ALMA MATER STUDIORUM

UNIVERSITÀ DI BOLOGNA – SEDE DI CESENA

FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI

Corso di Laurea Triennale in Scienze e Tecnologie Informatiche

MongoDB
Analisi e prototipazione su applicazioni di Social Business Intelligence

Tesi di Laurea in

Laboratorio di Basi di Dati

Relatore:

Chiar.mo Prof.

Matteo Golfarelli

Candidato:

Manuel Bianchi

II Sessione

2012/2013

Indice

Introduzione

L'ultimo decennio ha visto un radicale cambiamento del mercato informatico, con la nascita di un numero sempre maggiore di applicazioni rivolte all'interazione tra utenti. In particolar modo, l'avvento dei social network ha incrementato notevolmente le possibilità di creare e condividere contenuti sul web, generando volumi di dati sempre maggiori, nell'ordine di petabyte e superiori. La disponibilità di un numero così massivo di informazioni ha trasformato il valore percepito del singolo dato, giudicato sempre meno rappresentativo, in favore di analisi effettuate su quantità enormi di informazioni, allo scopo di individuare comportamenti comuni tra utenti e stabilire le tendenze di mercato. Tali possibilità hanno permesso la nascita di nuovi settori dell'informatica orientati all'analisi delle informazioni, con il preciso obiettivo di fornire maggiori strumenti in fase decisionale. Tra di essi figura la Social Business Intelligence, settore in cui lo studio delle informazioni collezionate dai social media permette di stabilire in tempo reale le reazioni degli utenti a un determinato prodotto o servizio, attraverso visualizzazioni dei risultati immediate e significative. La disponibilità di tali quantità di dati ha ampliato le possibilità di analisi, ma, al tempo

stesso ha generato nuove sfide e difficoltà nella loro gestione. Al crescere delle dimensioni di un database crescono anche i tempi richiesti per le interrogazioni, al punto che i sistemi di gestione dati basati sul modello relazionale non riescono più a modellare il problema garantendo alte prestazioni in fase di interrogazione. Tale situazione rende conveniente la ricerca di strumenti alternativi, creati appositamente per supportare operazioni di analisi massiva. In questi scenari viene posta grande attenzione alla suddivisione dei dati su server multipli (cluster) secondo modalità che prediligano le prestazioni e la semplicità di manutenzione. Il movimento NoSQL nasce come possibile soluzione a tali necessità, con un forte orientamento alla gestione in ambiente distribuito di grandi quantità di dati eterogenei tra loro. Lo scopo di questo studio di tesi è quello di illustrare come i sistemi NoSQL, nello specifico caso di MongoDB, cerchino di sopperire alle difficoltà d'utilizzo dei database relazionali in un contesto largamente distribuito. La tematica del cluster sarà dunque un argomento ricorrente e rappresenterà il punto di vista attorno al quale ruoterà la trattazione. Le motivazioni della scelta di MongoDB come caso di studio risalgono non solo nell'elevata popolarità che questo prodotto gode all'interno del panorama non relazionale, ma anche nello sviluppo rapido e costante a cui è soggetto. Attraverso un ciclo di rilasci frequenti, MongoDB ha visto espandere notevolmente le proprie caratteristiche negli ultimi anni, con l'aggiunta di funzionalità peculiari quali la ricerca full-text. Proprio quest'ultima caratteristica sarà motivo di studio ed elemento fondante di un prototipo di applicazione appositamente progettato. Lo scopo dell'applicazione è quello di fornire un primo demo tecnologico delle possibilità offerte da MongoDB attraverso funzionalità di ricerca di keyword in campi di testo indicizzati. Il prototipo sviluppato andrà poi ad aggiungersi come modulo a un'applicazione di analisi semantica già esistente e rappresenterà dunque un primo oggetto su cui valutare le potenzialità dei database non relazionali. Conclusa la progettazione dell'applicazione, si rende necessaria una valutazione delle prestazioni e dei benefici che tale programma può ottenere in un ambiente basato su cluster. E' logico supporre che in uno scenario

di distribuzione dati su più macchine ci sia un guadagno prestazionale rispetto ai tempi di query ottenuti effettuando interrogazioni su singolo server; verranno pertanto svolti appositi test per determinare, qualora esista, l'entità di questo incremento. Per una trattazione più approfondita si è deciso di estendere lo studio prestazionale a un numero di funzionalità maggiore di quelle previste nell'applicazione, includendo così operazioni quali conteggio di elementi, aggregazione di dati e distinzione di documenti.

Volendo fornire una panoramica sull'organizzazione degli argomenti esposti in questo studio di tesi, di seguito si segnalano le tematiche affrontate in ciascuna sezione. Nel primo capitolo verranno trattati i problemi del modello relazionale e le difficoltà nell'applicare tali soluzioni su cluster largamente distribuiti. Verranno dunque analizzate le metodologie proposte dai sistemi NoSQL per risolvere tali problematiche e si fornirà una descrizione delle principali caratteristiche di tali strumenti. Il secondo capitolo vedrà invece lo studio delle funzionalità messe a disposizione da MongoDB. Tra le caratteristiche che saranno motivo di attenzione si distinguono il peculiare modello dati e la gestione della memoria, due delle differenze più evidenti rispetto ai sistemi relazionali. Gli altri argomenti che saranno trattati riguardano la distribuzione dei dati su cluster, le modalità con cui i meccanismi di replicazione aumentino la tolleranza ai guasti, le operazioni rese possibili dalle varie tipologie di indici offerte e infine verrà offerta una panoramica sulle metodologie di aggregazione dei dati. Nel terzo capitolo verrà illustrata l'applicazione sviluppata, descrivendone le finalità, la modellazione e le caratteristiche offerte. Il quarto capitolo sarà invece destinato all'analisi prestazionale di MongoDB in ambiente distribuito. L'obiettivo di questa analisi sarà quello di determinare la scalabilità delle varie operazioni messe a disposizione da MongoDB.

Capitolo 1

Il movimento NoSQL

1.1 Il modello relazionale

I Database Management System (DBMS) basati sul modello relazionale rappresentano lo standard de facto nella gestione dei dati. Vennero introdotti nel 1970 da Edgar Frank Codd nell'articolo "A Relational Model of Data for Large Shared Data Banks" [**1**], con lo scopo di fornire un modello che strutturasse i dati tramite relazioni, identificabili visivamente come tabelle. Affinché un sistema per la gestione di basi di dati possa essere definito relazionale deve soddisfare 13 regole o forme normali [**2**] mirate a garantire una definizione coerente delle informazioni, l'accessibilità dei dati con relativa manipolazione e l'indipendenza dalla rappresentazione fisica e logica. Un RDBMS (relational DBMS) nasce dunque con un forte orientamento alla rappresentazione strutturata dei dati, modello che permette di essere sfruttato con buone prestazioni in un notevole numero di scenari applicativi; questo vantaggio in breve tempo portò il modello relazionale a essere l'elemento dominante nelle

tecnologie per la gestione dei dati. Un'analisi effettuata da Solid IT [**3**] mostra come le prime posizioni nella classifica di popolarità dei DBMS siano interamente occupate da soluzioni basate sul modello relazionale. Il risultato ottenuto non è necessariamente fedele al market share delle soluzioni analizzate, in quanto il punteggio è calcolato sulla base dell'interesse mostrato dagli utenti, ma è comunque indicativo di una forte preponderanza dei sistemi SQL-like.

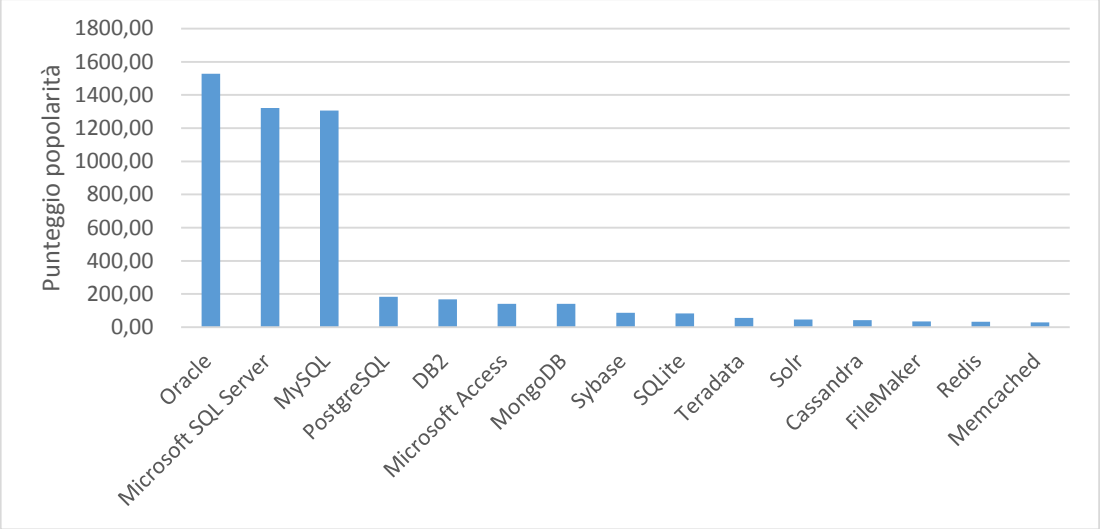

Figura 1.1 Popolarità dei principali DBMS (Luglio 2013)

1.2 Limiti del modello relazionale

Negli ultimi anni si è assistito a un rapido e importante cambiamento nel mercato relativo alla gestione dei dati, che ha evidenziato come il modello relazionale non possa essere utilizzato efficacemente in ogni situazione. Tra i settori che necessitano di nuove soluzioni create a misura del problema troviamo, a esempio, la gestione dei Big Data e meccaniche che supportino le nuove metodologie di sviluppo software, come l'*agile software development*. Con il termine Big Data si fa riferimento a dataset la cui dimensione sia diversi ordini di grandezza superiore a un tipico database; tale

definizione è ovviamente dipendente dal periodo storico in esame, infatti mentre nei primi anni ’90 si poteva ascrivere a questa categoria database con dimensioni dell’ordine dei gigabyte, oggi si considerano come Big Data collezioni dell’ordine dei petabyte o addirittura exabyte. Lavorando con quantità di dati così significative, la complessità strutturale dei RDBMS rappresenta un serio problema e diventano interessanti soluzioni alternative. Uno studio di Couchbase [**4**] mostra le motivazioni che spingono le aziende intervistate (su un campione di 1300) a cercare soluzioni alternative al modello relazionale nella gestione dei Big Data.

