](#).

Naturalmente, l'elaborazione parallela dei dati non è un campo in cui i database relazionali sono particolarmente brillanti. In una delle future versioni di MongoDB è comunque prevista una gestione migliore dei big data.

Geospazialità

Una caratteristica particolarmente potente di MongoDB è il suo supporto per gli indici geospaziali. Consente di archiviare coordinate x e y nei documenti, e in seguito di cercare documenti che sono `$near` (vicini) un set di coordinate, o `$within` (contenuti) in un rettangolo oppure un cerchio. E' una caratteristica più facile da comprendere visivamente, pertanto vi invito a provare il [tutorial geospaziale interattivo di 5 minuti](#) se volete saperne di più.

Strumenti e Maturità

Probabilmente lo sapete già, ma MongoDB è ovviamente più giovane della maggior parte dei database relazionali. Questo è fattore da considerare con attenzione. Quanto, dipende da cosa state facendo e da come lo state facendo. In ogni caso e semplicemente, un ragionamento serio non può ignorare il fatto che MongoDB è più giovane, e che gli strumenti a disposizione non sono fantastici (anche se bisogna dire che gli strumenti a disposizione di molti database relazionali maturi sono anch'essi terribili!). Per esempio, la mancanza di supporto per i numeri a virgola mobile in base 10 si rivelerà senz'altro un grattacapo (ma non necessariamente la fine dei giochi) per i sistemi che devono gestire del denaro.

Di buono c'è che esistono driver per molti linguaggi, che il protocollo è moderno e semplice, e che lo sviluppo avviene a grandissima velocità. MongoDB è usato in produzione da un numero tale di aziende che le preoccupazioni sulla sua maturità, pur valide, stanno rapidamente diventando una cosa del passato.

Riepilogo

Il messaggio da cogliere da questo capitolo è che nella maggior parte dei casi MongoDB può sostituire un database relazionale. E' molto più semplice e diretto; è più veloce e in generale impone meno restrizioni agli sviluppatori. La mancanza di transazioni può rivelarsi un limite significativo. Eppure, quando ci chiediamo quale sia il ruolo di MongoDB nel panorama dei nuovi sistemi di archiviazione, la risposta è semplice: **proprio nel mezzo.**

Capitolo 6 - MapReduce

MapReduce è un approccio all'elaborazione dati che vanta due vantaggi significativi rispetto alle altre soluzioni tradizionali. Il primo, e più importante, sono le performance. In linea teorica MapReduce può operare in parallelo elaborando grandi set di dati contemporaneamente su più core/CPU/computer. Come abbiamo già visto, però, attualmente MongoDB non è in grado di sfruttare pienamente questo aspetto. Rispetto a quel che è possibile fare con SQL, il codice di MapReduce è infinitamente più ricco e ci permette di spingerci molto avanti prima che una soluzione ancor più specializzata si renda necessaria.

La popolarità del modello MapReduce è cresciuta molto, ed è ora possibile usarlo praticamente ovunque; C#, Ruby, Java, Python e così via, tutti ne offrono una implementazione. Vi avverto, all'inizio MapReduce sembrerà molto diverso da ciò a cui siete abituati, e piuttosto complicato. Non demoralizzatevi, prendetevi il tempo di sperimentare voi stessi. Vale la pena comprendere MapReduce a prescindere dal fatto che lo usiate con MongoDB o meno.

Teoria e Pratica

MapReduce è un processo in due fasi. Prima si mappa (map) e poi si riduce (reduce). Il mapping trasforma i documenti del flusso di input emettendo una coppia chiave=>valore (chiave e valore possono essere complessi). La reduce prende la chiave e l'array di valori ad essa abbinati e li usa per produrre il risultato finale. Affronteremo entrambe le fasi vedendo l'output di ognuna.

L'esempio che useremo è la generazione di un report col numero di viste (hits) giornaliere che otteniamo per una data risorsa (diciamo una pagina web). E' lo *hello world* del MapReduce. Per raggiungere il nostro scopo ci affideremo a una collezione hits che conterrà due campi: resource (risorsa) e date. L'output che vogliamo ottenere è un elenco con le seguenti colonne: resource, year, month, day e count.

