UNIVERSITÀ DEGLI STUDI DI PADOVA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

CORSO DI LAUREA TRIENNALE IN
INGEGNERIA INFORMATICA

I DATABASE NON RELAZIONALI, LE PRESTAZIONI DI MONGODB

Relatore: Giorgio Maria Di Nunzio

Laureando: Marzia Marsura

Anno Accademico 2012/2013

Ringraziamenti

Innanzitutto ringrazio il Professor Giorgio Maria Di Nunzio per avermi fatto da relatore per la mia tesi.

Inoltre ringrazio i miei genitori per il sostegno dato, sia economico che morale, e per essermi sempre stati accanto durante sia i momenti felici che i momenti difficili che mi hanno portato a raggiungere questo traguardo.

Infine ringrazio tutti i miei amici e i miei compagni di corso che hanno condiviso con me molti momenti e mi sono stati vicino durante questo percorso di studi.

Indice

Sommario

L'obiettivo di questa tesi è lo studio dei database non relazionali, in particolar modo del database *document-oriented* MongoDB.

I database non-relazionali sono una tipologia di nuovi database che si stanno sviluppando in questi ultimi anni per soddisfare le nuove esigenze del web visto l'aumento impressionante del carico di dati da memorizzare.

Essi si allontanano dalla filosofia dei database relazionali che si basano su proprietà logico-relazionali ed hanno una struttura molto rigida.

Come vedremo infatti, MongoDB, non punta tanto a memorizzare i dati in strutture predefinite basate su regole rigide, ma piuttosto si occupa di rendere più performanti sia le operazioni di scrittura che quelle di lettura attraverso documenti *schemaless* che non richiedono operazioni costose per il recupero delle informazioni d'interesse. Essendo inoltre questi documenti separati tra loro, è resa possibile la distribuzione dei dati su più server.

Per dimostrare la capacità di MongoDB di gestire enormi quantità di documenti, dopo aver descritto dettagliatamente i metodi di creazione di un database, verrà presentato un esempio di database progettato sia con MongoDB che con un database relazionale in MySQL e di questi verranno analizzati gli inserimenti e i tempi di risposta alle interrogazioni, mettendo così in evidenza le nuove prestazioni di MongoDB.

Capitolo 1

INTRODUZIONE

Oggigiorno risulta quasi impossibile non imbattersi almeno una volta nell'arco della giornata in un sistema di gestione di basi di dati; ad esempio andando in banca a depositare o prelevare denaro, accedendo in internet a siti di vendita online, prenotando un volo o un albergo per una vacanza o tante altre azioni, prevedono l'accesso e/o l'aggiornamento ad una base di dati. I sistemi di basi di dati si possono infatti considerare la tecnologia predominante per la memorizzazione di dati strutturati.
Da quando Edgar F. Codd, nel 1970, per semplificare la scrittura di interrogazioni sui database e per favorire l'indipendenza dei dati, pubblicò l'articolo "*A relational model of data for large shared data banks",* introducendo così il modello relazionale, strutturato intorno al concetto matematico di relazione o tabella, la maggior parte di sistemi di basi di dati viene progettata seguendo questo modello.
Negli ultimi anni però, aumentando sempre di più il volume di dati da memorizzare e avendo la necessità di elaborare grandi quantità di dati in poco tempo, si sta andando incontro ad un nuovo modello di gestione dei dati che si allontana dal modello relazionale. Questo nuovo modello prende il nome di NoSQL, che sta per *"Not Only SQL"* o *"Not Relational",* proprio per sottolineare il distacco dai database relazionali (SQL è il più comune linguaggio di interrogazione dei dati nei database relazionali).
Questi archivi di dati, a differenza di quelli costruiti basandosi sul modello relazionale, non presuppongono una struttura rigida o uno schema dove si descrivono le proprietà che i dati dovranno avere e le relazioni tra di essi. I database non relazionali puntano infatti ad essere più flessibili, in quanto ormai i flussi di dati che arrivano dalle sorgenti massive del web presentato notevoli irregolarità e sono più sparpagliati e sparsi (Big Data).
Per aumentare il livello della prestazione nella gestione e interrogazione dei dati si punta dunque ad un sistema distribuito, il quale si basa sulle proprietà del Teorema del CAP, presentato nel 2000 da Eric Brewer in un KeyNote alla conferenza "*Principle of distributed Computing*".
Il teorema si basa sulle seguenti caratteristiche:

- ***Consistency*** **(Coerenza):** in un sistema con repliche, dopo una modifica tutti i nodi del sistema distribuito devono essere aggiornati prima di permettere accessi.

- ***Availability*** **(Disponibilità):** la garanzia che ogni richiesta riceva una risposta sia su ciò che sia riuscito o fallito; in altre parole, il sistema è sempre disponibile.

- ***Partition Tolerance*** **(Tolleranza al Partizionamento):** il sistema continua a funzionare nonostante arbitrarie perdite di messaggi che possono avvenire se le comunicazioni si interrompono tra due punti del sistema.

Il Teorema enuncia l'impossibilità di avere tutte e tre le caratteristiche nello stesso momento, perciò se ne possono implementare due a discapito della terza.
Pertanto se si sceglie di non avere una tolleranza al partizionamento avremo problemi sulla scalabilità verticale, proprietà spiegata in seguito, che è sicuramente più costosa. Se si rinuncia alla disponibilità dobbiamo attendere che vengano risolte alcune richieste a discapito delle prestazioni. Se si rinuncia alla coerenza si avrà per un periodo un disallineamento dei dati sui nodi. Questi ultimi due problemi sono quelli più noti e permettono di superare il problema della scalabilità. Per scalabilità orizzontale si intende la capacità di un'applicazione di crescere e decrescere in base alle necessità richieste dagli utenti, introducendo o togliendo dei nodi al sistema senza compromettere il suo funzionamento. In una qualche maniera si riesce dunque a parallelizzare il carico del lavoro. Grazie a questo approccio, avendo più nodi, l'errore in un nodo non pregiudica il funzionamento dell'intero database e quindi il sistema risulta più sicuro. Un altro vantaggio è sicuramente il costo, in quanto con un'applicazione scalabile si possono creare più nodi a basso costo. Gli unici svantaggi di questa scelta risiedono però nella progettazione dell'applicazione, che deve rispecchiare questa struttura scalabile e non comportare troppi problemi nell'installazione.

1.1 Classificazione dei database non relazionali

I database NoSQL possono essere implementati seguendo differenti approcci a seconda delle strutture dati con cui si rappresentano i record di dato.
Le principali categorie sono le seguenti quattro:

- ***Column-oriented database***: le informazioni sono memorizzate in **colonne.** Non c'è bisogno di definire subito le colonne. Tipicamente sono usati nell'ambito della memorizzazione distribuita dei dati.

- ***Key/Values store***: in questo caso i dati vengono immagazzinati in un elemento che contiene una chiave assieme ai dati veri e propri. É quindi del tutto analogo ad una *Hash Table*. Questo metodo è il più semplice da implementare, ma anche il più inefficiente se la maggior parte delle operazioni riguardano soltanto una parte di un elemento.

- ***Document store***: è l'evoluzione del metodo *key/value*, rispetto ai normali database relazionali invece che immagazzinare i dati in tabelle con dei campi fissi, questi vengono messi in un documento (rappresentato in XML, JSON o BSON) che può contenere illimitati campi di illimitata lunghezza, così se ad esempio di una persona conosciamo solo nome e cognome, ma magari di un'altra persona anche indirizzo, data di nascita

e codice fiscale, si evita che per il primo nominativo ci siano campi inutilizzati che occupano inutilmente spazio.

- ***Graph database***: i dati vengono immagazzinati sotto forma di strutture a grafi, rendendo più performante l'accesso a questi da applicativi orientati agli oggetti. Tipicamente si usa nei social network.

I principali database non relazionali sviluppatisi in questi anni sono riportati nella figura 1.1, specificandone la categoria.

Graph	Column	Document	Persistent Key/Value	Volatile Key/Value
neo4j	BigTable (Google)	MongoDB (~BigTable)	Dynamo (Amazon)	memcached
FlockDB (Twitter)	HBase (BigTable)	CouchDB	Voldemort (Dynamo)	Hazelcast
InfiniteGraph	Cassandra (Dynamo + BigTable)	Riak (Dynamo)	Redis	
	Hypertable (BigTable)		Membase (memcached)	
	SimpleDB (AmazonAWS)		Tokyo Cabinet	

Figura 1.1 : I principali database non relazionali suddivisi per categoria

1.2 Pro e Contro dei database non relazionali

Come in ogni nuovo modello nei database NoSQL si sono riscontrati sia dei vantaggi che dei svantaggi. Tra i vantaggi si notano sicuramente le performance più alte per i tempi di risposta, infatti nei database non relazionali un elemento contiene tutte le informazioni necessarie e dunque non serve usare i dispendiosi "join" come invece avviene per i database relazionali. Altro vantaggio deriva dalla semplicità di questi database: è proprio questa che permette di scalare in orizzontale in maniera così efficiente, permettendo di aggiungere nodi a caldo in maniera impercettibile dall'utente finale. Infine, scegliendo un database adatto alla mappatura più diretta alle classi d'oggetti del proprio applicativo si possono ridurre di molto i tempi dedicati allo sviluppo del metodo di scambio dati tra il database e l'applicativo stesso (il cosiddetto *object-relational mapping* che è invece necessario in presenza di database relazionali). La semplicità di questi database comporta però anche dei svantaggi: non essendoci dei controlli fondamentali sull'integrità dei dati, il compito ricade quindi totalmente sull'applicativo che dialoga col database.

Altra pecca è la mancanza di uno standard universale (come può essere l'SQL) ogni database ha infatti le proprie API e il suo metodo di storing e di accesso ai dati.
Nonostante ciò, i database non relazionali, seppur non essendo la soluzione definitiva al problema del salvataggio e recupero dei dati, visto l'avvento di Social Network e del cloud, risultano la miglior soluzione in questi contesti dove ogni giorno bisogna salvare un numero elevatissimo di dati e solo grazie alla scalabilità orizzontale dei database NoSQL e quindi all'aggiunta di server facili da gestire, è possibile ottenere delle performance soddisfacenti, molto migliori rispetto ai soliti database relazionali. Esempi di applicazioni che utilizzano questo modello sono infatti Twitter, Facebook e Amazon.

Capitolo 2

LE CARATTERISTICHE DI MONGODB

MongoDB (il cui nome deriva da “humongous”) è un database sviluppato in C++, open-source, document-oriented, scalabile e altamente performante. Esso è stato realizzato in maniera tale da avere alte prestazioni in lettura e scrittura. Le letture più consistenti infatti possono essere distribuite in più server replicati, le interrogazioni sono più semplici e veloci grazie all'assenza di join e l'approccio ai documenti rende possibile la rappresentazione di relazioni gerarchiche complesse attraverso documenti nidificati e array.
Ecco le sua caratteristiche distintive:

- *Database document-oriented:*

  i dati vengono archiviati sotto forma di documenti in stile JSON con schema dinamici, secondo una struttura molto semplice e potente;

- *Supporto completo agli indici:*

  indicizzazione di qualsiasi attributo;

- *Replicazione:*

  facilità nella replicazione dei dati attraverso la rete e alta scalabilità;

- *Sharding:*

  scalabilità orizzontale senza compromettere nessuna funzionalità;

- *Query document-based*

2.1 Database document-oriented

Questo modello di database eredita il meccanismo di storage dal paradigma doc-oriented che consiste nel memorizzare ogni record come documento che possiede caratteristiche predeterminate. Si può aggiungere un numero qualsiasi di campi con una qualsiasi lunghezza.
Nei doc-oriented si segue una metodologia differente rispetto al modello relazionale: si accorpano quanto più possibile gli oggetti, creando delle macro entità dal massimo contenuto informativo. Questi oggetti incorporano tutte le notizie di cui necessitano per una determinata semantica.