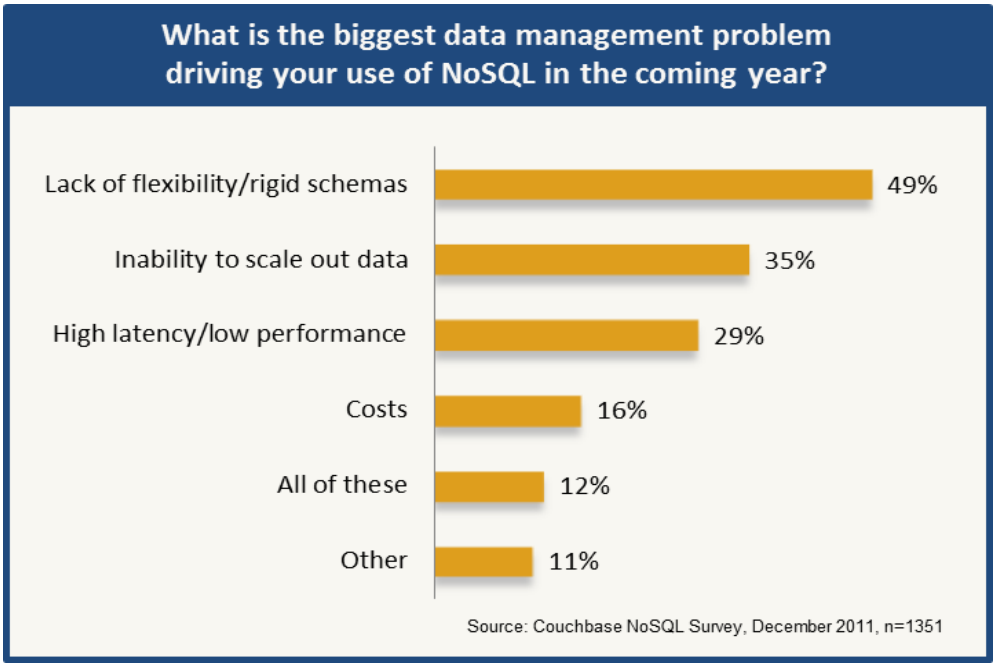

Figura 1.2 Fattori limitanti percepiti lavorando con Big Data (Dicembre 2011)

Si può notare come l’aspetto ritenuto più limitante sia la rigidità dello schema, in particolar modo la necessità di lavorare con dati non omogenei come email, video, conversazioni audio ed elementi di interazione tra utenti come tweet e post dei vari social network. Per il periodo 2010-2015 è prevista una crescita delle dimensioni dei

Big Data pari al 35% annuale [**5**] pertanto anche la scalabilità dei dati viene ritenuto un fattore molto importante, soprattutto dove sono richieste prestazioni elevate.

1.3 Il movimento NoSQL

Le difficoltà nell'uso dei DBMS relazionali in certi settori ha richiesto lo sviluppo di soluzioni alternative, comunemente raggruppate sotto il termine NoSQL (Not Only SQL). Il movimento NoSQL nasce nel 2009 come conseguenza di una serie di conferenze tenute da Jon Oskarsson e, in breve tempo, si sviluppano numerose soluzioni alternative dalle caratteristiche più disparate. Rick Cattell, nell'articolo "Scalable SQL and NoSQL Data Stores" [**6**] elenca le proprietà comuni alla maggioranza dei sistemi di tipo NoSQL:

- Possibilità di replicare e distribuire partizioni di dati su più server
- Possibilità di scaling orizzontale per "operazioni semplici"
- Un'interfaccia o protocollo di chiamata semplificato, a differenza delle implementazioni SQL basate su *binding*
- Un modello di concorrenza più debole rispetto a quello garantito dai DBMS che supportano tutte le proprietà ACID
- Uso efficiente della memoria e di indici distribuiti
- Capacità di aggiungere dinamicamente nuovi attributi

La *scalabilità orizzontale* è una caratteristica di particolare importanza quando si parla di Big Data. Con questo termine si intende la possibilità di distribuire i dati e le operazioni su macchine fisiche differenti, senza memoria o dischi rigidi in comune, al fine di parallelizzare le operazioni e ottenere un minore quantitativo di dati da elaborare per singolo server. Nonostante sia possibile aumentare le prestazioni anche tramite *scaling verticale* (aumentando il numero di core e processori di una singola macchina),

l'approccio distribuito permette di ridurre i costi utilizzando macchine assemblate e permette di superare i limiti hardware dati dal quantitativo massimo di core e memoria installabili su un singolo mainframe. La scalabilità ha assunto un ruolo particolarmente importante con l'avvento del social web 2.0, ambito nel quale le operazioni più comuni sono semplici ma molto frequenti, come l'aggiornamento di singoli contatori o piccoli gruppi di record; in queste situazioni si richiede la possibilità di effettuare velocemente queste operazioni per far fronte al carico di milioni di utenti contemporanei. Una distribuzione dei dati agile e automatica è una caratteristica di principale importanza per un DBMS non relazionale.

Una caratteristica fondamentale in un RDBMS è l'aderenza alle proprietà ACID:

- **Atomicity**: una transazione può solo terminare con un successo o con un *rollback*, non sono permesse esecuzioni parziali
- **Consistency**: una transazione deve lasciare il database in uno stato coerente, ovvero non devono verificarsi contraddizioni tra i dati archiviati
- **Isolation**: ogni transazione deve essere eseguita in modo isolato e indipendente dalle altre transazioni
- **Durability**: una volta che una transazione effettua un *commit*, i cambiamenti apportati non dovranno più essere persi

L'applicazione di queste proprietà in un sistema distribuito è stata analizzata da Eric Brewer nell'articolo "Towards robust distributed systems" [**7**], dal quale si è formalizzato il Teorema CAP. Esso specifica come in un contesto distribuito si possano identificare 3 proprietà fondamentali:

- **Consistency**: i dati rimangono consistenti alla fine dell'esecuzione di un'operazione, ovvero una volta effettuata un update tutti i client dispongono degli stessi dati

- **Availability**: ogni richiesta effettuata al database riceve una risposta su ciò che sia riuscito o fallito
- **Partition Tolerance**: il sistema continua a funzionare anche qualora la comunicazione tra server sia inaffidabile e si verifichino perdite di messaggi. L'unico evento che possa causare l'impossibilità di comunicazione è la mancanza globale di connettività (total network failure)

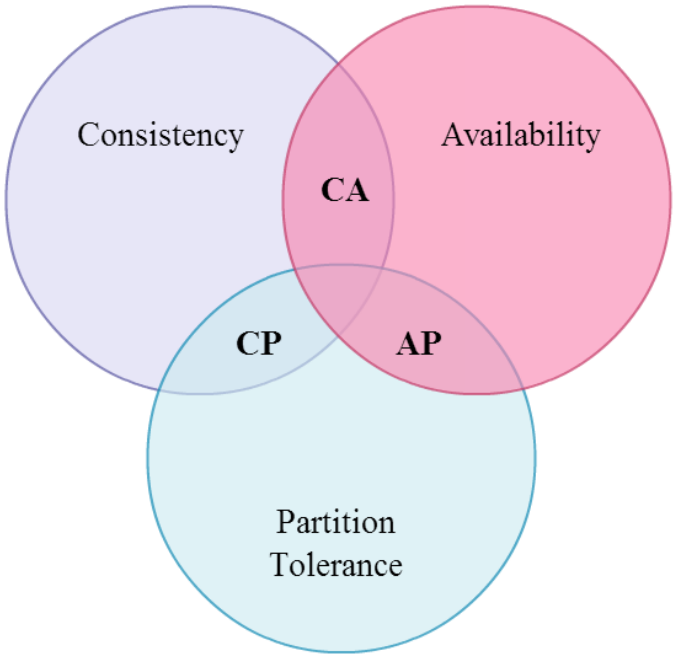

Figura 1.3 Diagramma Teorema CAP

Uno studio compiuto da Seth Gilbert e Nancy Lynch [**8**] afferma come sia possibile soddisfare contemporaneamente solo due delle tre proprietà presentate, ottenendo dunque i seguenti scenari:

- **Consistency + Availability (CA):** tutti i nodi sono in contatto tra di loro. Quando avviene una partizione della rete (le connessioni di rete tra due gruppi

di sistemi vengono a mancare contemporaneamente), il sistema si blocca. E' il caso dei DBMS relazionali quali PostgreSQL e MySQL

- **Availability + Partition Tolerance (AP):** il sistema è sempre disponibile, ma è possibile che alcuni dati restituiti siano inaccurati. Esempi di questo modello sono CouchDB, Cassandra, Riak e Voldemort
- **Consistency + Partition Tolerance (CP):** alcuni dati possono non essere accessibili, ma i rimanenti sono considerati consistenti. Tipici esempi di database fondati su questo modello sono MongoDB, Redis, HyperTable e Memcached

Le conseguenze del teorema non sono strettamente limitanti e focalizzarsi su due attributi non esclude completamente il terzo, ma semplicemente non rende possibile garantirlo a priori; per esempio, è possibile usare efficacemente un DBMS relazionale in ambiente cluster soddisfacendo la condizione di partition tolerance, purché il numero di nodi sia limitato [**9**].

A fronte dei risultati offerti dal teorema CAP, i database di tipo NoSQL, per poter essere scalabili come richiesto, devono sacrificare delle proprietà e dunque non possono in alcun modo aderire strettamente al modello ACID. Seppur non valga per tutti i DBMS NoSQL, si ha che la maggioranza di essi segue una logica operazionale denominata BASE (Basically Available Soft-state Eventual consistency) nella quale si pongono in primo piano le prestazioni e la disponibilità dei dati, a fronte di una minore consistenza. Quest'ultima viene definita "eventuale" in quanto non è possibile garantirla sempre a priori, ma è comunque presente in una forma più debole. Phyo Thandar Thant e Thinn Thu Naing nell'articolo "Improving the Availability of NoSQL Databases for Cloud Storage" [**10**] descrivono il ruolo della consistenza all'interno del paradigma BASE elencando le seguenti proprietà:

- **Causal consistency**: la sessione incaricata di leggere un dato ne vedrà sempre la versione più recente
- **Read your own writes**: la sessione che effettua un cambiamento al database vedrà immediatamente la modifica, anche qualora le altre sessioni subissero un ritardo
- **Monotonic consistency**: una sessione non vedrà mai un dato tornare al suo valore originario, leggendo un'informazione non sarà possibile ottenerne una versione datata.