Dati i seguenti contenuti di hits:

resource	date
index	Jan 20 2010 4:30
index	Jan 20 2010 5:30
about	Jan 20 2010 6:00
index	Jan 20 2010 7:00
about	Jan 21 2010 8:00
about	Jan 21 2010 8:30
index	Jan 21 2010 8:30
about	Jan 21 2010 9:00
index	Jan 21 2010 9:30
index	Jan 22 2010 5:00

Desideriamo ottenere i seguenti risultati:

resource	year	month	day	count
index	2010	1	20	3
about	2010	1	20	1

about	2010	1	21	3
index	2010	1	21	2
index	2010	1	22	1

La cosa bella di questo tipo di approccio è che, salvando l'output, i report sono veloci da generare e la crescita dei dati è controllata (per ogni risorsa che tracciamo aggiungeremo non più di 1 documento al giorno)

Per il momento concentriamoci sul concetto. Alla fine di questo capitolo vi fornirò dati e codice di esempio da usare per fare esperimenti per conto vostro.

Per prima cosa affrontiamo la funzione map. L'obiettivo della map è emettere un valore che può essere ridotto. E' possibile che map emetta 0 o più risultati. Nel nostro caso emetterà un solo risultato (cosa che capita spesso). Immaginiamo la map come un ciclo che scorre i documenti della collezione hits. Per ogni documento vogliamo emettere una chiave con resource, year, month e day, ed un semplice valore 1:

```
function() {
    var key = {
        resource: this.resource,
        year: this.date.getFullYear(),
        month: this.date.getMonth(),
        day: this.date.getDate()
    };
    emit(key, {count: 1});
}
```

this fa riferimento al documento trattato al momento. Vedere l'output del nostro mapping agevolerà la comprensione del procedimento. Usando i dati visti sopra, l'output completo sarebbe:

```
{resource: 'index', year: 2010, month: 0, day: 20} => [{count: 1}, {count: 1}, {count: 1}]
{resource: 'about', year: 2010, month: 0, day: 20} => [{count: 1}]
{resource: 'about', year: 2010, month: 0, day: 21} => [{count: 1}, {count: 1}, {count: 1}]
{resource: 'index', year: 2010, month: 0, day: 21} => [{count: 1}, {count: 1}]
{resource: 'index', year: 2010, month: 0, day: 22} => [{count: 1}]
```

Capire questo passaggio intermedio è la chiave per comprendere MapReduce. I valori della emit sono raggruppati, in forma di array, per ogni chiave. Gli sviluppatori .NET e Java possono immaginare che si tratti di qualcosa del tipo: IDictionary<object, IList<object>> (.NET) oppure HashMap<Object, ArrayList> (Java).

Cambiamo la nostra map function in modo piuttosto artificioso:

```
function() {
    var key = {resource: this.resource, year: this.date.getFullYear(), month: this.date.getMonth(), day: this.date.getDate()};
    if (this.resource == 'index' && this.date.getHours() == 4) {
        emit(key, {count: 5});
    }
}
```



```

    } else {
        emit(key, {count: 1});
    }
}

```

Il primo output intermedio cambierebbe in:

```

{resource: 'index', year: 2010, month: 0, day: 20} => [{count: 5}, {count: 1}, {count: 1}]

```

Notate come ogni emit genera un nuovo valore che è raggruppato in base alla nostra chiave (key).

La funzione reduce prende ognuno di questi risultati intermedi e genera l'output finale. Ecco come appare la nostra funzione reduce:

```

function(key, values) {
    var sum = 0;
    values.forEach(function(value) {
        sum += value['count'];
    });
    return {count: sum};
};

```

Che restituisce l'output seguente:

```

{resource: 'index', year: 2010, month: 0, day: 20} => {count: 3}
{resource: 'about', year: 2010, month: 0, day: 20} => {count: 1}
{resource: 'about', year: 2010, month: 0, day: 21} => {count: 3}
{resource: 'index', year: 2010, month: 0, day: 21} => {count: 2}
{resource: 'index', year: 2010, month: 0, day: 22} => {count: 1}

```

Tecnicamente, l'output in MongoDB è:

```

_id: {resource: 'home', year: 2010, month: 0, day: 20}, value: {count: 3}

```

Avrete notato che questo è proprio il risultato finale che stavamo cercando.

Se avete prestato attenzione vi sarete forse chiesti *perché non abbiamo usato semplicemente sum = values.Length?* Sembrerebbe un approccio efficace visto che stiamo essenzialmente sommando un array di 1. Il fatto è che non sempre reduce è usato con un set di dati intermedi perfetti e completi. Per esempio, se invece di essere chiamato con:

```

{resource: 'home', year: 2010, month: 0, day: 20} => [{count: 1}, {count: 1}, {count: 1}]

```

Reduce fosse chiamato con:

```

{resource: 'home', year: 2010, month: 0, day: 20} => [{count: 1}, {count: 1}]
{resource: 'home', year: 2010, month: 0, day: 20} => [{count: 2}, {count: 1}]

```

L'output finale è lo stesso (3) ma il percorso fatto è, semplicemente, diverso. Per questo motivo reduce deve sempre essere idempotente, il che significa che chiamare reduce più volte dovrebbe sempre generare il risultato che si otterrebbe con una sola chiamata.

