ALMA MATER STUDIORUM - UNIVERSITA' DI BOLOGNA
CAMPUS DI CESENA
SCUOLA DI SCIENZE

CORSO DI LAUREA IN SCIENZE E TECNOLOGIE INFORMATICHE

TITOLO DELLA RELAZIONE FINALE

Analisi e sperimentazione del DBMS NoSQL MongoDB: il caso di studio della Social Business Intelligence

Relazione finale in
Laboratorio di Basi di Dati

Relatore

Chiar.mo Prof Golfarelli

Presentata da

Alice Gambella

Sessione: III
Anno Accademico: 2013/2014

Indice

Introduzione

I sistemi di memorizzazione e gestione di dati di tipo NoSQL stanno suscitando un interesse sempre maggiore. In un ambiente mutevole, costantemente in rapida e fervente evoluzione come quello dell'informatica il modello relazionale ha rappresentato per decenni un indiscusso ed esemplare pilastro, conquistando e mantenendo un'assoluta egemonia nell'ambito della gestione ed amministrazione delle basi di dati. Per questo motivo il tentativo attuale, peraltro tutt'altro che isolato e passeggero, di delineare ed affermare delle alternative efficaci, accorte, originali, specializzate, ricche e poliedriche al modello relazionale e alle sue caratteristiche non può non attirare la nostra attenzione così come quella dell'intero panorama internazionale. Nei capitoli che seguono prenderemo dunque in considerazione quei cambiamenti intervenuti nel mondo delle basi di dati con l'inizio del nuovo millennio che sono riusciti a mettere in difficoltà i più che affermati sistemi di gestione di basi di dati fondati sul modello relazionale, facendo sì che l'esigenza di guardare oltre i suoi tratti peculiari e di ricercare delle soluzioni innovative e rivoluzionarie nel modo di approcciarsi ai problemi, di affrontarli e risolverli diventassero sempre più percepibili e diffuse fra tutti coloro che hanno attualmente necessità di memorizzare, trattare, analizzare ed organizzare le informazioni. All'univocità della tradizione si contrappone ora una molteplicità di nuove tecnologie e soluzioni, differenti non solo dal modello relazionale ma anche l'una dall'altra, poiché nate per rispondere ad esigenze e bisogni fra loro dissimili. In questa eterogeneità di approcci e di modelli è tuttavia possibile individuare dei tratti comuni, cercheremo pertanto di individuarli e di fornirne una presentazione organica e apprezzabile, seppure non esaustiva. Non essendo possibile in effetti offrire una presentazione completa ed approfondita di ogni tecnologia appartenente al movimento NoSQL, è stato necessario operare una scelta ed individuare, fra le altre, una tipologia di DBMS a cui dedicare maggiormente la nostra attenzione.

In questo studio di tesi ci siamo concentrati sui DBMS orientati al documento e, in particolare, su MongoDB che riesce a coniugare aspetti importanti, necessari e vantaggiosi tipici dei sistemi relazionali con la

flessibilità e la dinamicità nella gestione e nella modellazione dell'informazione propria dei sistemi NoSQL, così come l'elevata scalabilità che questo sistema consente con facilità di ottenere [1], permettendo il raggiungimento di performance invidiabili anche in presenza di dataset di dimensioni ragguardevoli, la cui gestione su singolo server potrebbe risultare complessa se non proibitiva anche per i sistemi più avanzati e potenti. La varietà di funzionalità anche complesse offerte da MongoDB combinate con la semplicità di utilizzo di questo DBMS lo rendono eclettico e polivalente, adatto dunque ad inserirsi in modo efficace, produttivo ed agevole in svariati ambiti e contesti. D'altronde, come si evince dall'articolo "The Forrester Wave: NoSQL Document Databases, Q3 2014" di Noel Yuhanna, la diffusione dei DBMS orientati al documento è in espansione e MongoDB ne è stato decretato il sistema leader da un'analisi condotta dalla Forrester Research già nel settembre del 2014, confrontandolo con altri tre DBMS della stessa tipologia e prendendo in considerazione quasi 60 criteri differenti fra i quali figurano le performance, la sicurezza, la scalabilità e l'elevata disponibilità del sistema [2]. Le prestazioni effettivamente ottenibili utilizzando MongoDB su singolo server e in ambito distribuito saranno al centro delle nostre analisi soprattutto relativamente all'utilizzo di una particolare funzionalità offerta dal sistema: la ricerca di testo, eseguita con l'ausilio di appositi indici costruiti sui campi testuali oggetto di ricerca.

Allo studio teorico di MongoDB affiancheremo quindi una verifica concreta e pratica delle possibilità che esso può offrire, eseguendo un insieme di test concentrati principalmente sull'esecuzione delle varie forme disponibili di ricerche di testo. Con questo progetto di tesi ci proponiamo quindi di condurre uno studio attento e critico di MongoDB e delle sue peculiarità e, attraverso di esso, delle caratteristiche dei sistemi NoSQL, dell'innovazione da essi introdotta, delle modalità con cui consentono il superamento di quei limiti che impediscono ai sistemi relazionali di fronteggiare con successo e di godere appieno delle moderne sfide e possibilità emerse e divenute centrali nell'ultimo decennio nel settore dell'elaborazione e memorizzazione dei dati.

I successivi capitoli della tesi saranno strutturati nel modo seguente: il primo avrà inizio con un'introduzione sul modello relazionale e sulle difficoltà che possono nascere nel tentativo di conformare le esigenze attuali di applicazioni ed utenti ai vincoli da esso imposti per lasciare poi spazio ad un'ampia descrizione del movimento NoSQL e delle tecnologie che ne fanno parte; il secondo capitolo sarà invece dedicato a MongoDB, alla presentazione delle sue caratteristiche e peculiarità, cercando di fornirne un quadro apprezzabile ed approfondito seppure non completo e del tutto esaustivo; infine nel terzo ed ultimo capitolo verrà approfondito il tema della ricerca di testo in MongoDB e verranno presentati e discussi i risultati ottenuti dai nostri test.

Capitolo 1 - Il movimento NoSQL

1.1 Il modello relazionale

Nel 1970 l'articolo "A Relational Model of Data for Large Shared Data Banks", pubblicato da Edgar F. Codd, presentava quello che in breve tempo sarebbe divenuto il modello logico largamente più utilizzato nella rappresentazione e gestione dei dati. Ancora oggi i Relational Database Management System dominano il settore dei sistemi informativi, basti pensare all'enorme diffusione di DBMS come Oracle, MySQL e Microsoft SQL Server, tutti quanti DBMS relazionali (RDBMS) e tutti attualmente ai massimi livelli di popolarità [3].

Il data model introdotto da Codd si basa su alcuni concetti principali, primo fra tutti quello di *relazione*, fondato sulla definizione matematica di *relation* ma non del tutto rispettoso delle sue proprietà in quanto viene eliminata l'importanza riconosciuta all'ordine in cui si dispongono i domini della relazione che, nel contesto del modello relazionale, si vedono assegnare ciascuno un nominativo univoco all'interno della relazione, detto *attributo*. In realtà, poiché gli n (con $n>0$) domini su cui è definita una relazione non debbono necessariamente essere distinti fra loro, è più corretto affermare che un differente attributo viene assegnato ad ogni singola occorrenza di ciascun dominio.

Per preservare l'integrità del modello dei dati da lui introdotto Codd definì nel 1985, con gli articoli "Is your DBMS really Relational" e "Does your DBMS run by the Rules?", tredici regole che stabiliscono le caratteristiche che un DBMS deve possedere ed i dettami che deve necessariamente soddisfare affinché si possa considerare davvero relazionale. Fra gli altri ricordiamo il principio fondamentale, asserito dalla "Regola zero" su cui tutte le regole successive si fondano, secondo il quale un RDBMS, per poter essere considerato tale, deve gestire ed operare sui dati esclusivamente attraverso le sue capacità (o funzionalità) relazionali. Le regole successive sottolineano gli aspetti principali per un database management system relazionale quali, ad esempio, il fatto (sancito dalla regola due, "*guaranteed access*") che ciascun valore di una relazione R deve essere accessibile a

partire semplicemente dal nome della relazione, il nome dell'attributo e lo specifico valore della primary key che identifica la tupla a cui quel valore appartiene oppure, come afferma la regola uno (*information representation*), che ogni informazione deve, a livello logico, essere rappresentata esclusivamente per mezzo di *valori* di tabelle. Fra le restanti è opportuno ricordare le regole otto e nove, che sanciscono un aspetto fondamentale del modello relazionale, di cui si soffriva la mancanza nei modelli di dati precedenti: le caratteristiche di indipendenza logica e fisica dei dati, ossia la possibilità per gli utenti ed i programmi applicativi di non doversi curare dell'effettiva strutturazione fisica dei dati né delle sue possibili variazioni, così come di eventuali modifiche apportate a livello logico che non giustifichino la necessità di variazioni sugli schemi esterni o nelle applicazioni. La *data independence* viene indicata dallo stesso Codd, nell'articolo "Relational Database: A practical foundation for productivity", come una delle principali motivazioni dello studio che ha condotto alla formulazione del modello relazionale. Al suo fianco vengono specificate anche altre motivazioni, quali ad esempio la necessità di proporre un modello che fosse strutturalmente semplice (obiettivo, quello della semplicità strutturale, indicato da Codd come *commiunicability objective*) e di introdurre sia un valido fondamento teorico per l'organizzazione e la gestione delle basi dati sia il concetto (ma non ancora la sintassi) di un linguaggio di alto livello che desse la possibilità agli utenti di applicare le operazioni desiderate non solo record per record ma anche su un più ampio insieme di dati. Questo in una realtà in cui la non separazione fra livello logico e fisico e l'assenza di alcun supporto per il *set processing* venivano individuate da Codd come due delle principali cause della crisi di produttività che affliggeva il settore della gestione ed elaborazione dei dati.

1.2 Limiti del modello relazionale

I DBMS relazionali hanno dominato in modo indiscusso il settore delle basi di dati per circa quarant'anni ma i cambiamenti avvenuti nell'informatica durante l'ultimo decennio ne hanno messo in luce alcuni limiti. Il modello relazionale è stato elaborato con il fine di gestire, memorizzare ed amministrare dati strutturati fra loro omogenei, in un mondo in cui la

flessibilità di interrogazione era più importante della flessibilità degli schemi [4]. Con l'inizio del nuovo millennio il panorama della gestione ed elaborazione dei dati però è cambiato radicalmente: con la nascita del web 2.0, l'affermarsi ed il diffondersi in modo sempre più massivo e permeante del mobile e dei social network abbiamo assistito ad una crescita esponenziale delle moli di dati disponibili in rete. Ora tutto è informazione [5], i dati da elaborare e gestire non sono esclusivamente numeri e stringhe ma immagini, video, e-mail, post pubblicati su forum e blog, emoticon e molto altro ancora. Non solo dati strutturati, ma anche, e soprattutto, dati non strutturati e semi-strutturati, generati dalle sorgenti più disparate, dagli smartphone ed i social network alla domotica, la sensoristica e tutto ciò che rientra nell'*Internet of Things*, e quindi anche di tipologie e formati estremamente differenti. Questa immensa mole di dati è al contempo un'enorme ricchezza ed una sfida non indifferente. L'immane quantità di informazioni rese disponibili on-line dagli utenti ne esprimono pensieri, gusti, opinioni ed interessi e costituisce quindi una vera e propria miniera d'oro per le aziende che, attraverso opportune analisi di tali dati, possono ottenere un feedback ampio ed immediato per le loro politiche di marketing, per i prodotti introdotti sul mercato, per ogni scelta ed iniziativa adottata e promossa dal loro brand, così come ottenere una più precisa conoscenza di gusti ed interessi del target a cui si rivolgono e confrontare i propri risultati con quelli dei propri competitors. Come è facile intuire però la gestione di una tale quantità di dati porta con sé notevoli problematiche.

Di fronte alla crescita smodata delle informazioni da mantenere ed elaborare, assieme all'aumento del numero di utenti che chiedono di avere accesso a tali informazioni in modo concorrente, ad esempio tramite siti web o social network, all'incremento della quantità di operazioni che su tali dati devono essere eseguite e, a causa delle dimensioni degli attuali dataset, della loro complessità, la *scalabilità* acquisisce un ruolo fondamentale. Una possibilità è offerta dallo scaling verticale, ossia adottare macchine più potenti, con un numero maggiore di processori, una maggiore quantità di memoria e hard disk di dimensioni superiori, ma questa soluzione rischia di essere molto costosa e comunque dalle possibilità inevitabilmente limitate. Una seconda e più efficace opportunità è rappresentata dallo scaling

orizzonatale, ossia dalla distribuzione del dataset e delle operazioni che su di esso si ha necessità di applicare su un cluster di macchine, senza che vi sia nessuna porzione di memoria condivisa fra i server. Oltre al guadagno economico in questo modo si rafforza anche la resilienza del proprio sistema poiché, come fanno notare P. J. Sadalage e M. Fowler nel testo "NoSQL Distilled", se guasti su una singola macchina sono comuni e potrebbero quindi facilmente rendere i dati non reperibili, avere a disposizione un cluster di macchine, che offra magari non solo distribuzione dei dati ma anche meccanismi di replicazione, consente di mettere in atto delle politiche che permettano di affrontare l'eventualità di guasti e difficoltà sia a livello dell'hardware che del software, preservando la disponibilità dei dati. Uno dei maggiori limiti del modello relazionale è proprio la difficoltà di realizzazione della scalabilità orizzontale. Alcuni DBMS relazionali consentono di costruire dei cluster basandosi però sul concetto di shared disk subsystem e lasciando quindi il sottosistema disco come unico point of failure. Si potrebbe pensare di limitarsi ad eseguire i DBMS relazionali come più server distinti che operano su porzioni separate dell'insieme complessivo di dati ma, sebbene in questo caso si riesca a suddividere effettivamente fra i nodi del cluster non solo il volume di dati da gestire ma anche il carico di operazioni ed elaborazioni che su di essi si ha bisogno di eseguire, anche questa opzione presenta delle serie difficoltà. Alle applicazioni viene ad esempio lasciato il compito di avere coscienza di quale server è necessario interrogare per ciascuna porzione dei dati, ovvero la dislocazione dei singoli bit di informazione fra i membri del "cluster", si perde inoltre ogni controllo di consistenza, integrità referenziale e querying oltre ai benefici delle transazioni che, nonostante permangano su singolo server, non possono essere garantiti fra un server e l'altro. Inoltre il costo, in caso di adozione di sistemi proprietari, può non essere trascurabile e certamente maggiore rispetto a quello di una soluzione non distribuita. In breve i sistemi di gestione di basi di dati relazionali non sono progettati per operare in modo efficiente su cluster [6], la scalabilità orizzontale, sempre più importante nella gestione ed elaborazione dei dati, è di difficile implementazione su tali sistemi, le soluzioni disponibili si dimostrano complesse ed onerose, sia in termini finanziari sia in termini di performance.

La rigidità dello schema propria di questo modello inoltre mal si adatta alla dinamicità e flessibilità della rete, dei requisiti dei dati che ne vengono attualmente estratti e delle applicazioni che su quei dati hanno bisogno di operare e con le quali gli utenti hanno la possibilità di interagire pressoché in ogni momento e da qualsiasi luogo. Tali motivazioni, insieme alla incapacità dei sistemi relazionali di continuare a garantire prestazioni elevate anche di fronte a moli di dati così ingenti, hanno fatto nascere la necessità di cercare nuove soluzioni, che andassero aldilà del modello relazionale e dei suoi limiti. Un'indagine condotta nel dicembre del 2011 da Couchbase rivela come l'interesse in sistemi di gestione di basi di dati che travalichino il tradizionale modello relazionale sia cresciuto e si sia diffuso portando un numero crescente di aziende ad abbandonare i relational database management system a favore di soluzioni più innovative, specificando come ragioni primarie di tale cambiamento proprio la rigidità propria del modello relazionale (per il 49% dei soggetti intervistati), la necessità di distribuire i dati ed il carico di lavoro da eseguire su di essi fra più macchine così da poter far fronte alle ingenti moli di informazioni e all'elevato numero di operazioni da eseguire su di esse (per il 35% dei rispondenti) ed il bisogno di migliorare le performance (nel 29% dei casi).

1.3 Il movimento NoSQL

La molteplicità di nuove tecnologie che si distaccano dai tradizionali RDBMS si riunisce sotto il termine NoSQL, acronimo di "Not only SQL". Il movimento NoSQL nasce infatti, non come antitesi della tradizione relazionale, ma con l'intento di sviluppare delle soluzioni innovative da affiancare ai classici RDBMS per poter affrontare in modo più efficace ed efficiente le nuove sfide del settore, prima fra tutte la gestione dei Big Data. Ciò che ci si può aspettare per il futuro, come suggeriscono Eric Redmond e Jim R. Wilson in "Seven Databases in Seven Weeks", è infatti non la scomparsa ed il superamento definitivo dei database management system fondati sul modello relazionale, bensì un ulteriore sviluppo ed una più capillare diffusione, al loro fianco, di molteplici tecnologie, diverse l'una dall'altra perché progettate ciascuna per far fronte a problematiche

differenti, per poter assolvere al meglio a determinati compiti, per rispondere ad insiemi specifici di esigenze, desideri e necessità di aziende, organizzazioni, enti e ricercatori.

Citando Dan McCreary e Ann Kelly nella loro opera "Making Sense of NoSQL. A guide for managers and the rest of us", potremmo dire che una delle sfide del NoSQL è data dalla difficoltà del fornirne una definizione completa ed adeguata. Proprio perché tale movimento riunisce in sé molteplici tecnologie, ciascuna con proprie caratteristiche e peculiarità, risulta difficile formulare una definizione adeguata. Ciò che possiamo fare è esporre alcuni degli aspetti comuni alla maggior parte di questi sistemi sottolineando però che nessuna delle caratteristiche che verranno di seguito enunciate è coercitiva o necessaria.

Come abbiamo visto nel precedente paragrafo una delle motivazioni principali che hanno reso sensibile la necessità di superare la tradizione relazionale per ricercare soluzioni nuove che fossero sviluppate e progettate alla luce delle nuove fattezze del web, dell'informatica in generale e del settore della gestione ed elaborazione di dati in modo particolare, è data dalla rigidità insita nel modello relazionale che richiede la definizione ed il rispetto di uno schema fisso ed invariabile. Al contrario i sistemi NoSQL sono solitamente *schemaless*, ovvero privi di schema, consentono quindi di aggiungere liberamente nuovi campi, senza che questo comporti modifiche o problematiche a livello strutturale. In questo modo si facilita la gestione di dati fra loro estremamente differenti per tipologia e formato, vantaggio tutt'altro che trascurabile data la grande varietà e molteplicità di forme che possono ora essere assunte dalle informazioni che si ha necessità di trattare.

Altro aspetto fondamentale riguarda la possibilità di scalare orizzontalmente con semplicità. La maggior parte dei DBMS NoSQL è stata infatti progettata appositamente per poter operare in modo agile ed efficiente su cluster, così da rispondere alla sempre più pressante esigenza di suddivisione del volume dei dati e del carico di lavoro su un insieme di macchine così da:

- riuscire a gestire moli di dati che, per la loro dimensione, non sarebbe possibile mantenere su singola macchina,
- migliorare le prestazioni,

- distribuire le operazioni di scrittura e lettura su più server, aumentando la quantità di scritture/letture che il sistema è in grado di effettuare in un certo intervallo di tempo.

Rick Cattel, nell'articolo "Scalable SQL and NoSQL Data Stores", riporta fra le caratteristiche chiave dei sistemi NoSQL, oltre alla già introdotta possibilità di inserire in modo semplice e dinamico nuovi attributi nei record di dati, proprio l'abilità di distribuire fra più macchine il dataset complessivo, introducendo anche un certo livello di ridondanza attraverso meccanismi di replicazione, e quella di scalare orizzontalmente il throughput di operazioni semplici, ossia ricerche di chiavi o scritture e letture che coinvolgono un solo record o un piccolo numero di record. Con l'avvento del web 2.0, infatti, le applicazioni web sono chiamate a far fronte ad un alto numero di utenti concorrenti le cui richieste si traducono per lo più in operazioni semplici che devono essere eseguite mantenendo prestazioni molto elevate, a causa della rapidità con cui nuove richieste si affiancano continuamente alle precedenti.
Lo scaling orizzontale consente di ottenere apprezzabili miglioramenti dal punto di vista delle performance, i sistemi NoSQL offrono infatti supporto anche alla scalabilità lineare (*linear scalability*), ovvero il fatto che per ogni nuovo server inserito nel cluster si possa ottenere un miglioramento prestazionale costante.
Per poter raggiungere la scalabilità e le performance desiderate è necessario però rinunciare ad alcune garanzie tipicamente offerte dai sistemi relazionali. Ricordiamo a questo proposito come il modello relazionale prescriva che ogni transazione debba soddisfare quattro proprietà fondamentali, normalmente indicate con l'acronimo ACID, ovvero:

- Atomicità (*atomicity*): poiché una transazione costituisce un'unità logica di elaborazione o tutte le operazioni di cui è costituita vanno a buon fine oppure nessuna delle modifiche applicate ai dati durante l'esecuzione della transazione permane nel db.
- Consistenza (*consistency*): al termine dell'esecuzione di una transazione il database deve essere sempre in uno stato consistente, ovvero nessuno dei vincoli di integrità deve essere stato violato.

- Isolamento (*isolation*): se *n* transazioni vengono eseguite in modo concorrente ciascuna di esse agisce sul db in modo isolato ed indipendente dalle restanti *n-1*, in altre parole l'esecuzione concorrente di *n* transazioni deve sempre risultare equivalente ad una possibile esecuzione sequenziale delle stesse *n* transazioni.
- Durabilità (*durability*): una volta eseguito il `commit` le variazioni applicate dalla transazione divengono permanenti, il DBMS deve garantirne pertanto la persistenza.

Questo tipo di approccio pone al centro dell'attenzione la consistenza ed è ideale, ad esempio, per l'effettuazione di transazioni finanziarie, in cui si ha bisogno di un alto livello di affidabilità e consistenza. Nel caso dei sistemi NoSQL invece, ruolo centrale viene spesso riconosciuto alla disponibilità dei dati (*availability*), frequentemente si accetta infatti di sacrificare parzialmente la consistenza per poter garantire disponibilità e tolleranza al partizionamento (*partition tolerance*).
Il teorema CAP, formulato da Eric Brewer nel 2000 e dimostrato poi due anni dopo da Seth Gilbert e Nancy Lynch del MIT, afferma infatti che in un sistema distribuito solo due delle tre proprietà da cui il teorema stesso prende il nome possono essere garantite, il terzo principio potrà invece essere presente soltanto in modo più debole, "eventuale". La sigla "CAP" è acronimo di:

- Consistency: il valore letto dagli utenti è in ogni caso quello più aggiornato, quindi due letture dirette nello stesso momento a due nodi distinti del cluster che richiedono di accedere ai medesimi dati riceveranno sempre risultati uniformi.
- Availability: ogni richiesta ricevuta da uno dei nodi attivi del cluster deve sempre ottenere una risposta, con cui si può specificare sia il completamento dell'operazione lanciata sia il suo fallimento. I dati memorizzati nel cluster, in altre parole, sono sempre accessibili agli utenti, purché i server che mantengono quei dati siano operativi.
- Partition Tolerance: il sistema deve continuare ad essere operativo nonostante l'inaffidabilità della rete, e quindi le possibili perdite di messaggi, ed essere in grado di fronteggiare eventuali

partizionamenti del cluster, ovvero la momentanea mancanza di connessione fra due o più nodi del sistema distribuito. Malgrado le possibili difficoltà di comunicazione fra i server, quindi, il servizio offerto dal cluster non deve venire meno, tranne nel caso di una globale assenza di connettività.

Per comprendere quanto asserito dal teorema consideriamo un sistema distribuito con replicazione dei dati, un'immagine peraltro tutt'altro che rara, dato che i sistemi NoSQL comunemente mantengono più copie dello stesso dato per motivi di sicurezza, per favorire la disponibilità dei dati e per poter distribuire fra i server le operazioni di lettura. Tali sistemi, infatti, spesso mettono a disposizione semplici ed efficaci politiche non solo di scaling orizzontale ma anche di *replication*. Supponiamo che i collegamenti fra due porzioni del cluster per qualche motivo al momento non siano attivi, il cluster si trova quindi suddiviso in due partizioni fra le quali non è al momento possibile alcuna comunicazione. Supponiamo ancora che un utente voglia eseguire un'operazione di scrittura contattando uno dei server di una delle due regioni del cluster, a questo punto il sistema può scegliere di:

a) accettare di modificare i dati del db, facendosi poi carico dell'onere di comunicare la variazione apportata ai dati agli altri nodi del cluster quando possibile,
b) rifiutare di eseguire l'operazione richiesta.

Nel primo caso non si garantisce la consistenza poiché tacitamente si accetta la possibilità che altri utenti, durante l'intervallo di tempo in cui le problematiche di rete impediscono il diffondersi a tutti i nodi del cluster delle nuove modifiche apportate al database, possano leggere dei valori non aggiornati. Bloccando invece l'operazione di scrittura richiesta si sceglie di salvaguardare la consistenza a scapito della disponibilità, poiché l'azione di modifica viene impedita e viene quindi negato all'utente l'accesso ai dati malgrado il sistema sia ancora operativo.

In presenza di un partizionamento pertanto non è possibile garantire egualmente consistenza e disponibilità di dati, per poter assicurare a priori entrambe queste proprietà è necessario impedire il verificarsi di

partizionamenti, in questo modo però si viola la terza proprietà, ovvero quella di partition tolerance.
In un sistema distribuito sono pertanto disponibili tre differenti approcci, poiché è possibile scegliere di garantire:

- Cosistency ed Availability (CA): in questo caso la proprietà non soddisfatta è la tolleranza ai partizionamenti, quindi i dati sono sempre accessibili ed i valori letti dagli utenti sono sempre consistenti ma il sistema non può continuare ad operare se le connessioni fra i nodi non sono tutte quante attive e disponibili. Fra gli altri esempi di DBMS che hanno adottato questa politica ci sono quelli relazionali, che garantiscono consistenza e disponibilità dei dati prevedendo però comunemente di mantenere l'intero dataset su singolo server.
- Availabiliy e Partition Tolerance (AP): è ciò che accade per molti DBMS NoSQL, in cui la consistenza è "eventuale", si accetta cioè che in determinati intervalli di tempo, a causa di difficoltà di comunicazione fra i nodi del cluster, possano avvenire letture di valori non consistenti perché non aggiornati. La propagazione delle modifiche apportate ai dati verrà estesa ai nodi del cluster che ne mantengono delle copie inconsistenti perché non aggiornate non appena le condizioni della rete renderanno possibile lo scambio di informazioni necessario, ma le operazioni, siano esse letture o scritture, non vengono bloccate, l'accesso ai dati non è mai negato agli utenti. Se i collegamenti fra i server sono tutti quanti attivi, comunque, anche la consistenza è salvaguardata e garantita.
- Consistency e Partition Tolerance (CP): in questo caso viene riconosciuta la priorità della consistenza sulla disponibilità dei dati, quindi le operazioni di lettura riceveranno sempre in risposta valori aggiornati e coerenti, ma in caso di partizionamenti della rete alcune porzioni dei dati potrebbero non essere accessibili.

Dato che una buona parte dei DBMS NoSQL nasce proprio per poter operare con semplicità ed efficienza in ambito distribuito appare chiaro, a questo punto, che tali sistemi non possono conformarsi alle proprietà ACID,

fondamentali invece per ogni sistema relazionale e considerate per anni ed anni la sola politica possibile nella gestione delle transazioni e dunque delle operazioni sui dati. La tolleranza ai partizionamenti per questi DBMS costituisce infatti una proprietà fondamentale, che difficilmente potrà essere sacrificata a favore di altre garanzie che potrebbero, in un contesto altamente decentralizzato, non essere strettamente necessarie. Alla tradizione (*ACID transaction*s) il NoSQL contrappone spesso un modello un po' più flessibile comunemente indicato con l'acronimo BASE, per esteso: Basically Available Soft-state Eventual Consistency. Questo modello consente ai sistemi di sacrificare parzialmente la consistenza a favore della disponibilità dei dati, massima importanza viene infatti riconosciuta al fatto di non rifiutare o ritardare l'esecuzione delle operazioni richieste dagli utenti, le modifiche vengono accettate ed applicate al db anche a costo di perdere momentaneamente la sincronizzazione con gli altri server del cluster. I sistemi che adottano questo tipo di approccio tendono ad essere più semplici e più rapidi e, a differenza dei DBMS relazionali, dimostrano un atteggiamento "ottimista", in quanto si basano sulla convinzione che, anche se in un secondo tempo, il servizio ritroverà sempre la consistenza.

In ogni caso i modelli ACID e BASE non si escludono necessariamente a vicenda, alcuni sistemi offrono la possibilità di scegliere quale approccio adottare per mezzo della definizione di file di configurazione e la tipologia di API adottate. È anche possibile scegliere di conformarsi alla logica operazionale tipica dei RDBMS in alcune aree chiave e adottare invece un approccio differente altrove, a seconda di quelle che sono le nostre esigenze. Uno dei risultati più apprezzabili raggiunti dalla diffusione del NoSQL è infatti la possibilità di scelta. Abbiamo ora a disposizione una molteplicità di tecnologie differenti, approcci dissimili con i quali è possibile e necessario confrontarsi per poter compiere una scelta consapevole ed efficace. Un'altra caratteristica comune ai DBMS NoSQL individuata da Dan McCreary e Ann Kelly in "Making Sense of NoSQL. A guide for managers and the rest of us" è infatti il carattere innovativo di ciascuno di essi, il fatto di aver costruito delle alternative valide e specializzate al modello relazionale, a quello che per anni ed anni è stato il solo modello di memorizzazione, elaborazione e recupero dei dati effettivamente utilizzato.

Un ulteriore elemento di distacco rispetto ai sistemi di gestione di basi di dati tradizionali può essere riconosciuto nel fatto che i prodotti di tipo NoSQL normalmente non prevedono le operazioni di join, largamente utilizzate invece nei sistemi relazionali, che sappiamo essere complesse ed onerose e quindi potenzialmente dannose dal punto di vista prestazionale.

I sistemi NoSQL possono consentire inoltre una rappresentazione de-normalizzata dell'informazione, in cui dati fra loro correlati vengono per lo più raccolti e memorizzati insieme assegnando a specifici campi valori complessi. Si contravviene così ad una delle caratteristiche primarie del modello relazionale: la prima forma normale (*first normal form*, 1NF), che prescrive che il valore memorizzato in ciascun attributo di ogni record debba necessariamente essere un *unico* valore *atomico*, ovvero semplice ed indivisibile. Tale pratica può essere adottata per rendere più semplici le elaborazioni future dei dati, consentendo ad esempio di ottenere tutte quante le informazioni fra loro correlate con una sola e veloce operazione di lettura così come di poterle aggiornare con una sola operazione di scrittura. Al contempo però non va dimenticato che questo tipo di approccio può avere delle implicazioni negative, implicazioni che debbono essere tenute presenti nel decidere come strutturare effettivamente i propri dati.