In scenari di replicazione la consistenza può anche essere descritta attraverso la notazione NRW dove N è il numero di copie totali di un dato che il database deve mantenere, R il numero di copie del dato a cui l'applicazione client accederà in lettura e W il numero di copie che devono essere effettuate in fase di scrittura per convalidare il successo dell'operazione. Secondo questa notazione otteniamo la seguente tabella:

Tabella 1.1 Analisi consistenza tramite notazione NRW

Attributi	Consistenza
W = N, R = 1	Forte consistenza, buone prestazioni in fase di lettura
W = 1, R = N	Forte consistenza, buone prestazioni in fase di scrittura
W + R <= N	Consistenza debole ed eventuale
W + R > N	Forte consistenza attraverso politiche di QUORUM

Le politiche di quorum permettono di ottenere buoni risultati in termini di consistenza e al tempo stesso un certo grado di tolleranza ai guasti.

Il fattore quorum può essere calcolato con la seguente formula:

QUORUM = (fattore di replicazione / 2) +1

Per esempio, con un fattore di replicazione pari a 3, si ha un quorum uguale a 2, capace dunque di tollerare guasti a 1 replica server. Con un fattore di replicazione pari a 6, si ha un quorum di 4 e quindi la possibilità di tollerare 2 server di replica offline [**11**]. Quando le politiche di quorum sono attivate, ogni scrittura deve essere registrata nel *commit log* e nelle opportune tabelle per un numero di nodi di replicazione pari al valore del fattore quorum. Al tempo stesso, una lettura viene considerata valida solo quando un numero di server pari al valore del quorum restituisce il dato richiesto, per poi offrire in output all'utente il valore, tra quelli ottenuti, con il timestamp più recente. Un esempio di implementazione efficace basata su quorum è quella offerta dal DBMS Cassandra, nel quale è possibile specificare un livello di dettaglio delle politiche di quorum personalizzabile in base alle proprie esigenze di consistenza dei dati.

I DBMS NoSQL che pongono l'attenzione sulla consistenza e sulla tolleranza alle partizioni di rete hanno invece la particolarità di non potere garantire una disponibilità dei dati in senso forte. Come si è già accennato, un tipico esempio di questa categoria è dato dal DBMS orientato ai documenti MongoDB. In questo caso è possibile identificare due scenari di applicazione del teorema CAP, secondo meccaniche di sharding e/o di replicazione.

A livello di cluster basato su sharding, MongoDB offre:

- Consistenza forte: un dato risiede su uno e un solo shard, pertanto non è possibile ottenere valori inconsistenti

- Piena tolleranza alle partizioni: anche nell'eventualità si abbia una partizione nella rete, le interrogazioni non ritornano dati incorretti. Gli shard continuano a lavorare in modo indipendente
- Disponibilità dei dati in senso debole: letture e scritture su uno shard offline non sono possibili e l'operazione non potrà terminare con successo

A livello di replicazione dei dati si hanno le stesse proprietà, ma garantite secondo differenti politiche:

- Consistenza forte: come comportamento predefinito, tutte le letture sono gestite da un singolo server, ovvero il nodo primario
- Piena tolleranza alle partizioni: se un numero sufficiente di nodi non sono più raggiungibili, viene automaticamente eletto un nuovo server primario, garantendo sempre la possibilità di ricezione delle richieste
- Disponibilità dei dati in senso debole: quando il server primario non è raggiungibile non è possibile accedere ad alcun dato

Bisogna comunque notare come MongoDB sia un caso particolare di database NoSQL dove non è possibile definire a priori il grado di fedeltà al teorema CAP. E' possibile ottenere, attraverso opportune configurazioni e richieste da parte dei client, una maggior disponibilità dei dati a fronte di una minore consistenza degli stessi; un esempio di configurazione possibile ma non predefinita è data dalla possibilità di effettuare letture direttamente sui server secondari, aumentando notevolmente la disponibilità dei dati, ma accettando implicitamente di leggere dati eventualmente non aggiornati.

1.4 Classificazione dei database NoSQL

Negli ultimi anni sono state fornite diverse categorizzazioni dei database NoSQL, alcune particolarmente dettagliate come quella di Yen [**12**], altre maggiormente orientate a specifici settori come quella fornita da Ken North che pone enfasi sull'ambiente cloud [**13**]. Una tassonomia che si distingue per semplicità ed efficienza è quella espressa da Rick Cattell [**6**] e di seguito sarà presa come riferimento.

Tabella 1.2 Tassonomia dei database NoSQL

Tipologia	Esempi
Chiave/valore	Redis, Scalaris, Tokyo Cabinet, Project Voldemort, Riak
Orientato ai documenti	MongoDB, SimpleDB, CouchDB, Terrastore
Extensible Record	Bigtable, HBase, HyperTable, Cassandra

I database chiave/valore rappresentano il modello dati più semplice, in quanto formati da una tabella associativa (dizionario) che permette agli utenti di ottenere un valore data la rispettiva chiave. Nonostante database di questo tipo esistano da diversi anni (Berkeley DB venne rilasciato nel 1991), recentemente si è vista una notevole crescita di questo modello dati, influenzata in larga parte da Dynamo, DBMS closed-source di Amazon. La caratteristica principale dei database chiave/valore è massimizzare la scalabilità dei dati e delle operazioni, pertanto vengono generalmente a mancare garanzie di consistenza, così come alcune funzionalità di interrogazione quali aggregazioni o join. La lunghezza delle chiavi è spesso limitata a un certo numero di byte, mentre generalmente ci sono meno imposizioni sui valori, sia come dimensione, sia come tipo di dato. Il formato dati chiave/valore trova applicazione immediata in

scenari di caching, dove l'informazione viene legata a una chiave univoca e si pone particolare interesse alla fase di recupero del dato. Un'implementazione molto diffusa basata su questi meccanismi è fornita da Memcached, un sistema di caching in ram particolarmente orientato alle prestazioni. Tale applicativo mantiene gli oggetti esclusivamente in memoria centrale e non sono offerte possibilità di persistenza dei dati; per assolvere a questa mancanza sono stati sviluppati diversi DBMS di tipo chiave/valore basati su Memcached che permettono la memorizzazione fisica delle informazioni. Esempi tipici di questi sistemi sono MemcacheDB e Couchbase Server.

Altri DBMS chiave/valore come Project Voldemort e Riak si differenziano dai concorrenti per l'uso del controllo di concorrenza multiversione (MVCC) invece di politiche fondate sull'uso estensivo di lock. Nei sistemi basati su MVCC a ogni oggetto viene associato un timer o un timestamp, grazie al quale è possibile tenere traccia delle varie versioni del dato in esame. I lock sono comunque presenti, ma in forma debole, infatti un lock acquisito in fase di lettura delle informazioni non entra in conflitto con uno acquisito in fase di scrittura. La mancanza di blocchi nel sistema aumenta le prestazioni delle operazioni, ma richiede al tempo stesso meccaniche che eliminino le versioni inutilizzate dei dati, aumentando la complessità del sistema.

Riak, in particolar modo, presenta un numero maggiore di funzionalità rispetto agli altri database chiave/valore, come la capacità di memorizzare oggetti in formato JSON e quindi disporre di campi multipli in modo analogo ai database orientati ai documenti.

Un esempio di oggetto memorizzato in Riak è presentato in figura 1.4.

```
{
    "bucket":"clienti",
    "key":"12345",
    "object":{
        "nome":"Mario",
        "telefono":"0123456789"},
    "links":[
        ["vendite","Luigi Bianchi","UffVendite"],
        ["ordini_cliente","01746","Ordini"] ],
    "vclock":"opaque-riak-vclock",
    "lastmod":"Mon, 03 Aug 2013 18:49:42 GMT"
}
```

Figura 1.4 Esempio di oggetto in Riak

Si può notare come la struttura base sia contraddistinta dal riferimento del *bucket* a cui appartiene l'oggetto, la chiave e il relativo valore (campo *object*). Il bucket rappresenta il primo livello di suddivisione concettuale dei dati ed è comparabile a una tabella dalla struttura non rigida. Il campo "*vclock*" è una diretta conseguenza del versioning tramite MVCC e rappresenta il vector clock dell'oggetto, usato per contraddistinguere la versione attuale. Il campo "*links*" è invece una caratteristica peculiare di Riak e permette di stabilire una relazione unilaterale con altri oggetti. Non vengono eseguiti controlli di integrità sui dati referenziati dai link e il loro utilizzo è destinato principalmente come input di altre operazioni (per esempio nel caso di map/reduce che scorrono gli oggetti collegati). Nonostante queste funzionalità, Riak non dispone di un linguaggio di interrogazione completo e la mancanza di supporto a indici secondari rende possibile solo la ricerca su chiave primaria.

Per un approccio più completo nella gestione dei dati, sono state sviluppate soluzioni alternative come i database orientati ai documenti.

Con il termine *document* si intende un insieme di dati concettualmente raggruppati che possono essere codificati in vari formati, come XML, YAML, JSON, BSON e altri tipi

binari. Una prima e importante differenza rispetto ai database relazionali è data dalla mancanza di pianificazione nella struttura dei dati, infatti ogni document, anche all'interno della stessa collezione, può avere attributi differenti, per numero e tipo (approccio *schema-less*). Come la maggioranza dei database NoSQL, i sistemi orientati ai documenti non supportano le proprietà ACID delle transazioni mentre, a differenza dei database chiave/valore, il linguaggio di interrogazione è generalmente più dettagliato e permette di fare query su range di attributi, così come operazioni di aggregazione complesse. I database orientati ai documenti nascono per gestire informazioni eterogenee in un ambiente distribuito e le principali differenze tra le varie implementazioni si basano sul grado di concorrenza fornite [**6**].