Pertanto MongoDB non possiede uno schema e ogni documento non è strutturato, ha solo una chiave obbligatoria: _id , la quale serve per identificare univocamente il documento; essa è comparabile, semanticamente, alla chiave primaria dei database relazionali.

2.2 Supporto completo agli indici

MongoDB utilizza le tecniche di indicizzazione; solitamente il campo _id è indicizzato automaticamente, inoltre conviene indicizzare anche quei campi dove è tipico eseguire ricerche o sui quali sono definiti ordinamenti. MongoDB, proprio per favorire l'indicizzazione, fornisce strumenti in grado di suggerire su quali campi sia opportuno definire indici.
Per usarli con buoni risultati conviene privilegiare applicazioni read-intensitive, ovvero collezioni con un alto rapporto letture/scritture per incrementare le letture rispetto alle scritture.
Nonostante siano molto utili vanno utilizzati solo se il campo è realmente selettivo in quanto indicizzare dei campi comporta sicuramente dei costi. Il tempo di scrittura infatti aumenta con l'aumentare degli indici utilizzati, questo perché una modifica o un'aggiunta di dati può comportare la modifica di molti indici.
Dunque affinché l'uso degli indici risulti veramente performante è importare fare una scelta ragionata considerando il tipo di interrogazioni che verranno effettuate sul sistema e quanto spesso quei dati verranno modificato o subiranno delle aggiunte.
In poche parole un indice è una struttura dati che cataloga le informazioni dei campi specificati nei documenti di una collezione. Attraverso questi indici risulta più facile e meno dispendioso ordinare i dati di una query e grazie ad essi si velocizzano i tempi di risposta in quanto per certe ricerche mirate non vengono più scannerizzati tutti i documenti ma si salta direttamente ai documenti che rispettano la clausola della query.
La maggior parte degli indici seguono la struttura ad albero, B-Tree, grazie alla quale con un semplice confronto del nodo padre si sceglie se considerare il braccio destro e sinistro scartando ogni volta un numero consistente di confronti.
Una volta definito un indice su uno o più campi sarà il sistema stesso a mantenere aggiornato l'indice.
E' possibile definire diverse tipologie di chiavi:

- **Chiavi semplici** : indicizzano un solo campo di una collezione (insieme di documenti con le stesse caratteristiche o simili, una sorta di tabella dei database relazionali);

- **Chiavi composte**: indicizzano due o più campi di una collezione;

- **Chiavi documento**: considerano come campo dell'indice un campo che contiene oggetti (utili per query che non richiedono interrogazioni ordinate);

- **Chiavi array**: considerano come campo dell'indice un campo che contiene un array

E’ possibile inoltre definire un indice sparso su un determinato campo, così facendo nell’indice saranno compresi solo gli oggetti che presentano il campo indicato, mentre quelli che non lo presentano verranno filtrati.
Caratteristica peculiare di MongoDB è la possibilità di utilizzare indici geospaziali bidimensionali i quali ci permettono di interrogare il sistema con query basate sulla posizione.

2.3 Replicazione

La replicazione è utilizzata per rimediare e prevenire malfunzionamenti. Essa infatti comporta ad una ridondanza dei documenti, la quale ritorna molto utile in caso di perdita dei dati. In poche parole consiste nel salvare più copie dei documenti in server diversi in modo tale di avere dei dati sicuri anche se accorrono degli errori in uno o più server.
Essenzialmente ci sono due tipi di replicazione: la Master-slave semplice e la Replica-Set.La prima, come suggerito dal nome, presenta un’unità centrale master che è sempre aggiornata, la quale periodicamente sincronizza tutti gli altri nodi (slave), che risultano quindi dipendenti ad essa. In questo modo in ambito distribuito si assicura una consistenza assoluta sul master ed una consistenza relativa sullo slave, nel senso che prima o poi lo slave risulterà sincronizzato con il master. In caso di malfunzionamento di un nodo, il ripristino deve essere eseguito manualmente.
La tecnica Replica-Set o insieme di replicazione è una elaborazione del modello Master-Slave che aggiunge il ripristino automatico dopo un malfunzionamento ed il supporto di cluster di server con membri primari (master) e secondari (slave).

2.4 Sharding

Lo sharding è la tecnica usata per partizionare orizzontalmente un database. Mentre nel partizionamento verticale, chiamato normalizzazione, si punta a separare ed isolare semantiche omogenee splittando una tabella in più tabelle dividendo i campi della tabella stessa, nel partizionamento orizzontale lo split è realizzato in modo da ottenere due o più insiemi di tuple da sistemare in due o più tabelle.
In MongoDB le tecniche di Sharding estendono proprio il partizionamento orizzontale permettendo agli shard, cioè alle partizioni ottenute dal database di partenza, di essere slegati completamente gli uni dagli altri. Questa separazione è possibile in quanto ogni shard può vivere in una istanza dello schema logico totalmente separato e può essere ospitato da un server fisico diverso da quello degli altri shard. Pure il data center può essere uno qualsiasi, addirittura trovarsi in un altro continente. In poche parole ogni shard non dipende dal sistema logico e fisico sottostante.
Questa tecnica comporta ovviamente dei vantaggi: essendo ridotto il numero di righe di ogni tabella coinvolta e dunque pure le dimensioni degli indici risultano ridotte, le ricerche sono più veloci e più efficienti. Si ottengono performance superiori grazie anche al

parallelismo reso possibile dalla possibilità di posizionare ogni shard su hardware differenti. Inoltre la segmentazione dei dati può seguire in maniera più naturale gli aspetti del mondo reale che possono coinvolgere spazi completamente separati.
MongoDB prevede un servizio automatico di sharding, grazie al quale una volta impostata la chiave sulla quale eseguire il partizionamento orizzontale, il sistema segmenta automaticamente la collezione. Le collezioni che vengono così a crearsi sono strutturalmente identiche: le dimensioni in termini di oggetti e occupazione di spazio sono bilanciate. Si generano delle shard key ovvero delle chiavi di frammentazione, partendo dai chunk creati dai cluster che contengono i dati.
Tutti i metadati si memorizzano in un server di configurazione e si creano dei nodi router che propagano le richieste ai nodi. Quando ci sarà una richiesta si interroga un nodo che possiede una tabella con una mappa del sistema per sapere a quali shard rivolgersi per ottenere gli oggetti richiesti.

2.5 Query document-based

Una delle più importanti caratteristiche di MongoDB è la sua capacità di supportare query dinamiche ad hoc, le quali, essendo caratteristiche dei database relazionali, ci permettono con facilità di migrare dati da un database relazionale verso un database di MongoDB trasformando in pochi passaggi query SQL in query del linguaggio *document-based*.
Per recuperare i dati in MongoDB esistono varie *query-object* che rispecchiano i costrutti principali delle query SQL.
Tra i principali ricordiamo:

- **il selettore di campo**, corrispondente allo SELECT in SQL, il quale recupera un sottoinsieme dei campi della nostra collezione. (In MongoDB, differentemente a quando accade in SQL, il campo _id è sempre ritornato.)

- **la selezione**, corrispondente alla WHERE in SQL, la quale indica attraverso dei criteri quali documenti recuperare in una collezione. Come in SQL vengono utilizzati operatori condizionali (and, or, nor, in, nin, all, exists, mod, size, type, ne), espressioni regolari, valori in array, valore negli oggetti embedded, meta operatori (not) e l'operatore di aggregazione (group). Il processo di selezione potrebbe causare un overhead per la ricerca di documenti nel database, per questo viene utilizzata la funzione ”Map-Reduce”: prima di tutto per ogni elemento viene invocata la funzione “Map”, che produce delle coppie chiave-valore da passare alla successiva funzione e in seguito viene eseguita la funzione “Reduce”, che aggrega i risultati ricevuti e restituisce la nuova collezione.

- **gli ordinamenti**, i quali, in maniera analoga alla clausola ORDER BY in SQL, specificando su quale campo eseguire l'ordinamento, ritornano risultati ordinati in maniera crescente o decrescente.

- **skip** e **limit**, i quali permettono una paginazione dei risultati.

E' importante sottolineare come i risultati di una query vengano gestiti: MongoDB usa tecniche di cursori, i quali sono usati per recuperare iterativamente tutti i documenti ritornati dalla query eseguita.

2.6 La nuova filosofia di MongoDB

Come possiamo ben capire dalle sue caratteristiche, MongoDB, essendo un database NoSQL , non si preoccupa tanto di soddisfare le proprietà ACID tipiche dei database relazionali, piuttosto si concentra su aspetti più performanti per quanto riguarda la velocità, la flessibilità e la facilità d'uso del database.
Esso, organizzando i dati in documenti *schemaless*, riesce a rispondere alle interrogazioni con tempi molto inferiori rispetto ai database relazionali dove i dati sono separati in tabelle multiple e necessitano quindi di join per raggruppare i risultati.
Senz'altro anche l'approccio alla scalabilità orizzontale attraverso lo sharding è un altro punto a favore per la velocità e le prestazioni del database. Lo sharding consente infatti di gestire più carico di lavoro senza richiedere l'utilizzo di macchine potenti più grandi e quindi più costose e riesce ad aumentare la capacità senza tempi di inattività, sfaccettatura molto importante nel web che, a causa dell'aumento di carico dei dati, può essere costretto a disattivare il servizio per lunghi periodi di manutenzione.
Per quanto riguarda la flessibilità, essa è data dal tipo di formato utilizzato nei documenti: JSON. Formato che si adatta perfettamente ai vari linguaggi di programmazione e non avendo uno schema fisso permette di modificare e ampliare il modello di dati in maniera molto semplice e veloce.
Infine MongoDB punta alla semplicità: è facile da installare, configurare, mantenere e usare. A tal fine, fornisce poche opzioni di configurazione e cerca automaticamente, quando possibile, di fare la “cosa giusta” permettendo agli sviluppatori di concentrarsi sulla creazione dell’applicazione piuttosto che perdere tempo in oscure configurazioni di sistema.

Capitolo 3

CONCETTI BASE PER COSTRUIRE DATABASE CON MONGODB

3.1 I documenti e le collezioni.

Un tipico documento di MongoDB ha una struttura JSON del tipo:

```
{
    _id: ObjectId("5099803df3f4948bd2f98391"),
    name: { first: "Alan", last: "Turing" },
    birth: new Date('Jun 23, 1912'),
    death: new Date('Jun 07, 1954'),
    contribs: [ "Turing machine", "Turing test", "Turingery" ],
    views : NumberLong(1250000)
}
```

Come si può osservare il documento è formato da un insieme di campi ognuno dei quali è una coppia chiave-valore *(key-value)*. Ogni chiave dei documenti è una stringa e ad essa sono associati dei valori; i valori possono essere diversi tipi di dato tra cui anche oggetti complessi come array, documenti o addirittura array di documenti.

MongoDB è type- e case-sensitive e le coppie *key-value* sono ordinate: { “x” : 1 , “y” : 2 } è diverso da {“y” : 2 , “x” : 1}.

Il campo _id rappresenta un id interno utile per l’identificazione univoca del singolo documento (una specie di chiave primaria) e in generale non va inserito a mano, ma verrà generato dal database stesso.

In poche parole un documento rappresenta quello che nei database relazionali viene chiamata riga o tupla e le varie coppie key-value sono le proprietà o colonne.

La differenza sostanziale con i database relazionali sta nel fatto che i dati sono rappresentati in modo aggregato e ciò permette di lavorare con un oggetto nella sua interezza, dimenticando i join, strumenti di aggregazione molto usati nei database relazionali e molto costosi.

La collezione è un gruppo di documenti e dunque gioca un ruolo simile a quello a cui siamo abituati per le tabelle su un database relazionale.

L'approccio schemaless (mancanza di uno schema preciso) si MongoDB fa si che la struttura risulti più flessibile: i campi non sono definiti in precedenza con rigidità come succede nelle tabelle dove ogni riga deve soddisfare tutti i campi delle colonne e i tipi di da-

ti; qui con semplicità è possibile aggiungere dinamicamente alcuni attributi di un documento senza alterare tutto lo schema della collezione.
Per assurdo, non essendoci regole di schema e potendo avere ogni documento la propria struttura, esso potrebbe essere inserito in qualsiasi collezione e quindi ci si potrebbe chiedere quale fosse la necessità delle collezioni. Pensando a database con un carico notevole di dati si capisce facilmente che la suddivisione in collezioni dei documenti facilita la ricerca dei documenti. Le collezioni sono identificate dal loro nome.