Un vantaggio non trascurabile dei DBMS NoSQL è dato anche dal risparmio, in termini economici, che le aziende possono trarre dall'uso di questi sistemi piuttosto che dei tradizionali data stores relazionali. Si tratta infatti per lo più di implementazioni open-source e, per di più, grazie al supporto offerto allo scaling orizzontale, non si ha necessità di acquistare hardware particolarmente performante e costoso per poter mantenere prestazioni elevate. Le buone performance di questi sistemi sono favorite anche da un utilizzo efficiente della RAM e degli indici distribuiti, come ricordato da Rick Cattell in "Scalable SQL and NoSQL Data Stores", pertanto i risultati ottenuti possono essere decisamente apprezzabili anche facendo uso di hardware ordinario.

Apprezzabile inoltre è il costante impegno profuso dalle community di sviluppatori che hanno dato vita a questi DBMS per migliorarli ancora, ampliandone e perfezionandone le funzionalità.

Possiamo anche sottolineare che, sebbene la sigla "NoSQL" non si riduca ad indicare semplicemente dei sistemi che evitano di fare uso di questo linguaggio di interrogazione di database, un altro aspetto comune alla maggioranza dei DBMS che si raccolgono comunemente sotto tale nome è dato proprio dal fatto che non fanno uso dello Structured Query Language (SQL), sebbene possano fare uso di linguaggi simili ad esso, come ad esempio il Cassandra Query Language (CQL), linguaggio di interrogazione adottato dal DBMS orientato alle colonne (*column-oriented*) Cassandra. La somiglianza fra i nuovi linguaggi ed il più che affermato SQL in realtà è ragionevole e certamente non casuale, facilita infatti l'apprendimento di tali linguaggi da parte di coloro che, essendo già inseriti nel settore dell'elaborazione dei dati, conoscono già approfonditamente l'SQL e sono in grado di utilizzarlo con semplicità.

Se il modello relazionale si fondava sul concetto di relazione, rappresentata comunemente attraverso tabelle di righe e colonne, i sistemi NoSQL si basano invece su concetti e strutture dati molto differenti fra di loro, organizzando e strutturando i dati secondo modelli profondamente diversi l'uno dall'altro. Proprio il discostarsi dal tipo di rappresentazione ed organizzazione delle informazioni caratteristico dei sistemi relazionali ha segnato un ulteriore passo in avanti nel superamento di quelle caratteristiche proprie del modello relazionale che venivano spesso percepite come limitanti. Uno dei problemi che era stato necessario affrontare infatti nello sviluppo di applicazioni che interagissero con db relazionali riguardava il mapping fra le strutture dati su disco, gestite dal DBMS, e quelle in memoria, gestite invece dall'applicazione. La connessione fra il mondo relazionale e quello delle applicazioni e delle loro strutture dati, sempre più spesso basate sui concetti della programmazione ad oggetti (*object-oriented programming, OOP*), è infatti tutt'altro che semplice ed immediata. Il termine "*impedance mismatch*", in una metafora che riprende concetti propri dell'ingegneria elettrica, fa riferimento in modo particolare al distacco che separa il mondo relazionale da quello della programmazione orientata agli oggetti, ormai enormemente diffusa e radicata nel settore dello sviluppo di software. La risoluzione di questo disallineamento richiede uno sforzo sia di tipo progettuale sia implementativo, che rischia di complicare e

rallentare la produzione e lo sviluppo delle applicazioni. I modelli di dati su cui si basano i sistemi NoSQL invece si prestano più facilmente all'interazione con le applicazioni orientate agli oggetti e le strutture dati che esse gestiscono ed utilizzano, soprattutto grazie alla flessibilità che li caratterizza, distaccandosi dalla rigidità tipica degli schemi del mondo relazionale, e alla possibilità di memorizzare valori complessi, comune agli oggetti dell'OOP ma estranea alle relazioni. Il mismatch tra modello relazionale e OOP è stato considerato per molto tempo dagli sviluppatori di software come uno dei principali limiti dell'utilizzo di RDBMS, tant'è vero che negli anni novanta si era creduto in una possibile destituzione del modello relazionale dal ruolo centrale e di indiscussa supremazia di cui indubbiamente godeva nel settore della gestione ed elaborazione dei dati a favore dei database ad oggetti che, come la programmazione ad oggetti, in quel periodo nascevano ed iniziavano a diffondersi. In realtà, sebbene il paradigma a oggetti sia effettivamente riuscito ad imporsi nel mercato del software, i DBMS orientati agli oggetti (*Object Database Management System, ODBMS*) non hanno avuto altrettanta fortuna e non hanno saputo imporsi su quelli relazionali.

A dire il vero la gravità del cosiddetto impedance mismatch è ora mitigata dai molteplici framework ORM (*Object-Relational Mapping*) che consentono di semplificare l'interazione fra database ed applicazioni ma il problema non può ancora dirsi completamente risolto.

1.4 Classificazione dei sistemi NoSQL

Come abbiamo già detto il movimento NoSQL offre una ricca gamma di soluzioni fra le quali è possibile scegliere sulla base della specifica classe di problemi che si ha necessità di affrontare. Questi sistemi si distinguono sulla base dell'approccio adottato nella rappresentazione, memorizzazione e gestione dell'informazione, adottando strutture dati differenti, offrendo funzionalità distinte, più o meno complesse, seguendo differenti politiche nella salvaguardia della consistenza e della disponibilità dei dati, così come nella gestione delle informazioni in ambienti distribuiti e replicati. La varietà di soluzioni attualmente disponibili può essere organizzata e

suddivisa in alcune "classi" principali di sistemi, si tratta delle quattro tipologie seguenti:

- database chiave/valore (*key/value store*),
- database orientati alle colonne (*column-oriented store* o *column family store*),
- database a grafo (*graph store*),
- database orientati al documento (*document-oriented store*).

Il modello chiave-valore è forse il più semplice dei quattro che stiamo per analizzare. Come è facile intuire le informazioni vengono memorizzate come coppie chiave-valore, dove ciascuna chiave individua univocamente il valore a cui è associata e costituisce la sola via di accesso ad esso. Questi DBMS infatti permettono di accedere ai dati esclusivamente a partire dalle loro chiavi e non prevedono la possibilità di costruire indici secondari (*secondary index*), il solo indice presente è quello costruito sulla chiave che consente di avere accesso in modo rapido e semplice alle informazioni mantenute nel db. Semplicità e genericità vengono indicate da Dan McCreary e Ann Kelly in "Making Sense of NoSQL. A guide for managers and the rest of us" come due delle caratteristiche vincenti che hanno fatto di questi sistemi una scelta appropriata ed accorta per affrontare un certo range di problematiche e contesti, come per la realizzazione di sistemi di caching o hash table distribuite. Altri vantaggi dati dall'uso di questo tipo di DBMS sono l'affidabilità e la scalabilità dei dati, la portabilità e le elevate prestazioni per operazioni semplici, quali ad esempio l'inserimento o la cancellazione di una o più coppie chiave-valore. Allo stesso tempo però si tratta di soluzioni poco adatte se si ha necessità di eseguire sui dati elaborazioni complesse, perché mancano di un meccanismo di interrogazione ampio ed avanzato, posseduto invece da altre categorie di sistemi NoSQL come ad esempio i database orientati al documento.

A differenza dei database relazionali i dati memorizzati non devono essere necessariamente semplici ed atomici, il "valore" di una coppia chiave-valore può infatti accogliere anche tipi di dato complessi, come ad esempio le liste. I "valori" possono infatti essere memorizzati dal sistema come generici "oggetti binari di grandi dimensioni" (*binary large object, BLOB*), indipendentemente dall'effettiva tipologia di informazione di ciascuna

coppia. In questo modo il sistema può memorizzare qualsiasi tipo di dato (immagini, pagine web, documenti, video, file XML, stringhe…) continuando a memorizzare e trattare ciascuno di essi semplicemente come un dato di tipo BLOB, e come tale verrà poi restituito come risultato delle operazioni di lettura eseguite dagli utenti, sarà dunque compito delle singole applicazioni individuare il tipo di informazione ricevuto. Questa flessibilità si estende, anche se in maniera più limitata, anche alle "chiavi", che possono assumere diverse forme come, ad esempio, quella di stringhe generate automaticamente dal sistema, URL di pagine web o percorsi di file. Nella figura 1.1 è possibile vedere alcuni esempi di possibili coppie chiave-valore di diverse tipologie.

	Key	Value
Image name →	image-12345.jpg	Binary image file
Web page URL →	http://www.example.com/my-web-page.html	HTML of a web page
File path name →	N:/folder/subfolder/myfile.pdf	PDF document
MD5 hash →	9e107d9d372bb6826bd81d3542a419d6	The quick brown fox jumps over the lazy dog
REST web service call →	view-person?person-id=12345&format=xml	<Person><id>12345</id .</Person>
SQL query →	SELECT PERSON FROM PEOPLE WHERE PID="12345"	<Person><id>12345</id .</Person>

Figura 1.1. Esempio di coppie chiave-valore di un possibile key-value store. [7]

Un esempio di DBMS chiave-valore è Project Voldemort, sistema open-source scritto in Java ed introdotto nel 2009. Voldemort offre supporto allo sharding automatico dei dati che possono essere distribuiti fra più nodi (fisici e/o virtuali) con un numero *n*, definibile dagli utenti, di copie di ciascun dato. Nuovi nodi possono essere inseriti nel cluster o, al contrario, alcuni di essi possono essere rimossi senza che questo metta in difficoltà il sistema che riesce automaticamente a riadattarsi alla nuova configurazione del cluster. Se uno o più server dovessero subire dei guasti o non essere più attivi Voldemort potrebbe individuare automaticamente il problema e procedere, laddove è possibile, a metterne in atto il recovery.

Lo sharding e la replicazione dei dati salvaguardano le performance e la disponibilità dei dati, in più ogni nodo è indipendente da tutti gli altri, non si hanno pertanto né single point of failure né nodi centrali fondamentali per la coordinazione dell'intero cluster. La consistenza però non è garantita, in

quanto le operazioni di aggiornamento vengono estese alle *n* repliche dei dati distribuite fra i nodi del cluster in modo asincrono, sono pertanto possibili letture di valori inconsistenti perché non ancora aggiornati. È tuttavia possibile aumentare la probabilità di ottenere valori aggiornati leggendo la maggioranza delle copie del dato e scegliendo poi fra i valori ottenuti quello più recente.

Accanto a Project Voldemort Rick Cattell, nell'articolo "Scalable SQL and NoSQL Data Stores", porta ad esempio di DBMS chiave-valore anche altri sistemi, fra i quali possiamo ricordare Riak, sviluppato da Basho Technologies. Per le sue caratteristiche Riak viene a volte descritto come un DBMS orientato al documento ma Cattell ha preferito presentarlo come DBMS chiave-valore avanzato perché, sebbene possegga, così come Voldemort, anche funzionalità e peculiarità normalmente non diffuse fra i sistemi chiave-valore, allo stesso tempo è privo di importanti caratteristiche proprie dei database document-oriented. Riak non supporta infatti alcun indice se non quello costruito sulla chiave primaria e consente soltanto ricerche eseguite a partire dalla chiave primaria, mentre i sistemi orientati al documento normalmente offrono possibilità decisamente più ampie, basti pensare all'impossibilità di eseguire, in Riak, query di range, e permettono la costruzione di molteplici tipologie di indici. Una caratteristica di Riak che tuttavia lo porta ad essere simile ad un sistema orientato ai documenti è legata alla modalità di rappresentazione ed organizzazione delle informazioni prevista da questo DBMS. Ciascun elemento può infatti essere rappresentato nel formato JSON, acronimo di *JavaScript Object Notation*, che permette a ciascun elemento, chiamato da Cattell "oggetto", di essere composto da più campi e che rende tali oggetti molto simili ai *documenti* memorizzati da DBMS quali MongoDB e CouchDB, tanto più che possono essere organizzati in bucket, così come i documenti di un DBMS document-oriented vengono tipicamente raggruppati in collezioni (*collection*). Campi ammessi o necessari per ciascun oggetto sono definiti a livello del singolo bucket.

L'immagine 1.2 mostra un esempio di oggetto in Riak.

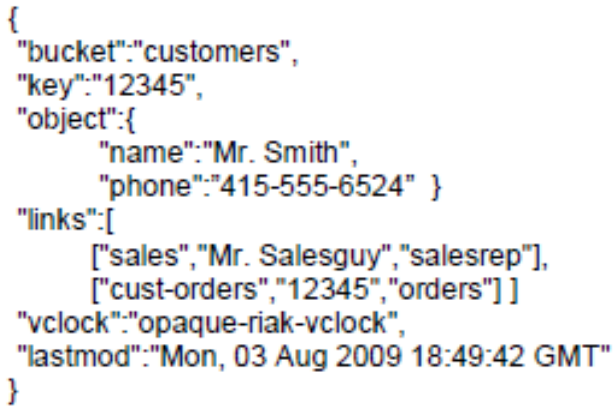

Figura 1.2. Esempio di oggetto in Riak, rappresentato nel noto formato JSON. [8]

In questo oggetto campione possiamo notare, fra gli altri, un campo "links". Caratteristica peculiare di Riak è infatti quella di poter memorizzare e rappresentare i collegamenti esistenti fra gli oggetti, ad esempio quelli che si hanno fra un ricercatore e gli articoli da lui redatti o fra un insegnante ed i corsi da lui tenuti in un certo anno accademico. La presenza del campo link, e dunque di questa riproduzione dei legami logici esistenti fra gli oggetti, riesce a mitigare il fatto che non sia concessa la costruzione (e dunque neppure l'utilizzo) di secondary index.

Così come Voldemort anche Riak utilizza un derivato del "controllo di concorrenza multiversione" (*multi-version concurrency control, MVCC*) per gestire gli aggiornamenti in cui i vector clock vengono associati ai singoli oggetti per mantenere un ordinamento fra le differenti versioni dell'oggetto presenti nel cluster. Pertanto, sebbene Riak ammetta la presenza di repliche dei dati inconsistenti per determinati intervalli di tempo, grazie ai vector clock il sistema è in grado di individuare la presenza di valori non sincronizzati e di risolvere il problema estendendo gli ultimi aggiornamenti apportati ai dati anche alle copie non ancora modificate.

Riak supporta non solo la replicazione dei dati ma anche, come la maggior parte dei sistemi NoSQL, la loro gestione in ambito distribuito ed offre un meccanismo di map/reduce che consente la suddivisione del lavoro fra tutti quanti i nodi disponibili.

L'attrattiva maggiore dei sistemi chiave-valore è costituita dalla possibilità di scaling e dalle performance elevate che questi sistemi sono in grado di offrire, grazie alla semplicità delle modalità con cui si ha accesso ai dati.

Questo vale anche per i DBMS orientati alle colonne, quali ad esempio HBase, HyperTable e Cassandra.

I sistemi orientati alle colonne riprendono la forma tabellare tipica dei database relazionali, anche questi DBMS raccolgono infatti i dati in tabelle bidimensionali, costituite da righe e colonne. Ci sono però delle sostanziali differenze, primo fra tutti il fatto, suggerito dall'appellativo assegnato a questa classe di DBMS, che le informazioni non sono in questo caso memorizzate per righe ma per colonne. Si sovverte quindi in qualche modo la concezione di tabella tipica dei database relazionali, contrapponendo ad un modello che pone al centro dell'attenzione le singole righe (row-oriented) l'idea di concentrarsi invece sulle singole colonne (column-oriented). Il confronto fra queste due differenti concezioni è ben rappresentato dalla figura 1.3.

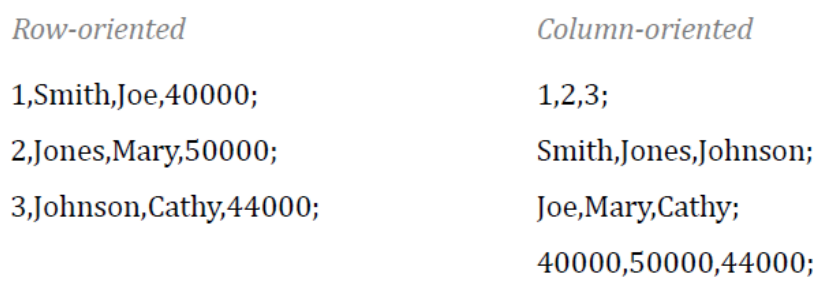

Figura 1.3. Confronto fra organizzazione dei dati row-oriented e column-oriented. [9]

I sistemi orientati alle colonne si lasciano alle spalle anche la rigidità tipica dei loro predecessori basati sul modello relazionale, le righe di una stessa tabella non sono infatti costrette a possedere esattamente gli stessi attributi, ciascuna riga è costituita esclusivamente dalle colonne che le sono necessarie ed è possibile aggiungerne di nuove in qualsiasi momento. In questo modo si ha un'evidente risparmio di memoria, soprattutto nel caso di tabelle sparse, e si rende il database estremamente più agile e più flessibile, caratteristiche che sappiamo essere attualmente molto apprezzate. In questi sistemi inoltre il nome di ciascuna colonna deve ancora essere univoco e, compatibilmente con i sistemi relazionali e differentemente, ad esempio, da DBMS orientati al documento come MongoDB, normalmente righe e colonne di ciascuna tabella accolgono esclusivamente tipi di dati semplici,

non è possibile cioè memorizzare un intero record in una singola cella di una tabella.

Ispirati e guidati dall'enorme successo ottenuto da BigTable, questi DBMS sono progettati per raggiungere alti livelli di scalabilità, riuscendo a gestire grandi quantità di dati suddividendoli su più server, sebbene ancora nessuno di questi sistemi sia riuscito ad eguagliare i risultati invidiabili ottenuti dal sistema sviluppato da Google a partire dal 2004. Rick Cattel, che si riferisce a questi DBMS con l'appellativo di "extensible record store", sottolinea come, per rafforzare ed aumentare la scalabilità, essi possano ricorrere contemporaneamente ad entrambe le tipologie di partizionamento: quella orizzontale e quella verticale. Il partizionamento orizzontale (*horizontal partitioning*), o *sharding*, prevede di distribuire su server distinti i record di un database, in altre parole, in un comune sistema relazionale, di suddividere fra i nodi del cluster le *righe* di ciascuna tabella; il partizionamento verticale (*vertical partitioning*) al contrario prevede di suddividere i dati di uno stesso record fra più server, la distribuzione delle informazioni fra i membri del cluster avverrebbe quindi, in questo caso, non per righe ma per colonne.

Nel caso dei DBMS column-oriented le righe sono spesso distribuite fra più server utilizzando come chiave di sharding la primary key e, a differenza di Riak dove la ripartizione fra i nodi viene determinata sulla base di una funzione hash, comunemente la suddivisione delle righe all'interno del cluster avviene per range. In questo modo si ha un livello di casualità decisamente inferiore nella distribuzione delle righe fra i vari nodi disponibili e ciò rende più efficiente l'esecuzione di query di range relative alla chiave primaria, poiché l'esecuzione di queste interrogazioni può essere isolata ad una specifica porzione del cluster invece di dover coinvolgere tutti i server.

Le colonne vengono invece organizzate in "gruppi di colonne" (*column group*) predefiniti e ripartite all'interno del cluster sulla base di questa suddivisione. Definendo i gruppi in cui raccogliere le colonne di una tabella gli utenti possono quindi specificare quali colonne sarebbe preferibile mantenere insieme e quali è invece possibile e conveniente separare, assegnandole a nodi differenti del cluster. Al concetto di "gruppi di

colonne" Cassandra, sistema column-oriented scritto in Java che è stato progettato inizialmente da Avinash Lakshman e Prashant Malik per Facebook e distribuito poi attraverso la licenza libera Apache License 2 a partire dal marzo del 2009, affianca anche quello di "*supercolumn*". Il singolo database, chiamato "keyspace", in Cassandra è organizzato in famiglie di colonne (*column family*), ossia un insieme di colonne semplici o "super". Quando si parla di supercolumn si fa riferimento ad un ulteriore livello di raggruppamento di colonne, quindi ad un ulteriore livello di profondità e complessità nella singola tabella. Ciascuna famiglia di colonne deve contenere elementi omogenei, non è possibile cioè che includa sia colonne semplici che "super" e, indipendentemente da quante siano le colonne coinvolte, ogni operazione di scrittura applicata ad un'unica column family viene sempre eseguita in modo *atomico*.

Da molti punti di vista Cassandra rispecchia le caratteristiche tipiche dei DBMS della sua categoria, come abbiamo già ricordato propone infatti una strutturazione delle colonne in gruppi (che in questo contesto prendono il nome di column family), consente alle singole righe di possedere un numero variabile di colonne, evitando così di riproporre il vincolo di uno schema rigido definito a livello delle singole tabelle, offre inoltre supporto al partizionamento e alla replicazione dei dati, permettendo l'individuazione ed il recovery in modo automatico di eventuali fallimenti più o meno gravi avvenuti nei nodi del cluster. Come molti altri sistemi, fra i quali possiamo ricordare HyperTable e HBase, altri popolari DBMS column-oriented scritti rispettivamente in C++ e Java, Cassandra prevede che gli aggiornamenti vengano dapprima applicati in memoria centrale, per poi memorizzarli in modo permanente su disco solo in un secondo momento.

Anche per questo DBMS è opportuno parlare di "eventual consistency" anzi, il modello di consistenza adottato da Cassandra si dimostra più debole di quello previsto da molte altre soluzioni NoSQL. Prevede infatti una distribuzione asincrona degli aggiornamenti fra le copie dei singoli dati e non fa uso di nessun tipo di lock, diversamente da HBase, ad esempio, che si serve invece di lock a livello di riga per realizzare transazioni di scrittura, che applichino cioè modifiche ai dati attraverso inserimenti, aggiornamenti (tramite operazioni *put*) e/o cancellazioni (*delete*).

Cassandra è però in grado di rispondere anche ad esigenze più marcate di consistenza, mette infatti a disposizione delle politiche di quorum che possono essere configurate dagli utenti in modo adeguato alle proprie necessità. Con un fattore quorum elevato sarà infatti necessario ottenere un numero altrettanto elevato di repliche di un singolo dato affinché l'operazione di lettura possa considerarsi riuscita, in questo modo è possibile leggere in ogni caso valori consistenti, anche se non tutti i nodi possiedono ancora la versione aggiornata del dato richiesto.

Cassandra è stato inizialmente sviluppato all'interno di Facebook ed è ancora utilizzato da questo social network, sebbene abbia avuto successo anche come soluzione open-source. Fra gli altri celebri utilizzatori di questo DBMS si annoverano anche Twitter e Digg.

DBMS largamente utilizzati nei social network sono quelli che adottano il modello a grafo, poiché per questi sistemi la rappresentazione delle associazioni e dei collegamenti logici esistenti fra le informazioni in essi memorizzate è il cuore pulsante, l'elemento cardine, di ogni database, sono pertanto adatti alla memorizzazione di dati caratterizzati da una complessa rete di connessioni e relazioni. La struttura dati di base per questi sistemi è, ovviamente, il grafo. Le informazioni vengono quindi rappresentate sotto forma di nodi, che memorizzano e descrivono i singoli dati, ed archi, che modellano le relazioni esistenti fra di essi. Spesso i nodi rappresentano elementi del mondo reale, potrebbero ad esempio modellare persone, pagine web, organizzazioni, i computer di una rete, svariate tipologie di prodotti, account di un social network, o addirittura le cellule di un organismo. I collegamenti e le connessioni fra elementi di questo tipo possono essere numerosi ed estremamente complessi e vengono rappresentati come archi che vanno a connettere fra di loro coppie di nodi. Un aspetto fondamentale dei sistemi a grafo è che essi non solo permettono di rappresentare le relazioni esistenti fra i dati ma anche, e soprattutto, di associare a ciascuna di queste correlazioni una descrizione dettagliata, permettendo cioè di tenere traccia non solo dell'esistenza di una qualche connessione fra due elementi della realtà modellata e rappresentata in un certo database ma anche della sua natura, della ragione della sua esistenza, del momento in cui è stata creata e così via. Tanto ai nodi quanto agli archi possono infatti essere

associate delle informazioni, sotto forma di *proprietà* (*property*). Come affermano Dan McCreary e Ann Kelly in "Making Sense of NoSQL. A guide for managers and the rest of us", se i key-value store rappresentano le informazioni attraverso due campi distinti, per l'appunto una chiave ed un valore per ogni dato, i sistemi a grafo fanno invece uso di tre "campi" fondamentali: i nodi, gli archi e le proprietà. Esistono in realtà molte varianti al modello di base, ci sono DBMS, come FlockDB, che non consentono di associare né ai nodi né agli archi informazioni aggiuntive, mentre altri sistemi consentono agli elementi costitutivi del grafo di mantenere delle informazioni per mezzo di coppie chiave-valore più o meno complesse. Ne è un esempio Neo4J, graph store open source sviluppato dalla Neo Technology. Ecco un'immagine, la figura 1.4, tratta dalla documentazione ufficiale di Neo4J, che presenta in modo chiaro ed efficace gli elementi di base di questo DBMS.

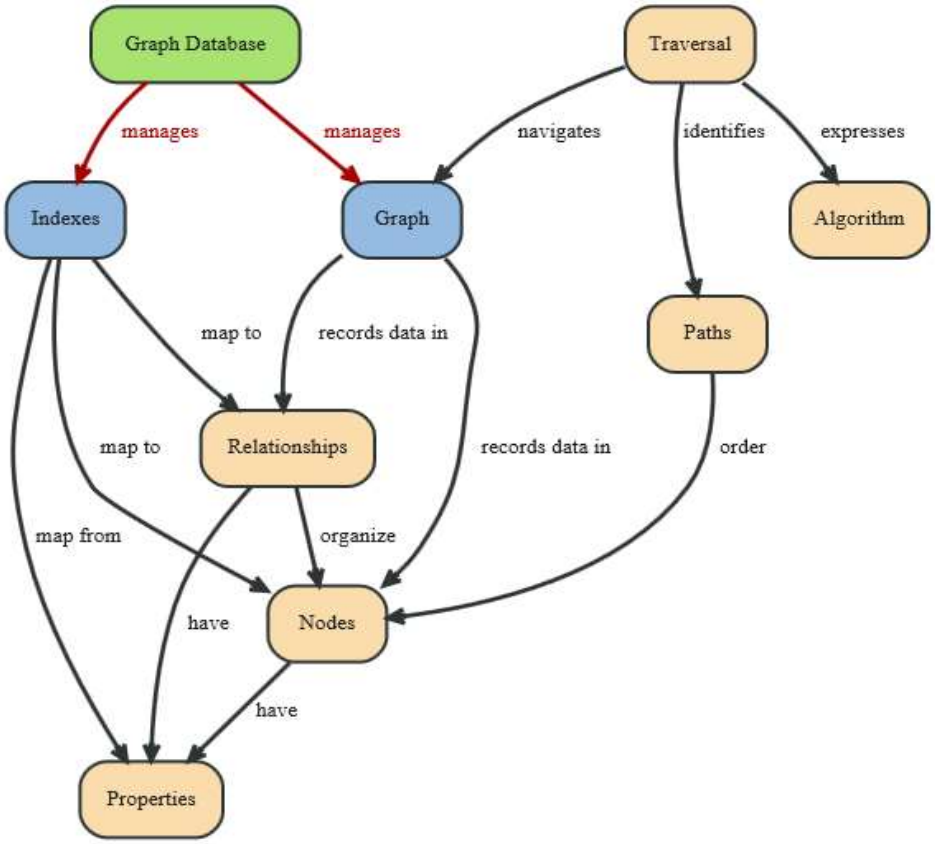

Figura 1.4 Elementi di base di Neo4J. [10]

Le proprietà, in Neo4J, sono delle coppie chiave valore dove la chiave è costituita necessariamente da una stringa non vuota mentre il valore può

essere costituito da un qualsiasi tipo primitivo di Java (boolean, byte, short, float, double, char ecc…), da una stringa o da un array. Sulle proprietà assegnate a nodi ed archi è possibile costruire degli indici, facilitando così la scrittura e l'esecuzione delle query. Una volta completata la costruzione di un indice su una certa proprietà (operazione che viene eseguita in background) tutte le query che operano su di essa sfrutteranno automaticamente l'indice disponibile, incrementando anche in modo sensibile le prestazioni, senza che l'utente debba richiederne esplicitamente l'utilizzo al momento della scrittura della query. Sempre in automatico ed in modo trasparente all'utilizzatore gli indici vengono mantenuti costantemente aggiornati dal sistema, che si occupa di estendere anche ad essi le modifiche apportate a nodi e relationship aventi proprietà indicizzate.

Un'altra possibilità offerta da Neo4J per migliorare le prestazioni nell'esecuzione di query di ricerca su grafo è costituita dall'uso delle *label*. Ogni label ha un nome (una stringa) ed un ID, che è invece di tipo int. Le label consentono di creare dei sottoinsiemi dei nodi di un grafo associando ad essi una "etichetta" che li caratterizzi. Uno stesso nodo può possedere un numero qualsiasi di etichette ma può anche esserne privo. È ad esempio possibile utilizzare le label "Company" e "product" per distinguere i nodi che rappresentano le aziende da quelli che modellano invece i loro prodotti, in questo modo una query del tipo "quali sono gli altri produttori con cui sono in contatto le aziende clienti della mia compagnia?" non dovranno navigare tutto il grafo ma si limiteranno a considerare i soli nodi aventi la label "Company". Poiché le label possono essere aggiunte ed eliminate liberamente in qualsiasi momento, possono essere utilizzate anche per codificare degli stati transitori dei nodi, supponendo quindi di avere un grafo dove ogni nodo rappresenta gli utenti di una certa applicazione distribuita è possibile utilizzare le label ad esempio per distinguere gli utenti attualmente on-line da quelli off-line.

Alle relationship non è possibile associare label ma, come abbiamo già visto, possono avere anch'esse, come i nodi, delle proprietà. Ciascuna relationship inoltre possiede uno ed un solo *tipo*, individuato univocamente dal suo nome, e collega fra loro due nodi non necessariamente distinti

secondo una *direzione* specifica. Un esempio molto semplice è fornito dalla figura 1.5.