Non lo faremo qui, ma è frequente concatenare metodi reduce quando si eseguono analisi più complesse.

Pratica Pura

Con MongoDB usiamo il comando mapReduce su una collezione. mapReduce accetta una funzione map, una funzione reduce e una direttiva di output. Nella nostra shell possiamo creare e passare una funzione JavaScript. Con la maggior parte delle librerie potete passare una stringa che contiene le vostre funzioni (il che è piuttosto brutto). Prima di tutto creiamo il nostro semplice set di dati:

```
db.hits.insert({resource: 'index', date: new Date(2010, 0, 20, 4, 30)});
db.hits.insert({resource: 'index', date: new Date(2010, 0, 20, 5, 30)});
db.hits.insert({resource: 'about', date: new Date(2010, 0, 20, 6, 0)});
db.hits.insert({resource: 'index', date: new Date(2010, 0, 20, 7, 0)});
db.hits.insert({resource: 'about', date: new Date(2010, 0, 21, 8, 0)});
db.hits.insert({resource: 'about', date: new Date(2010, 0, 21, 8, 30)});
db.hits.insert({resource: 'index', date: new Date(2010, 0, 21, 8, 30)});
db.hits.insert({resource: 'about', date: new Date(2010, 0, 21, 9, 0)});
db.hits.insert({resource: 'index', date: new Date(2010, 0, 21, 9, 30)});
db.hits.insert({resource: 'index', date: new Date(2010, 0, 22, 5, 0)});
```

Ora possiamo creare le nostre funzioni map e reduce (la shell MongoDB accetta comandi multi-riga, vedrete apparire ... dopo la pressione di invio ad indicare che ci si aspetta altro testo)

```
var map = function() {
    var key = {resource: this.resource, year: this.date.getFullYear(), month: this.date
        .getMonth(), day: this.date.getDate()};
    emit(key, {count: 1});
};

var reduce = function(key, values) {
    var sum = 0;
    values.forEach(function(value) {
        sum += value['count'];
    });
    return {count: sum};
};
```

Ora possiamo lanciare il comando mapReduce sulla nostra collezione hits:

```
db.hits.mapReduce(map, reduce, {out: {inline:1}})
```

Eseguendo il comando qui sopra dovreste ottenere il risultato desiderato. Impostare `out` a `inline` comporta che l'output di `mapReduce` ci venga immediatamente restituito. Attualmente c'è un limite a 16 megabyte o meno per gli output di questo tipo. Potremmo invece specificare `{out: 'hit_stats'}` per inviare i risultati alla nuova collezione `hit_stats`:

```
db.hits.mapReduce(map, reduce, {out: 'hit_stats'});  
db.hit_stats.find();
```

Facendo questo eventuali dati esistenti in `hit_stats` andranno perduti. Se facessimo `{out: {merge: 'hit_stats'}}` le chiavi esistenti verrebbero aggiornate coi nuovi valori e le nuove chiavi verrebbero inserite come nuovi documenti. Potremmo infine fare `out` su una funzione `reduce` per gestire casi più avanzati (come una `upsert`)

Il terzo parametro accetta opzioni aggiuntive, potremmo per esempio filtrare, ordinare e limitare i documenti che vogliamo analizzare. Possiamo inoltre fornire un metodo `finalize` da applicare ai risultati successivamente alla fase `reduce`.

Riepilogo

Questo è il primo capitolo in cui abbiamo affrontato qualcosa di veramente diverso dal solito. Se vi siete trovati a disagio, tenete presente che potete sempre ricorrere alle altre [capacità di aggregazione](#) offerte da MongoDB, adatte a scenari più semplici. In fin dei conti MapReduce è una delle caratteristiche più accattivanti di MongoDB. La chiave per comprendere a fondo come scrivere le vostre funzioni di `map` e `reduce` è visualizzare e comprendere l'aspetto che i dati intermedi assumeranno all'uscita della `map`, pronti per passare alla `reduce`.