Nei prossimi capitoli i database orientati ai documenti verranno approfonditi maggiormente analizzando il caso specifico di MongoDB.

Una terza macro-categoria di database NoSQL è formata da quelli *tabulari* (conosciuti anche con il termine "*Extensible Record Stores*" o "*Wide Column Stores*"). L'ampliamento nell'offerta di questo genere di database è cresciuta negli ultimi anni a seguito del successo di BigTable, database closed-source ad alte prestazioni implementato da Google. Tutti i database tabulari hanno forti influenze da esso, pertanto di seguito verranno analizzare brevemente le sue caratteristiche. BigTable effettua un mapping tridimensionale, associando al dato da memorizzare il suo identificatore di colonna, quello relativo alla riga e un timestamp, utilizzato per politiche di versioning e garbage collector.

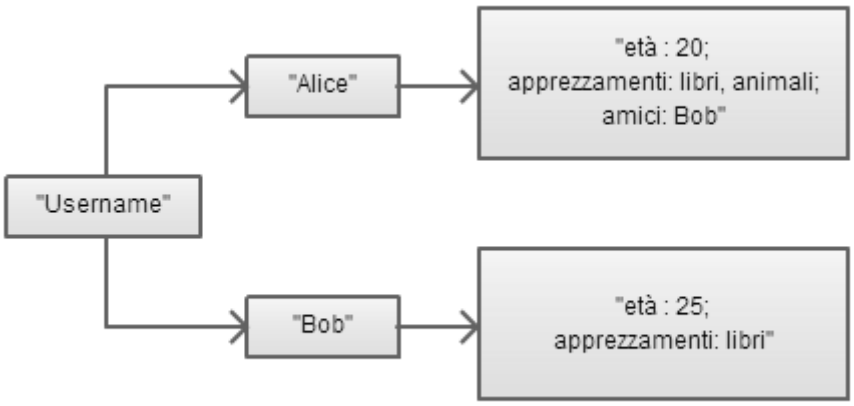

Figura 1.5 BigTable data mapping

Gli identificatori di colonna sono raggruppati in *column families*, i quali costituiscono l'unità base di accesso ai dati. Generalmente all'interno di un gruppo di colonne si hanno dati dello stesso tipo, in modo da agevolare la successiva compattazione. Le tabelle così create sono successivamente suddivise per righe in frammenti chiamati *tablet* in modo che ognuno di essi occupi una dimensione massima definita a priori. Quando i tablet raggiungono dimensioni prossime al limite, vengono compressi attraverso algoritmi di basso costo computazionale oppure, qualora non fosse possibile, ne viene aggiunto uno nuovo. Le informazioni riguardanti i tablet e la loro segmentazione vengono raccolti in opportuni *META-tablet*. Quando viene eseguita una query, si effettua l'interrogazione al nodo principale META0 che si occuperà di identificare su quale META-tablet server risieda il dato cercato, inoltrandone la richiesta; quest'ultimo provvederà a identificare e ritornare l'informazione cercata.

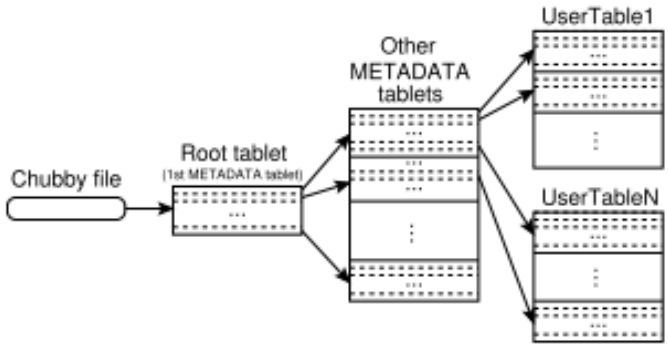

Figura 1.6 Localizzazione delle informazioni in BigTable

La distribuzione dei dati sia per righe che per colonne permette un'elevata scalabilità dei dati, mentre la suddivisione dei metadati in tablet su server differenti permette di avere un carico minimo per macchina (il server META0, nonostante sia il nodo centrale è particolarmente ottimizzato per operazioni di caching, riducendo così il rischio di colli di bottiglia nell'infrastruttura). L'infrastruttura di BigTable è progettata per supportare carichi a livello del petabyte ed è particolarmente rivolta alla distribuzione dei dati su migliaia di macchine [**14**].

Esistono ulteriori tipi di database, esclusi dalla tassonomia di R.Cattell utilizzata finora, che possono essere raggruppati sotto il movimento NoSQL. Tra gli esempi più comuni è possibile includere le seguenti categorie:

- Database a grafo: la rappresentazione dell'informazione è fornita per mezzo di nodi e archi che permettono di effettuare interrogazioni di adiacenza senza bisogno di indici. Esempi di questa categoria sono i database system come Neo4j e OrientDB

- Database orientati agli oggetti: combinano caratteristiche di memorizzazione dati con un linguaggio di interrogazione e manipolazione basato sul modello a oggetti. Questo approccio ibrido alla gestione dei dati ha portato i database

orientati agli oggetti a occupare una nicchia di mercato, nonostante la loro prima implementazione risalga ai primi anni 80. Esempi di questa categoria sono i DBMS Versant e ObjectDB

- Database XML: basano la persistenza dei dati su file XML. Rappresentano una variante dei database orientati ai documenti e permettono l'ausilio di database relazionali sui quali mappare il file XML. Alcuni esempi di questa categoria sono i database BaseX, eXist e Sedna

I database presentati finora rappresentano solo la parte più significativa delle tecnologie NoSQL disponibili sul mercato e la loro trattazione esaustiva è resa impossibile dall'elevato numero di soluzioni diverse presenti. Si può concludere osservando come l'enorme differenziazione tra le varie tipologie sia una chiara indicazione di un mercato che preferisce la creazione di sistemi per la risoluzione ad-hoc del problema, piuttosto che l'adattamento delle soluzioni già presenti.

Capitolo 2

Il database MongoDB

2.1 Introduzione

MongoDB è un sistema gestionale di basi di dati orientato ai documenti prodotto e supportato dall'azienda americana 10gen. Iniziato lo sviluppo il 9 ottobre 2007, il prodotto ha visto la prima release stabile nel febbraio 2009 a cui è seguito un ciclo di rilasci mirati non solo alla correzione di bug, ma anche all'aggiunta di funzionalità che in breve tempo lo hanno trasformato notevolmente, rendendolo adatto a un pubblico sempre più vasto. MongoDB è rilasciato sotto licenza Affero General Public License v.3 [**15**], mentre i driver che permettono ai linguaggi di programmazione di interfacciarsi al database sono rilasciati sotto licenza Apache 2.0 [**16**]; la flessibilità di tali licenze ha permesso alla community di MongoDB di ampliarne notevolmente le caratteristiche, arrivando così a supportare più di 30 linguaggi di programmazione differenti.

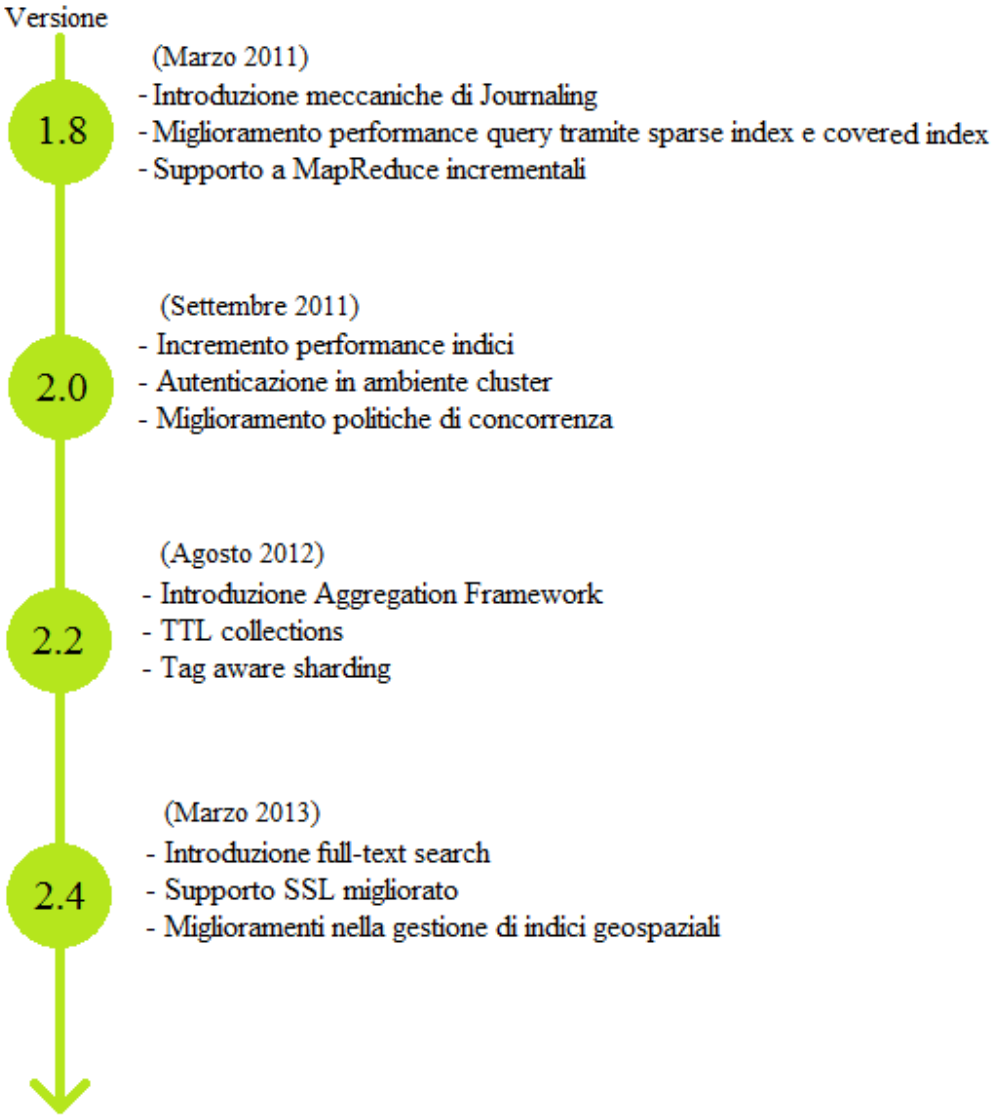

Figura 2.1 MongoDB Roadmap 2011-2013

Le motivazioni per cui, in questa sede, si è scelto di analizzare MongoDB si possono ricercare nell'alta popolarità di quest'ultimo, infatti diverse ricerche mostrano come tra le soluzioni NoSQL, MongoDB sia la più gradita da parte degli sviluppatori. I motivi di questa situazione si possono cercare principalmente nel modello dati fornito da

MongoDB e nella semplicità che contraddistingue il linguaggio di query. Le operazioni sul database vengono eseguite in console tramite Javascript e questa caratteristica permette un uso immediato per molti utenti, senza la necessità di imparare particolari sintassi o linguaggi. Un secondo motivo che ha portato MongoDB al successo è la maggiore versatilità rispetto ad altre soluzioni NoSQL, infatti la flessibilità del modello a documenti unito a un linguaggio di interrogazione semplice ma potente, permette a MongoDB di essere usato in un maggior numero di scenari applicativi, pur rimanendo nella fascia di mercato specializzata tipica del panorama NoSQL.