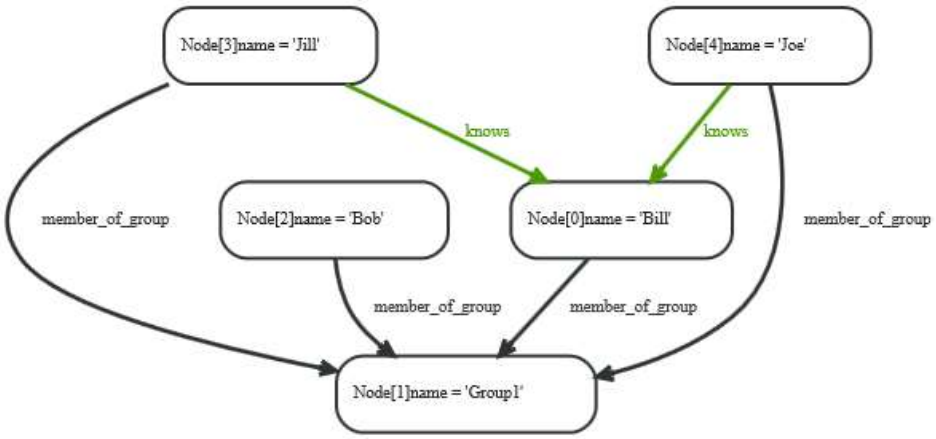

Figura 1.5. Esempio di grafo in Neo4J. [10]

In questa immagine possiamo vedere sei relationship a due delle quali è associato il tipo "knows" mentre le quattro restanti esprimono l'appartenenza ad uno specifico gruppo e sono quindi caratterizzate dalla tipologia "member_of_group".

Sebbene i grafi considerati da Neo4J siano *orientati*, in realtà i suoi archi possono essere percorsi in entrambi i versi, la direzione di ciascuna relationship non pone dunque limiti alle possibilità di muoversi da un nodo ad un altro navigando il grafo, ma viene presa in considerazione soltanto se necessario.

Il navigare il grafo muovendosi fra i suoi nodi seguendo le relationship esistenti fra di essi secondo uno specifico algoritmo viene indicato, in Neo4J, con il termine "traversal". Il risultato di una simile operazione costituisce un "percorso" (*path*), ossia una sequenza di nodi fra loro adiacenti. Un percorso di lunghezza 0 è costituito da un unico nodo mentre uno di lunghezza n (con $n > 0$) coinvolge $n+1$ nodi (non necessariamente tutti quanti distinti) collegati l'uno all'altro, in sequenza, da delle relationship.

I DBMS a grafo in generale sono ottimizzati per memorizzare in modo efficiente le informazioni sotto forma di nodi ed archi ed anche per consentirne la navigazione con prestazioni elevate, permettendo di risolvere in modo rapido ed efficace query che in un RDBMS tradizionale avrebbero

richiesto invece l'esecuzione di join complessi e dannosi dal punto di vista prestazionale. Neo4J mette a disposizione un linguaggio di interrogazione dichiarativo, Cypher, che consente di scrivere con facilità ed eseguire in modo efficiente anche query molto complesse di ricerca e di modifica dei dati memorizzati in nodi e relationship. Differentemente da quanto accade per la maggior parte dei DBMS NoSQL, in Neo4J ogni query che apporti delle modifiche ad un grafo deve essere eseguita all'interno di una transazione, concepita anche in questo caso come un'unità logica indivisibile la cui esecuzione può solo riuscire in modo completo oppure fallire.

Il "ciclo di vita" di una transazione in Neo4J prevede quattro fasi fondamentali:

1. l'apertura della transazione,
2. l'esecuzione delle query di modifica, inserimento o cancellazione, comunque operazioni che vadano in qualche modo ad intervenire sui nodi, sulle relationship del grafo e sulle loro proprietà, richiedendo eventualmente di apportare delle modifiche anche agli indici precedentemente costruiti.
3. La transazione viene segnalata come completata con successo oppure come transazione da annullare. Nel primo caso verrà, nella fase successiva, eseguito il commit e quindi tutte le modifiche, finora presenti esclusivamente in memoria centrale, verranno riportate su disco e rese permanenti; nel secondo caso si procederà invece con un rollback e nessuna delle variazioni apportate durante l'esecuzione della transazione permarrà nel database.
4. Infine si ha la chiusura della transazione. Solo a questo punto la transazione viene considerata formalmente conclusa e quindi solo in questo momento vengono rilasciati i lock acquisiti durante l'esecuzione della transazione così come la porzione di memoria destinata a mantenere i dati elaborati e gestiti dalla transazione stessa.

Neo4J riconosce nelle proprietà ACID tipiche dei sistemi relazionali la base fondamentale per la salvaguardia della consistenza e dell'affidabilità dei

dati, dunque, differenziandosi dalla stragrande maggioranza dei sistemi NoSQL, continua ad incapsulare le sue operazioni in transazioni che godono di tali proprietà. Per fornire anche un'elevata disponibilità dei dati e per evitare l'esistenza di un single point of failure che possa, in caso di guasti, danneggiare il sistema rendendolo non più operativo, Neo4J propone la costruzione di un cluster per la replicazione dei dati gestito secondo una politica master-slave dove il master può ricevere sia operazioni di scrittura che di lettura mentre a tutti gli altri nodi del cluster vengono inviate esclusivamente letture. In caso di guasti avvenuti in uno qualsiasi degli slave sarà possibile sostituire il server danneggiato semplicemente clonando uno degli slave ancora attivi, se invece il problema dovesse interessare il master sarà comunque possibile mantenere il cluster operativo innalzando a tale ruolo uno qualsiasi degli altri server.

Questa strategia viene presentata nella documentazione ufficiale di Neo4J con l'appellativo di "high availability cluster" ed è solo una delle tecniche offerte per ottenere fault tolerance, disponibilità ed affidabilità dei dati.

Un altro aspetto molto curato da questo DBMS è sicuramente quello delle performance. Le operazioni di lettura vengono gestite in modo da poter sfruttare al meglio l'hardware disponibile, avendo a disposizione più processori è possibile eseguire più operazioni di lettura in contemporanea, inoltre tali operazioni non prevedono l'acquisizione di lock né vengono bloccate o ritardate dal sistema. Si elimina in questo modo ogni pericolo di deadlock in lettura e non si ha bisogno di incapsulare in transazioni query che accedano ai dati senza modificarli in alcun modo. Le operazioni di scrittura vengono invece, come abbiamo già visto, normalmente eseguite in vere e proprie transazioni ACID, in realtà però Neo4J offre anche un'altra possibilità: il cosiddetto "Batch Inserter". Questa modalità è ottimizzata per operare su grandi moli di dati, prevede di agire direttamente sui file del database, scrivendo i dati in modo sequenziale e senza riportare le scritture eseguite sui log. In questo modo si possono ottenere prestazioni sensibilmente migliori, allo stesso tempo però si perdono le garanzie offerte dalle proprietà ACID delle transazioni, pertanto questa modalità di scrittura è utilizzabile solo in presenza di un unico thread di scrittura attivo nel sistema. In generale poi le interrogazioni condotte in Neo4J sono ottimizzate

per navigare fra i nodi dei grafi che racchiudono le informazioni di ciascun db nel modo più efficiente possibile. Tanto più che Cypher è un linguaggio dichiarativo, non procedurale, pertanto richiede di specificare *che cosa* si desidera ottenere come risultato della nostra interrogazione, la scelta di *come* ottenerlo è demandata al sistema che opera in modo tale da salvaguardare e massimizzare il più possibile le prestazioni.

I DBMS a grafo si dimostrano, in generale, piuttosto differenti rispetto alle altre tipologie di soluzioni NoSQL. Questo distacco è dato soprattutto dal fatto che la maggioranza dei sistemi NoSQL riconosce un ruolo centrale alla necessità di scalare orizzontalmente, arrivando spesso a fornire supporto automatico allo sharding. L'importanza di operare su cluster ha influenzato tali modelli, portandoli a gestire record con dimensioni anche notevoli caratterizzati da connessioni piuttosto semplici, al contrario i database a grafo non nascono con questa forte spinta verso lo scaling orizzontale e ciò ha avuto un evidente impatto sul modello dei dati su cui si basano, portandoli a gestire record di piccole dimensioni ma caratterizzati da correlazioni strette ed anche molto complesse che rendono difficile la distribuzione dei dati su più macchine. I sistemi a grafo pertanto si adattano con fatica allo scaling orizzontale, per poter aumentare le prestazioni in lettura ricorrono spesso alla replicazione dei dati su più nodi, accrescendo così la capacità complessiva di ricerca e di lettura del sistema, allo stesso tempo però, in questo modo, si ha bisogno di estendere gli aggiornamenti a tutte quante le repliche dei dati e di gestirne la sincronizzazione. Dunque l'effettiva distribuzione del carico di lavoro fra più server viene ottenuta solo in lettura, non anche in scrittura.

L'ultima delle tipologie di DBMS NoSQL che dobbiamo trattare è quella orientata al documento, o *document-oriented*. Si tratta di DBMS in cui l'unità primaria di informazione è costituita da documenti (*document*), comunemente rappresentati mediante formati semplici e noti come JSON, BSON (ossia *JSON Binario*) e XML (acronimo di eXtensible Markup Language). Un documento è incredibilmente simile ad un oggetto dei moderni linguaggi di programmazione, può infatti avere più campi (*field*)

contenenti valori tipizzati che possono essere semplici o complessi (come documenti innestati). In Apache CouchDB, ad esempio, ogni documento può includere un numero qualsiasi di coppie campo-valore, dove il "campo" possiede un nome univoco ed il "valore" può essere costituito da svariati tipi di dato, come testi, booleani, cifre o liste. Ad ogni documento inoltre CouchDB associa dei metadati, gestiti dal sistema, e consente di avere associati degli *allegati* (*attachment*). Possiamo considerare come "allegato" ed associarlo come tale ad un documento svariate tipologie di informazioni, dalle immagini ai file audio o video, ai documenti word, alle stringhe e altro ancora. Fra le informazioni associate in ogni caso ad un allegato si ha: un valore identificativo, il numero di byte che lo compongono ed il tipo MIME, che consente di identificare la natura delle informazioni contenute nell'allegato.

Un elemento comune a questi sistemi e, più in generale alla maggior parte delle soluzioni non relazionali, è legato alla flessibilità e quindi alla mancanza dell'imposizione di schemi rigidi ed immodificabili. In questo sono vicini anche a molti database a grafo, fra i quali si annovera lo stesso Neo4J che nella documentazione ufficiale viene presentato come DBMS "schema-optional", per sottolineare il fatto che la presenza di uno schema ben definito non è obbligatoria ma soltanto opzionale, è l'utente cioè a decidere eventualmente di farne uso, magari per esigenze di modellazione. Anche in CouchDB ogni database può contenere documenti con strutture differenti, caratterizzati cioè da un insieme di campi variabile, non rigidamente definito a livello di database. Elemento comune, in CouchDB, a tutti quanti i documenti è la presenza di un identificatore (ID) univoco all'interno del database di appartenenza ed il campo _rev, fondamentale per la corretta esecuzione di operazioni di scrittura in ambito concorrente. La modifica di un documento in CouchDB infatti non prevede l'acquisizione di nessun lock, il documento su cui si vuole operare viene semplicemente letto dall'applicazione utente che ha richiesto di eseguire la modifica, la copia del documento ottenuta viene quindi modificata e poi salvata nuovamente nel database di origine. Il valore del campo _rev è identificativo di una specifica "versione" del documento, per questo motivo nello scrivere una query di modifica rivolta ad un certo documento è necessario specificare anche il

valore aggiornato del campo _rev di tale documento, valore che viene modificato dal sistema non appena l'operazione di scrittura viene accettata e dunque eseguita.

Se il valore di _rev indicato nella query è inesatto (o se è assente) viene segnalato un errore e la scrittura viene bloccata, poiché la versione a noi nota del documento è probabilmente obsoleta e quindi procedendo con la modifica andremmo a sovrascrivere dei valori a noi sconosciuti, rischiando di commettere degli errori e portare il database in uno stato inconsistente. Supponiamo che un'applicazione utente P1 carichi un certo documento per modificarlo e che poi un'altra applicazione P2 legga una copia dello stesso documento, la modifichi e salvi per prima nel database la versione aggiornata. Quando P1, una volta modificata la propria copia del documento, tenterà di rendere permanenti nel db le modifiche da lei apportate l'operazione di salvataggio verrà negata, evitando così il verificarsi di un lost update. Per risolvere il problema P1 potrà leggere la versione aggiornata del documento e ripetere l'applicazione delle modifiche desiderate (con conseguente cambiamento anche del valore assegnato al campo _rev) per procedere poi finalmente con la memorizzazione del risultato ottenuto nel database. La tecnica utilizzata da CouchDB per gestire le modifiche concorrenti ai documenti raggruppati nei suoi database è un controllo della concorrenza multiversione (MVCC) applicato ai singoli documenti, in modo simile a quanto avviene in sistemi come Project Voldemort e Riak. Un esempio di documento in CouchDB viene proposto nella figura 1.6.

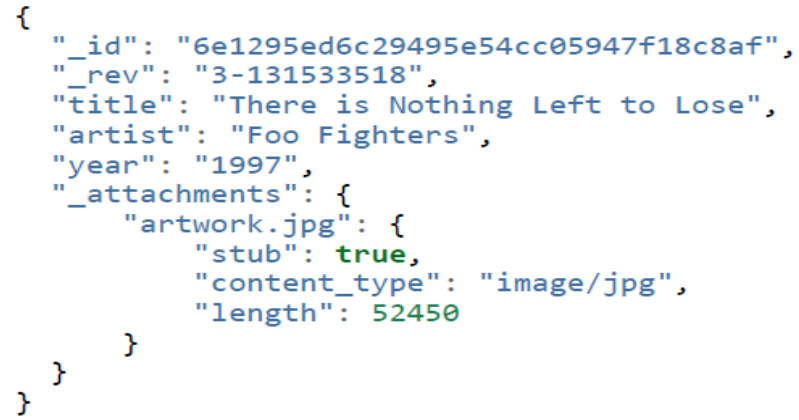

Figura 1.6. Esempio di documento JSON in CouchDB con un solo attachment di cui vengono mostrati unicamente i metadati. [11]

A differenza di quanto accade in altri sistemi NoSQL, come ad esempio Riak, i DBMS orientati al documento permettono solitamente la costruzione di indici su ogni campo dei documenti di un certo database o (se, a differenza di quanto accade in CouchDB, tale livello di organizzazione e raggruppamento di documenti esiste nello specifico sistema document-oriented considerato) di una certa collezione, mettendo spesso a disposizione degli utenti svariate tipologie di indici. MongoDB, noto DBMS document-oriented, ad esempio permette di scegliere fra indici costruiti su un unico campo (*single field index*), indici costruiti su più campi (*compound index*), indici di testo (*text index*), i cosiddetti *mulikey index*, ovvero indici costruiti su un campo i cui valori sono costituiti da array, ed indici hash. Gli indici di MongoDB possono inoltre avere importanti proprietà:

- possono essere sparsi: cioè non considerare necessariamente tutti i documenti di una certa collection ma esclusivamente quei document che possiedono il campo indicizzato, evitando così di sprecare memoria associando un valore null ad ogni documento che sia privo di tale campo. Si tratta di una possibilità fondamentale dal momento che anche MongoDB non prescrive una struttura fissa per i documenti di una stessa collezione.
- Possono imporre un vincolo di unicità: gli *unique index* impediscono infatti l'esistenza di valori duplicati per lo specifico campo su cui sono costruiti. Successivi tentativi di creazione di nuovi documenti che associno al campo indicizzato dei valori già presenti nella collection verranno negati, salvaguardando il vincolo imposto al momento della creazione dell'indice. In caso di indici composti l'unicità si riferisce alla combinazione di valori assegnati in ciascun documento all'insieme di campi indicizzati, pertanto, supponendo di avere uno unique index costruito sulla coppia di campi "name" e "grade", nella nostra collection potrebbero esistere due documenti aventi lo stesso nome o (in alternativa) lo stesso voto ma non è consentito che i valori di entrambi i campi siano coincidenti.
- Possono specificare un TTL, ossia un tempo di vita limitato, per i documenti: i cosiddetti *TTL index* possono essere utilizzati per far sì che i documenti di una certa collezione possano essere rimossi

automaticamente allo scadere di un determinato intervallo di tempo. Si tratta di una possibilità estremamente utile per tutti quei campi di applicazione in cui si vuole che le informazioni memorizzate restino disponibili soltanto per un certo periodo, come accade ad esempio per i log o per le informazioni associate ad una specifica sessione.

Un'ulteriore tipologia di indice disponibile in MongoDB sono i cosiddetti "geospatial index". Si tratta di indici molto particolari introdotti per consentire l'esecuzione efficiente di query "spaziali", ossia interrogazioni del tipo seguente: "quali sono le pizzerie più vicine a me in questo momento?" oppure "quali autofficine ci sono qui vicino in un raggio di 2km?". In generale dato un certo punto X, caratterizzato da una latitudine ed una longitudine, in MongoDB è possibile sfruttare un indice "spaziale" per determinare ad esempio gli *n* documenti (dotati di campi che li collocano nello spazio) più vicini ad X memorizzati in una specifica collection. La query dell'esempio 1.1 è un prototipo di una possibile query di tipo "spaziale", se eseguita restituirebbe infatti i dieci documenti della collection "legacy2d" con collocazione geografica più vicina al punto di cooridate (-73.96, 40.78), ordinati in base alla distanza da tale punto, dal più vicino al più lontano.

```
db.legacy2d.find({location: {$near: [-73.96, 40.78
]}}).limit(10);
```

Esempio 1.1. Query spaziale applicata alla collection legacy2d.[12]

Una query di questo tipo, per poter essere eseguita, necessita dell'esistenza di uno geospatial index costruito sui documenti della collection "legacy2d", di tipo flat (*2d geospatial index*) o sferico (*2dsphere geospatial index*) che si basano rispettivamente sulla geometria piana e su quella sferica per determinare e calcolare i risultati delle query di tipo "spaziale" che vengono inviate al sistema.

MongoDB consente l'esecuzione di differenti forme di query che richiedono di considerare la collocazione geografica dei documenti, ad esempio l'operatore $geoWithin consente di determinare tutti quanti i documenti di

una specifica collection collocati geograficamente all'interno di una certa area delimitata più o meno complessa e, a differenza di $near, non richiede necessariamente l'esistenza di uno geospatial index sebbene, come è facile immaginare, la possibilità di sfruttare un indice di questo tipo possa rendere più semplice e rapida la ricerca.

Dunque sono possibili query di prossimità e di "contenimento", ma anche di "intersezione" grazie all'operatore $geoIntersect. Capiamo bene quindi come MongoDB, ed in generale i sistemi orientati al documento, siano in grado di offrire un'ampia gamma di funzionalità avanzate che restano invece estranee a molti altri DBMS, fra i quali si annoverano sicuramente gli assai più semplici sistemi chiave-valore, privi di secondary index e che consentono l'accesso ai dati esclusivamente a partire dalla chiave, mentre nei DBMS orientati al documento è normalmente possibile accedere alle informazioni a partire da ciascuno dei campi di ogni singolo documento. La ricchezza del meccanismo di query, che mette a disposizione numerose possibilità di intervento sui dati, permettendo di operare su di essi anche ricerche ed operazioni complesse, unitamente alla genericità di questi sistemi, alla loro elevata scalabilità, alla flessibilità dovuta all'assenza di schemi rigidamente definiti a cui uniformarsi, alla semplicità del mapping fra il modello document-oriented ed il paradigma orientato agli oggetti, fa di questi DBMS le soluzioni NoSQL con più ampia applicabilità, rendendoli estremamente appetibili per una vastissima gamma di situazioni e problematiche.

Continueremo lo studio dei sistemi document-oriented attraverso un'analisi approfondita di MongoDB condotta nel prossimo capitolo, per ora possiamo concludere osservando che il traguardo più importante raggiunto dal movimento NoSQL e dallo sviluppo rapido, ampio, costante e dinamico delle svariate soluzioni da esso proposte è quello di aver dato vita ad una moltitudine di alternative, dando alle aziende e a chiunque altro abbia interesse a gestire ed elaborare grandi moli di dati la possibilità di sfruttare sistemi differenti, con qualità e caratteristiche distinte e peculiari, in ambiti diversi. Con il termine "*polyglot persistent model*" si fa riferimento proprio all'uso congiunto di DBMS differenti con il fine di ottenere un sistema complessivo potente e specializzato, capace di affrontare al meglio

condizioni e problematiche differenti sfruttando in modo abile e consapevole le caratteristiche specifiche di ciascuna delle sue parti.

Capitolo 2 - MongoDB

2.1 Concetti principali

MongoDB, il cui nome trae origine dal termine "humongous" cioè "gigantesco", "colossale", è un DBMS orientato al documento scritto in C++ e sviluppato a partire dal 9 ottobre del 2007 che ha dato vita alla prima release stabile nel febbraio del 2009 mentre la più recente, la 2.6.7, è stata rilasciata il 14 gennaio 2015. Come la maggior parte delle soluzioni NoSQL MongoDB è un sistema open-source, è infatti sottoposto alla licenza libera GNU Affero General Public License versione 3.0 [13] ed è dunque liberamente scaricabile dal suo sito ufficiale. Può operare su tutti i maggiori sistemi operativi (Windows, GNU/Linux, OS X e Oracle Solaris) e offre supporto per svariati linguaggi di programmazione, fra i quali: C, C++, C#, Java, Javascript, PHP, Python e Perl [14]. MongoDB è uno dei DBMS più diffusi e popolari, DB-Engines (nel febbraio 2015) lo presenta infatti come il quarto DBMS più popolare al mondo e come il più noto dei document-oriented data store e, più in generale, di tutti i sistemi NoSQL. Si tratta in effetti di un sistema potente, flessibile, veloce, scalabile in modo agevole ed efficiente con il quale è peraltro semplice iniziare a muovere i primi passi.

Come ogni altro sistema orientato ai documenti anche MongoDB si basa sul concetto fondamentale di documento, ossia un insieme di coppie campo-valore in cui, nel caso di MongoDB, il campo è costituito in ogni caso da una stringa mentre il valore, che è sempre tipizzato, può appartenere ad uno qualsiasi dei tipi di dato supportati da BSON, il formato adottato da MongoDB. Fra questi tipi di dato si hanno: double, stringhe, date, booleani, timestamp, espressioni regolari, simboli, integer a 32bit e a 64 bit, ma anche tipi di dati complessi, come gli array. MongoDB consente anche l'esistenza di documenti innestati e, addirittura, array di documenti memorizzati come valori di specifici campi. MongoDB è sensibile all'ordine in cui si dispongono le coppie campo-valore, i documenti {a: 1, b: "Hello"} e {b: "Hello", a: 1} sono pertanto da considerarsi diversi.

La dimensione massima per un documento BSON è di 16 megabyte, pertanto esiste un limite superiore al quantitativo di dati che è possibile

racchiudere in un unico documento, affinché un singolo documento non possa risultare eccessivamente pesante né per essere caricato in memoria centrale né per essere inviato ad un altro server (si limita quindi il consumo di RAM e di ampiezza di banda dovuti ad un singolo documento). MongoDB offre però in realtà la possibilità di travalicare anche il limite dei 16 megabyte, facendo uso delle API GridFS.
In questo modo i dati da memorizzare non vengono raggruppati in un unico file ma suddivisi in più porzioni (chiamate chunk) e ciascuna di esse viene memorizzata come un document separato. La dimensione di un singolo chunk nella versione 2.4 di MongoDB era di 256k, con il rilascio della release 2.6 tale valore è sceso leggermente, passando a 255k. Oltre che per raggirare la dimensione massima dei documenti BSON si può ricorrere a questa tecnica di memorizzazione e gestione dei dati anche per altri scopi come, ad esempio, per approfittare della possibilità di avere accesso ad una certa porzione di un file senza essere obbligati a caricarlo completamente in memoria centrale.

Se CouchDB organizza i documenti direttamente in database, MongoDB prevede invece un ulteriore livello di suddivisione ed organizzazione dei documenti: le collezioni (collection).
Ciascuna collection è identificata dal suo nome (ovvero una stringa UTF-8) e raggruppa documenti logicamente e strutturalmente simili ma non necessariamente identici. MongoDB non prevede infatti la definizione a priori di uno schema rigido ed immutabile, lascia agli utenti massima libertà nella scelta dei campi da inserire in ciascun documento, indipendentemente dalla struttura dei restanti documenti della medesima collection. Esiste però un'eccezione: il campo _id, necessariamente presente in ogni documento come primo campo. L'_id svolge un ruolo simile a quello della chiave primaria nei database relazionali, identifica infatti univocamente ciascun documento di ciascuna collection. Al momento dell'inserimento di un nuovo documento è possibile assegnare al campo _id qualsiasi valore desideriamo (di ogni tipo di dato BSON tranne gli array), purché ne sia garantita l'univocità, oppure, se nell'operazione di inserimento lanciata non viene riportato esplicitamente nessun valore per tale campo, il sistema ne

calcolerà automaticamente uno. Di default MongoDB assegna al campo _id valori di un particolare tipo di dato chiamato ObjectId, ossia una stringa esadecimale di 24 cifre codificate in 12 byte. I primi 4 byte di ciascun ObjectId rappresentano i secondi trascorsi dal 1 gennaio del 1970, i 3 byte seguenti invece costituiscono un identificativo della specifica macchina in uso, si tratta infatti per lo più del valore ottenuto da una funzione hash applicata all'hostname del calcolatore sul quale è avvenuta la generazione dell'ObjectId. Seguono altri due byte che dipendono invece dall'identificativo di processo (*Process Identifier, PID*) che caratterizza il particolare processo che ha determinato il calcolo dell'ObjectId, permettendo così di rendere univoci anche valori generati sulla stessa macchina. Infine gli ultimi 3 byte permettono di garantire l'univocità anche per ObjectId generati dallo stesso processo ad intervalli di tempo inferiori al secondo, si tratta di un contatore che assume valori crescenti e che consente di creare fino a 16'777'216 differenti identificatori univoci al secondo per ciascun processo di ciascuna macchina. La generazione di tali stringhe di cifre esadecimali è semplice e veloce e, soprattutto, assicura l'univocità anche fra server distinti. MongoDB è stato fin dall'inizio progettato come sistema destinato ad operare su cluster in modo quanto più possibile agevole ed efficiente, per questo fra svariate tipologie di identificatori si è preferito adottare quelli appena descritti di cui è possibile gestire con semplicità e rapidità la sincronizzazione anche fra più nodi.

La figura 2.1 mostra un esempio di documento in MongoDB, comprensivo ovviamente del campo obbligatorio _id e della stringa di caratteri esadecimali ad esso assegnata.

```
{
    _id : ObjectId("538ddc9519a492426476026b"),
    extract : "Although I am not disposed to maintain that the being born in a workhouse, is in itself the most fortunate and
              enviable circumstance that can possibly befall a human being, I do mean to say that in this particular instance,
              it was the best thing for Oliver Twist that could by possibility have occurred.",
    language : "en",
    link : "https://books.google.it/books?id=bMXAAAAYAAJ&printsec=frontcover&dq=Oliver+Twist&hl=it&sa=X&ei=
           GA7eV02eN4ntapnHgNAM&ved=CCMQ6AEwAA#v=onepage&q=Oliver%20Twist&f=false",
    author : {
                name: "Charles",
                surname: "Dickens"
    },
    title : "Oliver Twist"
}
```

Figura 2.1. Esempio di documento in MongoDB. Fra gli altri campi possiamo notare l'identificativo univoco _id ed un documento innestato.

Se da un lato, in MongoDB, i singoli documenti sono organizzati in collection, dall'altro le collection stesse sono riunite in database. Un database infatti raggruppa in sé più collection, memorizzate su disco in un unico insieme di file ed è anch'esso identificato dal nome che gli è stato attribuito dagli utenti. Una stessa istanza di MongoDB può infatti gestire più database, ciascuno del tutto indipendente dai db restanti.

Non è necessario richiedere esplicitamente la creazione di una collection o di un database, perché questa venga eseguita dal sistema è sufficiente "usare" quella collection o quel database come se esistessero già. L'inserimento di un documento in una collection "products" attualmente non presente nel database corrente, ad esempio, determina la nascita della collection e l'inserimento in essa del documento specificato. MongoDB mette in verità a disposizione degli utenti un metodo (`db.createCollection()`) che consente di richiedere esplicitamente la creazione di una nuova collection in un certo db, tale metodo però, di norma, viene utilizzato per costruire particolari tipologie di collezioni: le cosiddette *capped collection*. Si tratta di collection con dimensione fissa che vengono utilizzate come array "circolari": una volta occupata interamente la porzione di memoria riservata alla collection il sistema può continuare ad inserirvi nuovi document sostituendoli a quelli che ne fanno parte da più tempo. Gli aggiornamenti dei documenti raccolti in una capped collection sono consentiti soltanto se non portano al superamento della dimensione originaria dei documenti stessi, in questo modo il sistema impedisce che i documenti possano essere riallocati e garantisce quindi che essi siano in ogni momento allocati su disco nello stesso ordine in cui sono stati aggiunti alla capped collection. Una query che richieda quindi di restituire i documenti di una certa capped collection secondo l'ordine di inserimento degli stessi può essere eseguita senza bisogno di accedere ad essi tramite un indice, ottenendo così prestazioni migliori. Operazioni di scrittura operate su una capped collection possono, in generale, rivelarsi molto convenienti dal punto di vista prestazionale, l'aspetto forse più interessante delle capped collection resta tuttavia legato alla sua "fisionomia circolare", ossia al fatto di poter eliminare, se necessario, i documenti presenti da più tempo nella collection in modo automatico, senza doversene occupare personalmente.

Anche volendo comunque cancellare esplicitamente dei documenti da una capped collection non è possibile, l'unica opzione disponibile è rimuovere completamente la collection. L'uso di capped collection comporta infatti anche dei limiti, che riguardano non solo gli aggiornamenti e la cancellazione di documenti ma anche la capacità di scaling: non è infatti possibile suddividere fra più nodi i dati di una capped collection, la si deve necessariamente mantenere interamente su un singolo server. Le collection ordinarie invece, come approfondiremo più avanti nel capitolo, possono essere distribuite fra più nodi consentendo agli utenti di beneficiare dei vantaggi offerti dallo scaling orizzontale. Lo sharding viene attuato in MongoDB a livello della singola collection, così come ogni operazione, sia essa di lettura o di scrittura, viene sempre applicata ad una specifica collection e, di conseguenza, ai documenti che essa contiene.

MongoDB consente la modifica dei dati attualmente presenti in una specifica collection attraverso inserimenti, aggiornamenti e cancellazioni. Gli inserimenti possono essere effettuati attraverso il metodo `db.collectionName.insert()`, che permette di aggiungere alla collection specificata da "collectionName" un singolo documento oppure un array di documenti. Tale metodo consente anche di scegliere il livello di garanzia desiderato riguardo all'effettivo completamento dell'operazione, scegliendo fra differenti opzioni che costituiscono possibili trade-off fra la rapidità di completamento della scrittura e la sicurezza che tali modifiche vengano davvero applicate e rese permanenti. Stiamo parlando di quelli che MongoDB indica come "*write concern level*", ovvero i quattro livelli seguenti:

- Unacknowledged: in questo caso si massimizza la velocità, sacrificando però le garanzie di applicazione e permanenza nel db dell'inserimento richiesto e, più in generale, della modifica richiesta, dato che i differenti livelli di "write concern" si applicano ad una qualsiasi operazione di scrittura, non solamente agli inserimenti. In effetti in questo caso non si hanno garanzie del fatto che la scrittura sia andata a buon fine, in quanto il sistema non riferisce in alcun modo sulla corretta ricezione dell'operazione di scrittura da parte

del driver che, in ogni caso, cerca solitamente di cogliere il presentarsi di eventuali errori o problematiche.