3.2 La shell di MongoDB

MongoDB si basa su una shell JavaScript che permette l'iterazione con il database direttamente dalla linea di comando: eseguendo *mongod* il server del database viene attivato. Se non viene specificato nessun comando, viene selezionato un database di default chiamato test. Per poter avere la lista dei database presenti il comando da utilizzare é *show dbs*; poi grazie al comando *use* è possibile selezionare il database sul quale lavorare.
Un comando molto utile per vedere i comandi attraverso i quali la shell di MongoDB può interrogare i database e il sistema è *db.help().*

3.3 Creazione ed eliminazione di un database

Per creare un database nuovo si utilizza sempre la clausola *use* seguita dal nome del database e poi si può proseguire con la creazione della prima collezione.
Per eliminare un database in MondoDB è necessario selezionare il database che si vuole creare tramite il comando *use* per poi utilizzare il comando *db.dropDatabase().*

3.4 Creazione, modifica ed eliminazione di un utente

Per creare un utente in un database è necessario selezionare il database per poi utilizzare il comando *db.addUser(nome, password, [readOnly=false])* . Il campo readOnly se settato su true specifica che l'utente avrà solo permessi di lettura sul database.
Per associare un utente a tutti i database (utente globale) è sufficiente aggiungere tale utente al database admin .
Per modificare un utente è sufficiente utilizzare nuovamente il comando *db.addUser(nome, password, [readOnly=false])* .
Per eliminare un utente è possibile utilizzare il comando *db.removeUser(nome)*.

3.5 Creazione ed eliminazione di una collezione

Come per i database è disponibile un helper specifico anche per le collezioni è possibile utilizzare il comando *db.nome collection.help()* per avere informazioni sui comandi da poter utilizzare sulle collezioni.
Per creare una nuova collezione in un database è sufficiente selezionare il database di interesse ed invocare il metodo *db.createCollection(nome collezione).*
L'eliminazione di una collezione comporta l'eliminazione di tutti gli oggetti interni alla collezione stessa e si effettua tramite il comando *drop().*

3.6 Creazione, lettura, modifica ed eliminazione di un oggetto

Per **inserire** un oggetto in una collezione si utilizza il metodo *insert*.
Si può inserire direttamente l'oggetto, creandolo in maniera implicita direttamente all'interno della collezione:
> db.Calciatori.insert ({ nome : ” Pippo ” , cognome : ” Inzaghi ” , ruolo : ”attaccante ” })
In questo caso l'oggetto non ha una vita propria ma la sua esistenza è legata alla collezione in cui lo si inserisce.
Oppure si crea prima in modo esplicito l'oggetto:
> calciatore = {nome : ” Pippo ” , cognome : ” Inzaghi ” , ruolo : ” attaccante ”}
e poi per renderlo persistente nel database lo si inserisce nella collezione interessata.
> db.Calciatori.insert(calciatore)
In questa maniera l'oggetto appena creato sarà visibile a tutti solo all'atto dell'inserimento nella collezione.
Nell'operazione di inserimento, se non è già presente nel documento, viene inserita in automatico una chiave *_id,* che identifica il documento univocamente e globalmente. Solitamente di default la chiave _id è di tipo ObjectId ma può assumere qualsiasi altro tipo, purché sia supportato da MongoDB.
Per quanto riguarda l'esempio fatto sopra all'inserimento dell'oggetto calciatore il risultato dell'inserimento risulta essere qualcosa del tipo:
{ ”_id” : ObjectId (”4d148829610f000000001b33”) , nome : ” Pippo ” , cognome ” Inzaghi ” , ruolo : ”attaccante ”}
La dimensione massima di un documento è pari a 16MByte.
Se si devono inserire più documenti in una collezione, si può rendere più veloce l'inserimento utilizzando il *batchInsert,* metodo simile all'insert ma che permette si inserire più documenti contemporaneamente scrivendoli sotto forma di array:
> db.foo.batchInsert([{"_id" : 0}, {"_id" : 1}, {"_id" : 2}])
>db.foo.find()
{"_id" : 0 }
{"_id" : 1 }
{"_id" : 2 }

Per **leggere** i documenti inseriti all'interno di una collezione si utilizza il metodo *find* o *findOne*; il primo ritorna tutti i documenti all'interno di una collezione mentre il secondo ne ritorna solamente uno, scelto a caso.
Per vedere tutti i documenti della collezione Calciatori si esegue ad esempio:
db.Calciatori.find()
Il metodo find in SQL corrisponde alla query “SELECT * FROM Calciatori;”.

Per **modificare** un oggetto di una collezione si può usare il metodo *update,* il quale possiede almeno 2 parametri: il primo esprime il criterio per trovare il documento da aggiornare, il secondo rappresenta la modifica e quindi il nuovo documento.
Nel caso in cui voglio aggiungere il campo prezzo al documento calciatore creato esplicitamente, essendo esso esistente a livello di sessione, posso modificarlo tramite la ridefinizione di tutti gli attributi dell'oggetto; ad esempio, se l'oggetto è il seguente:
> calciatore
{ ”nome” : ”Pippo” ,
” cognome ” : ” Inzaghi ” ,
“ruolo” : “attaccante”,
” _i d ” : ObjectId (”4d148842610f000000001b33”) }
per modificarlo lo “sovrascrivo”:
> calciatore = {nome : ”Pippo” , cognome : ” Inzaghi” , ruolo : “attaccante”, prezzo : ” 10000 ” }
Nel caso in cui l'oggetto sia stato inserito implicitamente posso modificare usando direttamente il comando:
>db.Calciatori.update({cognome: “Inzaghi”}, {“$set” : {“prezzo” : “10000”}})
Ora se cerco il nuovo documento vedrò la chiave aggiunta:
>db.Calciatori.find()
{ ” _id ” : ObjectId (”4d148829610f000000001b33”) ,
nome : ” Pippo ” ,
cognome: ” Inzaghi ” ,
ruolo : ”attaccante ”,
prezzo : “10000”}

In MongoDB si prevedono varie opzioni di modifica per il comando update, i principali operatori sono:

- **$inc**: utilizzato per incrementare o decrementare un valore numerico di un particolare valore

- **$set**: utilizzato per impostare il valore di un campo; se il campo non è presente viene aggiunto, altrimenti viene sovrascritto con il nuovo valore. Attraverso il comando set si può anche trasformare il tipo di dato, creando ad esempio un array.

- **$unset**: utilizzato per rimuovere la chiave del campo fornito

- **$rename**: rinomina il campo con un nuovo nome

- **$push**: aggiunge un valore alla fine di un array se quest'ultimo esiste, altrimenti crea l'array con il primo valore.

Si possono aggiungere più valori in una operazione utilizzando il comando $each e si può decidere la lunghezza massima di un array attraverso l'operatore $slice e riordinare i valori dell'array attraverso il comando $sort.
Ad esempio con il comando
>db.movies.update({“genre” : “horror”}, {“$push” :{ “top 10” : { “$each” : [{“name” : “Nightmare on Elm Street”, “rating”: 6.6}, {“name” : “Saw”, “rating”: 8}], “$slice” : -10 , “$sort” : {“rating”: -1}}}})
si ordinano gli elementi dell'array a seconda del valore del campo “rating” e si tengono i primi 10.

- **$pushAll**: aggiunge più valori ad un array se quest'ultimo esiste, altrimenti crea l'array con i valori

- **$pull**: rimuove l'elemento da un array specificato dal valore dell'elemento da rimuovere

- **$pullAll**: rimuove una lista di elementi di un array specificati

- **$pop**: rimuove l'ultimo elemento di un array

- **addToSet**: aggiunge il valore specificato all'array solo se questo non è già presente. Questa funzione può essere sostituita anche dal comando push combinato con la clausola $ne: >db.movies.update({“genre” : {“$ne” : “horror”}}, { push : {“genre” : “horror”}})

L'operatore $ viene utilizzato sempre nei comandi di modifica e aggiornamento delle coppie chiave-valore.

Un altro tipo speciale di aggiornamento è l'***upsert***. Attraverso questo metodo se il documento su cui deve avvenire la modifica non viene trovato, ne viene creato in automatico uno nuovo con le proprietà descritte nella modifica, altrimenti se il documento è presente si prosegue normalmente con la modifica. Per creare un *upsert* si aggiunge un terzo parametro posto a true al metodo *update*:
>db.Calciatori.update({“cognome”: “Materazzi”}, {“$set” : {“prezzo” : “1000”}}, true)
Se non viene trovato nessun documento che soddisfi il criterio di ricerca viene creato il seguente documento:
{ ” _id ” : ObjectId (”4d148829610f000000001b34”) ,
“cognome” : ”Materazzi” ,
“prezzo” : “1000”}
L'operatore “$setOnInsert” permette di inserire la modifica solo quando il documento è stato creato; ad esempio scrivendo:
>db.users.update({} , { “$setOnInsert” : {“createdAt” : new Date()}}, true)

>db.users.findOne()
{ “_id” : ObjectId (”4d148829610f000000001c37 ”) ,
“createdAt” : ISODate(“2013-02-25T16:01:50.742Z”)}
Se poi richiamo l'update il documento non subirà modifiche ma sarà lo stesso di prima.

Gli *updates,* di default, aggiornano solamente il primo documento trovato che soddisfa i criteri di ricerca; perciò se ci sono altri documenti che rispecchiano la ricerca rimangono invariati. Affinché vengano modificati tutti i documenti bisogna aggiungere un quarto parametro posto a true al metodo *update*.
Per vedere quanti documenti sono stati aggiornati si può lanciare il comando *getLastError*: *>db.runCommand({getLastError : 1 }).*
Con questo comando però non si vedono i documenti modificati; per ottenere ciò si utilizza il metodo ***findAndModify*** *,* il quale ritorna gli oggetti e li modifica in un solo passo.
La struttura generale è del tipo:
db.runCommand ({ findAndModify : <collection>, <options>})
dove le opzioni possono essere le seguenti:

- **query** : esprime attraverso un “query document” i criteri con i quali cercare il documento da modificare.
- **sort**: esprime il valore su cui riordinare i vari documenti (facoltativo).
- update: l'aggiornamento da eseguire sui documenti trovati (deve esserci questo o remove).
- **remove**: valore booleano per specificare se il documento deve essere eliminato (deve esserci questo o update).
- **new**: valore booleano, se a true ritorna il documento modificato, se a false ritorna il documento prima della modifica (caso di default).
- **fields**: i campi del documento da ritornare (facoltativo).
- **upsert**: valore booleano che specifica se deve essere un upsert oppure no; di default non lo è.

Per quanto riguarda la **cancellazione** nel caso in cui un oggetto sia definito all’interno di una sessione per essere usato più volte, si può ottenere la cancellazione dell’oggetto in due modi: direttamente, cioè si cancella l’oggetto usando direttamente la delete *>delete calciatori;* indirettamente,cioè l’oggetto viene eliminato automaticamente alla chiusura della sessione.
Nel caso in cui si deve cancellare un oggetto inserito in una collezione, si procede usando la remove(query). Il metodo in base alla query passata, effettuerà una scansione della collezione, eliminando tutti gli oggetti che rispettino le condizioni della query.
Per rimuovere tutti i documenti di una collezione basta re *>db.nomecollezione.remove().*

Esso è differente rispetto al comando drop(), infatti mentre il comando drop() elimina la collezione con il comando remove() si eliminano tutti i documenti della collezione ma la collezione in sé non è eliminata: le meta-informazioni esistono ancora.
Il comando drop() risulta comunque molto più performante in termini di tempo.