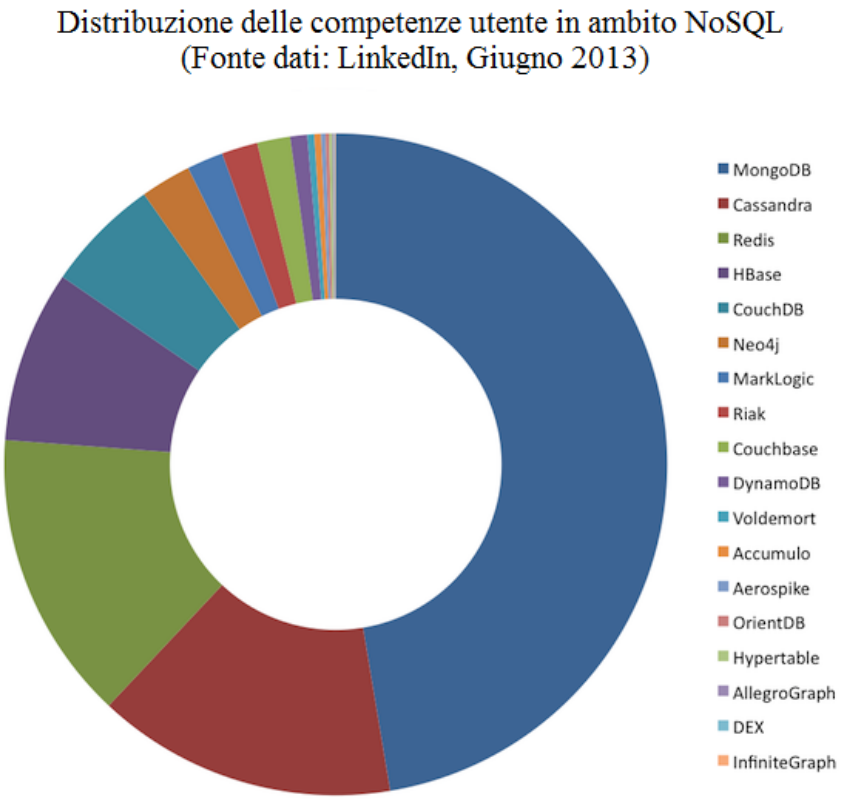

Figura 2.2 Popolarità MongoDB nel panorama NoSQL

2.2 Un database orientato ai documenti

In MongoDB l'unità fondamentale di memorizzazione delle informazioni è rappresentata dal *documento*. Tale termine non ha una definizione precisa e univoca, in quanto un documento è un qualsiasi insieme di informazioni concettualmente coerenti raggruppate secondo una data codifica che, nel caso di MongoDB, è il formato BSON. Un documento BSON è concettualmente simile a un oggetto JSON codificato in forma binaria, con alcune differenze quali il supporto alla memorizzazione delle date e performance migliori in fase di lettura e scrittura (gli integer, per esempio, sono memorizzati in formato binario, al contrario del formato JSON nel quale sono del semplice testo e necessitano di conversione in formato numerico).

Un esempio di documento in MongoDB è raffigurato in figura 2.3

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

Figura 2.3 Esempio struttura di un documento

Il documento mydoc in esame permette di mostrare l'eterogeneità del formato JSON/BSON, infatti all'interno dello stesso documento sono presenti integer a 64 bit (campo "views"), array (campo "contribs"), date (campi "birth" e "deaths"), documenti annidati (campo "name") e tipi speciali come ObjectID (campo "_id"). Quest'ultimo campo ha particolare importanza perché permette l'identificazione univoca del documento, concetto simile a quello di chiave primaria in un database relazionale. Se non viene esplicitato e dunque imposto manualmente per esigenze particolari, il campo _id viene autonomamente valorizzato con un ObjectID, un tipo di dato del formato

BSON che, oltre a essere generato con criteri di unicità, dispone di una struttura informativa nelle sue componenti. Un ObjectID ha una struttura di dimensione pari a 12 byte così composta:

- 4 byte contenenti i secondi trascorsi dal 1° gennaio 1970 al momento attuale di creazione (timestamp unix-like)
- 3 byte raffiguranti l'id della macchina di creazione
- 2 byte raffiguranti l'id del processo di creazione
- 3 byte di contatore incrementale

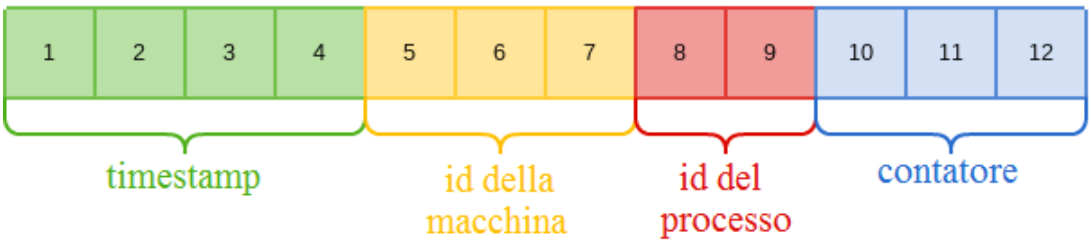

Figura 2.4 Struttura ObjectID

Una simile configurazione ha la proprietà che, in caso di inserimenti consecutivi, i record vengano memorizzati per la maggioranza in ordine di inserimento. Mentre i primi 9 byte si occupano di fornire informazioni sul sistema, gli ultimi 3 byte garantiscono l'unicità di generazione, per un massimo di 16.777.216 documenti creati al secondo per singolo processo.

I documenti vengono raccolti in *collezioni* (collection) le quali rappresentano una suddivisione concettuale dei dati in modo analogo alle tabelle dei database relazionali; a differenza di quest'ultime, in MongoDB una collezione non ha vincoli strutturali e pertanto i documenti presenti all'interno possono avere campi diversi, per tipo e valore.

A loro volta, le collection risiedono nel *database* che rappresenta l'universo del sistema di cui si vogliono gestire i dati.

La mancanza di una struttura prestabilita nelle collection è una caratteristica piuttosto diffusa nei sistemi NoSQL e rappresenta una prima, importante differenza rispetto ai database relazionali. Tale approccio viene definito *schema-less* e permette ai sistemi basati su MongoDB di cambiare agevolmente forma e modellare dati che cambiano nel tempo. Bisogna però notare come MongoDB non permetta alcune operazioni fondamentali tipiche dei database relazionali, quali join tra collection. La mancanza di controlli di integrità referenziale tra i dati obbliga il progettista a modellare lo schema logico con modalità molto diverse da quelle messe in atto durante lo sviluppo di soluzioni SQL-based, per esempio attraverso accorpamento dei dati in un'unica collection. Nel capitolo 3 si potrà vedere un esempio pratico di modellazione.

Tabella 2.1 SQL-MongoDB mapping chart

Termine SQL-like	Corrispettivo in MongoDB
Database	Database
Tabella	Collection
Riga	Document
Colonna	Field

Un tipo particolare di collezioni sono le *capped collection*, strutture di dimensione prestabilita che agiscono come buffer circolari. Una volta che si raggiungono le dimensioni massime imposte precedentemente, i documenti più vecchi vengono sovrascritti da quelli nuovi. In fase di aggiunta di nuovi record viene garantita la memorizzazione in ordine di inserimento, proprietà che permette di leggere i dati ordinati senza l'uso di indici (nel caso di capped collection, se non viene esplicitato diversamente, il sistema non crea un indice sul campo "_id"). La mancanza dell'indice permette di eliminare l'overhead dato dalla gestione di quest'ultimo, massimizzando

così le prestazioni del sistema. Le capped collection limitano le possibilità di manipolazione dei dati, infatti su di esse non è possibile eliminare i singoli documenti e non sono permesse modifiche ai dati che incrementino la dimensione dei documenti. In virtù di queste caratteristiche, le capped collection acquisiscono particolare importanza in scenari di logging dove sono richieste scritture rapide a fronte di un controllo debole sui dati.

2.3 Gestione della memoria

La gestione della memoria in MongoDB è mirata alla massima semplificazione e sfrutta il concetto di file mappato in memoria in modo persistente, dove ai segmenti della memoria virtuale viene assegnata una correlazione byte a byte con una porzione del file fisico in cui risiedono i dati.