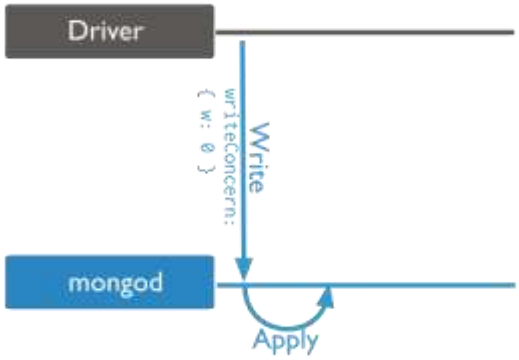

Figura 2.2. Operazione di scrittura con livello di write concern unacknowledged. [12]

- Acknowledged: è il livello utilizzato di default da MongoDB e prevede, da parte della specifica istanza che ha ricevuto l'operazione di scrittura, l'invio di una conferma di corretta esecuzione dell'operazione richiesta che verrà applicata dapprima in memoria centrale e poi, solo in un secondo momento, su disco. In questo caso il client deve restare in attesa di una conferma che comunichi l'avvenuta applicazione della modifica desiderata (o di un'eccezione che segnali il verificarsi di un errore), le prestazioni vengono quindi leggermente danneggiate.

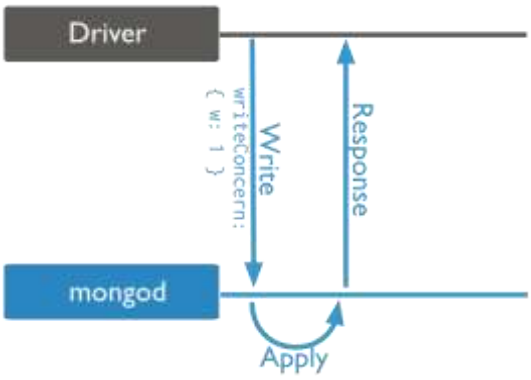

Figura 2.3. Operazione di scrittura eseguita con livello di write concern acknowledged, il livello di default. [12]

- Journaled: questo livello richiede che il journaling sia attivato e prevede che la conferma venga inviata dal sistema soltanto dopo che le modifiche richieste siano state salvate nel file di journal (un log utilizzato da MongoDB per garantire la persistenza delle modifiche apportate ai dati anche in caso di fallimenti a livello di sistema). Le scritture nel file di journal avvengono periodicamente, di default ogni 30-100 millisecondi ma l'intervallo di tempo fra una scrittura e la successiva può essere modificato dagli utenti tenendo presenti gli effetti che tale modifica può avere sulle prestazioni in fase di scrittura e sulla possibilità che degli aggiornamenti vadano persi perché non ancora riportati nel file di journal al verificarsi di un problema imprevisto che causi la perdita dei dati presenti in memoria centrale.

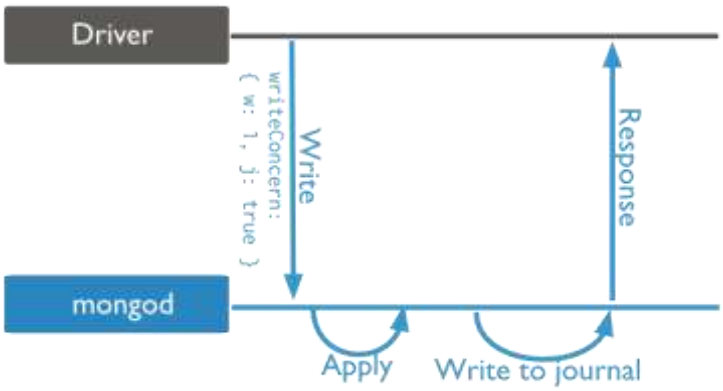

Figura 2.4. Operazione di scrittura eseguita con livello di write concern journaled. [12]

- Replica Acknowledged: questo livello si adatta ad un contesto con replicazione dei dati, cioè ad un cluster di nodi che mantengono più copie degli stessi document per migliorare la disponibilità dei dati, l'affidabilità del sistema e, in alcuni casi, per distribuire fa più server le operazioni di lettura. Adottando questo livello di write concern si richiede al sistema di assicurarsi che la modifica richiesta sia stata propagata non solo al nodo master ma anche ad almeno uno

dei nodi secondari prima di comunicarne la corretta esecuzione.

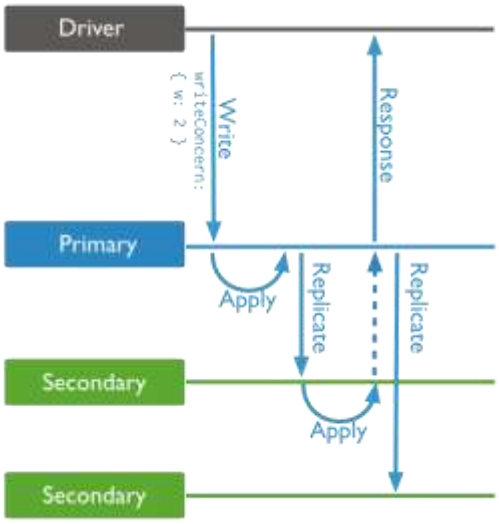

Figura 2.5. Operazione di scrittura eseguita con livello di write concern replica acknowledged. [12]

Per gli aggiornamenti e le cancellazioni, invece, è possibile utilizzare, rispettivamente, i metodi `db.collectionName.update()` e `db.collectionName.remove()`. Entrambi consentono di specificare delle condizioni da utilizzare come filtro per determinare su quali document, fra tutti quelli esistenti nella collection a cui si applica l'operazione di aggiornamento o di eliminazione, è necessario agire. Il metodo `db.collectionName.update()` in particolare consente di indicare: un document per specificare i criteri da considerare per filtare i documenti su cui operare, un altro documento per determinare gli aggiornamenti da applicare e, infine, un terzo document di specifica delle opzioni che definiscono il comportamento del metodo "update". Fra le altre possiamo ricordare le opzioni "multi" e "upsert". Di default `db.collectionName.update()` permette di aggiornare (o sostituire) un solo documento, se vogliamo che la modifica specificata venga applicata a tutti quanti i documenti della collection "collectionName" che soddisfano i criteri di selezione specificati è necessario richiederlo esplicitamente assegnando all'opzione "multi" valore true. Con `{upsert: true}` invece si precisa che, se non esiste alcun documento che soddisfi tutti quanti i criteri di selezione indicati nella query di modifica, il sistema deve inserire

nella collection un nuovo documento, a cui verranno applicate le modifiche indicate nel comando di update. Un prototipo di query di modifica è quello mostrato nell'esempio 2.1.

```
db.inventory.update({item: "TBD2"}, {$set: {details: {"model": "14Q3", "manufacturer": "IJK Co."},category: "houseware"}},{ upsert: true});
```

Esempio 2.1. Una possibile query di aggiornamento con l'opzione "upsert" settata a "true". [12]

Eseguendo tale comando si otterrebbe quanto segue:

- se esistesse almeno un documento con il campo "item" settato a "TBD2" allora in tali documenti il valore del campo "details" verrebbe aggiornato, assegnandogli un documento con i campi "model" e "manufacturer" ed i valori "14Q3" e "IJK Co.", così come il campo "category", a cui si assocerebbe il valore "houseware".
- Se invece nessun documento nella collection "inventory" possedesse il campo "item" con valore "TBD2" verrebbe inserito un nuovo documento contenente i campi "item", "details" e "category" con i valori corrispondenti indicati nella query.

Se di default `db.collectionName.update()` opera su un solo documento, la contrario `db.collectionName.remove()` normalmente elimina tutti quanti i documenti che soddisfano i criteri di selezione specificati dall'utente, è possibile però richiedere, attraverso un particolare flag, che l'eliminazione venga applicata ad un solo documento.
Tanto le cancellazioni quanto ogni altra operazione di scrittura in MongoDB viene eseguita in modo atomico ed indivisibile esclusivamente se applicata ad un unico documento, indipendentemente dalla complessità dei dati che esso contiene. Ricordiamo infatti che il valore assegnato ad un singolo campo può anche essere a sua volta un documento o, addirittura, un array di documenti. Questo consente di risolvere, almeno in parte, le difficoltà legate all'impossibilità di incapsulare le operazioni di scrittura in vere e proprie transazioni come accade invece nei DBMS relazionali, perché se le informazioni su cui si ha necessità di operare, sebbene siano complesse, sono racchiuse in un unico documento (magari contenente uno o più

document innestati) allora qualsiasi operazione di scrittura eseguita su di esse rimane in ogni caso atomica. Il problema nasce quando si ha necessità di modificare più documenti. Il sistema non è in grado di per sé di garantire l'isolamento di un'operazione di scrittura complessa impedendo in ogni caso ad altri utenti di accedere agli stessi dati, in lettura o in scrittura. Se ad esempio dovessimo registrare un incremento di costo per un certo insieme di prodotti, modellati ciascuno come un documento a se stante, durante l'applicazione di tale modifica altri utenti potrebbero leggere il costo non ancora aggiornato di alcuni prodotti oppure accedere ad essi per variarne il valore. Per evitare che questo accada MongoDB mette a disposizione un comando (`$isolated`) che consente di isolare, durante la sua esecuzione, un'operazione di scrittura che coinvolge più documenti. In questo modo nessuno potrà avere accesso ai prodotti interessati dall'aggiornamento finché quest'ultimo non sarà stato portato a termine. Anche utilizzando `$isolated` comunque non si rende un aggiornamento complesso atomico: se un errore dovesse verificarsi quando l'esecuzione dell'operazione di scrittura ha già avuto inizio il sistema non si occupa automaticamente di riportare i documenti già aggiornati ai valori precedenti. Il solo modo per avere atomicità è racchiudere tutte le informazioni da modificare in un solo documento, strutturandolo in modo complesso. Come vedremo in effetti, in MongoDB non è insolito scegliere una rappresentazione de-normalizzata dei dati, che prevede di mantenere unite informazioni che in un database relazionale avremmo dovuto assolutamente separare. Nei casi in cui le informazioni da aggiornare, inserire o cancellare siano distribuite fra più documenti, sappiamo già che l'operazione di scrittura da eseguire non può considerarsi complessivamente atomica, per tentare di emulare il comportamento delle transazioni tipiche dei sistemi relazionali si può però ricorrere ad una pratica suggerita da MongoDB ed indicata con il nome di "two phase commit". Tale pratica richiede fondamentalmente di modellare all'interno del database le singole "transazioni" (intese qui semplicemente come operazioni di scrittura che coinvolgono molteplici documenti) attraverso dei documenti appositi (raccolti in una collection dedicata), che ne descrivano la natura e le caratteristiche, e di associare al contempo a ciascuno dei document che dovranno essere oggetto di modifica delle

informazioni relative alla transazione che su quei documenti deve operare e al suo stato di avanzamento. Mantenendo costantemente aggiornati i dati che descrivono l'operazione da eseguire e quale parte di essa sia già stata eseguita si preserva la possibilità di ricondurre il sistema in uno stato consistente anche a fronte di possibili errori o fallimenti a livello di sistema. In presenza di più applicazioni che accedono in modo concorrente ai dati può essere utile memorizzare per ciascuna "transazione" anche un identificativo dell'applicazione che deve metterla in atto, così da evitare che più applicazioni eseguano per errore la stessa operazione di scrittura. In passato l'esecuzione di operazioni concorrenti era gestita mediante lock imposti sull'intera istanza di MongoDB, a partire dalla versione 2.2 invece a questi lock di tipo "globale" ne sono stati affiancati altri, definiti a livello di singolo database. Questa tipologia di lock è ora utilizzata per la maggior parte delle operazioni, fanno eccezione alcune operazioni che coinvolgono più database e che continuano pertanto ad utilizzare lock globali. Quelli gestiti da MongoDB sono lock *readers-writer* (noti anche con gli appellativi *multiple readers/single-writer* lock o *multi-reader* lock), ossia lock che consentono accesso multiplo in lettura ma che impongono un accesso esclusivo in caso di modifica dei dati, impedendo cioè l'avvenire di altre scritture e di letture finché tale lock non viene rilasciato o ceduto. MongoDB riconosce priorità alle scritture piuttosto che alle letture, in altre parole se due operazioni richiedono di accedere alla stessa porzione di dati una in lettura e l'altra in scrittura il sistema permetterà l'applicazione della modifica ai dati prima di consentirne la lettura. Si parla per questo di lock *writer greedy*. In alcune situazioni un lock acquisito, per operare in lettura o in scrittura, può essere ceduto. Nella versione 2.0 di MongoDB la cessione dei lock era regolata da algoritmi basati su una suddivisione in intervalli di tempo e sul numero di operazioni correntemente in coda per l'acquisizione di lock, sia per letture che per scritture. Ora (a partire dalla versione 2.2) la cessione dei lock è legata alla presenza o meno dei dati da leggere o modificare in memoria centrale. MongoDB utilizza delle euristiche per determinare se è o meno plausibile che un determinato document sia già presente nella RAM e, se l'esito dovesse essere negativo, il lock acquisito dall'operazione P1, che deve accedere ai dati di tale documento, viene

momentaneamente ceduto, viene concesso ad un'altra operazione P2 che deve invece agire su dati già presenti in memoria centrale. In questo modo, mentre P1 recupera i dati di suo interesse dal disco, P2 può operare e concludere la sua esecuzione poi, quando P1 avrà a disposizione in RAM i propri dati, potrà riacquisire il lock e procedere a sua volta. Riprendendo un esempio precedente in cui si aveva necessità di aggiornare i prezzi di un certo insieme di prodotti, tale operazione potrebbe cedere il lock esclusivo ottenuto sui dati fra l'applicazione della modifica desiderata ad un certo prodotto e la sua applicazione al prodotto successivo. Per questo motivo, sebbene l'azione su singolo document resti comunque atomica ed isolata e quindi sia impossibile, leggendo le informazioni racchiuse in un singolo documento, accedere a dati solo parzialmente aggiornati, le operazioni di lettura o di scrittura complesse non garantiscono l'isolation. Di qui la necessità, in alcuni casi, di forzare una gestione differente dei lock dalla quale ottenere maggiori garanzie. È ciò che viene fatto, ad esempio, scegliendo di utilizzare il comando `$isolated`. In questo modo infatti si richiede che il lock ottenuto sui dati venga conservato fino al termine dell'esecuzione dell'operazione, senza cessioni intermedie.

Ma quali sono, in MongoDB, le operazioni che richiedono l'acquisizione di lock? Ovviamente le operazioni di lettura, che richiedono un lock condiviso (più letture possono pertanto essere eseguite simultaneamente), e le operazioni di scrittura, quali inserimenti, cancellazioni ed aggiornamenti, per cui è invece necessario un lock esclusivo. Accanto a tali operazioni di base tuttavia vi sono anche altri casi in cui, per poter procedere, è necessario aver ottenuto un lock sui dati, ad esempio per la costruzione di un indice (se operata in foreground), per la creazione di una capped collection e per l'esecuzione del metodo `db.eval()`, che consente di eseguire codice JavaScript sui server di MongoDB, è necessaria l'acquisizione di un lock esclusivo sull'intero database.

È opportuno specificare che, in presenza di un ambiente distribuito, i lock acquisiti dalle operazioni per poter accedere ai dati conservati in uno specifico nodo del cluster non coinvolgono in alcun modo i server restanti, i cui documenti restano pienamente accessibili per gli altri utenti. Lo sharding, a cui MongoDB offre supporto automatico, consente quindi di

favorire ulteriormente la concorrenza, permettendo l'esecuzione simultanea anche di operazioni di scrittura su porzioni distinte dello stesso database, a patto che si trovino su server differenti.
Anche le tecniche di locking messe in atto dal sistema, così come l'assenza delle transazioni ACID, puntano dunque a massimizzare le prestazioni, acconsentendo a rinunciare ad alcune delle garanzie tipiche dei sistemi relazionali per rendere il sistema quanto più possibile leggero, rapido e prevedibile nelle sue performance [12].

2.2 Modellazione dei dati

MongoDB è, come sappiamo, un DBMS orientato al documento, si distacca pertanto notevolmente dai sistemi relazionali e le profonde differenze che lo separano da essi pongono nuove riflessioni e considerazioni alla base della strutturazione e modellazione dei dati che si vogliono rappresentare e delle relazioni esistenti fra di essi. MongoDB permette di scegliere fondamentalmente fra due differenti modalità di organizzazione dei dati: è possibile optare per una strutturazione "normalizzata" dei dati, in cui informazioni correlate ma logicamente distinte sono mantenute mediante documenti separati, eventualmente raccolti in collection e database differenti, allo stesso tempo però il sistema ammette anche una organizzazione differente, "denormalizzata", in cui informazioni fra loro strettamente connesse possono essere concentrate in un unico document, sfruttando la possibilità di gestire documenti innestati. Nel primo caso, per modellare comunque l'esistenza di un legame concettuale fra più document, avremo bisogno di rappresentare tali connessioni all'interno del database. Questo può essere fatto aggiungendo ai documenti dei campi che mantengano dei valori identificativi dei document ad essi legati. Ad esempio, supponendo di avere un database contenente le due collection "author" e "book", avremo bisogno di legare ciascuna opera letteraria al corrispondente autore. Legami di questo tipo vengono normalmente rappresentati in uno dei modi seguenti:

- è possibile sfruttare semplicemente il valore del campo `_id` che, come sappiamo, identifica il documento a cui appartiene in modo univoco.
- Altrimenti è possibile utilizzare i documenti DBRef. Questi prevedono tre campi (di cui uno opzionale) che consentono di far riferimento ad un altro document non solo attraverso il valore del suo `_id` ma anche del nome della collection e (opzionalmente) del database. In questo caso quindi ciascun document relativo ad un libro comprenderà anche un campo "writer" simile all'esempio 2.2.

```
"writer": {   "$ref": "author",
              "$id":
     ObjectId("5126bbf64aed4daf9e2ab771"),
      "$db" : "archivio_letterario"
           }
```

Esempio 2.2. Rappresentazione di una connessione logica fra due document, un libro ed il suo autore, per mezzo della convenzione DBRef. I tre campi "$ref", "$id" e "$db" codificano, rispettivamente, il nome della collection, il valore del campo _id ed il nome del database di appartenenza del document a cui si vuole far riferimento (ovvero dell'autore). Il campo "$db" è opzionale.

Un'organizzazione denormalizzata degli stessi dati prevedrebbe invece di incapsulare informazioni correlate ad un certo document sotto forma di documenti innestati o array. Riprendendo l'esempio precedente quindi dovremmo includere nel documento di ciascun autore un campo "books" a cui assegnare come valore un array di document che modellino la produzione letteraria completa di quello specifico scrittore. Si tratta, evidentemente, di un'opzione che in un sistema relazionale non sarebbe neppure stata presa in considerazione. In MongoDB, in generale, questo tipo di soluzione può invece essere adottata in presenza di relazioni di contenimento (semantica "*part of*") oppure relazioni "uno a molti" (*one-to-many*), come nell'esempio degli scrittori e delle loro opere. Scegliendo questo tipo di soluzione si favoriscono le prestazioni in lettura, poiché per recuperare tutte le informazioni fra loro correlate sarà sufficiente l'esecuzione di una sola query, mantenendo invece i dati separati avremmo

bisogno di accedere prima al documento di base e poi di sfruttare le informazioni ottenute da esso per poter accedere anche ai document correlati. Mantenendo concentrate le informazioni si può inoltre sfruttare l'atomicità garantita da MongoDB per tutte le operazioni che operino su un unico documento, evitando così le problematiche che possono nascere dall'esecuzione di operazioni complesse di cui il sistema non garantirebbe l'isolamento né l'esecuzione come singola, seppure articolata, unità di elaborazione. Informazioni che debbono necessariamente poter essere aggiornate in modo atomico è bene pertanto che vengano mantenute unite, se possibile, inglobate all'interno di uno stesso documento.

La strutturazione denormalizzata può tuttavia portare con sé una maggiore probabilità, a seconda del tipo di operazioni che si ha necessità di eseguire sui dati, del presentarsi della necessità di riallocare un documento a causa di un aggiornamento che ne abbia accresciuto eccessivamente la dimensione complessiva. MongoDB prevede di dedicare ad un certo documento una porzione di memoria un po' più grande di quella effettivamente necessaria, nel tentativo di evitare successive riallocazioni che possono influenzare negativamente le prestazioni (gli aggiornamenti che causano uno "spostamento" del document a cui sono stati applicati sono ovviamente più onerosi rispetto agli altri) e causare frammentazione, ma questo non sempre è sufficiente. Un ulteriore limite da ricordare è quello che definisce la dimensione massima concessa per ciascun document BSON (ovvero 16MB). In generale i casi in cui è bene optare per una rappresentazione normalizzata dei dati riguardano la modellazione di complesse relazioni molti a molti (*many-to-many*) e tutti quei casi in cui un'organizzazione di tipo denormalizzato causerebbe replicazione dei dati senza però che i benefici ottenuti, ad esempio in termini di prestazioni in lettura, valgano le problematiche che tale duplicazione può portare con sé. La scelta della modalità di rappresentazione dei dati più conveniente è infatti influenzata non solo dalla tipologia di informazioni che si ha necessità di gestire e memorizzare ma anche, ovviamente, dalle elaborazioni che si ha necessità di applicare ad essi.

MongoDB mette a disposizione degli utenti differenti pattern di modellazione che, attraverso l'uso dei riferimenti, organizzano i document

in una struttura gerarchica ad albero. Supponiamo di dover modellare una struttura del tipo rappresentato nell'immagine 2.6, fra i pattern di modellazione proposti da MongoDB abbiamo:

- Tree Structure with Child References: ogni elemento della gerarchia viene rappresentato come un document a sé stante e contenente, fra gli altri campi, un array di riferimenti a tutti gli eventuali documenti figli.
- Tree Structure with Parent References: anche in questo caso si ha un document separato per ciascun nodo dell'albero, ognuno di essi memorizza però un solo riferimento relativo al nodo padre.
- Tree Structure with Nested Sets: i nodi dell'albero vengono considerati nell'ordine in cui verrebbero raggiunti da una visita in pre-ordine dello stesso albero. Durante la visita ciascun nodo si considera attraversato due volte, la prima durante la "discesa" nel suo sottoalbero e la seconda durante la "risalita". Ebbene, ogni nodo dell'albero viene ancora rappresentato come documento a sé stante ma i riferimenti presenti in ciascuno di essi sono tre: l'`_id` del nodo padre e le due posizioni occupate dal nodo nell'ordine di attraversamento, la prima nel campo "left" e la seconda nel campo "right".

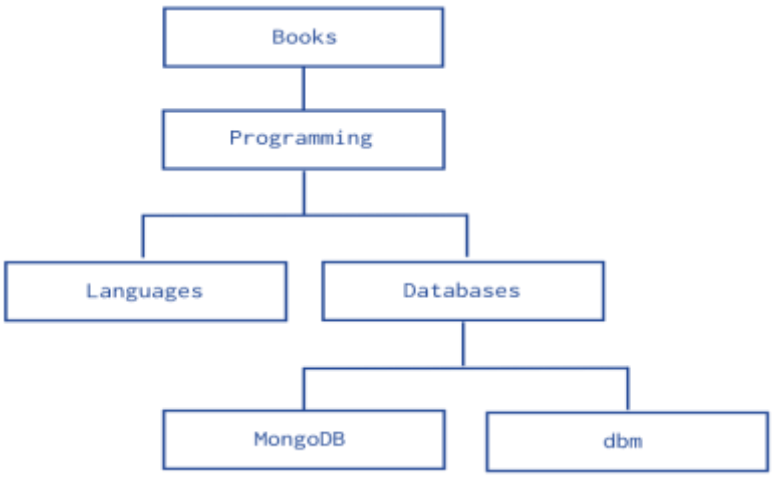

Figura 2.6. Esempio di gerarchia da modellare. [12]

2.3 Aggregazione in MongoDB

Come abbiamo già avuto modo di verificare se alcuni sistemi NoSQL, quali ad esempio molti DBMS chiave-valore, mettono a disposizione modalità di accesso ai dati piuttosto semplici e povere rispetto allo standard offerto dai principali DBMS relazionali, ciò non è vero per MongoDB. Oltre alle ricerche e alle semplici letture dei dati memorizzati in ciascun database questo DBMS mette infatti a disposizioni svariate possibilità e funzionalità più complesse, fra le quali differenti metodologie di aggregazione dei dati. Con il termine "aggregazione" si vuole far riferimento ad un ampio set di elaborazioni applicabili ai dati che consentono, grazie all'attuazione di una procedura specifica, di ottenere un risultato calcolato, raggiunto attraverso una computazione più o meno complessa.

Lo strumento più semplice di aggregazione offerto da MongoDB è costituito da alcuni comandi che consentono di mettere in atto operazioni specifiche e comuni, quali:

- il calcolo del numero di documenti di una collection che soddisfano determinati criteri di selezione,
- la restituzione di tutti e soli i valori distinti assunti da un certo campo fra i document di una data collection che rispettano specifiche condizioni,
- il raggruppamento dei document di una particolare collection che soddisfano i criteri fissati dall'utente. Il raggruppamento può essere definito su uno o più campi, sia realmente esistenti che calcolati, e può prevedere anche la definizione, e dunque l'applicazione, di semplici funzioni aggregate, come il conteggio del numero di elementi in ogni gruppo o la somma (o anche la media) dei valori assunti da un certo campo fra i membri dello stesso gruppo.

Per raggiungere questi semplici obiettivi è possibile far uso rispettivamente dei comandi `count`, `distinct` e `group` o, in alternativa, dei metodi corrispondenti (ovvero `count()`, `db.collection.distinct()` e `db.collection.group()`).

Il comando `count`, ad esempio, possiede una struttura del tipo mostrato nell'esempio 2.3.

```
{count: <collection>, query: <criteri di
selezione>, limit: <limit>, skip: <skip>, hint:
<hint>}
```

Esempio 2.3. Formato del comando count di MongoDB. [12]

Consente pertanto non solo di specificare la collection su cui operare ed eventuali criteri di selezione da applicare ai documenti in essa contenuti, ma anche di specificare (se lo si desidera) un numero massimo di document da prendere in esame (grazie al valore assegnato al campo "limit") ed uno che specifichi quanti document tralasciare prima di dare effettivamente inizio all'elaborazione dei dati (informazione specificata dal valore di "skip"). A partire dalla versione 2.6 di MongoDB inoltre è disponibile anche il campo "hint", grazie al quale si ha la possibilità di indicare al sistema un indice da utilizzare per l'esecuzione dell'operazione. A differenza di `group` il comando `count` può essere utilizzato anche in ambito distribuito, sebbene possa restituire un risultato impreciso in alcuni casi specifici (ovvero se esistono dei "documenti orfani" oppure se è in corso un trasferimento di dati fra più nodi del cluster). Fra questi comandi `group` è il più complesso, il suo utilizzo offre infatti maggiori possibilità, permettendo agli utenti di calcolare in modo semplice e intuitivo molteplici risultati, ma presenta dei limiti, come la già citata impossibilità di utilizzarlo su cluster, il limite massimo di 16MB fissato per la dimensione complessiva del risultato restituito al termine della computazione ed il limite superiore di 20'000 gruppi che si possono in questo modo determinare. In generale questi comandi offrono una facile via di accesso ad elaborazioni dei dati estremamente comuni e ciò li rende indubbiamente utili in molteplici circostanze, sono però operatori *single purpose*, mancano pertanto in flessibilità e ricchezza, soprattutto se confrontati con le restanti metodologie di aggregazione offerte da MongoDB. All'estremo opposto rispetto alla semplice metodologia già presentata si pone MapReduce, un meccanismo facilmente parallelizzabile che offre possibilità decisamente più ampie rispetto all'opzione precedente. Le operazioni di map-reduce comprendono due fasi fondamentali:

- map: una funzione definita dall'utente viene applicata a tutti i documenti della collection che soddisfano determinati criteri di selezione e produce come risultato delle coppie chiave-valore. Infine, in quella che viene a volte indicata come fase di "shuffle", il risultato ottenuto viene raggruppato sui valori delle chiavi.
- reduce: alle liste di valori associate a ciascuna chiave viene applicata un'ulteriore funzione che, come quella di mapping, è costituita da codice JavaScript e che, in generale, si occupa di ridurre ciascuna di tali liste ad un unico valore aggregato. Il risultato finale può essere restituito direttamente all'utente o memorizzato in una collection di output. Se si opta per la visualizzazione inline del risultato si deve rispettare il limite dato dalla dimensione massima consentita per i documenti BSON, fissata per ora a 16MB. In questo caso il risultato è infatti costituito da un unico document contenente un certo insieme di metadati ed un array di documenti innestati che riportano, ciascuno, una chiave ed il valore ottenuto dall'applicazione della funzione di reduce alla lista di valori precedentemente associati a quella stessa chiave. Il vincolo di 16MB complessivi non sussiste invece se si adotta una soluzione alternativa, preferendo dirigere l'output verso una specifica collection. In questo caso infatti ciascuna coppia chiave-valore ottenuta al termine della funzione di reduce può essere memorizzata in un documento a sé stante. Opzionalmente inoltre il risultato conseguito potrebbe essere sottoposto all'applicazione di un'ulteriore funzione (anch'essa composta da codice JavaScript) che consente di modificare ulteriormente i dati ottenuti e concludere l'elaborazione. Nella documentazione ufficiale di MongoDB ci si riferisce a tale funzione con l'appellativo di "*finalize function*".

Un esempio di applicazione di map-reduce è visibile nella figura 2.8 in cui viene mostrato un possibile utilizzo del metodo `db.collection.mapReduce()` ed i passi attraverso i quali esso viene eseguito.