3.7 I tipi di dato

Come abbiamo già visto i documenti di MongoDB possono essere pensati come dei "JSON-like" in quanto sono concettualmente simili agli oggetti in JavaScript. JSON (JavaScript Object Notation) in poche parole è un semplice formato per lo scambio di dati. Per le persone è facile da leggere e scrivere e per le macchine risulta facile da generare e analizzarne la sintassi. Esso supporta solo sei tipi di dato (null, boolean, numeric, string, array e object) ma MongoDB lo ha ampliato con altri tipi mantenendo sempre la tipica struttuta chiave-valore.
I tipi più comuni sono:

- **null**: rappresenta un valore nullo o un campo non esistente

  {"x" : null}

- **boolean**: tipo booleano con valori true o false

  {"x" : true}

- **number**: valori numerici; di default si usano numeri a 64bit con virgola mobile.

  {"x" : 3.14} or {"x" : 3}

- **string**: rappresenta stringhe dell'alfabeto UTF-8

  {"x" : "prova"}

- **date**: per rappresentare la data e l'ora

  {"x" : new Date()}

- **regular expression**: espressioni regolari che utilizzano le espressioni di JavaScript utilizzate nelle query

  {"x" : /foobar/i}

- **array**: insiemi o liste di valori rappresentate da array

  {"x" : ["a", "b", "c"]}

- **embedded document**: i documenti di MongoDB possono avere come

valori dei documenti veri e propri che vengono chiamati documenti nidificati

{"x" : {"foo" : "bar"}}

- **object id**: si tratta di un ID di 12-byte per documenti

  {"x" : ObjectId()}

- **binary data**: si tratta di una stringa di arbitrari byte, non si manipola dalla shell ed è l'unica soluzione per salvare stringhe non-UTF-8 nel database

- **code**: possono essere presenti codici JavaScript

  {"x" : function() { /* ... */ }}

3.8 Queries

Per interrogare il nostro database in MongoDB si utilizza il metodo *find*. Come in tutti i database un'interrogazione, o meglio una query, ritorna un sottoinsieme della collezione con la proprietà specificata nel primo parametro del metodo find.
Se i documenti richiesti devono rispettare più di una condizione (AND condition) basta elencarle nel primo parametro separandole da virgole; ad esempio se voglio trovare tutti gli users di 27 anni e di nome Joe scrivo:
>db.users.find({“username” : “Joe”, “age”: 27})
Se desidero invece visualizzare solo alcuni campi dei vari documenti di una collezione utilizzo il secondo parametro del metodo find; in questo parametro pongo ad 1 i campi che voglio visualizzare, ad esempio il seguente comando mi fa visualizzare solo l'email e il nome dell'utente:

```
> db.users.find({}, {"username" : 1, "email" : 1})
{
"_id" : ObjectId("4ba0f0dfd22aa494fd523620"),
"username" : "joe",
"email" : "joe@example.com"
}
```

Come si nota di default viene sempre ritornato l'_id. È comunque possibile specificare anche i campi che non si vogliono visualizzare ponendo il parametro di tali campi a 0, e così è possibile non visualizzare l'_id.

3.8.1 Operatori di confronto

Per confrontare certi campi è possibile utilizzare gli operatori "$lt", "$lte", "$gt" e "$gte" che corrispondono rispettivamente a <, <=, >, >=.
Ad esempio per cercare gli utenti con età compresa tra i 18 e 30 anni si scrive:

db.users.find({"age" : {"$gte" : 18, "$lte" : 30}})
Per verificare che un campo non sia uguale ad un certo valore si utilizza l'operatore “$ne” che sta per not equal.

3.8.2 L'operarore $and

Per cercare dei documenti che soddisfano contemporaneamente più proprietà si usa l'operatore $and in questa maniera:
> db.users.find({"$and" : [{“username” : “Joe”}, {“age”: 27}]})
Ma come abbiamo visto sopra si può omettere.

3.8.3 Operatori OR: $in e $or

In MongoDB ci sono due metodi per porre delle OR queries: “$in” e “$or”.
“$in” è utilizzato per cercare i documenti che soddisfano l'interrogazione se hanno uno tra i valori indicati di una singola chiave; ad esempio se voglio cercare gli utenti che si chiamano o Joe o Mike o Eros scrivo:
> db.users.find({"username" : {"$in" : [“Joe”, “Mike”, “Eros”]})
Esiste anche l'operatore contrario a “$in”, “$nin” che ritorna tutti i documenti che non rispecchiano i valori elencati nell'array.
L'operatore “$on” è invece più generale in quanto controlla che i documenti soddisfino almeno una delle condizioni che possono riguardare anche più chiavi; ad esempio se voglio i documenti degli utenti che si chiamano Joe oppure che hanno 27 anni scrivo:
> db.users.find({"$or" : [{"username" : “Joe”}, {"age" : 27}]})
Ovviamente si possono usare anche combinati.

3.8.4 L'operatore $not e l'operatore $ne

L'operatore $not nega il risultato di un qualsiasi altro operatore di MongoDB o di una regolare interrogazione.
Mentre l'operatore $ne viene usato per negare uno specifico valore.

3.8.5 L'operatore $exists

Questo operatore viene utilizzato per vedere se un documento contiene un particolare valore; l'esempio più significativo è con il valore null:
considero questi documenti
{"_id" : ObjectId("4ba0f0dfd22aa494fd523621"), "y" : null }
{"_id" : ObjectId("4ba0f0dfd22aa494fd523622"), "y" : 1 }
{"_id" : ObjectId("4ba0f148d22aa494fd523623"), "y" : 2 }
Facendo la ricerca:
>db.c.find({"z" : null})
ottengo il seguente risultato:

{"_id" : ObjectId("4ba0f0dfd22aa494fd523621"), "y" : null }
{"_id" : ObjectId("4ba0f0dfd22aa494fd523622"), "y" : 1 }
{"_id" : ObjectId("4ba0f148d22aa494fd523623"), "y" : 2 }
questo perché il valore null assume anche il valore di “non esiste”, dunque la query precedente ritorna tutti i documenti che non possiedono la chiave z.
Per ottenere effettivamente solo i documenti che hanno la chiave z con valore pari a null devo eseguire questa query:
> db.c.find({"z" : {"$in" : [null], "$exists" : true}})

3.8.6 Operatori per array: $all, $size, $slice

Se voglio trovare array che contengono più di un elemento uso l'operatore $all; ad esempio per cercare array che contengono sia l'elemento “apple” che “banana” scrivo:
> db.food.find({fruit : {$all : ["apple", "banana"]}})
Se voglio invece trovare array di una determinata lunghezza uso l'operatore $size:
> db.food.find({"fruit" : {"$size" : 3}})
Con l'operatore slice si sceglie invece il sottoinsieme di elementi che si vuole vedere;
per vedere i primi 10 commenti di un post ad esempio:
> db.blog.posts.findOne(criteria, {"comments" : {"$slice" : 10}})
gli ultimi 10:
> db.blog.posts.findOne(criteria, {"comments" : {"$slice" : -10}})
ma anche intervalli:
> db.blog.posts.findOne(criteria, {"comments" : {"$slice" : [23, 10]}})

3.8.7 Interrogazioni sui documenti nidificati

Funzionano essenzialmente allo stesso modo delle query normali; ad esempio se considero questo documento:
{ "name" : {
"first" : "Joe",
"last" : "Schmoe"},
"age" : 45}
per cercare qualcuno che si chiama Joe Schmoe posso scrivere:
> db.people.find({"name" : {"first" : "Joe", "last" : "Schmoe"}})
e mi ritroverà il documento precedente;
affinché la query risulti riutilizzabile nel caso avvengano delle modifiche nel documento, come ad esempio l'aggiunta di una chiave in name, posso usare questa query:
> db.people.find({"name.first" : "Joe", "name.last" : "Schmoe"})

3.8.8 Gli operatori limit, skip e sort

L'operatore limit permette di limitare il numero di risultati specificando il numero massimo:
> db.c.find().limit(3) .
L'operatore skip lavora in un certo senso nella maniera opposta a limit; esso scarta un numero di risultati (che viene indicato come parametro) e ritorna tutti gli altri risultati:
> db.c.find().skip(3) .
Infine l'operatore sort riordina i valori delle chiavi indicate in maniera crescente (1) o in maniera decrescente (-1) :
> db.c.find().sort({username : 1, age : -1}) .
Ovviamente questi operatori si possono combinare; ad esempio se voglio cercare in uno store online gli mp3 e sono interessato a vedere i primi 50 pezzi a prezzo più alto scrivo:
> db.stock.find({"desc" : "mp3"}).limit(50).sort({"price" : -1}) .

3.9 Funzioni di aggregazione

Attraverso le funzioni di aggregazione MongoDB ci permette di creare delle pipeline che contengono delle parti di documenti raggruppati o organizzati in modi particolati.
I principali operatori per creare pipeline sono:

- **$match**: filtra i documenti affinché abbiano un particolare valore desiderato. Se voglio gli utenti che vengono dall'Oregon: *>db.users.aggregate({$match : {"state" : "OR"}})*.

- **$project**: permette di estrarre alcuni campi dai documenti secondari e volendo è possibile anche rinominarli. Con la seguente operazione si ritorna solo il campo "author": *> db.articles.aggregate({"$project" : {"author" : 1, "_id" : 0}})* . Per rinominare il campo "_id" con "userId": *> db.users.aggregate({"$project" : {"userId" : "$_id", "_id" : 0}})* .

Assieme all'operatore $project si possono utilizzare anche molte altre operazioni:

Espressioni aritmetiche:

- "$add" : [expr1[, expr2, ..., exprN]]

  Prende una o più espressioni e le somma.

- "$subtract" : [expr1, expr2]

  Prende due espressioni e sottrae la seconda alla prima.

- "$multiply" : [expr1[, expr2, ..., exprN]]

  Prende una o più espressioni e le moltiplica.

- "$divide" : [expr1, expr2]

  Prende due espressioni e divide la seconda per la prima.

- "$mod" : [expr1, expr2]

  Prende due espressioni e ritorna il resto della divisione della prima per la seconda.

Espressioni riguardanti date:

"$year", "$month", "$week","$dayOfMonth", "$dayOfWeek", "$dayOfYear", "$hour", "$minute", and "$second".

Espressioni su stringhe:

- "$substr" : [expr, startOffset, numToReturn]

  Ritorna la sottostringa della prima espressione partendo dallo startOffset fino al numToReturn compreso.

- "$concat" : [expr1[, expr2, ..., exprN]]

  Concatena ogni espressione data.

- "$toLower" : expr

  Ritorna l'espressione che deve essere una stringa in minuscolo.

- "$toUpper" : expr

  Ritorna l'espressione che deve essere una stringa in maiuscolo.

Espressioni logiche:

- "$cmp" : [expr1, expr2]

  Compara le due stringhe e torna 0 se sono identiche, un numero negativo se expr1 è minore all'expr2 e un numero positivo viceversa.

- "$eq"/"$ne"/"$gt"/"$gte"/"$lt"/"$lte" : [expr1, expr2]

  Valuta il confronto tra la prima e la seconda espressione e ritorna true se è corretto, false altrimenti.

- **$group**: permette di raggruppare i documenti a seconda dei campi indicati.

Anche insieme a questo operatore si possono utilizzare degli altri operatori:

Espressioni aritmetiche:

- "$sum" : value

  Si aggiunge il valore value ad ogni documento e lo si ritorna aggiornato.

- "$avg" : value

  Ritorna la media di tutti i valori trovati nel raggruppamento (group).

Operatori di limite:

- "$max" : expr

  Ritorna il valore massimo tra tutti.

- "$min" : expr

  Ritorna il valore minimo tra tutti.

- "$first" : expr

  Ritorna il primo valore presente nel raggruppamento; esso è utile quando i valori sono ordinati.

- "$last" : expr

  Al contrario di "$first" ritorna l'ultimo valore.

- **$sort**: aggrega i documenti ordinandoli secondo un campo in maniera crescente o decrescente.

- **$limit**: specifica il numero massimo di elementi che si vogliono considerare.