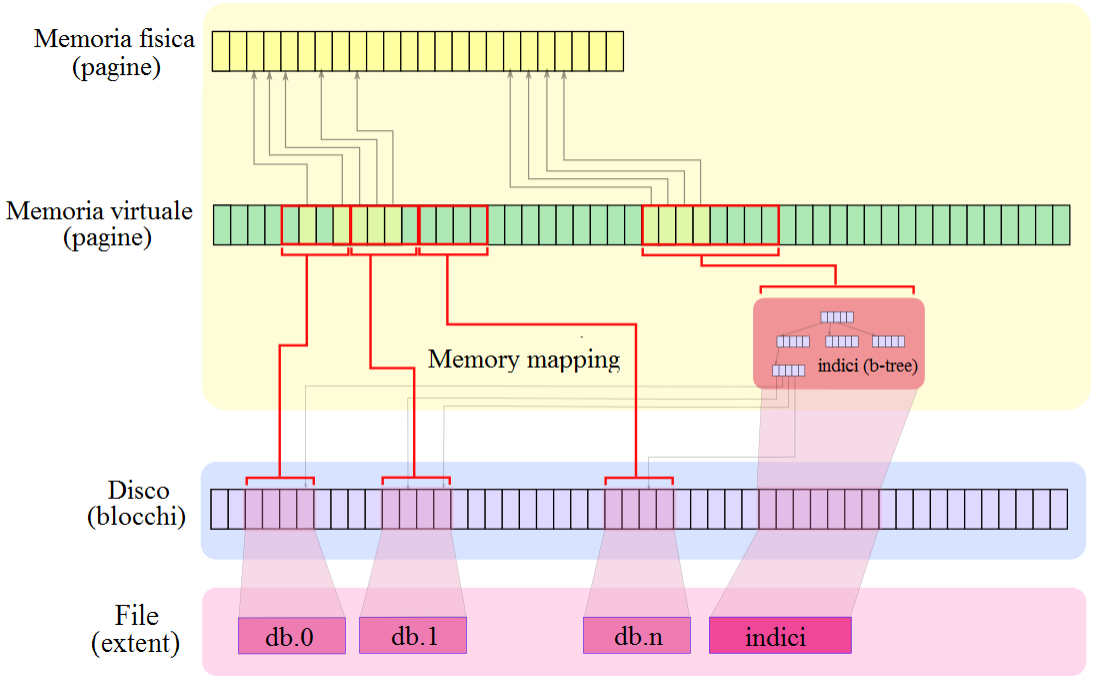

Figura 2.5 Mappatura di un file in memoria

La memoria virtuale è una rappresentazione privata a livello di processo della memoria fisica e permette di astrarre l'indirizzamento su di essa. Dal punto di vista del processo, un file mappato in memoria virtuale ha le stesse metodologie di accesso di un array di byte. L'indirizzamento della memoria virtuale su quella fisica è compiuta in modo trasparente dal sistema operativo, che si occupa di gestire la complessità di tutte le operazioni di accesso. Quando il processo richiede una pagina non ancora mappata in memoria, si genera un *page fault*, a cui il sistema operativo farà seguire la ricerca dell'informazione su disco. Alla conclusione di questa operazione si possono presentare due scenari: nel caso la memoria fisica non sia interamente occupata, la nuova pagina viene semplicemente caricata in RAM, mentre se non esiste spazio libero a disposizione, il sistema operativo si occuperà di scambiare la nuova pagina con una già presente in memoria (*swap-out*). Sia l'accesso su disco che lo scambio di pagine sono operazioni costose in termini di tempo d'esecuzione e, per minimizzarne l'occorrenza, MongoDB tende a caricare e mantenere in memoria l'intero database (o una sua porzione significativa, qualora la dimensione del database sia superiore al quantitativo di RAM disponibile).

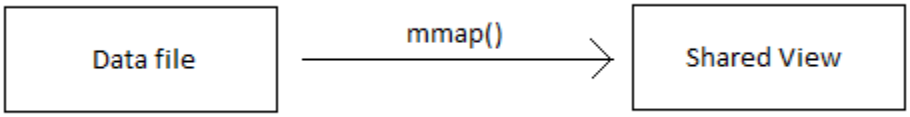

Figura 2.6 MongoDB memory mapping

All'avvio dell'istanza di MongoDB, i data file vengono mappati in memoria attraverso la system call mmap(). La funzione mmap() è una system call conforme agli standard POSIX, pertanto viene garantita la piena compatibilità tra i principali sistemi operativi, quali Linux, Mac OS X e BSD, mentre per ambiente Windows sono disponibili funzionalità equivalenti. Una volta mappato il contenuto del file nella sezione di memoria virtuale denominata *shared view*, è possibile manipolare i dati lasciando al

sistema operativo la complessità di salvataggio su disco delle modifiche effettuate al database (a sua volta suddiviso in *extent*, ovvero porzioni fisiche di dati di dimensione preallocata), operazione che viene eseguita in piena autonomia da quest'ultimo a intervalli regolari (per offrire garanzie di periodicità, MongoDB impone comunque al sistema operativo di aggiornare il data file ogni 60 secondi).

2.4 Journaling

Si è visto precedentemente come generalmente i database NoSQL non supportino caratteristiche tipiche dei database relazionali come le proprietà ACID. Pur difettandone un supporto completo, MongoDB offre alcune garanzie rudimentali quali l'atomicità delle scritture nello stesso documento e la durabilità single-server dei dati. Quest'ultima caratteristica è prevista dalla versione 1.8 e viene garantita attraverso politiche di Journaling. Il *journal* è un sistema di write-ahead logging che permette il recupero dei dati in caso di situazioni impreviste quali system failure. Più precisamente, le informazioni memorizzate nel journal permettono di recuperare operazioni quali modifiche ai documenti (insert/update/remove), modifiche agli indici e modifiche al namespace (file contenenti descrizioni di collection e indici).

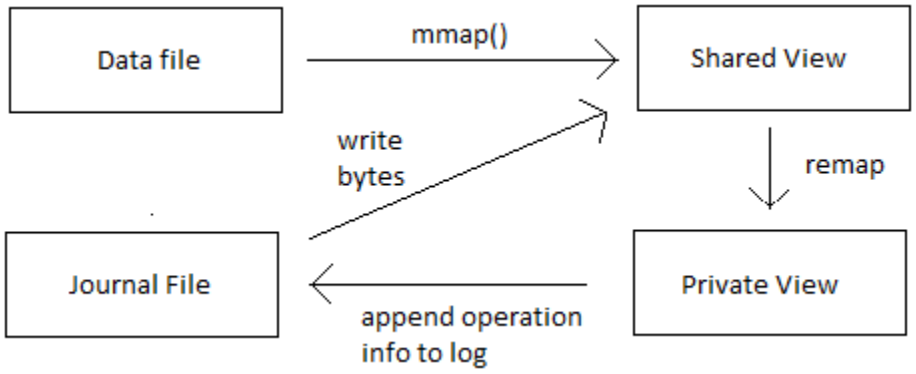

Figura 2.7 Gestione della memoria con Journaling attivato

L’attivazione del Journaling introduce due nuovi elementi nella gestione della memoria, ovvero la *private view* e il file di journal. La private view è un’area di memoria virtuale che viene inizializzata con il contenuto della shared view ma senza supporto al datafile, dunque le modifiche ai dati presenti in quest’area non si rifletteranno su di esso. Il file di journal è invece un file binario di dimensione predefinita e preallocata (1GB di default) che agisce come file di log. Ogni volta che il server MongoDB dovrà effettuare modifiche ai dati, quest’ultime verranno effettuate all’interno della private view e successivamente scritte nel file di journal entro un intervallo di tempo personalizzabile, il cui valore predefinito è 100ms. La scrittura delle modifiche eseguite non avviene per singola operazione, ma in batch di esse, in modo da rendere minore l’impatto prestazionale del journaling. Il file di Journal conterrà dunque la descrizione dei byte che dovranno essere modificati nel datafile; in caso di system failure, al successivo riavvio le informazioni presenti nel Journal permetteranno di riportare i dati a uno stato consistente. Una volta che le informazioni sono salvate nel Journal, il DBMS si occuperà di replicare queste operazioni nella shared view, per poi rimappare la private view in modo da riflettere il contenuto aggiornato. Periodicamente, verranno infine salvate le informazioni sul datafile fisico, con le stesse meccaniche e tempistiche che si hanno senza la presenza del journaling.

La concorrenza nelle interazioni multi utente viene garantita tramite meccaniche di locking *shared-exclusive*, configurazione che prevede l’esistenza di due tipi di lock: condivisi ed esclusivi. Durante le operazioni di scrittura viene assegnato un lock esclusivo e si hanno le seguenti proprietà:

- Quando un oggetto ottiene un lock esclusivo, non possono essere assegnati lock condivisi su quella risorsa
- Il lock esclusivo è unico e, una volta assegnato a un oggetto, non possono sussistere altri lock esclusivi su tale risorsa

Nelle operazioni di lettura viene assegnato un lock condiviso, dalle seguenti proprietà:

- Se su una risorsa esistono uno o più lock condivisi, non è possibile imporre un lock esclusivo
- Il lock condiviso non è unico e possono pertanto coesistere sul medesimo oggetto

Il meccanismo di gestione della concorrenza tramite lock si applica anche al file di journal, pertanto se si verifica un evento di system failure, all'avvio del DBMS devono essere applicate le operazioni memorizzate nel journal non ancora replicate sui dati, operazione possibile imponendo un lock in scrittura, mantenuto fino a che l'integrità dei dati non sia ristabilita.

La presenza di meccanismi di journaling garantisce la durabilità dei dati, ma al tempo stesso dispone di alcuni svantaggi che possono essere significativi in particolari occasioni:

- Prestazioni ridotte: per mantenere garanzie accettabili di persistenza dei dati l'intervallo del commit su journal non deve essere elevato. A tempi minori corrispondono overhead maggiori e dunque prestazioni ridotte
- Maggiore consumo di memoria volatile: quando il journaling è attivato, parte della memoria centrale viene destinato al "write working set", ovvero l'ammontare di dati che vengono modificati tra un re-map della private view e quello successivo

2.5 Indici

Un indice è una struttura dati che permette la localizzazione rapida delle informazioni relative al campo su cui è costruito e riveste una parte fondamentale in fase di ottimizzazione delle query. In MongoDB tutti gli indici sono basati su B-tree. Quando un indice "copre" la query (ovvero quando tutti i campi della query fanno parte dell'indice, sia come condizione sia come valore ritornato) MongoDB può eseguire l'interrogazione senza accedere alla collection, massimizzando così le prestazioni. In MongoDB è possibile creare un indice su qualsiasi campo di un documento, indipendentemente dal tipo di dato, rendendo così possibile l'indicizzazione di campi complessi quali array e documenti. Sono supportati indici composti (indici costruiti su campi multipli) con la possibilità di specificare l'ordinamento ascendente o discendente per ogni campo. Un esempio di creazione di indice composto è mostrato in figura 2.8

```
db.anagrafica.ensureIndex({
        "nome": 1,
        "cognome": 1,
        "data_nascita": -1
})
```

Figura 2.8 Esempio creazione indice

L'esecuzione di questo comando crea un indice sui campi nome, cognome e data di nascita. Per i primi due elementi è specificato un ordinamento ascendente, mentre la data di nascita è ordinata iniziando dalla più recente. E' importante notare come le query possano utilizzare anche parti di un indice composto, purché i campi richiesti siano un prefisso di quelli indicizzati; nell'esempio fornito l'indice può assolvere query che esprimono condizioni su nome e cognome, ma non interrogazioni su nome e data di nascita. Analogamente ai principali database relazionali, anche in MongoDB è

possibile imporre vincoli di unicità sui campi indicizzati, con la possibilità di eliminare i duplicati presenti in fase di creazione.