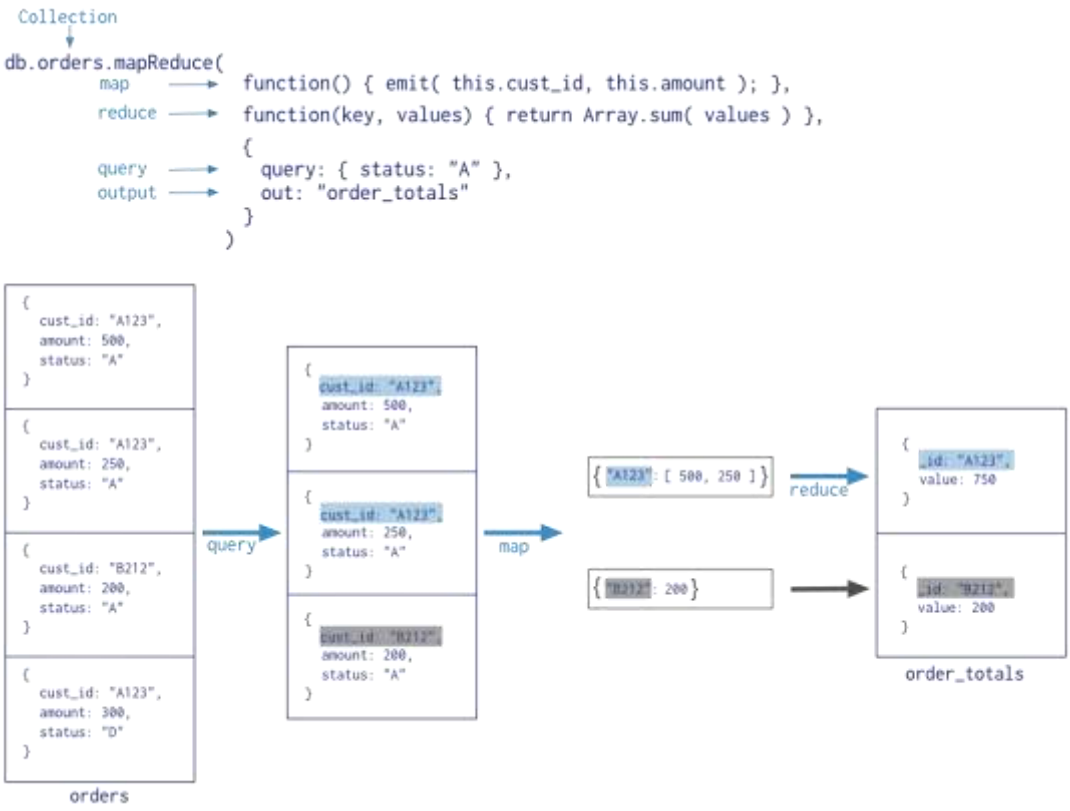

Figura 2.8. Esempio di operazione map-reduce. [12]

I parametri di questo metodo comprendono:

- le funzioni da applicare nelle fasi di map e di reduce, così come un'eventuale funzione cui sottoporre l'output finale per modificare ulteriormente il risultato (finalize function).
- Eventuali condizioni da applicare come filtri di selezione ai documenti appartenenti alla collection a cui l'operazione di map-reduce viene applicata possono essere specificate dall'utente grazie al campo "query" che è possibile indicare nel documento di opzioni che segue le funzioni di map e di reduce fra i parametri del metodo considerato.
- Il campo "out" consente di scegliere come gestire il risultato finale dell'operazione, se visualizzarlo inline oppure dirigerlo verso una collection di output. In questo secondo caso potremmo scegliere di dare origine ad una collection dedicata, non ancora presente nel db, oppure di sfruttarne una creata in precedenza, con la possibilità di utilizzare anche collection che siano distribuite fra più server. Scegliendo di memorizzare il risultato in una collection già esistente

potremo anche decidere se sovrascriverne il contenuto oppure se unire il nuovo risultato con quello già presente nella collection, specificando anche come gestire eventuali document con chiavi duplicate, se sovrascrivendo il nuovo documento a quello già esistente che riporta tale chiave oppure se applicare ancora, ad entrambi i documenti, la funzione di reduce memorizzando infine nella collection il risultato ottenuto da questo ulteriore livello di computazione.

- Grazie al campo "sort" è possibile richiedere che i document di input vengano elaborati secondo un ordinamento specifico per favorire le prestazioni, con la sola restrizione che la chiave su cui si definisce l'ordinamento deve essere indicizzata.
- "Limit" offre invece la possibilità di porre un limite superiore al numero di documenti da elaborare, snellendo anche in questo caso il carico di lavoro e favorendo dunque la rapidità dell'operazione da eseguire.
- In "scope" si indicano invece eventuali variabili globali che debbono essere accessibili tanto dalla funzione di map quanto da quella di reduce e, se esiste, anche dalla finalize function.
- Con il valore assegnato al campo "verbose" decidiamo se dovranno essere visualizzati o meno nel risultato finale dati relativi alle prestazioni ottenute dall'esecuzione della nostra operazione. Il valore di default è "true", queste informazioni vengono quindi riportate automaticamente nel risultato.
- Resta da considerare il campo "jsMode" che consente di specificare se i dati debbono essere convertiti da oggetti JavaScript a documenti BSON già fra la fase di map e quella di reduce o no. Evitando di effettuare questa conversione non sarà necessario neppure riportare poi i dati al formato precedente così da poterli fornire in input alla funzione di reduce. In questo modo si possono pertanto migliorare le performance, MongoDB permette però questo tipo di esecuzione soltanto se il numero di chiavi restituite dalla funzione di map non supera le 500'000 unità. Il comportamento di default, che prevede di tradurre in documenti BSON i risultati restituiti dalla fase di map e

di memorizzarli temporaneamente su disco, consente invece, seppure con prestazioni peggiori, di operare su moli arbitrariamente grandi di dati.

Map-reduce è certamente lo strumento più potente offerto da MongoDB per quanto riguarda le possibilità di aggregazione dei dati, il tallone di Achille di questa metodologia è dato proprio dalla sua complessità e dalle prestazioni che consente di ottenere. In molti casi infatti non si ha bisogno di sfruttare appieno la potenza offerta dalle operazioni di map-reduce e la complessità di questo meccanismo appare eccessiva, portando gli sviluppatori a preferire altre opzioni. Un trade-off apprezzabile fra le due realtà già presentate è costituito dalla cosiddetta *Aggregation Pipeline*, implementata in MongoDB a partire dalla versione 2.2. Si tratta di un framework per l'aggregazione di dati basato sul concetto di *pipeline*, ovvero una sequenza ordinata di operatori che vengono applicati ai dati uno di seguito all'altro per poter calcolare risultati aggregati più o meno elaborati.
Più precisamente in MongoDB si parla di "*stage*" della pipeline, non di operatori. I documenti in input vengono elaborati da ciascuno stage in sequenza, rispettando cioè l'ordinamento specificato al momento della scrittura dell'operazione di cui si richiede l'esecuzione, e lo stesso "livello" (stage) può normalmente comparire più volte (in posizioni distinte) all'interno di una stessa pipeline, tranne per alcune eccezioni. Alcuni stage accettano come operandi delle espressioni, tali espressioni specificano in che modo agire sui document per procedere con l'elaborazione e possono includere vari elementi, fra i quali:

- path field: si tratta del nome di uno dei campi presenti nei document della collection presa in esame preceduto dal simbolo `$`. L'uso di "path field" consente di fare riferimento in una espressione al valore assegnato ad uno specifico campo del document considerato ed elaborato correntemente dalla pipeline ad esempio per confrontarlo con altri valori o per utilizzarlo nel calcolo di risultati aggregati.
- Variabili: MongoDB mette a disposizione dei suoi utilizzatori specifiche variabili di sistema, come CURRENT e ROOT, e ne consente l'utilizzo negli stage della pipeline così come permette di

sfruttare variabili definite precedentemente dall'utente e di avere accesso al loro valore facendone precedere il nome dal prefisso `$$`.

- Valori letterali: sono ammessi valori letterali di ogni tipologia, alcuni di essi però in determinate situazioni potrebbero essere fraintesi dal sistema (come nel caso di una stringa che inizi con `$` o con `$$`), per evitare che questo accada MongoDB offre la possibilità di far uso dell'operatore `$literal` che impone al sistema di non sottoporre a parsing i suoi argomenti e di considerarli semplicemente come dei valori.
- Operatori: MongoDB mette a disposizione svariati operatori, fra gli altri ricordiamo:
    - Operatori booleani: si tratta di `$and`, `$or` e `$not`, tre operatori che mettono in atto la logica booleana e la applicano ai propri operandi valutando come `false` non solo il vero e proprio valore booleano ma anche `null`, lo zero, e `undefined` (tipo di dato ormai deprecato).
    - Operatori aritmetici: comprendono le più comuni operazioni matematiche, ovvero la somma (`$add`), la moltiplicazione (`$multiply`), la divisione (`$divide`), la sottrazione (`$subtract`) ed il modulo (`$mod`). `$add` e `$subtract`, in particolare, possono operare non solo con valori numerici ma anche con le date. Nel caso di `$add` se fra gli operandi considerati figura un valore di tipo `date` i restanti operandi, tutti necessariamente numerici, vengono considerati come millisecondi da sommare alla data specificata. `$subtract` consente invece di considerare in ogni caso solo due operandi ed essi possono essere entrambi numerici, entrambi di tipo `date` oppure una data (che deve essere specificata necessariamente come primo operando) ed un numero. `$subtract` consente quindi non solo di sottrarre fra loro due valori numerici ma anche di operare la differenza fra due date o di detrarre ad una data un certo valore numerico

inteso, anche in questo caso, come espressione di una certa quantità di millisecondi.

- Operatori di comparazione: consentono di confrontare fra loro valori dello stesso tipo ma anche valori di tipologie differenti, MongoDB ha infatti organizzato in una gerarchia ben definita tutti i tipi di dato BSON ed è dunque in grado di confrontarli fra di loro. Nella tabella 2.1 è riportato l'ordine definito ed utilizzato da MongoDB per le comparazioni di tipi di dato distinti.

1	MinKey (internal type)
2	Null
3	Numbers (ints, longs, doubles)
4	Symbol, String
5	Object
6	Array
7	BinData
8	ObjectId
9	Boolean
10	Date, Timestamp
11	Regular Expression
12	MaxKey (internal type)

Tabella 2.1. Tipi di dato BSON disposti in ordine crescente secondo l'organizzazione definita in MongoDB. [12]

MongoDB mette a disposizione: l'operatore `$cmp`, che restituisce 1 se il primo operando è maggiore del secondo, 0 se i due operandi sono uguali e -1 se il primo è minore del secondo; l'operatore di maggioranza `$gt` e quello di minoranza `$lt`, a cui si affiancano `$gte` e `$lte` che si distinguono dai precedenti perché restituiscono `true` anche in caso di uguaglianza fra gli operandi, e l'operando `$eq`, che verifica che i due operandi siano equivalenti, al quale si contrappone l'operando $`ne` che, al contrario, restituisce `true` in caso di non uguaglianza fra gli operandi.

- Operatori insiemistici: si tratta di operatori pensati per agire sugli array considerandoli come insiemi di valori, ignorando

di conseguenza tanto l'ordinamento dei loro elementi quanto eventuali valori duplicati presenti in essi. Se l'operatore utilizzato restituisce un nuovo insieme, in quanto tale il risultato ottenuto non presenterà ripetizioni, eventuali valori duplicati vengono quindi automaticamente rimossi dal sistema. Si tratta di operatori che consentono di mettere in atto le più comuni operazioni fra insiemi, quali unione (`$setUnion`), differenza (`$setDifference`) e intersezione (`$setIntersection`), ma anche operazioni di tipo differente. `$allElementsTrue` ad esempio restituisce `true` se tutti gli elementi presenti nell'operando considerato sono differenti da `false` e da valori che possano essere valutati come tali (`null`, 0, `undefined`).

L'Aggregation Pipeline consente anche l'esecuzione di operazioni complesse, come vedremo è infatti possibile effettuare delle ricerche di testo all'interno della pipeline e fra gli stage che gli utenti possono scegliere di adoperare a partire dalla release 2.4 figura anche `$geoNear`, che restituisce una sequenza di documenti ordinati sulla base della prossimità rispetto ad un certo punto (geospatial point), operando su un piano o su una sfera. `$geoNear` consente di limitare il risultato attraverso i valori assegnati dall'utente a `$limit` e `$num`, che permettono entrambi di fissare un numero massimo di documenti da restituire, oppure considerando una distanza massima dal punto indicato entro la quale devono trovarsi i documenti considerati per poter entrare a far parte del risultato finale o per essere passati allo stage successivo della pipeline. È inoltre possibile sottoporre i documenti esaminati a dei filtri, specificando le condizioni che vogliamo siano soddisfatte come valore del campo "query" del document di opzioni accettato come parametro da `$geoNear`, del quale fanno parte anche i già citati `$limit` e `$num` ed anche `$maxDistance`, che consente di fissare la distanza massima accettata. L'uso di questo stage è sottoposto però a delle restrizioni, può infatti comparire soltanto una volta nella stessa pipeline e deve necessariamente essere il primo fra gli stage che la compongono, richiede inoltre l'esistenza di un geospatial index (flat o

sferico) costruito sul campo recante la collocazione "spaziale" dei singoli documenti. Fra gli altri stage che possono essere utilizzati ricordiamo:

- `$group`: consente di raggruppare i documenti ricevuti in input su un determinato campo o su un insieme di campi. Il risultato è costituito da un document per ciascuno dei gruppi individuati. Ognuno di essi possiede necessariamente un campo `_id` in cui è visibile la combinazione di valori che, nei document di quel particolare gruppo, sono stati assegnati ai campi che costituiscono la chiave attorno alla quale è stato realizzato il raggruppamento. Lo stage `$group` permette anche di fare uso di svariati operatori per calcolare dei valori aggregati più o meno complessi. I cosiddetti "*accumulator operator*" sono operatori utilizzabili esclusivamente all'interno dello stage `$group` e di essi fanno parte:
  - `$sum` e `$avg`: che consentono, rispettivamente, di calcolare la somma e la media aritmetica dei valori numerici ottenuti dall'applicazione di una specifica espressione ai membri di uno stesso gruppo.
  - `$min` e `$max`: consentono di individuare i valori minimo e massimo ottenuti valutando una stessa espressione su tutti i documenti che appartengono al medesimo gruppo.
  - `$push`: restituisce un array contenente i valori ottenuti applicando una espressione definita dall'utente a tutti i document riuniti in uno stesso gruppo.
  - `$addToSet`: è analogo a `$push`, in questo caso però vengono eliminati dall'array risultante eventuali valori ripetuti.
  - `$first` e `$last`: consentono di sfruttare un ordinamento precedentemente imposto ai documenti ricevuti in input, permettono infatti di ottenere, per ogni gruppo, il valore risultante dall'applicazione di una certa espressione esclusivamente al primo (nel caso di `$first`) o all'ultimo (nel caso di `$last`) dei document

che ne fanno parte, ed una simile azione è sensata solo in presenza di un ordinamento significativo.

Se lo si desidera è anche possibile considerare tutti quanti i documenti ricevuti in input come facenti parte di un unico gruppo, così da applicare gli operatori precedenti su tutti quanti i documenti complessivamente. Per raggiungere questo obiettivo è necessario evitare di fornire una combinazione di campi su cui effettuare il raggruppamento. Il campo `_id` del document richiesto come argomento da `$group` è quello riservato alla specifica del campo o dei campi su cui si vuole raggruppare ed è obbligatorio, non può quindi essere omesso ma può assumere valore `null`, ad indicare che i documenti dovranno essere considerati come facenti parte di un unico gande gruppo cumulativo.

- `$limit`: consente di limitare il numero di documenti da restituire al seguente livello della pipeline (se esiste) o come risultato finale. Ad esempio `{$limit: 10}` fa sì che soltanto i primi dieci documenti ricevuti in input dallo stage `$limit` vengano passati a quello successivo, senza variare in alcun modo i campi presenti in ciascun di essi.
- `$project`: al contrario di `$limit`, `$project` consente di proiettare verso il successivo stage della pipeline una versione modificata nel contenuto dei documenti ricevuti in input. Permette infatti di:
    - specificare i campi che si vogliono preservare nei documenti restituiti,
    - scegliere se rimuovere dal risultato il campo `_id`, che è presente di default, ed i valori ad esso assegnati,
    - introdurre nuovi campi ed assegnare ad essi dei valori oppure modificare i valori precedentemente assegnati a campi già esistenti. Questo è possibile settando il campo di interesse con un'espressione oppure un valore letterale specificato attraverso l'uso di `$literal`, visto che l'assegnamento ad un certo campo di un valore numerico

o booleano è inteso in questo specifico contesto come volontà di mantenere o rimuovere quel particolare campo dai document di output. In particolare i valori 0 e `false` vengono utilizzati per richiedere la soppressione del campo mentre valori quali 1 o `true` per ottenere la sua inclusione nei documenti restituiti da `$project`. Con il codice `{ $project: { title : 1 , author : 1 } }` si sta pertanto richiedendo di sopprimere dai documenti di output ogni campo tranne il titolo, l'autore e l'`_id`. In presenza di uno stage `$project` infatti i soli campi preservati nel risultato sono quelli di cui è stata esplicitamente richiesta la visibilità e l'`_id`, restituito di default a meno che non ne venga richiesta esplicitamente la rimozione scrivendo, ad esempio, `{$project: {_id: 0} }`.

- `$match`: grazie a questo stage è possibile filtrare i documenti da passare al successivo stadio della pipeline, per questo motivo è conveniente anticipare quanto più possibile l'inserimento di $match fra gli stage previsti nella pipeline, così da ridurre il numero di document su cui è necessario operare nei livelli successivi. I criteri di selezione che possono essere fissati dagli utenti possono essere più o meno complessi, a partire da condizioni di uguaglianza imposte su un certo campo fino ad arrivare all'uso dell'operatore `$text` per l'esecuzione di ricerche di testo e dunque a filtrare i documenti ricevuti in input sulla base dell'esistenza o meno, in un certo campo dal valore testuale, di una certa parola, una frase specifica o un particolare insieme di termini. Affinché l'uso di `$text` nello stage `$match` sia possibile quest'ultimo deve trovarsi esattamente all'inizio della pipeline. D'altronde un simile posizionamento di `$match` è ugualmente consigliabile, anche nei casi in cui non deve essere posto alcun filtro sul contenuto di campi testuali, poiché permette di sfruttare l'esistenza di eventuali indici per

poter velocizzare l'esecuzione della ricerca e dunque della selezione richiesta.

- `$sort`: consente di fissare un ordinamento per i documenti di input. I document non vengono pertanto né filtrati né modificati da questo stage, ma solo disposti secondo un certo ordine. L'ordinamento può essere fissato, in modo crescente o decrescente, su uno o più campi (reali o calcolati) oppure sul textScore assegnato ai singoli documenti a seguito dell'effettuazione di una ricerca di testo operata tramite l'operatore `$text`. Il textScore è una misura dell'attinenza di un certo document alla specifica ricerca di testo condotta e viene associato automaticamente ad ogni documento che, soddisfacendo le condizioni di selezione della ricerca stessa, entri a far parte del suo risultato. Per poter richiedere, nello stage `$sort`, che l'ordinamento venga effettuato su un certo insieme di campi è sufficiente elencare i nomi di tali campi separati da virgole e seguiti dai due punti (:) e da un valore numerico che specifica il tipo di ordinamento da imporre su quel particolare campo: crescente (in questo caso il valore numerico da usare è 1) oppure discendente (simboleggiato dal numero -1). Se fra gli elementi su cui ordinare figura il textScore è invece necessario far uso dell'operatore `$meta` con la sintassi seguente:

  ```
  { $sort: { newFieldName: { $meta: "textScore" }, … } }
  ```

  L'ordinamento definito sul textScore è necessariamente decrescente.

Questi ed altri stage sono disponibili sia attraverso il comando `aggregate` sia facendo uso del metodo `db.collection.aggregate()`. Questo framework per l'aggregazione dei dati è stato sviluppato dall'equipe di MongoDB ponendo al centro dell'attenzione le performance e la semplicità di utilizzo. Il gran numero di operatori e stage a disposizione degli utenti offrono una buona flessibilità, sebbene comunque inferiore di quella offerta dal meccanismo di Map-Reduce. Come quest'ultimo anche l'Aggregation Pipeline offre la possibilità di operare tanto con collection mantenute su un

unico server quanto su collection distribuite, al contrario dei comandi single purpose come `group` che non possono essere utilizzati in caso di suddivisione dei dati fra più membri di un cluster. Con la release 2.6 di MongoDB sono inoltre stati superati due importanti limiti che fino a quel momento avevano caratterizzato l'Aggregation Pipeline restringendone sensibilmente le possibilità di utilizzo. La prima di queste limitazioni riguardava l'impossibilità di avere risultati con dimensione maggiore di 16MB poiché l'esito dell'elaborazione poteva essere restituito all'utente soltanto inline racchiuso in un unico documento dalla struttura più o meno complessa. Questo problema è ora superabile poiché il comando `aggregate` può restituire un cursore o memorizzare il risultato dell'elaborazione in una collection, pertanto il limite dato dalla dimensione massima per i document BSON può essere raggirato configurando in modo appropriato il funzionamento di `aggregate` così da permettergli, se necessario, di gestire risultati arbitrariamente grandi. Analogamente il metodo `db.collection.aggregate()` è in grado di gestire risultati di qualunque dimensione poiché non prevede la restituzione inline ma tramite cursore. Nelle release precedenti inoltre la quantità di RAM utilizzabile da ciascuno stage della pipeline era limitata a 100MB, un limite piuttosto restrittivo per l'elaborazione di grandi quantità di dati che, se violato, avrebbe causato un errore. Grazie all'opzione `allowDiskUse` è ora possibile, invece, consentire al sistema di far uso di file temporanei su disco, permettendo così la trattazione di moli di dati anche ingenti. Avremo modo di visualizzare alcuni esempi di utilizzo dell'Aggregation Pipeline nel prossimo capitolo, in cui verrà proposta un'analisi dei risultati ottenuti da un insieme di test operati su MongoDB su singolo server ed in ambito distribuito per poterne saggiare e studiare le prestazioni.

2.4 Replication

MongoDB offre la possibilità di duplicare il proprio dataset e mantenere le copie dei dati su server distinti così da evitare l'esistenza di un single-point-of-failure, incrementare la disponibilità dei dati e favorire, se necessario, le prestazioni del sistema attraverso politiche di prossimità geografica. Con il termine *replica set* si fa appunto riferimento ad un insieme di istanze

mongod finalizzate alla gestione di più repliche dei medesimi dati, suddivisi in server primari e secondari. In ogni replica set deve esistere esattamente un server primario (*primary*), tutti gli altri nodi sono considerati secondari e mantengono dei duplicati del dataset gestito dal primary. Il solo membro del replica set abilitato a ricevere le operazioni di scrittura è il server primario che, una volta applicata la modifica richiesta, si occupa di registrare la scrittura effettuata nell'*oplog* (abbreviazione di "operation log"). L'oplog è una capped collection dedicata alla registrazione delle modifiche applicate ai dati in presenza di un replica set ed è proprio copiando il contenuto dell'oplog del server primario che tutti gli altri nodi possono mantenere aggiornato, seppur in modo asincrono, il proprio dataset, riproducendo sulle proprie copie dei dati le operazioni che sono già state applicate sul server primario. Di default anche le operazioni di lettura, non solo quelle di scrittura, sono dirette esclusivamente verso il primary, questo comportamento è però modificabile e adattabile alle esigenze degli utenti e delle loro applicazioni. Ovviamente dirigendo delle letture verso membri secondari non si ha la garanzia di ottenere in ogni caso le versioni più aggiornate dei dati, il solo modo per garantire letture consistenti è mantenere la configurazione di default e limitare le operazioni di lettura esclusivamente al server primario. Allo stesso tempo ampliare ai server secondari (che possono essere molteplici) la possibilità di ricezione delle operazioni di lettura può ovviamente migliorare le performance e la *read capacity* del sistema, ovvero il numero di letture che esso è in grado di soddisfare, in media, in un dato intervallo di tempo. Poiché i membri di un replica set possono appartenere a data center differenti è anche possibile sfruttare, come già accennato, il vantaggio dato dalla possibile vicinanza geografica di un certo server secondario ad un dato utente. Esistono cinque differenti modalità di gestione delle letture fra cui è possibile scegliere nel configurare il proprio replica set e sono:

- primary: è la modalità di default e prevede che tutte le letture vengano inviate esclusivamente al server primario.
- PrimaryPreferred: questa seconda modalità prevede che normalmente le operazioni di lettura vengano eseguite dal server primario e solo nel caso in cui tale nodo sia momentaneamente non

disponibile tali operazioni verranno deviate verso un altro membro del replica set.

- Secondary: si tratta dell'opzione opposta rispetto a quella di default, prevede infatti che le letture vengano suddivise esclusivamente fra i server secondari. Una simile scelta è sensata, naturalmente, solo se si è in grado di tollerare la ricezione di valori datati, non aggiornati. La propagazione di una operazione di scrittura (sia essa un inserimento, una cancellazione o un update) fra i membri del replica set non è, infatti, immediata, dato che i server secondari ripetono sulla propria copia di dati le modifiche già applicate al dataset del server primario solo dopo aver aggiornato il proprio duplicato dell'oplog, mantenuto in una collection chiamata "local.oplog.rs".
- SecondaryPreferred: si tratta della modalità inversa rispetto a "primaryPreferred" in quanto prevede che le letture vengano normalmente dirette verso i server secondari con la possibilità di reindirizzarle verso il nodo primario solo nel caso in cui non vi siano server secondari attualmente disponibili.
- Nearest: in questo caso la caratteristica fondamentale considerata nel decidere quale server dovrà eseguire una certa operazione di lettura è data dalla sua collocazione geografica, le letture vengono infatti in ogni caso dirette al server per cui la latenza risulti essere minima, indipendentemente dal fatto che si tratti di un server primario o secondario.

Una caratteristica dei replica set in MongoDB è quella di essere in grado di determinare automaticamente se un certo server sia o meno raggiungibile ed operativo in un dato istante. I membri del replica set si scambiano infatti periodicamente (ogni 2 secondi) dei ping, dunque l'assenza prolungata di segnale da parte di uno dei server verso gli altri segnala la presenza di un problema, l'impossibilità di comunicare con il server perché non è più attivo o per via di problemi di rete. Se tale server dovesse essere proprio quello primario, una volta trascorsi dieci secondi senza ricevere alcun ping da tale server, i restanti membri del replica set metteranno in atto un meccanismo di "votazione" per elevare uno dei nodi secondari al ruolo di nuovo server primario. In realtà non tutti i membri di un replica set possono partecipare

alla votazione così come non tutti possono essere eletti. Possono esistere dei nodi cosiddetti "non votanti", il numero massimo di membri per un replica set è infatti pari a 12 ma al più sette di questi possono esprimere il proprio voto in caso di elezione, dunque la possibilità di avere dei nodi non votanti è ciò che permette di avere replica set con più di sette elementi. Il server che diverrà il nuovo primary è il primo, fra tutti quelli eleggibili, che riceverà la maggioranza dei voti. In passato, per favorire l'elezione di un certo nodo o di un certo numero di nodi, era possibile consentire ad alcuni membri del replica set di esprimere più di un voto durante le elezioni. Ora questa pratica è deprecata, piuttosto che variare il numero di voti che determinati nodi possono esprimere è consigliabile intervenire sulla priorità. Tanto i server primari quanto quelli secondari hanno infatti associato un valore di priorità che di default è pari ad 1, è però possibile aumentarne il valore se si desidera favorire l'elezione di un particolare nodo, oppure, al contrario, porla a zero per rendere impossibile l'ascesa a server primario di alcuni membri del replica set. Il ruolo giocato dai valori di priorità è fondamentale nell'ambito delle elezioni, il server secondario prescelto dovrà infatti essere il nodo eleggibile con più alta priorità che sia attualmente disponibile nel replica set. Se in seguito inoltre, a causa di variazioni di configurazione apportate ai membri del sistema, dovesse essere presente nel replica set un server secondario con valore di priorità più alto di quello assegnato al nodo primario verrà innescata una nuova votazione per offrire a tale nodo la possibilità di acquisire il ruolo di server primario, a patto che, ovviamente, tale nodo sia eleggibile e dunque anche che il suo valore di optime sia sufficientemente elevato. L'optime di un nodo detiene il timestamp dell'ultimo aggiornamento previsto dall'oplog che sia stato applicato ai documenti da quel particolare server ed è dunque un indicatore del livello di aggiornamento dei dati mantenuti da ciascun nodo.

Accanto a server primari e secondari i replica set possono includere anche un'altra tipologia di componenti: gli arbitri (*arbiter*). Un arbitro non detiene una copia del dataset del server primario e non è eleggibile, il suo unico compito è quello di partecipare, con il proprio voto, alle elezioni che si terranno per scegliere il nodo più appropriato per ricoprire il ruolo del server primario. Gli arbitri vengono utilizzati in presenza di un numero pari di

server, così da poter garantire il raggiungimento della maggioranza al momento delle votazioni senza essere obbligati a mantenere un'ulteriore replica dei dati. Specifici nodi di un replica set potrebbero essere inoltre dedicati al backup o al disaster recovery. Spesso tali server vengono mantenuti celati alle applicazioni utente, si tratta dei coiddetti "hidden member" che mantengono, come ogni altro server primario o secondario, una copia del dataset ma che si distinguono nella loro configurazione per l'opzione "hidden: true" che permette a tali nodi di essere invisibili agli occhi degli utilizzatori del replica set. Questi nodi hanno sempre priorità nulla, dunque non possono divenire server primari, ma normalmente possono partecipare alle elezioni esprimendo il proprio voto. Essendo invisibili gli hidden member non ricevono operazioni di lettura dagli utenti, per questo motivo vengono spesso sfruttati per altre attività, quali backup e reporting. Un esempio particolare di hidden member è dato dai *delayed replica set member*, caratterizzati dal fatto di mantenere un certo "ritardo" prestabilito nell'aggiornare il dataset. Questi server mantengono una versione datata dei dati, applicando le modifiche dell'oplog in ritardo di un certo numero di secondi scelto dall'utente al momento della configurazione del server. Eseguendo ad esempio i comandi di configurazione riportati nell'esempio 2.4 si predispone un delayed member (in questo caso il server collocato all'inizio dell'array "members") che mantenga i dati aggiornati con un'ora di ritardo:

```
cfg = rs.conf()
cfg.members[0].priority = 0
cfg.members[0].hidden = true
cfg.members[0].slaveDelay = 3600
rs.reconfig(cfg)
```

Esempio 2.4. Comandi necessari per configurare un delayed member che aggiorni i dati con un ritardo di 60 minuti. [12]

Il ritardo desiderato è dunque espresso in secondi (3600 nell'esempio precedente) ed assegnato all'opzione "slaveDelay". Mantenendo una versione obsoleta dei dati questi server possono risultare incredibilmente utili per risolvere numerosi problemi derivanti da errori umani che abbiano

portato, ad esempio, alla cancellazione erronea di alcune collection o di un intero database o all'applicazione di aggiornamenti scorretti ai dati. Le motivazioni che si pongono alla base della costruzione di un replica set sono infatti da ricercare nell'esigenza di maggiori garanzie di sicurezza, fault tolerance e disponibilità dei dati, non nella necessità di distribuire le operazioni di lettura fra più server così da suddividere il carico di lavoro e raggiungere più alte prestazioni, obiettivo per cui MongoDB mette infatti a disposizione un meccanismo decisamente più efficace: lo sharding.