Chapter 7 - Performance e Strumenti

In quest'ultimo capitolo tratteremo le questioni di performance e daremo un'occhiata ad alcuni strumenti a disposizione degli sviluppatori MongoDB. Per ognuno degli argomenti esamineremo gli aspetti più importanti, senza scendere troppo nel dettaglio.

Indici

All'inizio del libro abbiamo incontrato la collezione speciale `system.indexes` che contiene informazioni su tutti gli indici del nostro database. Gli indici in MongoDB funzionano in maniera molto simile a quella dei database relazionali: migliorano le performance di ricerche e ordinamenti. Gli indici vengono creati con `ensureIndex`:

```
db.unicorns.ensureIndex({name: 1});
```

E cancellati con `dropIndex`:

```
db.unicorns.dropIndex({name: 1});
```

Un indice univoco si crea passando un secondo parametro e impostando `unique` a `true`:

```
db.unicorns.ensureIndex({name: 1}, {unique: true});
```

Gli indici possono venire creati su campi incorporati (ancora una volta ricorrendo alla notazione col punto) e sugli array. E' anche possibile creare indici composti:

```
db.unicorns.ensureIndex({name: 1, vampires: -1});
```

L'ordine dell'indice (1 per ascendente, -1 per discendente) non importa per gli indici singoli, ma può avere un impatto sugli indici composti, quando si combinano condizioni di range (gamma) e ordinamenti.

Maggiori informazioni sugli indici sono reperibili sulla [pagina sugli indici](#) del sito ufficiale.

Explain

Per capire se le nostre query stanno usando un indice possiamo ricorrere al metodo `explain` applicato a un cursore:

```
db.unicorns.find().explain()
```

L'output ci dice che è stato usato un `BasicCursor` (non-indicizzato), che 12 oggetti sono stati trattati, quanto tempo c'è voluto, quale indice è stato eventualmente usato, e qualche altra informazione utile.

Se facciamo in modo che la nostra query ricorra a un indice scopriremo che è stato usato `BtreeCursor`, e ci verrà detto il nome dell'indice applicato:

```
db.unicorns.find({name: 'Pilot'}).explain()
```

Scritture 'Fire and Forget'

Abbiamo già detto prima che, per default, le write in MongoDB sono del tipo fire-and-forget. Questo comporta un miglioramento delle performance al costo di un aumento del rischio di perdita dati in caso di crash. Un effetto interessante di questo tipo di approccio alla scrittura dati è che non viene restituito alcun errore quando una insert/update finisce per violare un vincolo di univocità. Per informarci sugli eventuali errori di scrittura dobbiamo chiamare esplicitamente il metodo `db.getLastError()` dopo una insert. Molti driver astraggono questo dettaglio gestendolo internamente, e forniscono un metodo diretto per lanciare scritture *sicure* - spesso ricorrendo a un parametro extra.

Sfortunatamente la shell esegue automaticamente delle scritture sicure, pertanto non possiamo vedere facilmente questo comportamento in azione.

Sharding

MongoDB supporta l'auto-sharding. Lo sharding è un approccio alla scalabilità che ripartisce i dati su server multipli. Una implementazione banale potrebbe salvare tutti i dati degli utenti con nome che comincia per A-M sul server 1, e il resto sul server 2. Per fortuna le capacità di sharding di MongoDB sono nettamente superiori. L'argomento Sharding è ben al di là degli scopi di questo libro, ma sappiate che esiste e che dovrete considerarlo nel caso le vostre necessità vadano oltre il singolo server.

Replicazione

La replicazione in MongoDB funziona in modo simile a quella dei database relazionali. Le scritture vengono inviate a un singolo server, il master, che in seguito si sincronizza con uno o più server, gli slave. Possiamo controllare se le letture possono avvenire o meno sugli slave, il che ci può aiutare a distribuire il carico di lavoro correndo però il rischio di leggere dati leggermente obsoleti. Se il master fallisce, uno slave può venir promosso al ruolo di master. Anche la replication è al di là degli scopi di questo libro.

E' vero che la replication può migliorare le performance (distribuendo le letture), ma il suo scopo principale è migliorare l'affidabilità. Combinare replication e sharding è un approccio molto diffuso. Per esempio, ogni shard potrebbe essere configurato con un master e uno slave. (Tecnicamente ci sarà anche bisogno di un arbitro per decidere chi promuovere nel caso di due slave che tentano entrambi di diventare master. Ma un arbitro richiede poche risorse e può essere usato per più shard.)