Normalmente quando viene imposto un indice su un campo, qualora quest'ultimo non fosse presente in un documento lo si considera indicizzato con il valore NULL. Questo comportamento potrebbe essere indesiderato in caso si abbiano numerosi documenti il cui valore indicizzato sia assente, pertanto è possibile specificare l'attributo "sparse" dell'indice in modo da non considerare i valori non presenti.

L'ultimo attributo degli indici che merita attenzione è il Time To Live (TTL). Questa proprietà, applicabile solo con campi di tipo datetime, permette a MongoDB di rimuovere automaticamente il documento dopo un certo intervallo di tempo definito a priori. Un indice TTL assume particolare importanza in scenari di logging o memorizzazione di sessioni utente, dove documenti datati non ricoprono ruoli significativi.

E' necessario notare come non tutti i tipi di indici siano mirati all'ottimizzazione delle query, infatti alcuni di essi aumentano le possibilità di interrogazione offrendo nuove funzionalità nella ricerca dei dati. Gli indici che fanno parte di questa categoria sono i seguenti:

- Indici hashed: introdotti nella versione 2.4 di MongoDB, permettono di indicizzare uno o più campi attraverso il corrispettivo valore hash. Questa possibilità acquisisce particolare importanza in ambiente distribuito, qualora si volesse imporre come chiave di shard (ovvero il campo su cui viene basata la suddivisione dei dati tra le diverse macchine) un attributo monotonicamente crescente quale una data. In questa situazione, usando un campo su cui è costruito un indice hashed, si evita l'addensamento dei risultati più recenti e si permette una distribuzione dei dati maggiormente omogenea

- Indici geospaziali: basati anch'essi su B-tree, permettono interrogazioni multidimensionali tramite coordinate, sia su superfici piatte che sferiche. Le operazioni permesse sono query di inclusione (determinazione degli elementi all'interno di un dato poligono), intersezione (applicabili solo su superfici sferiche, permettono di individuare gli elementi che intersecano un particolare oggetto) e prossimità (individuazione degli elementi più vicini a una data posizione)

- Indici geohaystack: sono una versione prestazionalmente più efficace degli indici geospaziali, applicabili solo su superfici piatte. Implementano una logica a bucket per raggruppare documenti appartenenti a una specifica regione geografica, pertanto la determinazione della capacità del bucket permette di definire la granularità dell'indice. Tramite indici geohaystack sono disponibili solo funzionalità di ricerca di prossimità

- Indici text: introdotti in versione sperimentale con la release 2.4, permettono la ricerca full-text sui campi indicizzati. Questo particolare tipo di interrogazione prevede la ricerca di una o più keyword su dati di dimensione notevole, operazione che sarebbe computazionalmente sconveniente da effettuare tramite ricerca tradizionale. Sono inoltre disponibili funzionalità di stemming (determinazione della radice di una parola) che permettono una ricerca più completa, seppur con la possibilità di generare falsi positivi. Effettuata una query, a ogni elemento risultante dall'interrogazione viene attribuito un punteggio, in modo da ottenere un vettore risultati ordinato per significatività rispetto ai termini cercati. Le funzionalità di full-text search rappresentano una distinzione importante di MongoDB rispetto alla concorrenza NoSQL e verranno ulteriormente approfondite nei capitoli successivi in quanto oggetto di test dimostrativi e prestazionali.

2.6 Replicazione

La possibilità di replicare un database su server differenti è un aspetto fondamentale in un ambiente di produzione. La replicazione crea copie multiple dei dati su nodi diversi e la ridondanza così ottenuta permette di avere dei server di backup o disaster recovery sempre aggiornati, in modo da fare fronte a eventi potenzialmente dannosi per l'integrità dei dati come l'interruzione dell'alimentazione. In MongoDB la replicazione permette di aumentare l'accessibilità dei dati e le performance di lettura, infatti per un client è possibile impostare il driver (le librerie che si occupano di interfacciarsi con il database) in modo che sia consentita la lettura dei dati da qualsiasi membro di un replica set; questo aspetto è particolarmente importante quando il nodo principale preferito per la comunicazione abbia un carico di lavoro eccessivo e l'utente possa tollerare la possibilità di ottenere dati non aggiornati.

MongoDB implementa i meccanismi di replicazione dei dati basandosi su un modello master/slave con failover automatico, chiamato replica set.

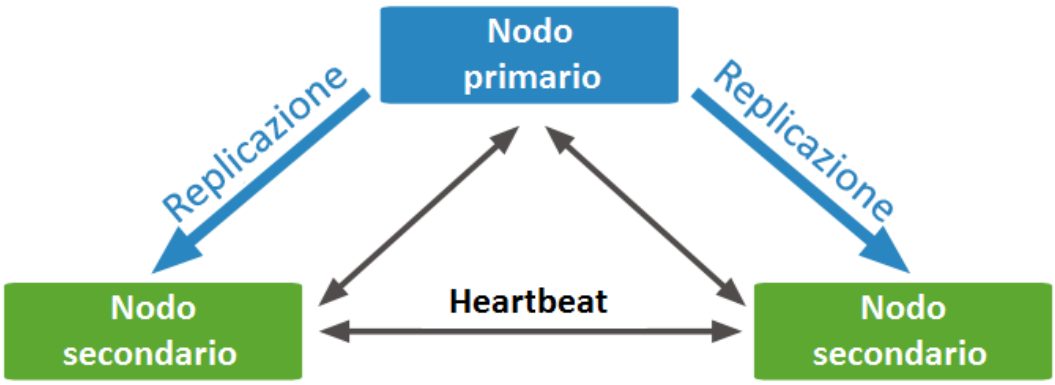

Figura 2.9 Replica set con 3 nodi

Un replica set prevede un nodo primario a cui vengono indirizzate tutte le operazioni sui dati, sia di lettura che di scrittura. Effettuata l'interrogazione, il server primario

registra l'operazione sul proprio "oplog", una capped collection salvata nel database local che si occupa di memorizzate un set di informazioni minime necessarie per la riproduzione dell'operazione. Un esempio di informazione memorizzata nell'oplog è data in figura 2.10

```
{
 "ts" : { "t" : 1296864947000, "i" : 1 },
 "op" : "i",
 "ns" : "biblioteca.libri",
 "o" :   {  "_id" : ObjectId("4d4c96b1ec5855af3675d7a1"),
            "title" : "Oliver Twist" }
}
```

Figura 2.10 Esempio operazione memorizzata nell'oplog

Il campo "ts" rappresenta il timestamp, dove il primo valore è dato dai secondi in formato unix-time e il secondo è un contatore (senza di esso si avrebbe una granularità dettagliata al secondo, eventualità che non permette di identificare correttamente il record). Il successivo campo "op" mostra l'operazione eseguita, codificata in forma abbreviata; in questo caso il valore "i" indica un inserimento. Il terzo campo mostra il namespace di riferimento, ovvero l'accoppiata database e collection. Infine, l'ultimo campo è l'elemento che è stato soggetto a manipolazione; nel caso di inserimento si ha il documento aggiunto.

Periodicamente i server secondari analizzeranno il proprio oplog ottenendo il timestamp dell'ultima operazione; svolta questa fase si occuperanno di interrogare l'oplog del server primario alla ricerca di tutte le operazioni temporalmente successive al timestamp cercato precedentemente. Le operazioni individuate verranno copiate nell'oplog del nodo secondario, per poi essere applicate sulla propria copia dei dati concludendo il ciclo di replicazione. Il procedimento descritto è asincrono e nonostante il periodo di tempo con cui i secondari interroghino i primari sia molto basso (nell'ordine dei millisecondi) è possibile che i server secondari presentino copie dei

dati più vecchie dell'attuale. Al contrario, la scrittura delle informazioni nel journal (qualora sia attivato) e nell'oplog avviene attraverso una transazione atomica.

Una delle caratteristiche più diffuse nei database NoSQL è la capacità di autogestione del sistema in caso di malfunzionamento. MongoDB implementa una logica di *automate failover* basata su votazione dei server secondari. All'interno di un replica set, tutti i nodi sono in contatto tra loro attraverso lo scambio di piccoli messaggi di stato, denominati *heartbeat*. Quando il server primario non è più raggiungibile, i nodi secondari riconoscono l'interruzione dei messaggi e si preparano a eleggere un successore. Ogni server ha una priorità, attributo numerico impostato a priori in fase di configurazione. Quando i server secondari dovranno stabilire quale sarà il nuovo nodo primario, i membri con maggiore priorità confronteranno tra di loro il periodo di tempo trascorso dall'ultima sincronizzazione dei dati. Il nodo secondario con la maggior priorità e i dati più recenti verrà eletto a server primario.