2.5 Sharding

Una delle motivazioni del grande successo e del diffuso utilizzo di MongoDB è proprio la semplicità con cui rende possibile la distribuzione fra più nodi di un cluster tanto dei dati raccolti nelle collection e nei database quanto delle operazioni che su quei dati si ha bisogno di eseguire. MongoDB offre infatti supporto automatico allo sharding, consentendo agli utenti di sfruttarne i benefici, per poter gestire in modo efficiente anche grandi dataset e l'elevato tasso di letture e scritture che possono interessarlo, mediante la costruzione di un cluster di server. Uno *sharded cluster* in MongoDB consta di tre componenti fondamentali:

- gli shard: ciascuno shard può essere costituito da un singolo server oppure da un replica set e si occupa di gestire una certa porzione del dataset complessivo amministrato dal cluster. In ambito aziendale è consigliabile far sì che ciascuno shard sia costituito da un replica set, così da poterne sfruttare i benefici, in contesti differenti però, ad esempio se il cluster viene realizzato con il solo scopo di eseguire dei test, è possibile mantenere un'architettura più semplice facendo coincidere ciascuno shard con un'unica istanza mongod (il processo fondamentale di MongoDB, si tratta di un demone, come suggerito dalla lettera 'd' al termine del nome). MongoDB consente di abilitare lo sharding a livello di collection, in uno stesso database potrebbero pertanto coesistere collezioni di documenti per cui è stata richiesta la distribuzione fra tutti quanti gli shard del cluster ed altre che sono invece ancora mantenute interamente su singolo shard. Il

server (o il replica set) che mantiene le collection non distribuite appartenenti ad un certo database costituisce il *primary shard* di quel particolare database. Il termine "primary" viene pertanto utilizzato anche in relazione agli sharded cluster, senza tuttavia fare in alcun modo riferimento al significato che quello stesso termine assume nel contesto dei replica set.

- i config server: si tratta di istanze mongod che conservano i metadati necessari al funzionamento del cluster. Tali dati sono raccolti nel database config di ciascun config server e comprendono informazioni relative agli shard, alla suddivisione del dataset fra di essi, alle collection e ai database esistenti, ai lock attualmente in uso e molto altro ancora. Questo tipo di informazioni è fondamentale per lo sharded cluster, ragion per cui è necessario, in ambito aziendale, mantenere più di un solo config server, l'architettura prevista per l'utilizzo professionale di MongoDB in ambito distribuito prevede infatti l'esistenza di esattamente tre config server, in esecuzione su macchine distinte. In questo modo se uno o due di questi server dovessero essere temporaneamente non attivi o comunque non raggiungibili, il cluster potrebbe continuare ad operare correttamente, eseguendo letture e scritture senza problemi. I metadati risulterebbero però accessibili solo in lettura, ciò significa che operazioni che ne richiederebbero un aggiornamento, come una riorganizzazione della suddivisione dei dati fra gli shard, non potrebbero essere eseguite finché i config server non saranno nuovamente tutti operativi e normalmente contattabili. Se invece tutti e tre i config server esistenti dovessero risultare non attivi o comunque non contattabili dai restanti componenti del cluster, allora i document del dataset gestito dallo sharded cluster sarebbero ancora accessibili in lettura/scrittura unicamente finché i *query router* non vengono riavviati, essi mantengono infatti una copia in cache dei metadati necessari alla gestione delle operazioni di lettura e scrittura che può consentire il perdurare del funzionamento del sistema fino alla risoluzione dei problemi manifestatesi. Se scegliessimo di includere in uno sharded cluster un unico config server quest'ultimo

costituirebbe un single point of failure che, se inattivo, potrebbe portare alla inoperatività dell'intero sistema. Una simile scelta non è dunque consigliabile in ambito professionale sebbene possa essere comunque consapevolmente adottata nel caso in cui il cluster abbia scopi più semplici e a breve termine come, ad esempio, la conduzione di test concentrati in un breve periodo di tempo.

- i query router: sono i processi che si occupano di instradare le operazioni di lettura e di scrittura ricevute dagli utenti verso gli shard che detengono la porzione di dati interessata e di restituire poi ai medesimi utenti il risultato ottenuto. Per fare questo hanno ovviamente bisogno di far riferimento ai metadati mantenuti dai config server, per questo motivo mantengono una copia di tali dati nella cache. Ogni query router è un'istanza mongos (altro processo fondamentale in MongoDB) che si interpone fra lo sharded cluster e le applicazioni utente che, pertanto, non accedono mai direttamente ai dati, ogni operazione inviata al sistema viene sempre gestita attraverso la mediazione di un query router che rende fra l'altro del tutto trasparente agli utenti la complessità data dalla gestione di dati distribuiti. In uno stesso sharded cluster è possibile avere anche più query router, così da suddividere fra di essi il carico di lavoro.

Un esempio di sharded cluster potrebbe dunque essere quello mostrato nella figura 2.9.

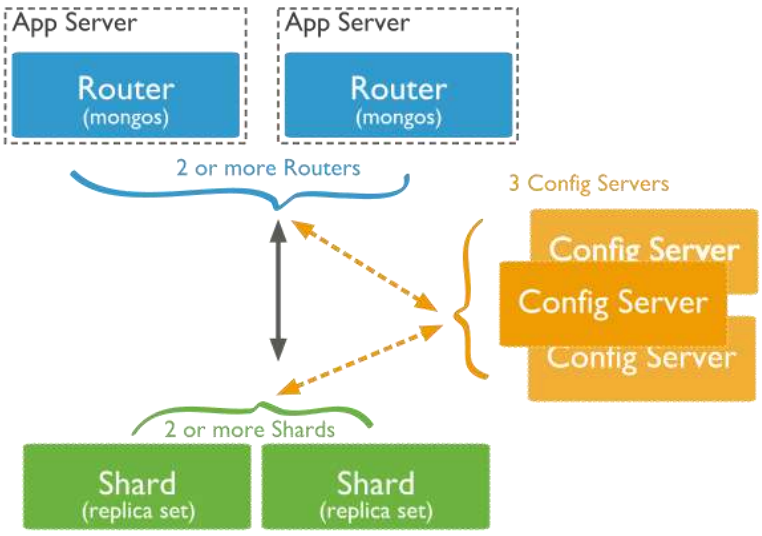

Figura 2.9. Possibile architettura di uno sharded cluster utilizzato in ambito aziendale. [12]

La suddivisione fra gli shard dei document raggruppati in ciascuna collection per cui è stato predisposto lo sharding viene deciso sulla base dei valori assunti da un campo o da un insieme di campi specifici prescelti dall'utente come *shard key*. Poiché lo sharding viene applicato a livello di collezione per ciascuna collection che si vuole gestire in modo distribuito è necessario specificare una shard key distinta, tenendo presente che la scelta dei campi che la costituiranno è un'importante passo di progettazione in quanto potrà influenzare sensibilmente le prestazioni future del sistema. Prendere consapevolmente e coscientemente questa decisione richiede di considerare attentamente sia i dati raccolti nella specifica collection di cui si vuole abilitare lo sharding sia le operazioni che su di essi avremo bisogno di eseguire. Una shard key dovrebbe principalmente consentire di:

- suddividere in modo equo i document fra gli shard,
- distribuire fra i nodi del cluster gli inserimenti di nuovi document (*write scaling*),
- permettere di circoscrivere l'esecuzione di ciascuna lettura ad una sezione quanto più possibile limitata del cluster (preferibilmente un unico shard), così da rendere più celere il suo completamento e migliori le prestazioni. MongoDB fa riferimento a questa importante caratteristica con il termine *query isolation*. Riuscire ad isolare la maggior parte delle operazioni di lettura su un singolo shard fa sì che l'eventuale inattività temporanea di un certo shard renda impossibile soddisfare solo un certo range di operazioni di lettura, ovvero quelle che coinvolgono la porzione di dati mantenuta dal particolare shard attualmente non interrogabile, mentre tutte le altre richieste possono ancora essere soddisfatte. Il venir meno di un singolo shard sarebbe decisamente più dannoso in presenza di una suddivisione dei dati che non consente di circoscrivere le operazioni di lettura ma che tende a proiettarle comunque sulla maggior parte degli shard, poiché risulterebbe non più eseguibile una percentuale maggiore di operazioni di lettura.

Prima di tutto è necessario sapere che il campo o i campi che sceglieremo dovranno essere necessariamente presenti in tutti quanti i document della

collection e che su quella particolare combinazione di campi sarà necessario costruire un indice. A partire dalla release 2.2 a dire il vero, se la shard key è costituita dalla combinazione di campi $f_1,f_2,\ldots,f_k$, è possibile accettare un qualsiasi indice costruito su una sequenza di campi $f_1,f_2,\ldots,f_n$, con $k \leq n$. Esistono ancora tuttavia delle restrizioni sulle tipologie di indici accettabili, non sono infatti ammessi indici "speciali", come geospatial index, multikey index e text index, ovvero gli indici utilizzati per poter eseguire ricerche di testo, tema che avremo modo di approfondire nel prossimo capitolo.

Se la collection è ancora vuota al momento dell'abilitazione dello sharding sarà il sistema in automatico a verificare se un indice adeguato è già presente e, se così non fosse, ad avviare la costruzione di un indice realizzato esattamente sui campi che costituiscono la shard key, se invece nella collection di interesse ci sono già dei document è necessario che tale indice esista già al momento dell'esecuzione del metodo `sh.shardCollection()` con cui si richiede di avviare lo sharding sulla collezione.

MongoDB organizza i document della collection in *chunk*, ossia in porzioni, spezzoni della collection nella sua interezza costituiti ciascuno da un numero intero di document. Tale suddivisione avviene sulla base del valore assunto in ciascun documento dalla shard key ed i chunk ottenuti vengono poi ripartiti fra gli shard del cluster. La dimensione massima di un singolo chunk di default è fissata a 64 MB (ma è modificabile dall'utente) ed i dati che ne fanno parte debbono essere mantenuti necessariamente uniti, ovvero sullo stesso shard. I document in cui la shard key ha esattamente lo stesso valore appartengono necessariamente ad uno stesso chunk e, di conseguenza, ad uno stesso shard. Normalmente se la dimensione di un chunk dovesse divenire eccessiva (se dovesse cioè superare i 64MB) il sistema procederebbe automaticamente a scindere il chunk in due sezioni distinte, dividendolo a metà, se però i document presenti nel chunk dovessero avere tutti quanti uno stesso valore della shard key la scissione sarebbe impossibile, anche se il limite dei 64MB viene superato. Nel definire la shard key è dunque importante scegliere una combinazione di campi tale che i document della collection si distribuiscano in modo quanto più possibile uniforme nello spazio delle combinazioni di valori che tali

campi possono assumere, così da evitare di avere un elevato numero di document con esattamente lo stesso valore della shard key e consentire una distribuzione equa dei dati fra gli shard. È altresì importante che la quantità di valori assumibili dalla shard key sia elevata e preferibilmente non limitata, così da non avere limiti neppure nella possibilità di suddivisione dei dati in un numero elevato di chunk distribuibili fra i nodi del cluster. Difficilmente si può ottenere una buona shard key utilizzando un unico campo, nella maggior parte dei casi è necessario considerarne più di uno per ottenere un efficace mix di casualità, uniformità nella distribuzione dei valori e capacità di assicurare write scaling da un lato e query isolation dall'altra. In alcuni casi poi, ovviamente, non tutte queste caratteristiche sono egualmente importanti. In un sistema con un bassissimo tasso di inserimenti, ad esempio, il concetto di write scaling avrà certamente un ruolo secondario rispetto alla capacità di gestire in modo efficace le operazioni di lettura o di mantenere bilanciata la distribuzione dei dati ed anche delle interrogazioni fra gli shard, così da evitare che i document e le query sottoposte al sistema da parte degli utenti si concentrino di più su una certa porzione del cluster lasciando altri nodi quasi privi di lavoro. Il mantenimento di una suddivisione bilanciata dei dati fra i membri del cluster è in realtà compito del *balancer*, un processo messo a disposizione dal sistema che si occupa di gestire la distribuzione iniziale dei chunk fra gli shard e la conservazione di una loro suddivisone equilibrata fra i nodi per tutta la durata della sopravvivenza del cluster, rendendone la gestione estremamente più semplice. Il balancer però opera sulla base dei valori assunti nei singoli document dalla shard key, quindi una scelta inadeguata dei campi che la costituiscono renderà comunque limitata la capacità del sistema di mantenere effettivamente equilibrata la suddivisione dei dati fra gli shard. La definizione della shard key si può dunque considerare a buon diritto la scelta più importante e l'arma più efficace a disposizione dell'utente per decidere il futuro del proprio cluster, il suo successo o il suo fallimento. Una volta stabiliti i campi che costituiranno la shard key il sistema può definire i chunks basandosi su due differenti tecniche: *range based partitioning* e *hash based partitioning*. Nel primo caso il sistema si limita a considerare l'insieme delle combinazioni di valori possibili per la

specifica shard key prescelta e a suddividerli in range fra loro non sovrapposti, ogni range costituisce un chunk distinto ed i dati che ne fanno parte saranno tutti quei documenti per cui la shard key assume una combinazione di valori inclusa in quel particolare intervallo. Una dimostrazione efficace ed intuitiva di questa tecnica di suddivisione è mostrata dalla figura 2.10, in cui si assume, per semplicità, che la shard key designata sia costituita da un solo campo, x, che assume valori numerici.

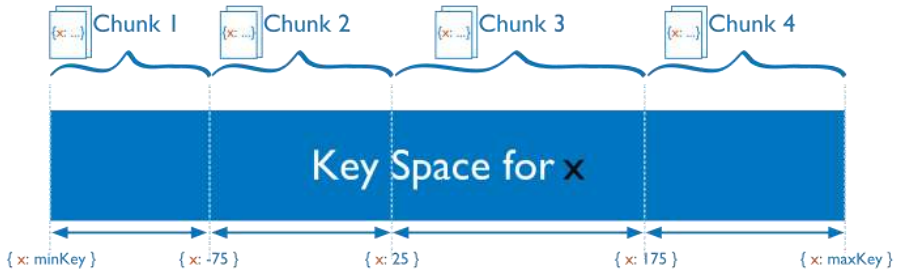

Figura 2.10. Esempio di suddivisione in chunk sulla base della tecnica del range based partitioning.

A partire dalla versione 2.4 di MongoDB si ha anche la possibilità di utilizzare una *hash shard key*. In questo caso la shard key deve essere costituita da un unico campo (preferibilmente con un'elevata cardinalità), sul quale viene costruito un indice hash che mantiene, per ogni entry, il valore ottenuto dall'applicazione di una specifica funzione hash al valore assegnato alla shard key. La distribuzione dei dati fra gli shard non avviene dunque per range ma si basa sulla funzione hash utilizzata dal sistema, assicurando una suddivisione equilibrata dei document fra i nodi del cluster che favorisce al contempo anche la capacità di distribuire in modo equo i futuri inserimenti così da sfruttare appieno la *write capacity* del sistema facendola coincidere con la somma delle capacità di gestione ed elaborazione delle operazioni di scrittura di ogni suo shard. L'uso di una hash shard key comporta infatti un elevato grado di casualità nella ripartizione dei dati fra i nodi del cluster e, se questo da un lato favorisce le operazioni di scrittura, dall'altro può danneggiare le prestazioni in lettura, rendendo decisamente più onerosa soprattutto l'esecuzione di query di range. Interrogazioni di questo tipo sarebbero al contrario favorite da una suddivisione dei document per range di valori, poiché il range based

partitioning rende probabile che documenti con valori vicini della shard key si trovino entrambi nello stesso shard, l'uso di una shard key di tipo hash può invece sparpagliare documenti con valori quasi uguali della shard key in tutto il cluster danneggiando la query isolation.

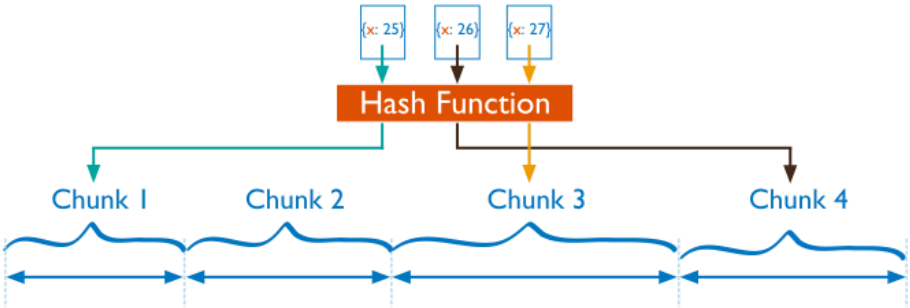

Figura 2.11. Esempio di suddivisione basata su hash based partitioning.

La distribuzione dei documenti rispetto ai valori della shard key è importante perché in uno sharded cluster le ricerche più efficienti sono quelle che il sistema riesce ad isolare su un unico shard o comunque su un sottoinsieme ristretto dei membri del cluster. Se la query richiede di accedere ai documenti con uno specifico valore della shard key (cioè se un valore preciso della shard key è esplicitato nell'interrogazione da eseguire) il query router riesce a dirigere quell'interrogazione verso uno shard specifico. In caso di shard key composta, affinché la query venga diretta solo ad un sottoinsieme degli shard è sufficiente specificare il valore anche solo di alcuni dei campi che la compongono, a patto che si tratti di quelli iniziali. In altre parole, se la shard key fosse costituita, *nell'ordine*, dai campi f_1, f_2, f_3, per ridurre il numero di shard a cui tale interrogazione verrà inviata è necessario limitare i valori accettabili per f_1, f_2 e f_3, oppure per f_1 e f_2, o solamente per il campo f_1, una condizione di uguaglianza posta invece, ad esempio, sui valori posseduti alla coppia di campi f_2 e f_3 non sarebbe di alcun aiuto. Nel caso in cui la query non dovesse contenere alcun filtro sulla shard key o ne individuasse un certo range di valori suddiviso fra la maggior parte dei nodi per via dell'adozione di una hash shard key, sarebbe necessario inviare la query a tutti quanti gli shard, attendere la ricezione dei risultati parziali da ciascuno di essi e poi fornire il risultato complessivo all'utente, aumentando il tempo di esecuzione. Per questo motivo è

importante che la shard key sia scelta a partire dai campi che figureranno più frequentemente nelle interrogazioni che si avrà bisogno di rivolgere al sistema, per lo meno in quelle interrogazioni per cui è rilevante il raggiungimento di performance elevate.

Capitolo 3 – Un caso di studio reale

3.1 Introduzione

All'analisi teorica sinora condotta vogliamo a questo punto affiancare un caso di studio reale, abbiamo infatti eseguito numerosi test per mettere alla prova le funzionalità offerte da MongoDB e le performance che questo DBMS permette effettivamente di ottenere, sia su singolo server sia in ambito distribuito. Per fare questo abbiamo considerato un database contenente una sola collection (tranne "`system.indexes`", creata automaticamente dal sistema in ogni db) chiamata "clip" di dimensione pari a 8,54063 gigabyte e contente esattamente 1.431.090 documents. I dati considerati derivano dall'ascolto delle discussioni avvenute on-line relativamente alle elezioni europee del maggio 2014, si tratta pertanto di documenti che contengono post di blog, commenti, articoli di giornali ed ogni altra forma di informazione testuale legata ai temi della politica e dell'attualità. Fra gli altri abbiamo i campi "TITLE" e "CONTENT", due campi testuali che contengono rispettivamente il titolo ed il contenuto dello specifico intervento rappresentato in un certo document. Ad essi si affiancano molti altri campi in cui si raccolgono alcune informazioni che è stato possibile reperire circa l'autore di quel particolare testo (ad esempio il genere, rappresentato nel campo "AUTHOR_GENDER", o la sua età, riportata in "AUTHOR_AGE"), la fonte da cui il testo considerato è stato reperito (in campi quali "SOURCE" e "SOURCECHANNEL") e molti altri metadati. I dati qui considerati sono infatti parte integrante di un ampio progetto di Social Business Intelligence che si ripropone un'analisi estremamente più avanzata e complessa di quella condotta per questo studio di tesi, che si prefigge semplicemente di testare, confrontare e valutare le

funzionalità offerte da un DBMS NoSQL di grande diffusione ed attrattiva come MongoDB che mette a disposizione strumenti interessanti, potenzialmente fruttuosi ed apprezzabili nonché pratici ed agevoli nell'utilizzo, come le ricerche di testo. Dopo aver proposto una presentazione complessiva di MongoDB e delle sue caratteristiche è proprio sulle ricerche di testo e sui benefici che si possono effettivamente trarre dalla costruzione di uno sharded cluster, e dunque dall'operare in un ambito distribuito, che abbiamo voluto concentrare la nostra attenzione ed i nostri studi. Per condurre i nostri test abbiamo considerato quattro calcolatori aventi la medesima architettura hardware e software descritta nella tabella 3.1, confrontando i risultati ottenuti operando dapprima su singolo server e poi su un cluster costituito da un numero crescente di nodi, da due fino a quattro.

Sistema Operativo	Microsoft Windows 7 Professional 64bit
Processore	Intel(R) Pentium(R), 2 core, 3.40GHz
RAM	4,00 GB
Hard Disk	Hitachi GST Deskstar T7K500 HDT725025VLA380 (0A33423) 250GB 7200 RPM 8MB Cache SATA 3.0Gb/s 3.5" Hard Drive Bare Drive

Tabella 3.1. Caratteristiche delle quattro macchine utilizzate per l'esecuzione dei test.

Dato il ridotto numero di macchine a disposizione e la tipologia di progetto che avevamo intenzione di impostare ed affrontare abbiamo optato per un'architettura più semplice di quella consigliata in ambito professionale, realizzando un cluster in cui:

- ogni shard non è costituito da un replica set ma da un'istanza mongod stand alone,
- si ha a disposizione un unico query router ed un solo config server.

La scelta della shard key è stata effettuata a partire dall'analisi delle operazioni che avremmo dovuto effettuare e dei dati considerati, ovvero della struttura dei document raccolti nella collection "clip" e dei valori ad essi assegnati (la loro distribuzione e l'esistenza o meno di document in cui quei campi non fossero stati settati, acquisendo così valore `null`, o fossero

completamente assenti). Come sappiamo la shard key si dovrebbe comporre dei campi più utilizzati nelle query che verranno lanciate sulla collection, cercando così di ottenere i benefici che possono derivare dalla query isolation. Nel nostro caso al centro delle interrogazioni da eseguire si trovavano le ricerche di testo, operate sui campi "CONTENT" e "TITLE", con qualche condizione di selezione posta a volte sul campo "SOURCE". Fra quelli elencati l'unico campo sinceramente candidato ad entrare a far parte della shard key è proprio il campo "SOURCE", poiché sui campi restanti non vengono posti dei filtri, sono oggetto esclusivamente di ricerche di testo, che non traggono benefici dalla costruzione di indici di tipologie comuni (quali sono gli indici costruiti sulla shard key) ma unicamente dall'esistenza di indici di testo, di cui non possono fare a meno. Tali campi inoltre non sempre nei document della collection "clip" risultano valorizzati. La distribuzione dei valori assunti da "SOURCE", che specificano la fonte (la "sorgente") da cui sono state tratte le informazioni racchiuse in quel particolare documento, appare però estremamente sbilanciata, per questo motivo abbiamo scelto di adottare una shard key composta, costituita dalla sorgente e dal campo `_id` che, assumendo un valore distinto in ogni document, assicura il raggiungimento di un'elevata cardinalità. Se da un lato l'alta cardinalità, ossia la capacità di assumere una gran quantità di valori distinti suddivisibili in modo equo fra gli shard, è un ingrediente fondamentale per l'ottenimento di una distribuzione bilanciata dei dati, al contempo essa non assicura il conseguimento di altri due importanti obiettivi, cioé: query isolation e write scaling. Nel nostro caso, a dire il vero, la capacità del sistema di suddividere equamente le operazioni di inserimento fra i nodi del cluster gioca un ruolo secondario, poiché non abbiamo bisogno di inserire nuovi documenti ma solo di interrogare un database già costituito. Ciò che ci interessa è dunque per lo più tentare di ottimizzare le prestazioni in lettura. Proprio per favorire, laddove possibile, la query isolation scegliamo dunque di adoperare come prima componente della shard key il campo "SOURCE", così che il filtro posto in alcune query su tale campo costituisca una condizione di selezione specificata su un prefisso della shard key, favorendo l'isolamento di tali interrogazioni

esclusivamente sulla porzione del cluster effettivamente interessata da ciascuna query.
Come vedremo in realtà fra le interrogazioni che abbiamo testato figurano anche delle query che includono delle condizioni di uguaglianza poste su un campo differente da "SOURCE", ovvero "SOURCECHANNEL". L'esecuzione di questo gruppo di interrogazioni è stata però decisa in un momento successivo, quando la costruzione del cluster era già stata avviata, per questo motivo il campo "SOURCECHANNEL" non è stato preso in considerazione al momento della scelta della shard key, che è stata definita nel modo seguente: `{SOURCE:1, _id:1}`.
Alcune delle interrogazioni eseguite sono state ripetute anche su un secondo cluster, caratterizzato da un'architettura hardware più avanzata. Le macchine che fanno parte di tale cluster sono quattro personal computer HP aventi le caratteristiche riportate nella tabella 3.2

Processore	Intel Core i7 (quad-core), 3.6GHz
RAM	32GB
Hard Disk	4TB, 15'000rpm

Tabella 3.2. Caratteristiche HW dei nodi del secondo cluster.

Anche in questo caso l'architettura del cluster è minimale, prevede infatti:

- 4 shard costituiti ciascuno da un singolo server,
- 1 solo query router,
- 1 solo config server.

Dato che i test che sono stati condotti e dei quali si vogliono ora presentare i risultati si concentrano sull'attuazione di ricerche di testo crediamo opportuno dedicare dapprima un po' di spazio ad un piccolo approfondimento su tale argomento.

3.2 Text Search

Affinché sia possibile eseguire una ricerca di testo è prima di tutto necessario costruire un indice di testo (text index) sui campi che saranno oggetto delle nostre interrogazioni. A differenza di quanto accadeva nella

release 2.4, a partire dalla versione 2.6 di MongoDB la ricerca di testo è già abilitata di default nel sistema, non si ha più bisogno dunque di procedere manualmente alla sua abilitazione prima di poter proseguire con la costruzione di indici di testo ed il loro successivo utilizzo. Tuttavia sono ancora presenti dei limiti nella costruzione e nell'uso di text index, non è possibile, ad esempio, avere più di un indice di testo per ciascuna collection così come non è consentito includere nella medesima query sia una ricerca di testo che altri operatori che fanno uso di tipologie speciali di indici (come i geospatial index). Ciò impedisce, per esempio, di ricorrere agli operatori `$text` e `$near` nella stessa query ma anche di inglobare nella medesima interrogazione più di un'espressione recante l'operatore `$text` e quindi di affiancare più ricerche di testo distinte. Quella delle ricerche di testo è, d'altronde, una funzionalità aggiunta al sistema piuttosto recentemente, dato che è stata introdotta solo nella release 2.4, ed è già stata sviluppata in modo apprezzabile rispetto al passato. A partire dalla versione 2.6 è infatti possibile far uso di `$text` anche nello stage `$match` dell'Aggregation Pipeline e quindi operare ricerche di testo anche nell'ambito dell'aggregazione dei dati. L'operatore `$text` può essere adoperato anche per la scrittura di comuni operazioni di lettura, quindi facendo uso del metodo `db.collectionName.find()` o della sua variante `db.collectionName.findOne()` che, indipendentemente dal numero di document che soddisfano i criteri di selezione specificati dall'utente, restituisce sempre un solo risultato. La sintassi da adoperare per poter effettuare una ricerca di testo è mostrata nell'esempio 3.1.

```
{$text: {$search: <string>, $language: <string>} }.
```

Esempio 3.1. Formato di un'espressione contenente l'operatore $text. [12]

L'operatore `$text`, sostituito nella release 2.6 al precedente `text`, richiede dunque un solo parametro, espresso in forma di documento e contente due campi distinti: "search", che consente di specificare la particolare stringa da ricercare nei campi indicizzati con il text index, e "language", per indicare sulla base di quale lingua, fra le quindici supportate da MongoDB, si vuole basare la ricerca. Come vedremo `$text` permette di

effettuare differenti tipi di ricerche, consentendo all'utente di indicare: un'unica keyword, un insieme di keyword poste in OR fra loro, una o più frasi esatte oppure una lista di keyword a cui vengono affiancate anche delle parole chiave negate, delle quali si richiede cioè non la presenza ma l'*assenza* nei document della collection in esame. La lingua specificata dall'utente è a sua volta molto importante, è proprio sulla base del valore scelto per il campo "language", infatti, che il sistema basa l'individuazione delle stop word e la definizione delle regole di stemming e tokenization. Le ricerche di testo in MongoDB operano infatti sulla radice di ciascuna keyword fornita in input dall'utente dato che ognuna di esse viene elaborata dallo stemmer prima di essere effettivamente ricercata nei campi di testo indicizzati. Con il termine *stemmer* si fa riferimento ad un processo che consente di individuare lo *stem*, cioè la forma base, la radice, di una qualsiasi parola derivata o coniugata. Lo stem di una keyword non deve necessariamente coincidere con la sua radice morfologica, consente semplicemente di ricondurre ad una base comune termini fra loro correlati. Ricercando, ad esempio, la parola "politician" verranno selezionati anche tutti quanti i document in cui è presente "politicians" ma non quelli contenti solo il termine "policy". MongoDB inoltre, nell'ambito delle ricerca di testo, per quanto riguarda i caratteri non accentati è case-insensitive, pertanto le keyword "minister" e "MINISTER" vengono considerate equivalenti. Se si vuole modificare il comportamento di default di MongoDB per impedire l'intervento dello stemmer e l'individuazione delle stop word ciò è possibile assegnando al campo "language" il valore "none". Nonostante la sua importanza tale campo è in realtà opzionale, può quindi essere omesso nella scrittura di una query contenente una ricerca di testo. Se l'utente non specifica la lingua da utilizzare verranno considerate le regole e la lista di stop word proprie della lingua associata all'indice di testo al momento della sua creazione. La costruzione di questi tipi di indici avviene, come per ogni altra tipologia ammessa da MongoDB, tramite il metodo `db.collectionName.ensureIndex()` che consente di specificare, oltre ai campi da indicizzare, un certo numero di opzioni generiche (ossia presenti per ogni tipo di indice) ed altre che sono invece specifiche per gli indici di testo. Fra le opzioni dedicate alla realizzazione di indici di testo si

ha “default_language”, che permette di specificare proprio la lingua da utilizzare nella costruzione dell’indice e successivamente nell’esecuzione delle ricerche di testo che ne faranno uso (a meno che nella query non venga esplicitamente richiesto dall’utente di utilizzare una lingua differente). MongoDB offre anche la possibilità di gestire l’esistenza, nella medesima collection, di document o sub-document (cioè documenti innestati) contenenti testi scritti in lingue differenti, come nell’esempio mostrato dalla figura 3.1.

```
{
  _id: 1,
  language: "portuguese",
  original: "A sorte protege os audazes.",
  translation:
  [
     {
       language: "english",
       quote: "Fortune favors the bold."
     },
     {
       language: "spanish",
       quote: "La suerte protege a los audaces."
     }
  ]
}
{
  _id: 2,
  language: "spanish",
  original: "Nada hay más surrealista que la realidad.",
  translation:
  [
     {
       language: "english",
       quote: "There is nothing more surreal than reality."
     },
     {
       language: "french",
       quote: "Il n'y a rien de plus surréaliste que la réalité."
     }
  ]
}
```

Figura 3.1. Esempio di documenti recanti campi testuali contenenti frasi scritte in linguaggi differenti. MongoDB offre comunque la possibilità di indicizzarli con un unico test index. [12]

Anche in una situazione di questo tipo è possibile costruire un unico indice di testo, nell’indicizzare i termini presenti in ciascun document MongoDB farà infatti riferimento al valore assegnato al campo “language” di ciascuno

di essi. Nei document in cui il valore di tale campo è "spanish" verranno, ad esempio, utilizzate le regole di stemming e tokenization proprie dello spagnolo e verranno ignorate tutte quelle parole che in tale lingua indicano articoli, aggettivi possessivi, congiunzioni, preposizioni e, in generale, vocaboli che non aggiungono particolare significato alle frasi (ovvero le così dette stop word). Per quei document e quei sub-document in cui non è presente il campo "language" verrà utilizzato il linguaggio di default, ovvero l'inglese, con le sue regole e la sua lista di stop word. Se nei document da indicizzare la funzione propria del campo "language" viene svolta da un altro campo è necessario indicare il nome di tale campo come valore dell'opzione "language_override" nel richiedere la creazione dell'indice di testo. Un'altra proprietà molto importante messa a disposizione dal metodo `db.collectionName.ensureIndex()` è "weights", essa permette infatti di associare "pesi" distinti ai vari campi indicizzati dal text index, così da valutare in modo differente, ai fini del calcolo del textScore, un matching avvenuto in un campo piuttosto che in un altro. Abbiamo fatto uso di tale opzione, ad esempio, nel costruire l'indice di testo che ci ha consentito di eseguire i nostri test. Nel nostro caso i campi su cui eseguire ricerche di testo erano due: "CONTENT" e "TITLE" ed il comando che abbiamo eseguito per costruire il nostro text index è quello riportato nell'esempio 3.2.

```
db.clip.ensureIndex({CONTENT:"text",TITLE:"text"},{
name:"content_title_index",weights:{CONTENT:1,TITLE
          :2},default_language:"english",
          language_override:"LANGUAGE"});
```

Esempio 3.2. Comando lanciato sulla collection clip per costruire il nostro text index.