Statistiche

Possiamo ottenere informazioni su un database digitando `db.stats()`. Gran parte delle informazioni riguardano le dimensioni del database. Possiamo anche ottenere informazioni su una collezione, per esempio `unicorns`, digitando `db.unicorns.stats()`. Anche in questo caso le informazioni riguardano più che altro le dimensioni della nostra collezione.

Intefaccia Web

Tra le informazioni disponibili quando abbiamo lanciato MongoDB c'era un link a uno strumento amministrativo su web (potrebbe essere ancora visibile se fate scorrere la finestra di comando/terminale fino al punto in cui avete lanciato mongod). Lo strumento amministrativo è accessibile puntando il browser all'indirizzo <http://localhost:28017/>. Varrà la pena aggiungere `rest=true` al config e riavviare il processo mongod. L'interfaccia web fornisce molte informazioni sullo stato del server.

Profiler

Possiamo attivare il profiler di MongoDB eseguendo:

```
db.setProfilingLevel(2);
```

Una volta attivato, possiamo lanciare un comando:

```
db.unicorns.find({weight: {$gt: 600}});
```

Quindi esaminare il profiler:

```
db.system.profile.find()
```

L'output ci dirà che cosa è stato eseguito e quando, quanti documenti sono stati considerati e quanti dati sono stati effettivamente restituiti.

Il profiler si disattiva chiamando di nuovo `setProfileLevel` ma questa volta impostando l'argomento a 0. Un'altra opzione è specificare 1, che profilerà solo le query che richiedono più di 100 millisecondi. Oppure possiamo specificare il tempo minimo, in millisecondi, ricorrendo a un secondo parametro:

```
//profile per qualunque cosa che impieghi più di 1 secondo
db.setProfilingLevel(1, 1000);
```

Backup e Restore

Nella cartella bin di MongoDB c'è l'eseguibile `mongodump`. Semplicemente lanciandolo, `mongodump` si connette al local-host ed esegue il backup dei database in una sotto-cartella `dump`. Possiamo digitare `mongodump --help` per scoprire le opzioni aggiuntive. Le più comuni sono `--db NOME` per fare il backup di uno specifico database e `--collection NOMECOLLEZIONE` per fare il backup di una certa collezione. Successivamente potremo ricorrere all'eseguibile `mongorestore`, sempre nella cartella bin, per ripristinare un backup precedente. Anche in questo caso potremo usare `--db` e `--collection` per ripristinare database o collezione specifici.

Per esempio se volessimo eseguire il backup della collezione `learn` in una cartella `backup`, dovremmo eseguire (questi sono programmi indipendenti, non funzioneranno dall'interno della shell di mongo):

```
mongodump --db learn --out backup
```

Per ripristinare solo la collezione `unicorns` potremmo eseguire:

```
mongorestore --collection unicorns backup/learn/unicorns.bson
```

Vale la pena ricordare che `mongoexport` e `mongoimport` sono altri due programmi che possono essere usati per esportare e importare dati da JSON o CSV. Per esempio possiamo ottenere un output JSON eseguendo:

```
mongoexport --db learn -collection unicorns
```

E un output CSV eseguendo:

```
mongoexport --db learn -collection unicorns --csv -fields name,weight,vampires
```

Tenete presente che `mongoexport` e `mongoimport` non sono sempre in grado di rappresentare i dati. Solo `mongodump` e `mongorestore` dovrebbero essere usati per ottenere dei veri backup.

Riepilogo

In questo capitolo abbiamo scoperto vari comandi, strumenti e alcuni dettagli relativi alle performance di MongoDB. Non abbiamo visto tutto, ci siamo limitati a quelli usati più spesso. L'indicizzazione in MongoDB è simile a quella dei database relazionali, così come molti degli altri strumenti. In MongoDB, tuttavia, molti di questi strumenti sono davvero facili da usare.

Conclusione

Ora dovrete essere in possesso di informazioni sufficienti per usare MongoDB in un progetto reale. In MongoDB c'è più di quel che abbiamo trattato, ma giunti a questo punto la vostra priorità è mettere a frutto quel che avete imparato e acquisire familiarità col driver che userete. Il [sito MongoDB](#) è ricco di informazioni utili e il [MongoDB user group ufficiale](#) è il posto ideale dove porre le vostre domande.

NoSQL è nato non solo per necessità ma anche dall'interesse genuino verso la sperimentazione di nuovi approcci. E' risaputo quanto il nostro settore sia in continua evoluzione e che, se non tentiamo, a volte anche fallendo, non otterremo alcun successo. Credo che questo sia un buon approccio per condurre la nostra vita professionale.