- **$skip**: specifica i primi n elementi da scartare e ritorna tutti gli altri.

Oltre a questi operatori, per le funzioni di aggregazione, esistono comunque dei comandi che si possono utilizzare direttamente nelle collezioni senza creare un "aggregation framework" come si è visto prima.
Questi comandi sono:

- **count**: ritorna il numero di documenti di una collezione; *> db.users.count()*

- **distinct**: ritorna una lista di valori distinti per una particolare chiave specificata: *>db.users.distinct("age")*

- **group**: permette di raggruppare la collezione secondo la chiave scelta. Per far funzionare bene questo comando deve essere del tipo:

```
db.runCommand({"group" : {
"ns" : "stocks",
"key" : "day",
 "initial" : {"time" : 0},
```

"$reduce" : function(doc, prev) { ... } })
dove "ns" determina quale collezione deve essere raggruppata, “key” specifica la chiave secondo cui i documenti della collezione devono essere raggruppati, “initial” è un documento che verrà utilizzato come documento base per i risultati dell'aggregazione, “reduce” è la funzione JavaScript che esegue l'aggregazione.

MapReduce

MapReduce è un algoritmo di aggregazione molto potente per la condensazione di grandi volumi di dati.
Esso può essere riassunto in due passi; nella prima parte, detta appunto Map, MongoDB considera tutti i documenti interessati all’aggregazione(quelli che rispecchiano la *query condition*) e applica loro la funzione Map scritta opportunamente. Questa fase deve invocare il metodo *emit()* per selezionare le chiavi e i valori da aggregare. Solitamente la Map viene scritta utilizzando funzioni JavaScript.
Nella seconda fase, Reduce, vengono condensati e aggregati in una nuova maniera i dati seguendo le informazioni e le chiavi date dalla fase precedente ed è possibile ottenere i risultati sotto forma di una vera e propria nuova collezione.
Per chiarire le idee, ecco un semplice esempio:
supponiamo di avere questi dati:

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

e vogliamo ottenere:

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

usiamo un algoritmo MapReduce in cui Map emette 1 per ogni chiave (risorsa, data) e Reduce somma i valori;

popoliamo il database

```
> 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:'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)})
```

costruiamo la funzione Map

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

la funzione reduce

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

eseguiamo l'algoritmo

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

e leggiamo i risultati

```
> db.hit_stats.find()
{ "_id" : { "resource" : "about", "year" : 2010, "month" : 0,
"day" : 20 }, "value" : { "count" : 1 } }
{ "_id" : { "resource" : "about", "year" : 2010, "month" : 0,
"day" : 21 }, "value" : { "count" : 2 } }
{ "_id" : { "resource" : "index", "year" : 2010, "month" : 0,
"day" : 20 }, "value" : { "count" : 3 } }
{ "_id" : { "resource" : "index", "year" : 2010, "month" : 0,
"day" : 21 }, "value" : { "count" : 2 } }
{ "_id" : { "resource" : "index", "year" : 2010, "month" : 0,
"day" : 22 }, "value" : { "count" : 1 } }
```

Capitolo 4

PROGETTAZIONE DI UN DATABASE DI VENDITA LIBRI ONLINE IN MONGODB E MYSQL

4.1 Il database Library Management

Per verificare le prestazioni dei database costruiti con MongoDB, consideriamo un database d'esempio sulla vendita di libri online, simile al modello presentato da Emily Stolfo nello *SlideShare* del sito 10gen di MongoDB.
Essenzialmente verranno memorizzati per quanto riguarda i libri il titolo, gli autori e la casa editrice, specificando se sono ancora disponibili o meno, per i clienti verranno salvati dei dati personali, username, password e gli indirizzi a cui mandare la consegna. Inoltre ogni cliente può scrivere delle recensioni riguardanti i libri.

4.1.2 Progettazione e realizzazione in MongoDB

Inizialmente osserviamo come potrebbe essere rappresentata la situazione generando dei documenti nella maniera più semplice possibile.
Ci sarà un documento "author", il quale memorizza nome e città natale dell'autore. Un documento "publisher" che specifica nome, data di fondazione e posizione della casa editrice. Un documento "book" con il titolo del libro, il prezzo, il numero di pagine, la lingua, la data di pubblicazione, una breve lista sull'argomento del libro che esprime il genere, l'ambientazione, ecc,; vengono inoltre indicati attraverso un riferimento che li identifica gli autori del libro e la casa editrice e infine è presente un campo che indica la disponibilità del libro o meno e una lista di recensioni sul libro fatte da eventuali clienti. Un documento "patron" che memorizza nome del cliente, username, password e la lista dei libri acquistati con le rispettiva date d'acquisto. Un documento "address" che specifica l'indirizzo completo di un cliente riferito attraverso il suo codice identificativo.
Un documento "note" che riporta il testo effettivo della nota con la data di recensione e i riferimenti al libro riferito e all'utente che l'ha scritta.
Ricordiamo che di default in ogni documento, in MongoDB, viene generato un _id di tipo *ObjectID*, ma in questo cosa ne specifichiamo uno numerico per chiarire i collegamenti tra i vari documenti.

Documento per l'autore

author = {
_id: "111111",
name: "Kristina Chodorow",
hometown: "New York" }

Documento per la casa editrice

publisher = {
_id: "667788"
name: "O'Reilly Media",
founded: "1980",
location: "CA" }

Documento per un libro

book={
_id: "123456"
title : "MongoDB: The definitive guide",
authors :["111111"]
published_date: ISODate("2010-09-24"),
pages: 216
language: "English"
publisher_id: "667788"
price: 34,70
subjects: [string, string]
notes:[{ "333222" }]
available: "true" }

Documento del cliente

patron = {
_id: "888888",
name: "Joe Bookreader"
username: "JoeB"
password:"test"
checked_out:
[{_id: "123456", checked_out: "2012-10-15"},
{_id: "758344", checked_out: "2012-09-12"}] }

Documento dell' indirizzo

address= {
_id: "232323"
patron_id: "888888",

street: “123 Fake St.”,
city: “Faketon”,
state: “MA” }

Documento di una nota

note = {
_id: “333222”,
book_id : “123456”,
date : ISODate(“2011-11-01”),
patron_id : “888888”,
text : “Very helpful book” }

Ora, vista la possibilità con MongoDB di incorporare documenti all’interno ad altri documenti, per ottenere delle prestazioni maggiori, dobbiamo decidere come gestire i nostri dati. Le operazioni all’interno di un documento in generale, per come è strutturato MongoDB, sono più facili da eseguire in quanto avvengono sul lato server; i link, ovvero i riferimenti tra documenti, invece devono prima essere trattati dal lato client per poi essere ritornati al server ed essere così pronti a rispondere alle interrogazioni.
Solitamente si incorporano gli oggetti che non vengono quasi mai utilizzati in maniera separata rispetto al documento padre; per progettare uno schema performante con MongoDB è infatti essenziale conoscere dettagliatamente l’applicazione che si sta andando a creare per sapere che interrogazioni e che modifiche generalmente il nostro database subirà e quindi costruirlo ad hoc per ottenere risultati migliori.
Nel nostro caso si capisce facilmente che i documenti essenziali possono ridursi a un documento utente, che incorpora gli indirizzi a lui riferiti(uno o più) e i libri da lui acquistati e un documento libro, che contiene tutte le informazioni ad esso riguardanti: autori, casa editrice e recensioni.

```
patron = {
        _id: “888888”,
        name: “Joe Bookreader”,
        username: “JoeB”,
        password:”test”,
        address: {
                street: “123 Fake St.”,
                city: “Faketon”,
                state: “MA”
                }
        checked_out:
                [{_id: “123445”, checked_out: “2012-10-15”},
```

```
            {_id: “7583445”, checked_out: “2012-09-12”}]
        }
```

A seconda degli indirizzi messi a disposizione dall’utente per la consegna della merce, il campo “address” può contenere un singolo documento o un array di documenti; allo stesso modo il campo “cheched_out” che memorizza i libri acquistati dall’utente può non essere presente, se non sono ancora stati acquistati libri, o contenere uno o più documenti.

```
book={
        _id: “123456”
        title : “MongoDB: The definitive guide”,
        authors : [
                {name: “Kristina Chodorow”,
                hometown: ”New York”}
                ]
        published_date: ISODate(“2010-09-24”),
        pages: 410
        language: “English”
        publisher: {
                name: “O'Reilly Media”,
                founded: “1980”,
                location: “CA”}
        available: true
        price: 34,70
        subjects: [string, string]
        notes:[{
                patron_id: “888888”,
                date: ISODate(“2011-11-01”),
                text: “Very helpful book”
                }]
        }
```

In questo documento il campo “authors” può essere un semplice documento nidificato oppure un array di documenti a seconda del numero di autori ; inoltre anche il campo “notes” può contenere una o più note sotto forma di array di documenti embedded o non essere presente se ancora nessuno ha recensito un parere sul libro interessato.

4.1.3 Progettazione e realizzazione in MySQL

Rappresentiamo la stessa situazione descritta dal database creato in MongoDB sotto forma di un database relazionale.

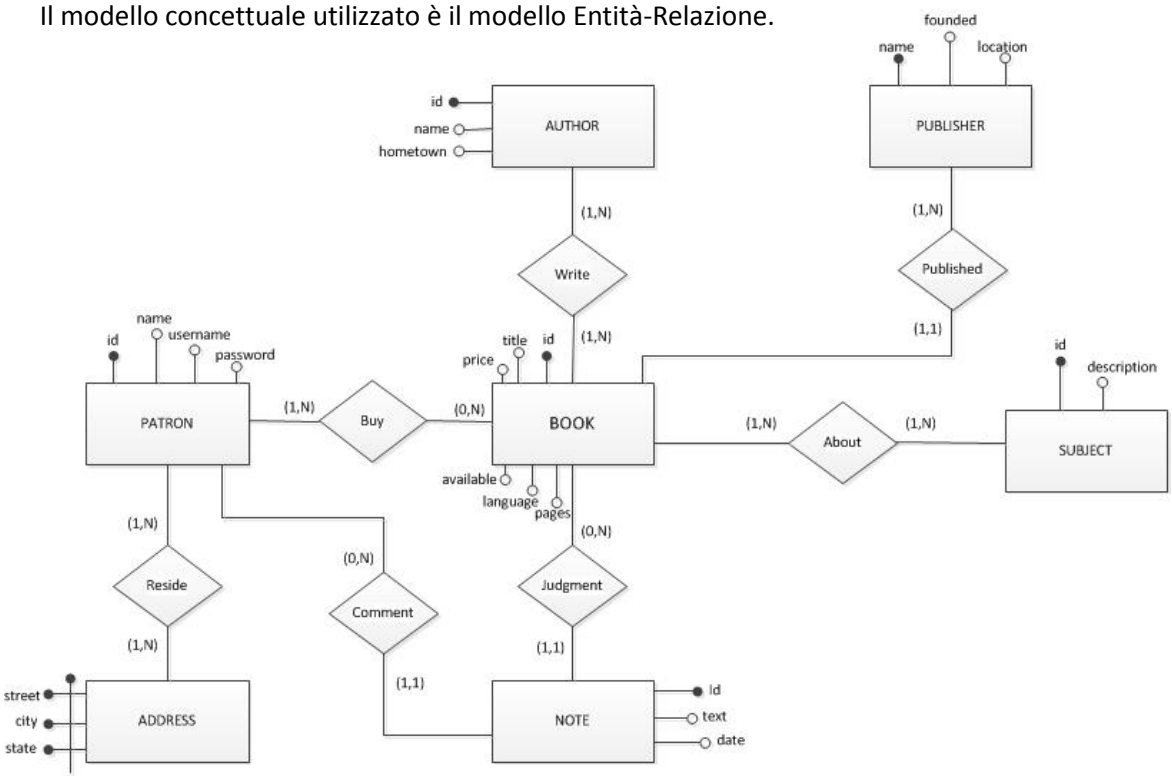
Per prima cosa ci siamo concentrati sulla progettazione concettuale creando lo schema ER relativo al nostro database di vendita libri online, per poi passare alla progettazione logica, trasformandolo in uno schema logico e infine è stata affrontata la fase di progettazione fisica implementandolo con MySQL.