Il primo parametro passato al metodo `db.clip.ensureIndex()` è un document recante l'elenco dei campi da indicizzare e la specifica della tipologia di indice che si desidera costruire su ciascuno di essi, indicata nel nostro caso dalla stringa "text". Il parametro seguente è un document dedicato alle opzioni, fra le quali notiamo:

- “name”: con cui viene semplicemente assegnato un nome al nuovo indice. Se l’utente non dovesse indicare alcun nome il sistema ne produrrebbe uno automaticamente.
- “weights”: con cui si è mantenuto il “peso” di default (cioè 1) per il campo “CONTENT” e si è assegnato un peso doppio al campo “TITLE”. In questo modo ritrovare la keyword cercata nel titolo avrà valore doppio rispetto ad individuarla nel campo “CONTENT”. Il textScore, che esprime la rilevanza di uno specifico documento per una certa ricerca di testo, viene infatti calcolato, per ogni document restituito nel risultato, come somma pesata dei matching avvenuti in ciascuno dei campi indicizzati.

Gli indici di testo possono occupare una quantità di memoria non indifferente in quanto prevedono una nuova entry per ciascuno stem distinto individuato considerando le parole esistenti nei campi indicizzati di ogni document della collection su cui è stato costruito, eccezion fatta ovviamente per le stop word dello specifico linguaggio adottato. Tali indici non sprecano però spazio per indicizzare document in cui i campi di testo considerati non esistono o in cui non è ancora stata assegnata loro alcuna stringa o un array di stringhe ma valori come `null` o array vuoti, i text index sono infatti “sparsi” di default, indicizzano cioè esclusivamente i document necessari, ignorando totalmente gli altri.

3.3 Presentazione dei risultati

Possiamo partire considerando la ricerca di un’unica keyword ed i risultati ottenuti da questa categoria di test. Quella che considera una sola parola chiave è infatti la più semplice delle tipologie di ricerca di testo messe a disposizione da MongoDB. Il tempo di esecuzione della ricerca si dimostra strettamente legato al numero di document in cui la keyword cercata è presente, tale valore costituisce in effetti il numero di document che il sistema è costretto a recuperare dal disco ed influenza dunque pesantemente le prestazioni. Proprio per questo motivo abbiamo scelto di effettuare i nostri test considerando undici keyword differenti facendo variare il numero delle occorrenze in un range abbastanza ampio da poter notare una

variazione nei tempi necessari per il completamento delle ricerche: l'intervallo considerato è compreso fra 307.545 e 7.943.
Il grafico 3.1 mostra la relazione fra le medie dei tempi di esecuzione registrati su singolo server per le ricerche di ciascuna delle keyword considerate ed il numero di occorrenze di ognuna di esse.

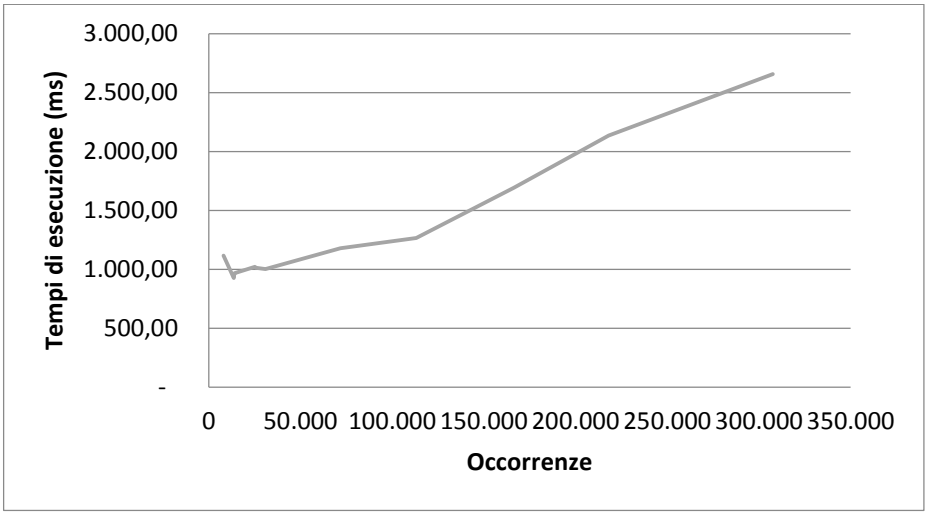

Grafico 3.1. Relazione esistente fra i tempi di esecuzione delle ricerche di keyword singole su singolo server e il numero delle occorrenze dei termini ricercati.

Dal risultato ottenuto è evidente come all'aumento del numero delle occorrenze segua normalmente un incremento nell'intervallo di tempo richiesto dall'esecuzione della ricerca di testo, nonostante vi siano delle piccole oscillazioni nei tempi di esecuzione. La relazione riscontrata e mostrata nel grafico precedente rispetta pienamente le nostre attese, lo stesso non si può dire purtroppo del variare dei tempi di esecuzione registrati passando ad operare in ambito distribuito, aumentando progressivamente il numero di shard considerati. Il grafico 3.2 mostra i risultati ottenuti confrontandoli anche con i tempi di query registrati sul secondo cluster, che, come abbiamo già detto, si distingue dal primo per essere costituito da macchine caratterizzate da hardware decisamente più performante.

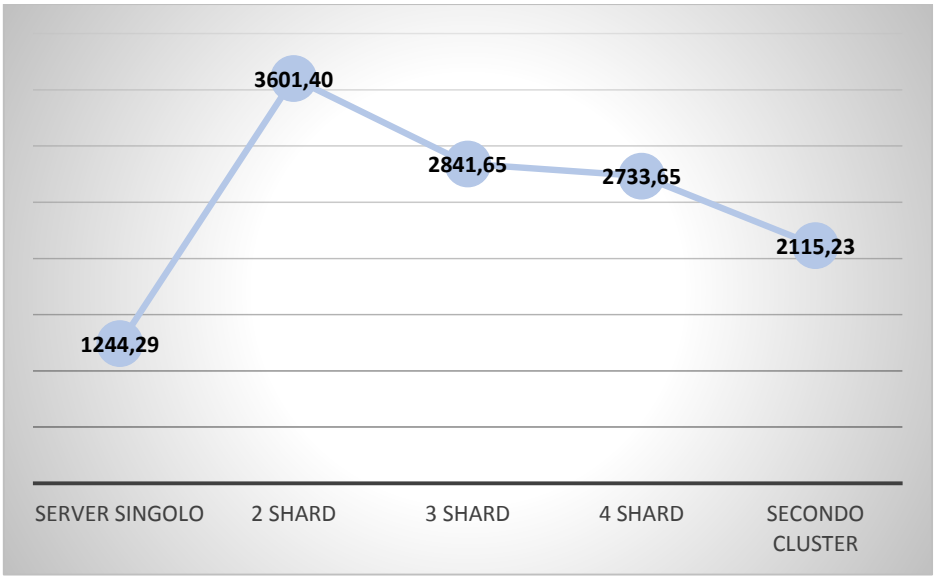

Grafico 3.2. Confronto fra i tempi di esecuzione ottenuti su singola macchina e su cluster per quanto riguarda la ricerca di keyword singole. I valori riportati nel grafico sono espressi in millisecondi e sono in realtà, per ciascuna configurazione esaminata, la media complessiva di tutti quanti i tempi di esecuzione registrati per la ricerca di ciascuna delle undici keyword prese in esame.

Le anomalie riscontrate sono probabilmente dovute al fatto che le operazioni considerate sono estremamente veloci già su singolo server, hanno infatti ottenuto tutte tempi di esecuzione inferiori ai 3.000ms, pertanto l'overhead dato dalla necessità per i vari nodi del cluster di scambiarsi informazioni attraverso la rete riesce probabilmente a nascondere i benefici che si sarebbero potuti ottenere dall'opportunità di distribuire il carico di lavoro fra più macchine. Risultati simili sono stati registrati anche relativamente alla ricerca di più keyword con un'unica operazione di ricerca di testo. In questo caso, come abbiamo già spiegato, le keyword indicate dall'utente si considerano legate da OR logici, in altre parole affinché un documento possa essere selezionato e restituito dall'operatore $text è sufficiente che almeno uno dei termini ricercati sia presente nei campi indicizzati con il text index. Nei nostri test abbiamo considerato gruppi di keyword costituiti da un numero crescente di parole, da un minimo di due fino ad un massimo di sei, selezionando anche in questo caso differenti possibili combinazioni di parole chiave così da prendere in considerazione gruppi di keyword aventi, complessivamente, un numero variabile di

occorrenze. Il range di occorrenze considerato è in questo caso compreso fra 66 e 619'784.

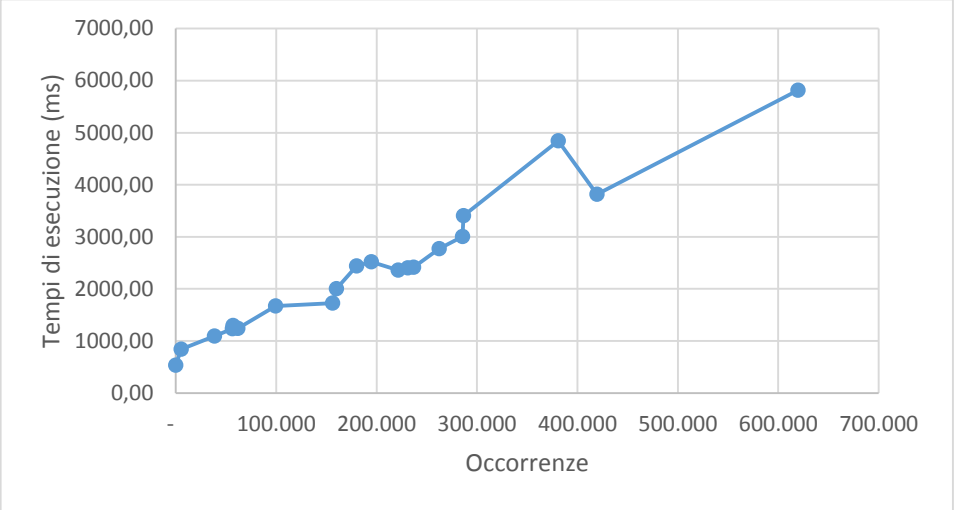

Grafico 3.3. Confronto delle medie dei tempi di esecuzione delle query con ricerche di keyword multiple eseguite su singolo server con le occorrenze di ciascun gruppo di keyword ricercato.

Ancora una volta, sebbene in presenza di alcune piccole oscillazioni nei tempi di esecuzione registrati, è possibile notare l'esistenza di una relazione fra l'aumento del numero di document selezionati dalla ricerca e le performance della sua esecuzione. È opportuno sottolineare come l'incremento delle occorrenze non coincida necessariamente con un aumento anche nel numero di keyword considerate, al contrario una ricerca di testo concentrata su un numero inferiore di parole chiave potrebbe selezionare un numero elevato di document e risultare dunque più lenta rispetto alla ricerca di un gruppo più numeroso di termini. Ad esempio le quattro keyword "minister austerity sanction tax" nel nostro caso erano caratterizzate da ben 380.845 occorrenze, mentre "poverty pensions childcare school research" soltanto da 221.530. Ebbene, la ricerca delle prime quattro parole chiave ha impiegato 4.845,5ms per essere completata mentre quella del secondo gruppo, composto da cinque keyword, solo 2.359ms. Nel grafico 3.4 vengono nuovamante confrontati i risultati ottenuti su singolo server con quelli raggiunti operando su cluster. Le quantità in millisecondi mostrate nel grafico sono in realtà il risultato della media

aritmetica, operata per ciascuna delle configurazioni prese in esame, dei tempi di esecuzione effettivamente ottenuti nel ricercare venti differenti gruppi di keyword (gli stessi considerati anche nel grafico precedente), suddivisi equamente sulla base del numero delle parole chiave presenti in ogni gruppo.

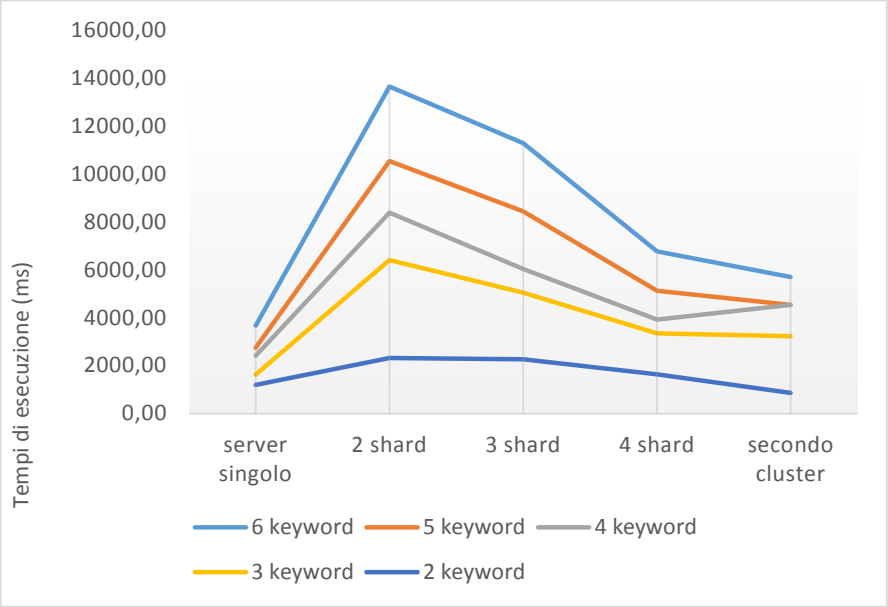

Grafico 3.4. Variazione delle performance nell'esecuzione di ricerche di testo con keyword multiple operando su singolo server piuttosto che su cluster, considerando un numero di shard crescente e confrontando anche differenti architetture hardware.

Anche in questo caso la scelta di operare su cluster non ha portato benefici evidenti nelle performance, al contrario pare per lo più averle danneggiate, con la sola eccezione della ricerca di due keyword sul secondo cluster. La spiegazione che possiamo trovare a questo comportamento è ancora una volta legata ai tempi di esecuzione estremamente contenuti registrati dalle operazioni eseguite che, nel caso di un unico server, sono ancora mediamente inferiori ai 4 secondi. Possiamo inoltre notare come l'aumentare dei nodi presenti nel cluster abbia comunque fatto registrare una riduzione nei tempi di query, seppure senza riuscire in generale ad eguagliare le performance ottenute operando su una sola macchina. Confrontando i risultati ottenuti sui due cluster con 4 nodi notiamo come i benefici dello scaling verticale (ossia il potenziamento dell'hardware utilizzato) siano anch'essi contenuti, seppure evidentemente presenti.

Risultati più significativi sono stati ottenuti prendendo in considerazione un'ulteriore tipologia di ricerca di testo resa possibile da MongoDB: la ricerca di frasi esatte. Questo tipo di ricerca, per via di come sono strutturati i text index, risulta essere generalmente decisamente più onerosa delle tipologie differenti. Gli indici di testo non mantengono infatti alcuna informazione circa la prossimità fra più keyword all'interno dei campi indicizzati, è dunque possibile che, nel ricercare ad esempio la frase "europian parliament", il sistema sia costretto inutilmente ad accedere ad un gran numero di document in cui i termini "european" e "parliament" siano sì entrambi presenti ma non nel modo in cui vorremmo, ossia uno di seguito all'altro. La ricerca di frasi esatte può dunque facilmente raggiungere tempi di esecuzione estremamente elevati, soprattutto aumentando il numero delle frasi ricercate. Al contrario di quanto avviene nel caso della ricerca di keyword multiple infatti scrivendo una query in cui si specificano più frasi esatte (indicate racchiudendo ciascuna di esse fra una coppia di simboli \") separate da spazi si richiede al sistema di ricercare tutti i document in cui tali frasi siano contemporaneamente presenti. Nel grafico 3.5 vengono mostrate le medie dei risultati ottenuti dalla ricerca di un numero crescente di frasi esatte (da una a cinque) operate come al solito sia su cluster che in sua assenza.

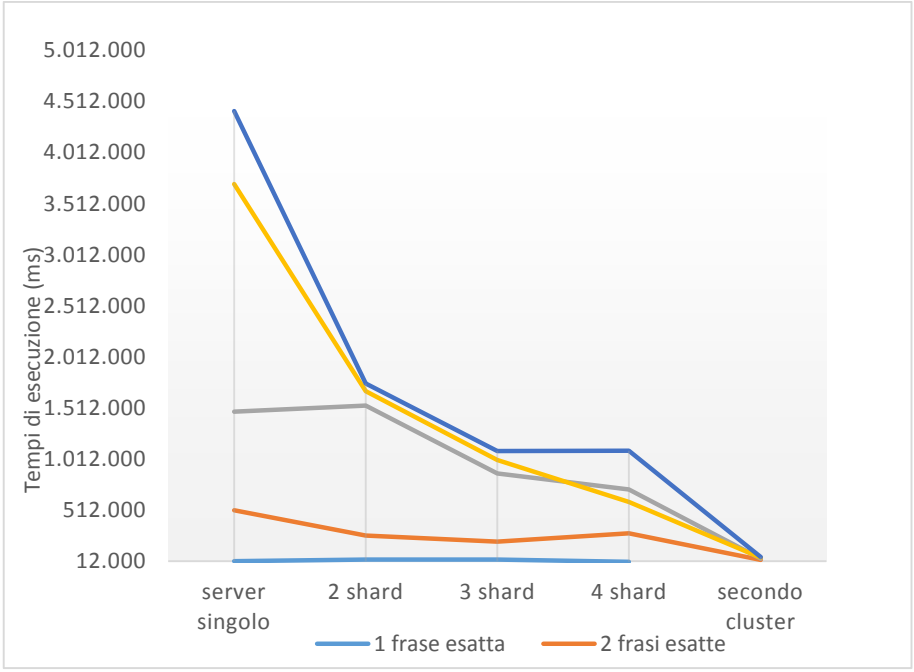

Grafico 3.5. Performance ottenute testando query per la ricerca di frasi esatte.

Anche se con qualche anomalia, iniziamo finalmente a riscontrare dei benefici nell'optare per una suddivisione dei dati e del carico di lavoro fra più shard. Il miglioramento appare evidente soprattutto accrescendo il numero di frasi considerate. Le ricerche di quattro o di cinque frasi racchiuse in un'unica query su singola macchina raggiungono infatti dei tempi di esecuzione così elevati da risultare assolutamente inaccettabili per molti ambiti applicativi, passando invece ad operare su cluster i tempi, seppure non trascurabili, si riducono sensibilmente, arrivando a richiedere solo pochi secondi sulla configurazione più avanzata. Possiamo notare infatti come in questo caso sia evidente la differenza prestazionale fra i due cluster con quattro nodi, l'architettura hardware (HW) più avanzata del secondo cluster si dimostra qui decisamente significativa nel determinare un ulteriore miglioramento nelle performance, consentendo il raggiungimento di tempi di esecuzione che, se paragonati a quelli ottenuti su singolo server, appaiono incredibilmente contenuti.

La quarta ed ultima forma di ricerca di testo consentita da MongoDB si distingue da tutte le precedenti perché consente non solo di specificare quali termini ricercare nei campi indicizzati dal text index, ma anche di indicare delle keyword da escludere. Si tratta delle così dette keyword negate, ossia di parole precedute da un trattino (dunque dal carattere -), simbolo che viene interpretato da `$text` come prova del fatto che di quello specifico termine si richiede non la presenza ma l'assenza nei document del risultato. Va sottolineato che per poter eseguire una ricerca di testo non è possibile specificare esclusivamente keyword negate, si deve necessariamente indicare almeno una keyword priva di negazione, altrimenti il sistema non restituirà alcun risultato. Così come la ricerca di keyword singole e quella di keyword multiple anche questa tipologia di ricerca di testo richiede normalmente intervalli di tempo piuttosto piccoli per essere completata, ciò porta ancora una volta ad osservare un comportamento anomalo nell'eseguire questo tipo di operazioni su cluster piuttosto che su un solo server. Proponiamo nuovamente i risultati ottenuti mediante un grafico, il grafico 3.6, che pone a confronto le prestazioni ottenute in ambito distribuito e operando su singola macchina.

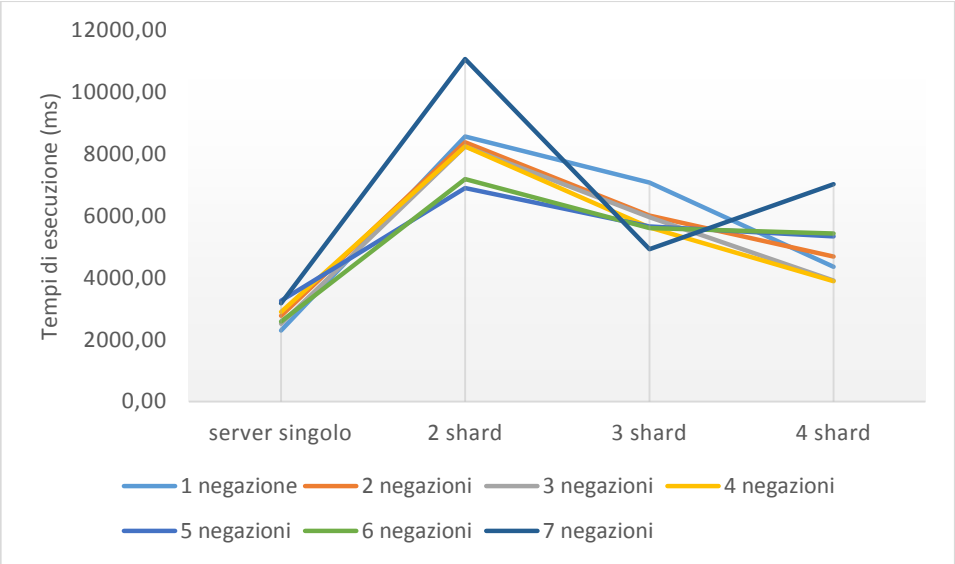

Grafico 3.6. Tempi di esecuzione ottenuti dalle ricerche di testo con keyword negate operate su un singolo server o in presenza di un cluster con 2, 3 o 4 shard.

Ciò che stupisce è anche la relazione fra i tempi di esecuzione ottenuti, sulla stessa configurazione, aumentando il numero di keyword negate specificate. Dato che introdurre nuove negazioni diminuisce il numero di document che soddisfano la ricerca (e dunque la quantità di documenti che il sistema ha bisogno di recuperare dal disco) ci si aspetterebbe infatti di assistere al ridursi degli intervalli di tempo necessari al completamento dell'esecuzione con l'aumentare delle keyword negate specificate nella query ma questo non sempre si verifica. Ciò appare evidente soprattutto nel grafico 3.7 che si concentra sulle performance ottenute su singolo server.

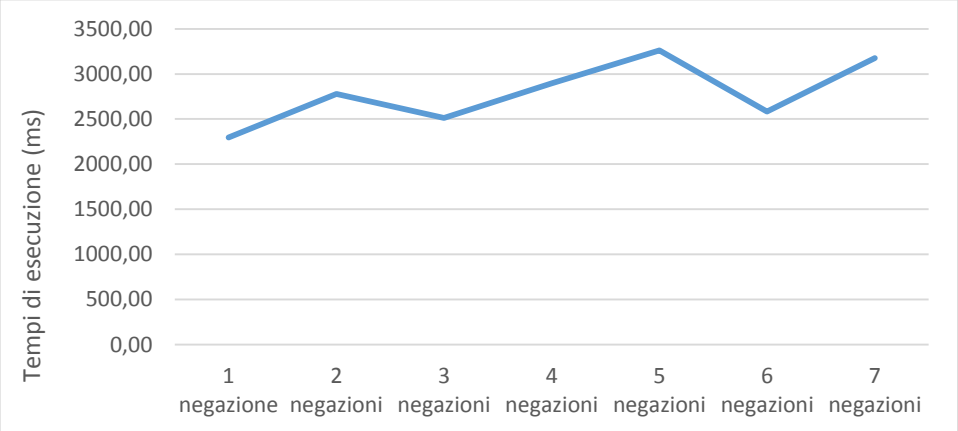

Grafico 3.7. Prestazioni delle ricerche di keyword negate su singolo server.

Poiché l'operatore `$text` viene eseguito attraverso il metodo `db.collectionName.find()`, è possibile affiancare alle ricerche di testo ulteriori condizioni di selezione per affinare la ricerca condotta, andando a definire ancor più precisamente le caratteristiche che debbono essere possedute da un certo document affinché esso possa entrare a far parte del risultato. La presenza di ulteriori filtri non riesce però a migliorare le prestazioni delle interrogazioni perché se l'operatore `$text` è presente la ricerca di testo viene in ogni caso eseguita per prima. Supponiamo, ad esempio, di avere la query dell'esempio 3.3.

```
db.clip.find({ SOURCECHANNEL:
"harrowtimes.co.uk_news", $text:{$search:
"minister", $language:"en" }});
```

Esempio 3.3. Query che restituisce tutti i document presenti nella collection "clip" in cui il valore del campo "SOURCECHANNEL" sia "harrowtimes.co.uk_news" ed in cui sia presente la keyword "minister" in almeno uno dei campi indicizzati dal text index.

In questo caso la ricerca della keyword "minister" viene eseguita per prima e, solo in seguito, viene applicato il filtro specificato sul campo "SOURCECHANNEL", ai documenti restituiti dall'operatore `$text`. L'unico modo per modificare questo tipo di comportamento è quello di costruire un indice composto, che indicizzi dapprima il campo "SOURCECHANNEL" e poi, come test index, i campi sui quali deve essere operata la ricerca di testo. Per avviare la costruzione dell'indice sarà pertanto necessario eseguire un comando del tipo riportato nell'esempio 3.4.

```
db.clip.ensureIndex({SOURCECHANNEL: 1, CONTENT:
"text", TITLE: "text"}, … );
```

Esempio 3.4. Codice necessario per costruire un indice composto che anteponga un campo indicizzato secondo un normale ordinamento ascendente a campi sui quali dovrà invece assumere la forma di un text index.

A riprova di quanto affermato proponiamo nei grafici 3.8 e 3.9 le prestazioni ottenute ripetendo l'esecuzione della ricerca di una e più farsi esatte preceduta questa volta da una condizione di uguaglianza sul campo "SOURCECHANNEL" ed eseguite, rispettivamente, con un normale indice di testo costruito sui campi soli "CONTENT" e "TITLE" e con un compound index del tipo specificato dall'esempio 3.2.

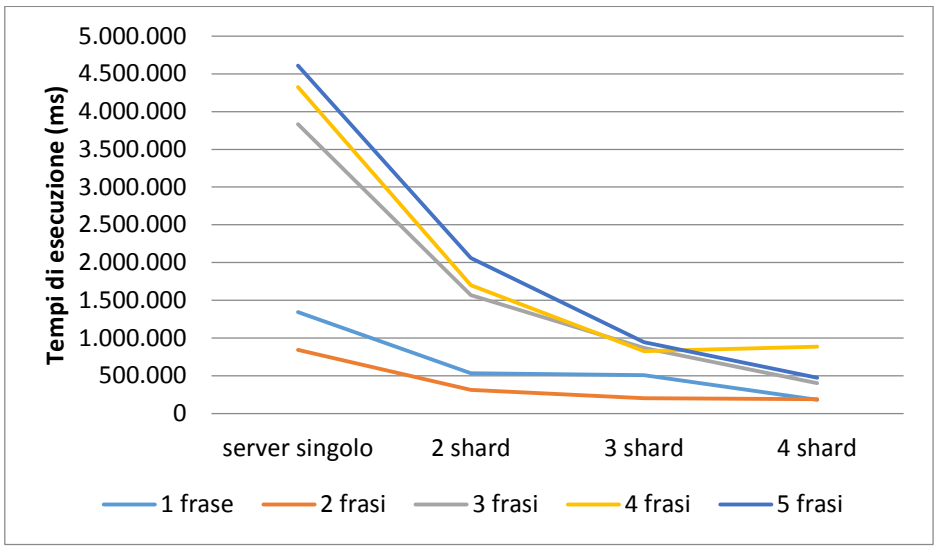

Grafico 3.8. Tempi di query ottenuti testando interrogazioni comprensive di ricerche di frasi esatte e di una condizione di selezione posta sul campo "SOURCECHANNEL" sfruttando un normale text index costruito sui campi testuali "TITLE" e "CONTENT".