Requisiti strutturali

Ogni **libro** viene identificato univocamente da un id; di esso si memorizza titolo, lingua, numero di pagine, prezzo e viene specificato se è disponibile o meno.
Inoltre si conoscono gli autori, la casa editrice e gli argomenti del libro.
Per ogni **autore**, identificato univocamente da un id, si memorizza nome e città di appartenenza, mentre delle **case editrici**, identificate dal nome, si conosce la data di fondazione e la location.
Ogni **argomento** del libro viene identificato da un id e viene descritto da alcune parole chiave che possono indicare il genere o l'anno d'ambientazione, ecc.
Di ogni **cliente**, identificato da un id si memorizza nome, username e password.
L'**indirizzo** è invece caratterizzato dalla via, paese e stato ed è relativo ad uno o più clienti (possono essere familiari che vivono nella stessa abitazione).
Infine le **note**, identificate da un id, memorizzano il testo e la data in cui sono state recensite. Si conosce inoltre il cliente che ha emesse la nota e il libro a cui è riferita.

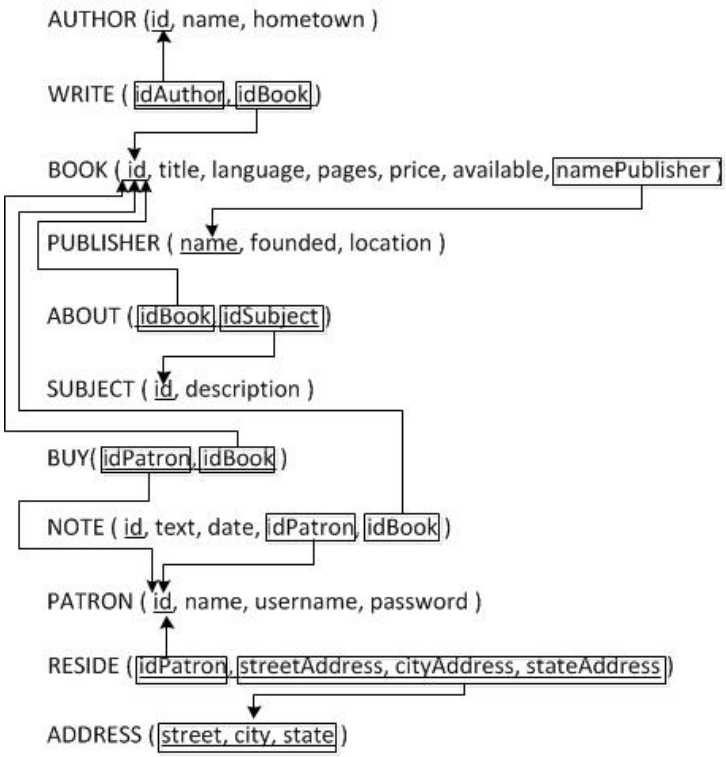
Progettazione concettuale

Il modello concettuale utilizzato è il modello Entità-Relazione.



Progettazione logica

Non essendoci nello schema ER precedente delle generalizzazioni o altro da dover semplificare, si può passare direttamente alla traduzione logica dello schema che viene riportata qui di seguito. Essa rappresenta esattamente le stesse informazioni dello schema ER e ci sarà d'aiuto per l'implementazione in MySQL.



Progettazione fisica

La progettazione fisica in SQL dello schema del database è riportata nell'appendice A, parte 1.

Capitolo 5

ANALISI DELLE PRESTAZIONI

Per valutare le prestazioni di entrambi i database "Library" presentati nel capitolo precedente sono state effettuate due tipologie di esperimenti; uno riguardante l'inserimento dei dati e uno riguardante l'interrogazione dei database.
Infine viene anche presentata la denormalizzazione nei sistemi di gestione MongoDB e MySQL e viene creato ad hoc per l'interrogazione scelta un database denormalizzato del database Library in MySQL per osservare i nuovi tempi di risposta.
Il pc utilizzato è a 64 bit e ha le seguenti specifiche:

Produttore: Sony
Modello: Vaio VPCCA
Processore: Intel Core i5
Memoria(RAM): 4,00 GB
Tipo di sistema: Ubuntu 12.10

5.1 Inserimenti

Affinché la quantità di dati da inserire in entrambi i database sia pressoché la stessa si è deciso di osservare i tempi di inserimento, per quanto riguarda MongoDB, del documento per la casa editrice e, per MySQL, della rispettiva tabella Publisher, entrambi riportati qui sotto.

Documento per la casa editrice

publisher = {
name: "O'Reilly Media",
founded: "1980",
location: "CA"
}

Tabella Publisher

PUBLISHER(<u>name</u>, founded, location);

5.1.1 Inserimenti in MongoDB

Per valutare le performance sui tempi di inserimento sono state inserite varie quantità di documenti importando i dati da un file txt attraverso la seguente istruzione dal promt dei comandi:

mongoimport --db library --collection Publisher --file nomefile.txt

Il file txt è stato creato opportunamente con un programmino scritto in Java in modo tale che creasse una quantità prestabilita di documenti che rispecchino i campi *name, founded* e *location* scritti nel formato per gli inserimenti:

{ "name" : "Axoxeoduy", "founded" : "1818", "location" : "OV"}.

Nella tabella 5.1 sono riportati i risultati dei vari inserimenti.

NUMERO DOCUMENTI INSERITI	TEMPO DI ESECUZIONE
500	< 0,001 sec
1.000	< 0,001 sec
2.000	0.13 sec
5.000	0.31 sec
10.000	0,58 sec
50.000	1,52 sec
100.000	2,49 sec
200.000	5,11 sec
300.000	7,62 sec
500.000	12,91 sec
1.000.000	28,18 sec
3.000.000	1 min 20,28 sec
5.000.000	2 min 15,59 sec

Tabella 5.1: Inserimenti MongoDB

5.1.2 Inserimenti in MySQL

Come per MongoDB, anche in questo caso sono stati sperimentati vari inserimenti di diverse quantità di dati per valutare la velocità di queste operazioni.

Le varie righe da inserire sulla tabella Publisher sono state recuperate da un file CSV, Comma-Separated Values, formato basato su file di testo utilizzato per l'esportazione ed importazione dei dati in tabelle. In particolar modo, non essendoci delle regole precise, il file ha i valori dei vari campi strutturati nel seguente modo:

name|founded|location
Anch'esso è stato realizzato mediante un programmino Java che inserisse dei valori casuali per i vari campi.
Una volta creato il file è stato caricato dalla shell di MySQL con il seguente comando:
LOAD DATA LOCAL INFILE 'nomefile'
INTO TABLE Publisher
FIELDS TERMINATED BY '|'
LINES TERMINATED BY '\n';
Nella tabella 5.2 sono riportati i risultati dei vari inserimenti.

NUMERO DOCUMENTI INSERITI	TEMPO DI ESECUZIONE
500	0,09 sec
1.000	0,10 sec
2.000	0.12 sec
5.000	0.15 sec
10.000	0,19 sec
50.000	0,42 sec
100.000	1,83 sec
200.000	7,17 sec
300.000	15,03 sec
500.000	30,16 sec
1.000.000	50,78 sec
3.000.000	11 min 49,27 sec
5.000.000	1 h 7 min 21,59 sec

Tabella 5.2: Inserimenti MySQL

5.1.3 Analisi degli inserimenti

Come si può notare dalle 2 tabelle finché la quantità dei dati è inferiore al milione MySQL riesce a tenere testa a MongoDB dopo di ché i tempi di esecuzione di MySQL aumentano esponenzialmente rispetto al carico di dati inserito. Infatti mentre per MongoDB l'inserimento di 5 milioni di dati si effettua in poco più di due minuti, MySQL ci impiega addirittura un'ora abbondante.
La differenza così elevata di tempi è dovuta in parte senz'altro al fatto che in MySQL l'imposizione dei vingoli *not null*, *primary key* e di *foreign key* richiedano il controllo della

validità dei dati al momento dell'inserimento, cosa che MongoDB non si preoccupa minimamente e lascia il compito al progettista di controllare la correttezza dei documenti.
Nonostante questa mancanza di controllo dei dati, per le applicazioni che richiedono un numero elevato di inserimenti, le migliorie apportate con questo database non-relazionale sono senz'altro esorbitanti.

5.2 Interrogazioni

Analizziamo ora i tempi di risposta di entrambi i database a delle interrogazioni equivalenti.
Prima di poter interrogare i database bisogna ovviamente impegnarsi a popolarli.
Innanzitutto sono stati creati dei metodi che salvassero in array i dati creati casualmente per tutti i campi dei documenti delle collezioni (ad esempio un array con i nomi dei clienti, uno con gli username, uno con i titoli dei libri, ecc..).
Una volta pronti tutti gli array dei dati, per quanto riguarda il database di MongoDB, sono stati creati due programmi: uno per creare i documenti della collezione Patron e uno per i documenti della collezione Book. Questi programmi utilizzando i dati degli array prima creati, stampano in un file txt una quantità prestabilita di documenti pronti per l'inserimento nel database "Library" creato in MongoDB. Affinché tutto funzioni coerentemente si è posta attenzione che, i riferimenti "_id" nel campo "checked_out" nella collezione Patron, che indicano I libri comprati, siano effettivamente esistenti all'interno della collezione Book. In MongoDB infatti, a differenza di MySQL, non esiste un controllo sui vincoli d'integrità referenziale, che crea il documento solo se i dati, che si riferiscono ad altri campi di altri documenti, effettivamente esistono.
Per il popolamento del database di MySQL sono stati creati dei programmi per ogni tabella del database, i quali stampano anch'essi su un file txt una quantità prestabilita di insert utilizzando i dati salvati negli array.
Per valutare le prestazioni di entrambi gli array sono state testate due interrogazioni:

- la prima richiede la selezione dei libri (specificando titolo e descrizione) con più di 50 pagine, di costo inferiore ai 50 euro, suddividendoli per casa editrice e ordinandoli per prezzo;

- la seconda utilizza una funzione di aggregazione e vuole ricavare per ogni autore il libro meno costoso specificandone il titolo e le informazioni inerenti la casa editrice.

5.2.1 Interrogazioni in MongoDB

La prima interrogazione viene posta nel seguente modo al database:

db.Book.find({ "pages" : {"$gt" : 50}, "price" : {"$lt" : 50} } , { "_id" : 0, "publisher.name" : 1, "publisher.founded" : 1, "title" : 1, "authors.name" : 1, "price" : 1, "pages" : 1, "subjects" : 1 }).sort({"publisher.name": -1, "price" : -1 })
Nel primo campo del metodo find si specificano i criteri di selezione, in questo caso il libro deve avere più di 50 pagine e costare meno di 50 euro; mentre nel secondo campo vengono specificati i campi che si vogliono visualizzare attraverso il valore 1; venendo sempre rappresentato di default il campo _id, per non visualizzarlo, non essendo di particolare interesse, lo si pone uguale a 0. Infine i risultati vengono ordinati a seconda della casa editrice e del prezzo del libro con l'operatore sort.
Come si può notare, l'interrogazione riguarda esclusivamente la collezione Book; per valutare i tempi di risposta è stata posta la precedente interrogazione al database popolato con diverse quantità di documenti.
I risultati si possono osservare nella tabella 5.3.

NUMERO DOCUMENTI INSERITI NELLA COLLEZIONE BOOK	**TEMPO DI ESECUZIONE DELLA QUERY**
2.000	0,044 sec
5.000	0,054 sec
10.000	0,081 sec
20.000	0,164 sec
50.000	0,350 sec
100.000	0,712 sec
200.000 *	1,402 sec
500.000 *	3,216 sec
1.000.000 *	6,323 sec
1.500.000 *	9,661 sec

Tabella 5.3: Prima interrogazione MongoDB

*per ottenere i risultati dell'interrogazione è stato richiesto di aggiungere un indice per l'ordinamento, che è stato creato con il seguente comando:
db.Book.ensureIndex({"publisher.name" : -1, "price" : -1}).

Passiamo alla seconda interrogazione:

db.Book.aggregate({ "$group" :{"_id" : {"authorsname" : "$authors.name"}, "price" : {"$min" : "$price"}, "title" : { $addToSet: "$title"}, "namePublisher" : {$addToSe :"$publisher.name"},"foundedPublisher" : { $addToSe t: "$publisher.founded"},"locationPublisher" : { $addToSet:"$publisher.location" }})

Essa attraverso il metodo aggregate raggruppa per autore il libro a minor prezzo (*{"$min" : "$price"}*) specificandone, come richiesto, il titolo e le informazioni della casa editrice attraverso il comando *"$addToSet*".

Anche per essa vengono osservati i tempi di risposta del database popolato con diverse quantità di documenti.

I risultati si possono osservare nella tabella 5.4.

NUMERO DOCUMENTI INSERITI NELLA COLLEZIONE BOOK	TEMPO DI ESECUZIONE DELLA QUERY
2.000	0,069 sec
5.000	0,086 sec
10.000	0,142 sec
20.000	0,285 sec
50.000	0,695 sec

Tabella 5.4: Seconda interrogazione MongoDB

Il test di questa interrogazione è stato bloccato solo a 50.000 documenti poiché il risultato successivo del database popolato con 100.000 documenti genera un documento troppo grande (supera i 16MB).

5.2.2 Interrogazioni in MySQL

Per entrambe le interrogazioni viene utilizzata la solita tipologia di interrogazioni tipica dei database relazionali: SELECT-FROM-WHERE.

A differenza delle query poste nel database di MongoDB che chiedono la risposta da un'unica collezione, qui sono necessari più JOIN di diverse tabelle per ottenere i risultati richiesti.

In particolare la prima interrogazione viene espressa nel seguente modo:

```
SELECT P.name, P.founded, B.title, A.name, B.price, B.pages, S.description
FROM Author A INNER JOIN WriteP W ON A.id=W.idAuthor
        INNER JOIN Book B ON W.idBook=B.id
        INNER JOIN Publisher P ON B.namePublisher=P.name
```

INNER JOIN About Ab ON B.id=Ab.idBook
INNER JOIN Subject S ON Ab.idSubject=S.id
WHERE B.pages > 50 AND B.price<70
ORDER BY P.name, B.price;

Come si può notare essa richiede ben cinque inner join e viene utilizzata la clausola ORDER BY per riordinare i risultati.

Nella tabella 5.5 si osservano i tempi di risposta del database; in particolare il dato sul numero di documenti inseriti riguarda la quantità di libri inseriti nella tabella Book; di conseguenza per ottenere la risposta desiderata si sono popolate in maniera congrua le altre tabelle: ad esempio nel primo caso essendo stati inseriti 2000 libri sono stati inseriti 1500 autori nella tabella Author, supponendo che un autore abbia scritto più di un libro, 2700 righe nella tabella WriteP e 3000 righe nella tabella About. Per quanto riguarda la tabella Publisher e la tabella Subject sono stati inseriti per tutte le interrogazioni 400 case editrici e 100 descrizioni fisse.

NUMERO DI DOCUMENTI INSERITI NELLA TABELLA BOOK	**TEMPO DI ESECUZIONE DELLA QUERY**
2.000	0,03 sec
5.000	0,05 sec
10.000	0,06 sec
20.000	0,12 sec
50.000	0,23 sec
100.000	0,63 sec
200.000	1,40 sec
500.000	12,66 sec
1.000.000	55,13 sec
1.500.000	2 min 46,51 sec

Tabella 5.5: Prima interrogazione MySQL

La seconda interrogazione posta al database è invece la seguente:

SELECT A.id, A.name, B.title, MIN(B.price), P.name, P.founded, P.location
FROM Author A INNER JOIN WriteP W ON A.id=W.idAuthor
INNER JOIN Book B ON W.idBook=B.id
INNER JOIN Publisher P ON B.namePublisher= P.name
GROUP BY A.id;

Essa utilizza la clausola “GROUP BY” che specifica rispetto a quale campo viene poi valutata la funzione di aggregazione “MIN”. In questo caso si ricava il prezzo più basso del libro di ogni autore.
I risultati sui tempi di risposta sono scritti nella seguente tabella.
(Per quanto riguarda il dato sui documenti inseriti è inerente solo alla tabella Book come spiegato sopra)

NUMERO DI DOCUMENTI INSERITI NELLA TABELLA BOOK	TEMPO DI ESECUZIONE DELLA QUERY
2.000	0,03 sec
5.000	0,04 sec
10.000	0,08 sec
20.000	0,11 sec
50.000	0,30 sec
100.000	0,62 sec
200.000	2,90 sec
500.000	9,27 sec
1.000.000	53,49 sec
1.500.000	1 min 46,86 sec

Tabella 5.6: Seconda interrogazione MySQL

5.2.3 Analisi delle interrogazioni

Per quanto riguarda la prima interrogazione, osservando separatamente i tempi di esecuzione dei due database, si nota che, mentre in MongoDB, all’aumentare del carico dei dati, i tempi aumentano in maniera lineare o addirittura in proporzione diminuiscono, in MySQL con il crescere dei dati aumentano quasi esponenzialmente i tempi di risposta. Come per gli inserimenti si nota dunque che mentre MySQL con carichi di dati elevati non da buone prestazioni, MongoDB riesce a stupirci mantenendo dei tempi di risposta sorprendenti.
Per quanto riguarda la seconda interrogazione, nel caso del database di MongoDB i risultati ottenuti attraverso il metodo di aggregazione “aggregate” non sono altrettanto sod-

disfacenti. In questo caso infatti MySQL riporta dei tempi senz'altro migliori dall'inizio alla fine del nostro test.
Ritornando alla prima interrogazione e confrontando i tempi di esecuzione di entrambi i database si nota che, fino ai 100.000 dati inseriti, seppur di poco MySQL è più veloce di MongoDB, ma con l'aumentare del carico di dati, le prestazioni di MySQL calano precipitosamente. Questo è dovuto molto probabilmente al costo elevato delle operazioni di join, che devono collegare un numero elevato di documenti. Per verificare se effettivamente la causa delle cattive prestazioni di MySQL, è questa, nel prossimo esperimento si è deciso di eliminare i join dall'interrogazione, creando un database adatto, attraverso la tecnica della denormalizzazione.

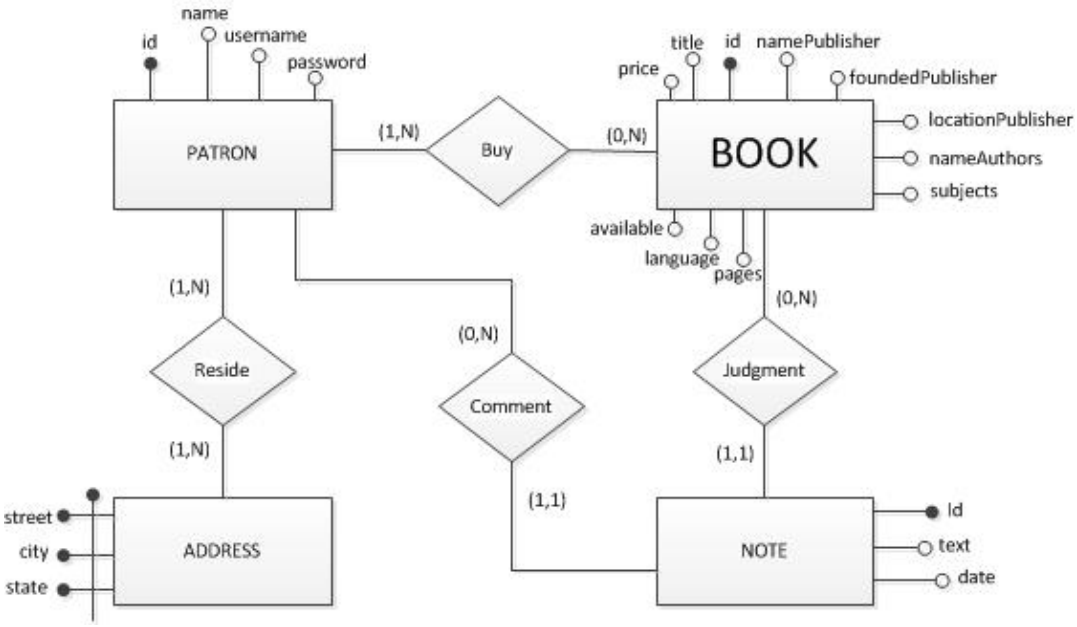
5.3 Denormalizzazione

Prima di spiegare in cosa consiste la denormalizzazione concentriamoci sulle principali differenze tra MongoDB e i database relazionali, come MySQL.
La novità del modello dei dati di MongoDB non è solamente la nuova struttura dati utilizzata per la memorizzazione, ma proprio la nuova maniera di memorizzare i dati; mentre nei database relazionali si cerca di creare documenti con caratteristiche indipendenti separati tra loro, dette tabelle, le quali si collegano attraverso delle relazioni, nei database sviluppati con MongoDB si cerca di inglobare tutte le informazioni in un documento unico, senza preoccuparsi della possibile ridondanza che viene a crearsi. In poche parole MongoDB non segue la teoria della normalizzazione degli schemi, seguita nei database relazionali; uno schema normalizzato è uno schema privo di ridondanze che divide ogni relazione se possibile in relazioni più piccole e indipendenti; ciò garantisce la consistenza dei dati, in quanto in caso di cancellazione o modifiche dei dati non bisogna preoccuparsi di aggiornare le altre entità che avrebbero potuto presentare dei dati ridondanti.
Questa caratteristica di MongoDB di disinteressarsi ad avere uno schema fisso dei documenti, facilita la distribuzione dei dati su più macchine, cosa difficile da ottenere nei database relazionali dove ogni tabella è fortemente riferita ad altre, ma qualità molto richiesta ai giorni d'oggi visto l'aumento consistente del carico di dati nelle applicazioni.
Dopo tutto questo discorso relativo alla normalizzazione, risulta chiaro che denormalizzare un database consiste nell'introdurre delle ridondanze o nell'inglobare più entità indipendenti per ottenere un particolare guadagno. Normalmente nei database in cui le operazioni di scrittura sono molto frequenti, un database normalizzato è più performante, in quanto l'inserimento dei dati riguarda pochi campi visto che le tabelle sono state separate il più possibile e inoltre gli aggiornamenti riguardanti un campo comportano solo una modifica, non essendoci dei duplicati in altre tabelle del database. Invece, nelle applicazioni in cui sono più frequenti le operazioni di lettura, le quali comportano solitamente l'utilizzo dei costosi join, risulta più performante un database denormalizzato.
Proprio per questo motivo, si è deciso di denormalizzare il database "Library" realizzato con MySQL, con lo scopo di risultare più performante rispetto alla prima interrogazione

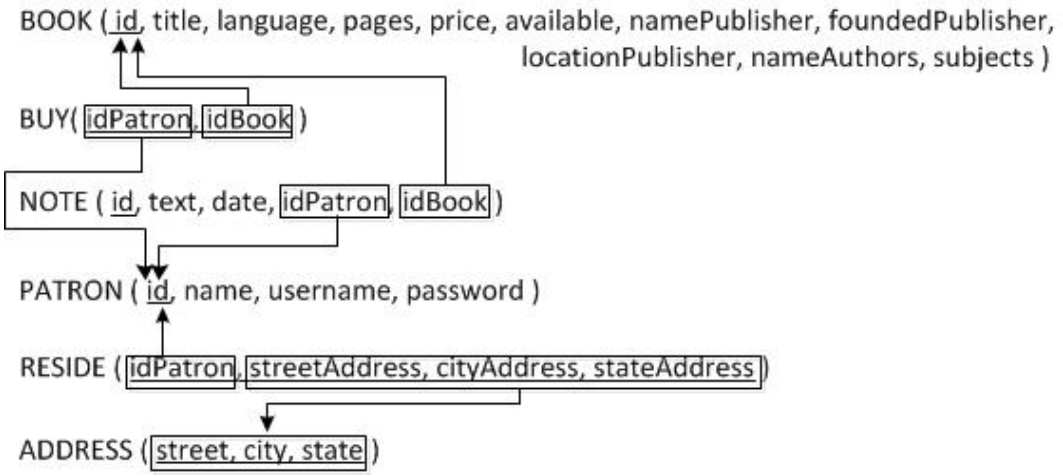
considerata nel paragrafo precedente. Ricordiamo che richiedeva la selezione dei libri con più di 50 pagine, di costo inferiore ai 50 euro, suddividendoli per casa ce ,ordinandoli per prezzo e specificandone titolo e descrizione.