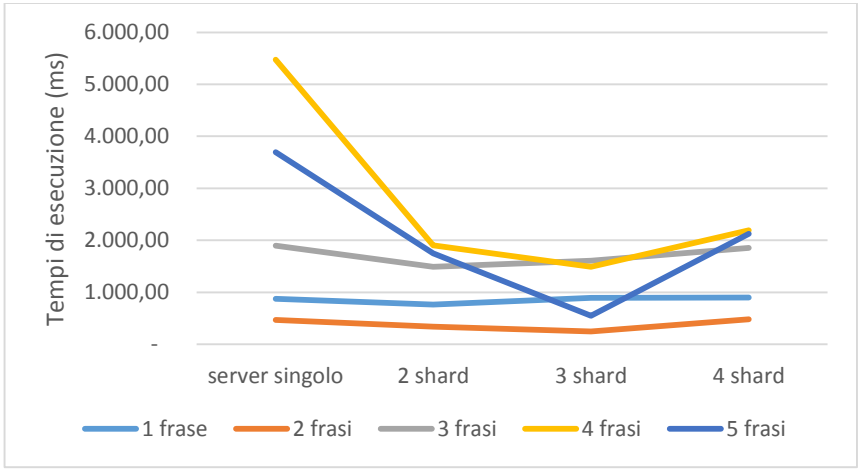

Grafico 3.9. Tempi di query ottenuti testando interrogazioni comprensive di ricerche di frasi esatte e di una condizione di selezione posta sul campo "SOURCECHANNEL" sfruttando un compound index costruito sul campo "SOURCECHANNEL" e sui campi testuali "TITLE" e "CONTENT".

Il vantaggio prestazionale ottenuto dalla costruzione dell'indice composto è evidente, ricordiamo però che MongoDB non ammette l'esistenza di più indici di testo per una stessa collection, prima di avviare la costruzione di un compound index del tipo considerato è stato quindi necessario eliminare il text index già costruito sulla collection "clip". A ciò si aggiunge il fatto che, una volta costruito, tale indice rende impossibile l'esecuzione di ricerche di testo che non siano affiancate ad una condizione di uguaglianza posta sul campo "SOURCECHANNEL". Ciò può costituire un serio problema per coloro che hanno bisogno di eseguire rapidamente ricerche di testo specificando al contempo delle condizioni di selezione su altri campi. Facendo ancora riferimento ai grafici 3.8 e 3.9 possiamo anche notare come l'esecuzione delle query testate abbia beneficiato, in special modo in assenza dell'indice composto, della costruzione del cluster e dell'incremento degli shard che ne facevano parte, trasformando i 4.610.925ms necessari per l'esecuzione dell'interrogazione contenente la ricerca di cinque frasi esatte condotta su singolo server nei 476.253ms sufficienti per ultimare l'esecuzione di quella stessa query sul cluster comprensivo di quattro shard.

Nonostante i limiti a cui abbiamo accennato l'esecuzione di ricerche di testo mediante il metodo `find()` costituisce un importante passo in avanti rispetto alla release 2.4 in cui non era possibile, ad esempio, far uso di `sort()` per imporre un preciso ordinamento al risultato di una ricerca di testo che, di per sé, non è definito. Combinando metodi come `sort()` e `limit()` è ora possibile ottenere in modo semplice dei risultati interessanti come selezionare soltanto gli *n* document più significativi per una data ricerca di testo quale, ad esempio, la ricerca del gruppo di keyword "research university education" come mostrato nella query dell'esempio 3.5.

```
db.clip.find({$text:{$search: "research university
   education", $language: "en"}},{score:{$meta:
        "textScore"}}).sort({score:{$meta:
            "textScore"}}).limit(100);
```

Esempio 3.5.Query che seleziona e restituisce i 100 document a cui l'operatore $text ha assegnato un textScore maggiore fra quelli contenenti, almeno una volta, nel campo "CONTENT" o nel campo "TITLE" una o più delle keyword seguenti: research, univesity education.

Ad arricchire ancora di più le possibilità di applicazione e di utilizzo delle ricerche di testo partecipa la possibilità ora offerta da MongoDB di farne uso anche all'interno dell'Aggregation Pipeline, seppure con delle restrizioni, ampliando così al contempo la capacità di elaborazione e manipolazione dei dati dello stesso aggregation framework. Abbiamo quindi considerato alcune delle query contenenti ricerche di testo che è possibile costruire grazie all'aggregation pipeline e ne abbiamo testato le prestazioni per saggiarne il comportamento al variare della specifica configurazione di MongoDB adottata.

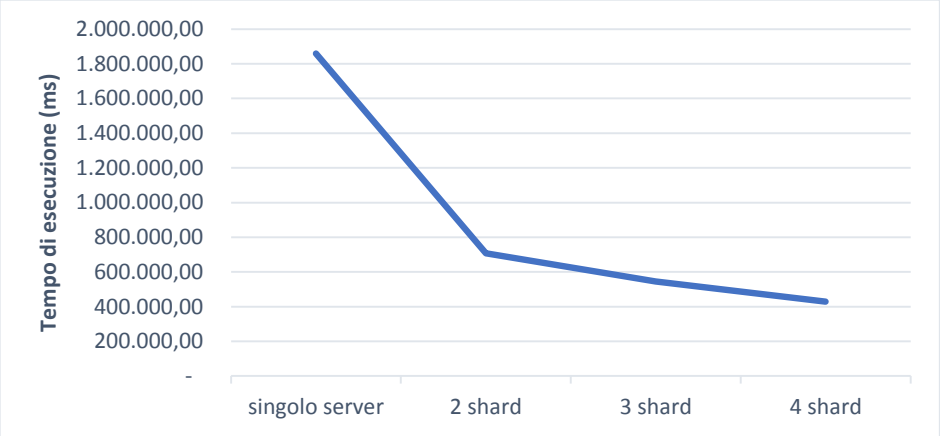

Grafico 3.10. Confronto delle prestazioni dell'esecuzione della ricerca della keyword "minister" e della successiva selezione, fra i document restituiti dall'operatore $text, dei cento risultati più recenti.

Il grafico 3.10 mostra le performance ottenute su singolo server ed in ambito distribuito da una query particolare, realizzata proprio grazie all'introduzione dell'operatore `$text` nello stage `$match` dell'Aggregation Pipeline. Questa query consente di ricercare fra i document della collection "clip" la keyword "minister" e di restituire poi, fra tutti quanti i document così individuati, soltanto i cento a cui sono stati associati, nel campo "DATE", i valori più recenti. È evidente che i tempi di esecuzione registrati non sono affatto trascurabili, è altrettanto visibile però come lo sharding abbia influito positivamente sulle prestazioni ottenute dal sistema, permettendo di ridurre passo dopo passo i tempi di query ottenuti di un fattore circa pari al numero degli shard che costituivano il cluster.

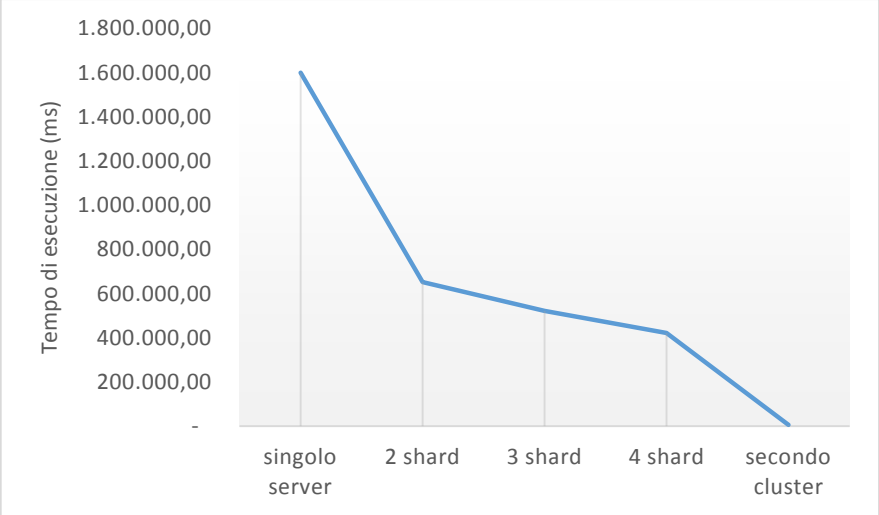

Grafico 3.11. Confronto tra le performance ottenute su singola macchina e su cluster dalla query che, operata la ricerca della keyword "minister", raggruppa il risultato ottenuto sul campo "SOURCE" e poi restituisce unicamente quelle sorgenti che hanno ottenuto un textScore complessivo maggiore o uguale di una specifica soglia minima (2.5).

Il grafico 3.11 propone un altro esempio di query complessa che riesce a mettere bene in luce i benefici che si possono trarre dallo scaling orizzontale, oltre che la ricchezza e la grande espressività offerta dall'aggregation framework e, più in generale, da MongoDB. Ancora una volta, come nel caso della query precedente, il fattore di miglioramento raggiunto grazie al passaggio ad una gestione distribuita riesce a dimostrarsi pressappoco pari al numero di shard considerati. L'esecuzione di questa stessa query inoltre, operata sul secondo cluster (costituito come sappiamo da quattro macchine più performanti di quelle considerate nel cluster sul quale sono stati registrati i tempi riportati nel grafico 3.11) ha richiesto solamente 6.411,5ms, un tempo davvero irrisorio rispetto a quello ottenuto su singolo server (1.599.715ms).

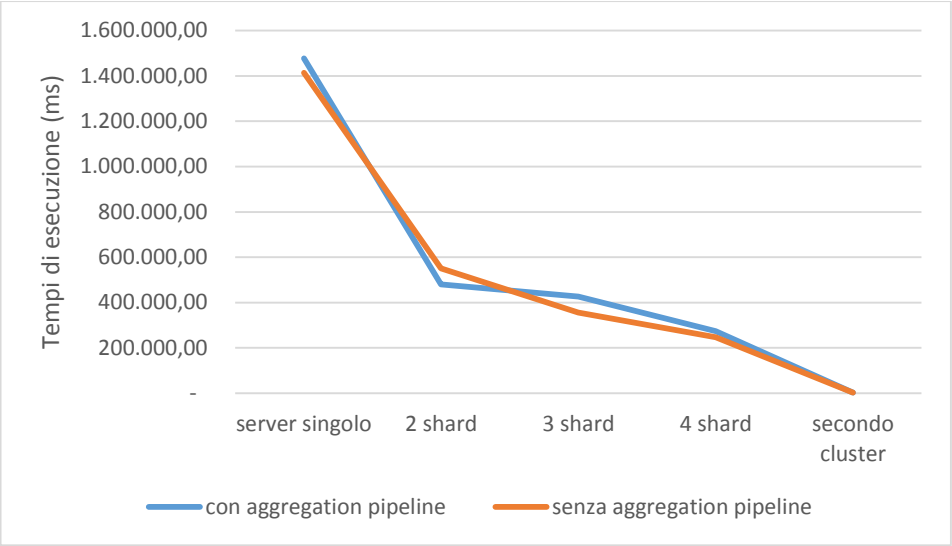

Grafico 3.12. Confronto fra le prestazioni ottenute utilizzando da un lato il metodo find() e dall'altro l'aggregation pipeline per raggiungere il medesimo obiettivo.

Come terzo ed ultimo esempio consideriamo ora l'esecuzione di una determinata ricerca di testo, affiancata dalla selezione dei document di una specifica sorgente e seguita poi dall'imposizione di un certo ordinamento del risultato (definito sul textScore). Questo tipo di operazione, di per sé abbastanza semplice, può essere eseguita sia mediante il metodo `find()` sia attraverso l'Aggregation Pipeline, abbiamo pertanto testato entrambe queste possibili implementazioni confrontandone le prestazioni. I risultati ottenuti nei due casi appaiono piuttosto simili ed entrambi traggono evidenti benefici dalla costruzione dei cluster considerati e dall'esecuzione distribuita delle interrogazioni. Ricordiamo che il campo "SOURCE" è stato selezionato come prima componente della shard key, il filtro riportato in queste query è pertanto una condizione di uguaglianza specificata su un prefisso della shard key e, in quanto tale, può partecipare a rendere ancora più sensibile il miglioramento ottenuto grazie allo scaling orizzontale, sebbene il ridotto numero di shard a nostra disposizione renda difficile il raggiungimento della query isolation.

Quelle considerate sono solo alcune delle interrogazioni che è possibile eseguire sfruttando il connubio di Aggregation Pipeline e ricerche di testo, altri esempi con le relative prestazioni registrate dal sistema in presenza di uno o più server possono essere trovate nell'appendice che segue il presente capitolo e le successive osservazioni conclusive.

Conclusioni

Questo lavoro di tesi si è concentrato sullo studio e l'analisi delle caratteristiche di una specifica tipologia di DBMS NoSQL: MongoDB, di cui abbiamo già sottolineato il ruolo centrale nel panorama attuale dell'elaborazione e della gestione dei dati. MongoDB si sta affermando sempre di più in un contesto in cui una sempre più rapida, ampia ed eclettica evoluzione della realtà, delle abitudini, degli interessi e delle necessità delle applicazioni ha posto ormai in luce limiti e difficoltà dati da alcuni aspetti e caratteristiche intrinseche del modello relazionale. Dopo aver dominato in modo indiscusso l'ambito delle basi di dati per anni ed anni, tale modello, seppure molto probabilmente non lascerà completamente il passo alle nuove tecnologie, deve comunque sempre più frequentemente essere affiancato da nuovi tipi di sistemi, che possano rispondere in modo più efficace, semplice ed economico alle nuove esigenze di flessibilità e scaling.

Lo studio che abbiamo condotto ha verificato come MongoDB sia un prodotto estremamente interessante e promettente, probabilmente destinato a radicare ancor più la sua presenza sul mercato. Abbiamo visto come il suo supporto automatico allo sharding consenta di raggiungere facilmente apprezzabili vantaggi prestazionali, permettendo il raggiungimento di obiettivi ambiti come quello della linear scalability, ossia la possibilità di introdurre un fattore di miglioramento nelle prestazioni che sia proporzionale al numero di shard presenti nel cluster. Sebbene fondamentale in molti contesti applicativi, lo scaling orizzontale non è tuttavia la risposta adeguata ad ogni situazione. Abbiamo verificato infatti che non tutte le tipologie di interrogazioni possono trarre dei benefici dall'operare in un ambito distribuito, nel decidere quindi come strutturare il proprio sistema è necessario analizzare in modo critico ed attento quelle che sono le proprie esigenze, la quantità e la tipologia di dati da gestire così come le operazioni che ad essi si avrà bisogno di applicare e la frequenza con cui la loro elaborazione verrà sottoposta al sistema. Come sottolineato nella stessa documentazione di MongoDB infatti in alcuni casi la costruzione di una architettura distribuita non farebbe altro che aggiungere inutilmente

complessità al sistema, senza trovare il giusto riscatto nella gestione e manipolazione distribuita dei dati.
Gli aspetti da noi analizzati di questo DBMS ne hanno messo in luce comunque la ricchezza, la flessibilità e l'ampia varietà di feature e funzionalità che esso può offrire che, seppure non prive di limiti e problematiche più o meno evidenti e inibenti, sono in rapida e costante evoluzione. È infatti già in fase di sviluppo la nuova release di MongoDB, la versione 3.0, che, nonostante non preveda attualmente miglioramenti dal punto di vista delle ricerche di testo, preannuncia numerosi ed interessanti passi in avanti nell'ambito della gestione degli indici, dei replica set e degli sharded cluster così come nella gestione dei lock, con l'introduzione della possibilità di amministrarli a livello di singola collection e non di un intero database o addirittura di un'intera istanza. Anche le potenzialità dell'Aggregation Pipeline e, più in generale, del linguaggio di interrogazione offerto dal sistema sono in evoluzione, nella release 3.0 è infatti prevista anche l'introduzione di nuovi operatori ed il raffinamento di una delle caratteristiche forse più interessanti e particolari di MongoDB ovvero le query di tipo "spaziali".

In conclusione dunque i risultati ottenuti e le analisi condotte, nonostante abbiano rilevato dei livelli di complessità a volte inattesi ed abbiano indubbiamente individuato dei limiti e delle problematiche ancora aperte, hanno comunque tratteggiato un quadro complessivamente positivo del sistema, con la forte speranza che il costante impegno volto al suo miglioramento ed al raffinamento delle sue promettenti potenzialità possa nei mesi e negli anni a venire consolidarne i punti di forza e migliorare gli aspetti che appaiono, al contrario, ancora deboli, contestabili o comunque limitanti.

Bibliografia

[1] MongoDB, http://www.mongodb.org/, 2014.

[2] MongoDB Inc., "MongoDB named a leader by independent research firm", Palo Alto, California, and New York—October 1, 2014. http://www.mongodb.com/press/mongodb-named-leader-independent-research-firm

[3] DB-Engine, "DB-Engines Ranking", 2015, http://db-engines.com/en/ranking

[4] E.Redmond, J.R.Wilson, *Seven Databases in Seven Weeks* , Jaqueline Carter, United States of America, 2012.

[5] R. Boraso, D. Guenzi, "Architetture scalabili per memorizzazione, analisi, condivisione e pubblicazione di grosse moli di dati", *Archeologia e Calcolatori*, Supplemento 4, 2013, pp. 139-146.

[6] Pramod J. Sadalage e Martin Fowler, *NoSQL Distilled A Brief Guide to the Emerging World of Polyglot Persistence*, Addison-Wesley, Crowfordsville, IN, 2012.

[7] Dan McCreary e Ann Kelly, *Making Sense of NoSQL. A guide for managers and the rest of us*, Manning Publications Co., Shelter Island, NY, 2014.

[8] Rick Cattell, "Scalable SQL and NoSQL Data Stores", pubblicato originalmente nel 2010, ultima revisione nel dicembre del 2011.

[9] Manuel Scapolan, "NoSQL. Il database relazionale va in pensione, avanza il movimento NoSQL.", 2012, http://www.manuelscapolan.it/2012/05/il-database-relazionale-va-in-pensione-avanza-il-movimento-nosql/

[10] The Neo4j Team, "The Neo4j Manual v2.1.7", http://neo4j.com/docs/2.1.7/ 2015

[11] "Apache CouchDB™ 1.6.1 Documentation",

http://docs.couchdb.org/en/1.6.1/contents.html, 2014

[12] MongoDB Inc., *The MongoDB 2.6 Manual*, 2014, http://docs.mongodb.org/manual/, 2014

[13] "The GNU Affero General Public License per MongoDB" http://blog.mongodb.org/post/103832439/the-agpl , 2010

[14] DB-Engine, "MongoDB System Properties", 2015, http://db-engines.com/en/system/MongoDB

[15] "Top five NoSQL considerations", *MongoDB white paper*, luglio 2014.

[16] A. Poggi, "Object-Relational Mapping", 2010-2011, http://www.dis.uniroma1.it/~poggi/didattica/progettoas11/lezioni/9-ORM.pdf

[17] Project Voldemort Home Page, 2014, http://www.project-voldemort.com/voldemort/

[18] Eric A. Brewer, "Towards Robust Distributed Systems", 2000, https://www.cs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf

[19] Wikipedia, http://it.wikipedia.org/wiki/BigTable, 2014

[20] L. Hofhansl, "Acid in HBase", 2012, http://hadoop-hbase.blogspot.it/2012/03/acid-in-hbase.html

[21] Wikipedia, http://it.wikipedia.org/wiki/Neo4j, 2014

[22] Apache CouchDB Relax, http://couchdb.apache.org/, 2014

[23] Wikipedia, http://it.wikipedia.org/wiki/Base_di_dati_orientata_al_documento, 2014

[24] Apache HBase Team, "Apache HBase ™ Reference Guide", 2014, http://hbase.apache.org/book.html

[25] Kristina Chodorow e Michael Dirolf, "*MongoDB. The definitive guide"*, United States of America, O'Reilly Media, 2013

[26] Wikipedia, http://it.wikipedia.org/wiki/MongoDB#Caratteristiche_Principali, 2014

[27] Wikipedia, http://en.wikipedia.org/wiki/Readers%E2%80%93writer_lock, 2014

[28] Manuel Bianchi, “MongoDB Analisi e prototipazione su applicazioni di Social Business Intelligence”, 2013

[29] Simone Marini, "Il movimento NoSQL. Il caso di studio di CouchDB", 2013

[30] Dario Maio, “Il modello relazionale”, 2014, http://bias.csr.unibo.it/maio/courses/bd/bd.htm#mat

[31] Forrester Research Inc. “The Forrester Wave™: NoSQL Document Databases, Q3 2014,” by Noel Yuhanna, September 30, 2014.

Appendice A – Altri esempi di query che coniugano Aggregation Pipeline e ricerche di testo

La prima query considerata permette di raggruppare sulla sorgente per poi ottenere nel risultato: il numero di document restituiti dalla ricerca di testo per ciascuna sorgente e l'insieme (eventuali elementi duplicati verranno rimossi automaticamente) degli score dei document ottenuti per ogni sorgente. Il codice che consente di realizzare questa interrogazione è mostrato nell'esempio A.1 mentre risultati ottenuti dalla sua esecuzione grafico A.1.

```
db.clip.aggregate([{$match: {$text: {$search:
    "minister",$language: "en"}}},{$group:
{_id:"$SOURCE", numDocuments: {$sum:1}, scores:
       {$addToSet:{$meta: "textScore"} }
    }}],{allowDiskUse: true});
```

Esempio A.1. Testo della prima query dell'appendice A.

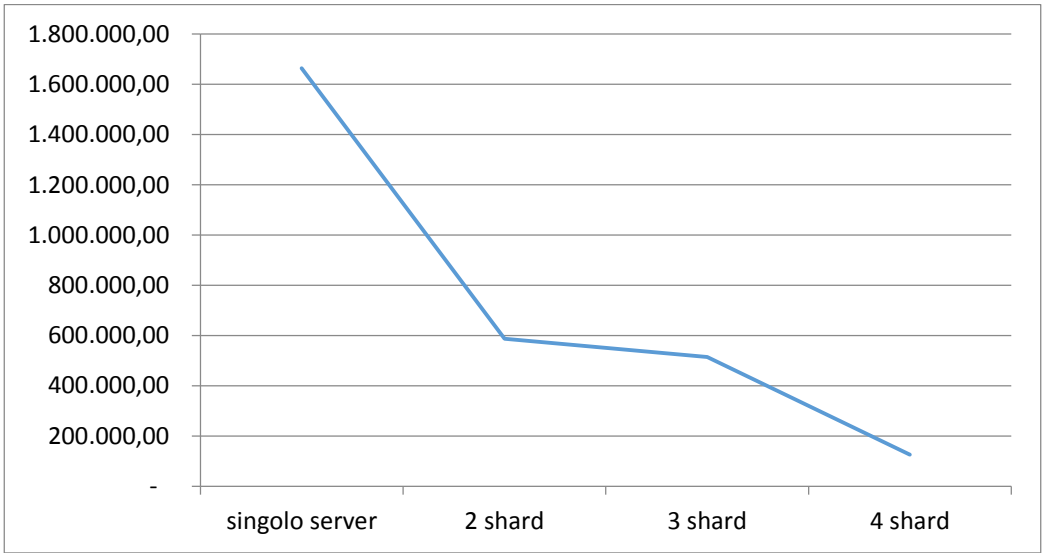

Grafico A.1. Risultati ottenuti dall'esecuzione della prima query considerata nell'appendice A

La seconda query che esaminiamo restituisce solo quei document a cui è stato associato un textScore più alto di una certa soglia minima (1.0) ricercando nei campi indicizzati dal text index la keyword “minister”. Di ogni document vengono riportati nel risultato solo il titolo ed il textScore. Codice della query nell’esempio A.2 e risultati visualizzabili nel grafico A.2.

```
db.clip.aggregate( [{ $match: { $text:{ $search:
"minister", $language:"en" }}},{ $project: { TITLE:
1, _id: 0, SOURCE: 1, score: { $gt:[ {$meta:
"textScore"}, 1.0]} }}]);
```

Esempio A.2. Testo della seconda query considerata nell’appendice A.

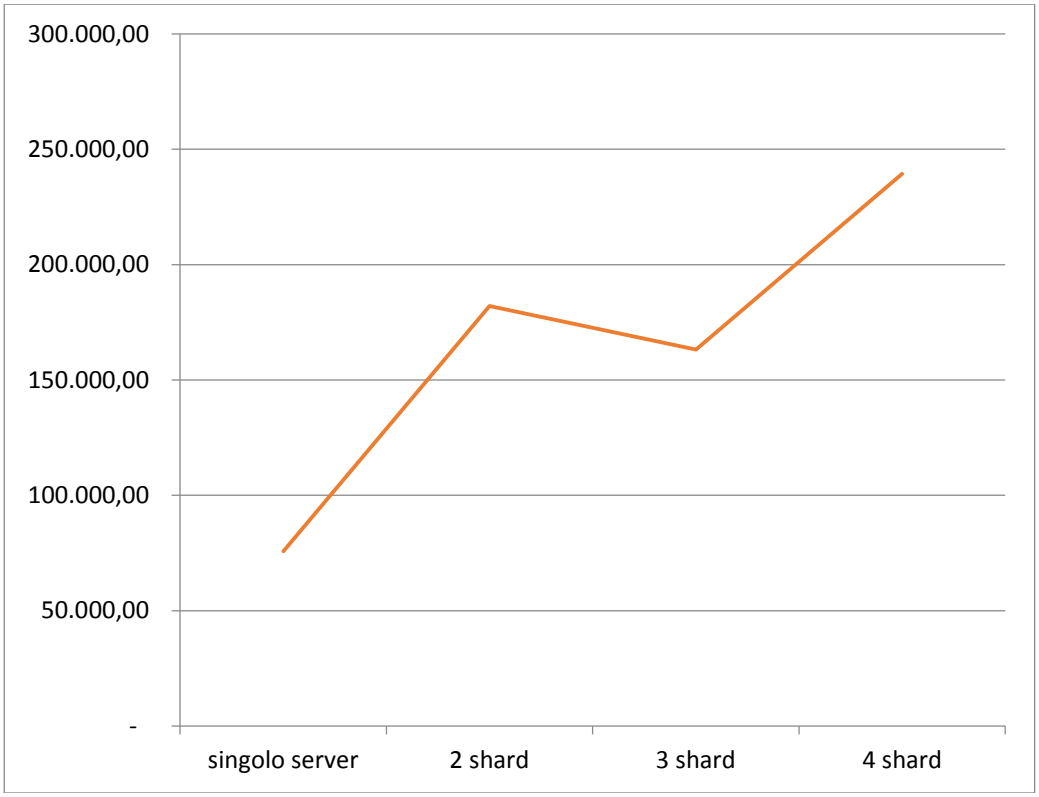

Grafico A.2. Risultati ottenuti dai test effettuati per la seconda query riportata nell’appendice A.

Con la query che stiamo ora per prendere in considerazione si ottiene, per ogni SOURCE, la media, il massimo, il minimo e la somma dei textScore associati ai document di quella particolare sorgente a seguito della ricerca di testo della keyword "minister". Il codice, come al solito è riportato nell'esempio A.3 ed il confronto fra i risultati ottenuti eseguendo la query su singolo server e su cluster è mostrato nel grafico associato, ovvero il grafico A.3.

```
db.clip.aggregate([{$match: {$text: {$search:"minister", $language:"en"}
}},{$group:{_id:"$SOURCE", sommaScore: {$sum: {$meta:
"textScore"}},mediaScore: {$avg: {$meta: "textScore"}}, minScore:
{$min:{$meta: "textScore"}}, maxScore: {$max:
{$meta:"textScore"}}}}],{allowDiskUse: true});
```

Esempio A.3. Codice della terza query esaminata nell'Appendice A.

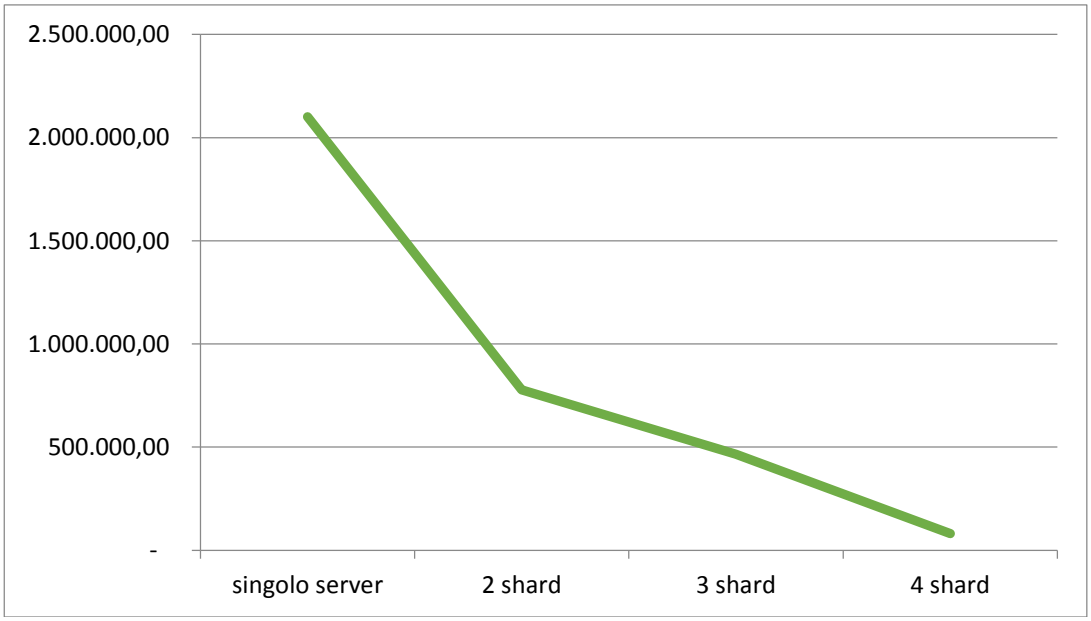

Grafico A.3. Risultati ottenuti eseguendo la terza query dell'appendice A.

La quarta ed utlima query considerata permette di operare un raggruppamento sul campo “SOURCE” e per ogni gruppo viene restituito solo il titolo ed il textScore del document che, in quel particolare gruppo, ha ottenuto il textScore massimo dalla ricerca della keyword singola “minister”. Ancora una volta abbiamo riportato il codice della query, visibile nell’esempio A.4, ed i risultati registrati dai test su di essa eseguiti, mostrati nel grafico A.4.

```
db.clip.aggregate([{$match:{$text:{$search:"ministe
r",$language:"en"}}},{$group:{_id:{source:"$SOURCE"
,score:{$meta:"textScore"},title:"$TITLE"}}},{$sort
:{"_id.score":1}},{$group:{_id:"$_id.source",maxSco
reTitle:{$last:"$_id.title"},maxScore:{$last:"$_id.
score"}}}],{allowDiskUse:true});
```

Esempio A.4. Codice dell’ultima query dell’appendice A.

Grafico A.4. Risultati ottenuti eseguendo la quarta query dell’appendice A.