5.3.1 Progettazione concettuale, logica e fisica del database denormalizzato

Lo schema ER, come si può osservare nella figura seguente, è stato modificato affinché nell'entità Book, oltre ai dati strettamente relativi al libro, vengano inglobate tutte le informazioni riguardanti la casa editrice, gli autori e le descrizioni del libro.



Rispettivamente alle modifiche dello schema ER, è stato modificato anche lo schema logico relazionale, riportato nell'immagine seguente.



Infine il codice SQL dello schema denormalizzato è riportato nell'appendice A, parte 2.

5.3.2 Analisi dell'interrogazione

Nella tabella 5.7 vengono riportati i nuovi dati sui tempi di risposta della prima interrogazione rispetto al database denormalizzato.
La formulazione della query è la seguente, e come si può notare risulta priva di join:

```
SELECT namePublisher, foundedPublisher, title, nameAuthors, price, pages, subjects
FROM BookDen
WHERE pages > 50 AND price<70
ORDER BY namePublisher, price;
```

NUMERO DI DOCUMENTI INSERITI NELLA TABELLA BOOK	TEMPO DI ESECUZIONE DELLA QUERY
2.000	0,01 sec
5.000	0,02 sec
10.000	0,03 sec
20.000	0,05 sec
50.000	0,10 sec
100.000	0,22 sec
200.000	0,42 sec
500.000	3,56 sec
1.000.000	20,26 sec
1.500.000	33,95 sec

Tabella 5.7: Prima interrogazione MySQL denormalizzato

Confrontando questi tempi di risposta rispetto a quelli del database precedente di MySQL si nota subito un netto miglioramento sulla velocità di risposta, anche per quanto riguarda la valutazione del database con un carico di dati notevole.
Nonostante ciò, confrontando questi risultati con quelli del database implementato con MongoDB, si nota che MongoDB per quanto riguarda l'interrogazione di database con oltre un milione di dati rimane comunque molto più performante.

Capitolo 6

CONCLUSIONI

In questa tesi si sono voluti studiare i database non relazionali, soluzione alternativa ai database relazionali per memorizzare grosse quantità di dati.
Prima di tutto siamo partiti da una spiegazione generale su come e perché si siano sviluppati questi nuovi database, descrivendo le classificazioni principali e i pro e i contro di questi nuovi "archivi" di dati.
In particolare ci siamo poi focalizzati sul caso di studio di MongoDB, descrivendo prima di tutto le sue principali proprietà e presentando poi le regole di sintassi per l'inserimento, l'aggiornamento, la cancellazione e le interrogazione sui dati.
Infine, ci siamo concentrati sulle nuove prestazioni di lettura e scrittura di questo nuovo database, mettendolo a confronto con un database progettato con MySQL.
Sostanzialmente abbiamo implementato una stessa situazione in entrambi i database e poi abbiamo eseguito dei test: il primo test riguarda l'inserimento dei dati, il secondo l'interrogazione sui database e il terzo constata le prestazioni generate dai database denormalizzati.
In tutti questi esperimenti è spiccata la capacità di MongoDB di non deludere nelle prestazioni anche in database con un carico di dati elevato, cosa che suscita scarsi risultati nei database relazionali.
Si può concludere quindi che, nonostante MongoDB rappresenti i dati in documenti privi di schemi e a prima vista poco organizzati, se sono stati costruiti in maniera consona rispetto l'utilizzo che viene fatto, fornisce ottime prestazioni in scrittura e lettura e grazie proprio ai suoi documenti privi di relazioni rigide permette la scalabilità su più server.
Questo studio ci fa riflettere su come a volte certe tecniche che ormai sembrano consolidate, debbano poi essere sostituite da nuovi metodi per soddisfare le nuove esigenze del mercato tecnologico, che in anni come questi si sta sviluppando in maniera esponenziale.

Appendice

PROGETTAZIONE FISICA

1 Progettazione fisica database MySQL

Qui di seguito viene riportato il codice SQL dello schema del database “Library”.

```
CREATE DATABASE library;
```

Tabella AUTHOR:

```
CREATE TABLE Author(
        id INTEGER NOT NULL PRIMARY KEY,
        name VARCHAR(32),
        hometown CHAR(2)
);
```

Tabella PUBLISHER:

```
CREATE TABLE Publisher(
        name VARCHAR(64) NOT NULL PRIMARY KEY,
        founded INTEGER,
        location CHAR(2)
);
```

Tabella BOOK:

```
CREATE TABLE Book(
        id INTEGER NOT NULL PRIMARY KEY,
        title VARCHAR(128),
        language CHAR(3),
        pages INTEGER,
        price FLOAT,
        available BOOLEAN,
        namePublisher VARCHAR(64),
```

```
        FOREIGN KEY(namePublisher) REFERENCES Publisher(name) ON UPDATE
        CASCADE
);
```

Tabella WRITE:

```
CREATE TABLE Write(
        idAuthor INTEGER NOT NULL,
        idBook INTEGER NOT NULL,
        PRIMARY KEY(idAuthor, idBook),
        FOREIGN KEY(idAuthor) REFERENCES Author(id) ON UPDATE CASCADE,
        FOREIGN KEY(idBook) REFERENCES Book(id) ON UPDATE CASCADE
);
```

Tabella SUBJECT:

```
CREATE TABLE Subject(
        id INTEGER NOT NULL PRIMARY KEY,
        description VARCHAR(128)
);
```

Tabella ABOUT:

```
CREATE TABLE About(
        idBook INTEGER NOT NULL,
        idSubject INTEGER NOT NULL,
        PRIMARY KEY(idBook, idSubject),
        FOREIGN KEY(idBook) REFERENCES Book(id) ON UPDATE CASCADE,
        FOREIGN KEY(idSubject) REFERENCES Subject(id) ON UPDATE CASCADE
);
```

Tabella PATRON:

```
CREATE TABLE Patron(
        id INTEGER NOT NULL PRIMARY KEY,
        name VARCHAR(32),
        username VARCHAR(32),
        password VARCHAR(128)
);
```

Tabella BUY:

```
CREATE TABLE Buy(
        idPatron INTEGER NOT NULL,
        idBook INTEGER NOT NULL,
        PRIMARY KEY(idPatron, idBook),
        FOREIGN KEY(idPatron) REFERENCES Patron(id) ON UPDATE CASCADE,
        FOREIGN KEY(idBook) REFERENCES Book(id) ON UPDATE CASCADE
);
```

Tabella NOTE:

```
CREATE TABLE Note(
        id INTEGER NOT NULL PRIMARY KEY,
        text VARCHAR(256),
        dated DATE,
        idPatron INTEGER,
        idBook INTEGER,
        FOREIGN KEY(idPatron) REFERENCES Patron(id) ON UPDATE CASCADE,
        FOREIGN KEY(idBook) REFERENCES Book(id) ON UPDATE CASCADE
);
```

Tabella ADDRESS:

```
CREATE TABLE Address(
street VARCHAR(128) NOT NULL,
city VARCHAR(64) NOT NULL,
state CHAR(2) NOT NULL,
PRIMARY KEY(street, city, state)
);
```

Tabella RESIDE:

```
CREATE TABLE Reside(
idPatron INTEGER NOT NULL,
streetAddress VARCHAR(128) NOT NULL,
cityAddress VARCHAR(64) NOT NULL,
stateAddress CHAR(2) NOT NULL,
PRIMARY KEY(idPatron, streetAddress, cityAddress, stateAddress),
FOREIGN KEY(idPatron) REFERENCES Patron(id) ON UPDATE CASCADE,
FOREIGN KEY(streetAddress, cityAddress, stateAddress) REFERENCES Ad-
dress(street, city, state) ON UPDATE CASCADE );
```

2 Progettazione fisica database MySQL denormalizzato

Qui di seguito viene riportata la formulazione in SQL del database “Library” denormalizzato.

```
CREATE DATABASE library;
```

Tabella BOOK:

```
CREATE TABLE Book(
        Id INTEGER NOT NULL PRIMARY KEY,
         title VARCHAR(128),
        language CHAR(3),
        pages INTEGER,
        price FLOAT,
        available BOOLEAN,
        namePublisher VARCHAR(64),
        foundedPublisher INTEGER,
        locationPublisher CHAR(2),
        nameAuthors VARCHAR(256),
        subjects VARCHAR(256)
);
```

Tabella PATRON:

```
CREATE TABLE Patron(
        id INTEGER NOT NULL PRIMARY KEY,
        name VARCHAR(32),
        username VARCHAR(32),
        password VARCHAR(128)
);
```

Tabella BUY:

```
CREATE TABLE Buy(
        idPatron INTEGER NOT NULL,
        idBook INTEGER NOT NULL,
        PRIMARY KEY(idPatron, idBook),
        FOREIGN KEY(idPatron) REFERENCES Patron(id) ON UPDATE CASCADE,
        FOREIGN KEY(idBook) REFERENCES Book(id) ON UPDATE CASCADE
);
```

Tabella NOTE:

```
CREATE TABLE Note(
        id INTEGER NOT NULL PRIMARY KEY,
        text VARCHAR(256),
        dated DATE,
        idPatron INTEGER,
        idBook INTEGER,
        FOREIGN KEY(idPatron) REFERENCES Patron(id) ON UPDATE CASCADE,
        FOREIGN KEY(idBook) REFERENCES Book(id) ON UPDATE CASCADE
);
```

Tabella ADDRESS:

```
CREATE TABLE Address(
street VARCHAR(128) NOT NULL,
city VARCHAR(64) NOT NULL,
state CHAR(2) NOT NULL,
PRIMARY KEY(street, city, state)
);
```

Tabella RESIDE:

```
CREATE TABLE Reside(
idPatron INTEGER NOT NULL,
streetAddress VARCHAR(128) NOT NULL,
cityAddress VARCHAR(64) NOT NULL,
stateAddress CHAR(2) NOT NULL,
PRIMARY KEY(idPatron, streetAddress, cityAddress, stateAddress),
FOREIGN KEY(idPatron) REFERENCES Patron(id) ON UPDATE CASCADE,
FOREIGN KEY(streetAddress, cityAddress, stateAddress) REFERENCES Ad-
dress(street, city, state) ON UPDATE CASCADE );
```

Bibliografia

[1] Kristina Chodorow, (2013) MongoDB: The Definitive Guide, O'Reilly Media

[2] Kyle Banker, (2012) MongoDB In Action, New York, Manning

[3] Karl Seguin, The Little MongoDB Book

[4] http://www.mongodb.org/

[5] http://www.paperplanes.de/2010/2/25/notes_on_mongodb.html

[6] http://www2.mokabyte.it/cms/article.run?permalink=mb160_nosql-1

[7] http://nosql-database.org/links.html

[8] http://www.slideshare.net/savez/mongodb-15962567

[9] http://www.slideshare.net/mongodb/mongodb-schema-design-20356789

[10] http://www.slideshare.net/fullscreen/mongodb/schema-design-by-example/1

[11] http://docs.mongodb.org/master/MongoDB-manual.pdf

[12] http://sroselli.altervista.org/dispense/Linguaggi/ASP_PHP_MYSQL/Guida%20a%20MySQL.pdf