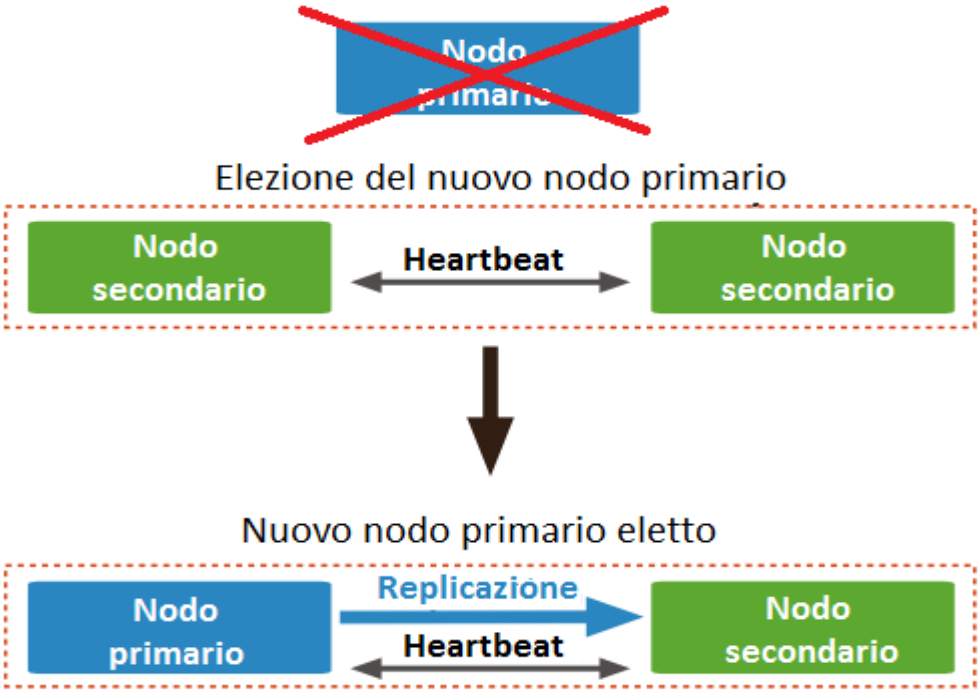

Figura 2.11 Elezione di un nuovo membro primario in un replica set

All’elezione del nuovo server primario è possibile che si verifichino scenari di spareggio, dove più server siano eleggibili. Per ovviare queste situazioni, è possibile istituire una macchina “arbitro” il cui scopo sia quello di stabilire il successore. Un server arbitro non richiede necessariamente hardware dedicato in quanto non contiene frammenti o copie dei dati, dunque il consumo di risorse è minimo e può risiedere su macchine già in uso. Quando è in corso una votazione viene imposto un lock sul database e non sono permesse scritture per tutta la durata dell’elezione, operazione che generalmente impiega un tempo inferiore al minuto.

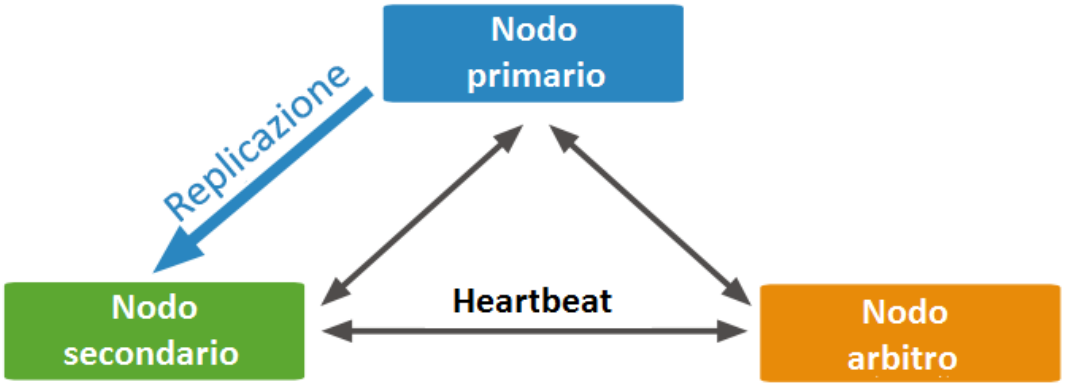

Figura 2.12 Struttura semplificata con nodo primario, secondario e arbitro

Come si è visto, in MongoDB i criteri di replicazione sono automatizzati ovunque sia possibile e conveniente, ma questo comportamento non preclude la possibilità di configurare situazioni ad-hoc. Alcune operazioni non sono configurate lato server ma disponibili a livello di driver e, dunque, lato utente. Si è già accennato alla possibilità per l’utente di leggere i dati da un server secondario, piuttosto che da un primario, operazione attivabile da driver. Sempre lato client è anche possibile richiedere un *Replica Acknowledgment*, dove la corretta scrittura di un dato viene confermata solo quando replicata su un certo numero di nodi.

2.7 Distribuzione su cluster

La distribuzione dei dati e delle operazioni su un cluster di macchine viene eseguita in MongoDB attraverso politiche di *sharding*, un meccanismo di scaling orizzontale che divide il dataset su server differenti, denominati shard. Ogni shard conterrà una partizione dei dati in un database indipendente capace di esistere in modo autonomo. Al crescere delle dimensioni di un database, le meccaniche di scaling orizzontale trovano sempre maggiore utilizzo, soprattutto quando è possibile dividere i dati in porzioni sufficientemente piccole per risiedere interamente in memoria centrale, annullando lo swapping su disco e ottenendo quindi un notevole incremento prestazionale. Lo sharding è particolarmente importante anche per la parallelizzazione offerta in fase di recupero dei dati, infatti ogni shard si troverà a lavorare su un numero minore di documenti e potrà concludere le interrogazioni in breve tempo. Un cluster basato su sharding prevede i seguenti componenti:

- Cluster server: definiti anche con il termine "query router", si occupano della ricezione delle query e del relativo instradamento agli shard. Sono processi non persistenti, le cui informazioni vengono memorizzate in memoria centrale e non richiedono supporto su unità fisiche. Un sistema distribuito può avere diversi cluster server, sia per distribuire il carico di richieste su più macchine, sia per motivi di tolleranza ai guasti

- Configuration server: garantiscono la memorizzazione e la persistenza dei metadata del cluster. Tra le informazioni salvate di maggiore importanza troviamo il percorso di rete degli shard facenti parte dell'infrastruttura, la suddivisione dei valori con cui avviene la distribuzione e uno storico delle migrazioni dei dati. La loro presenza è resa necessaria a causa della volatilità delle informazioni gestite dai cluster server, i quali dispongono così di un nodo centrale a cui fare riferimento

- Shard: server stand-alone in cui risiedono le porzioni del dataset distribuito. A uno shard può anche corrispondere un replica set, in modo da incrementare la tolleranza ai guasti

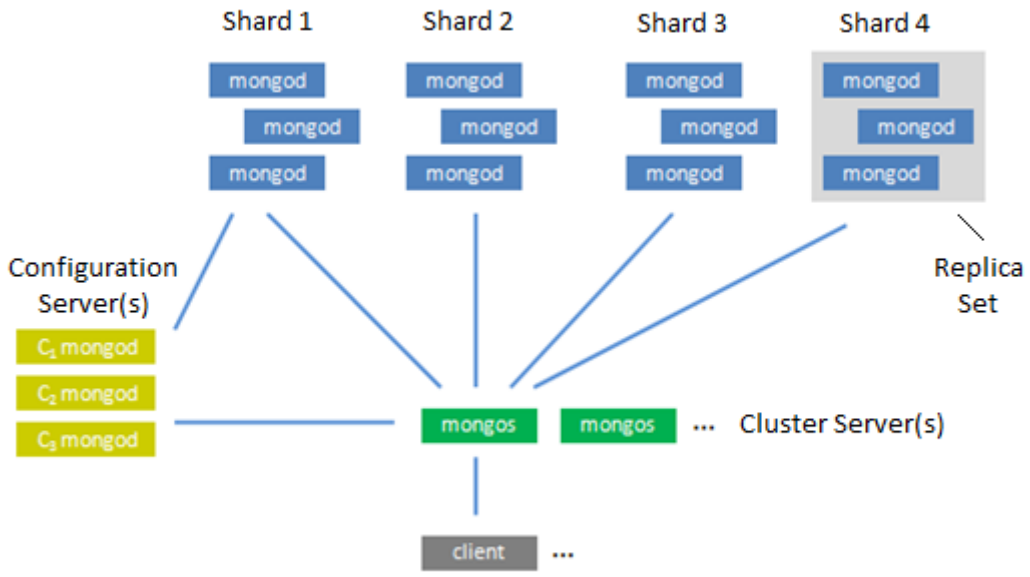

Figura 2.13 Struttura di un cluster basato su sharding

Come si nota in figura 2.13, la distribuzione tramite sharding può coesistere con i replica set e rappresenta lo scenario tipicamente applicato in ambienti di produzione. In questo caso, il cluster server instraderà le interrogazioni al server primario del replica set, il quale eseguirà la query e permetterà la replicazione delle operazioni nei server secondari, come mostrato nel paragrafo 2.6

La distribuzione dei dati viene effettuata tramite una *shard key*, ovvero un campo (o combinazione di essi) di un documento su cui effettuare la partizione per range di valori. La scelta della chiave di shard riveste un ruolo di fondamentale importanza e determina sia l'equità della distribuzione, sia le prestazioni del sistema. Una suddivisione ben bilanciata permette di ottenere un carico di lavoro medio uguale per

tutte le macchine del sistema, diminuendo così la probabilità di avere colli di bottiglia e code di attesa. Non sempre è possibile ottenere una distribuzione perfettamente bilanciata, per esempio nel caso si avesse la necessità di usare come chiave di shard un campo monotonicamente crescente quale una data. In quest'ultimo caso esemplificativo sussiste un ulteriore problema, qualora le operazioni più frequenti e computazionalmente pesanti avvenissero solo sui dati più recenti; in questo caso la maggioranza delle macchine risulterebbe inutilizzata, mentre le rimanenti sarebbero sottoposte a un pesante carico di lavoro. Per ovviare a queste situazioni è possibile utilizzare come chiave di shard uno o più campi su cui sia stato costruito un indice hashed, in modo da ottenere una distribuzione equa su tutti gli shard.

L'elemento che ha il compito di mantenere il più possibile equilibrata la distribuzione è il *balancer*. Il processo di bilanciamento agisce sui *chunk*, piccole porzioni di dati di dimensione prefissata (normalmente 64 MB). Periodicamente il balancer controlla che la differenza in numero di chunk tra lo shard più popolato e quello meno popolato sia inferiore a un certo limite, espresso in tabella 2.2

Tabella 2.2 Differenza chunk tra shard per considerare il sistema sbilanciato

Chunk totali nello shard	Differenza limite
< 20	2
21 – 80	4
> 80	8

Quando viene rilevato un numero di chunk superiore al limite concesso, viene avviato il processo di migrazione dei dati dallo shard sovrappopolato a quello numericamente in difetto. In caso di esigenze particolari è possibile disattivare il balancer e provvedere alla migrazione manuale dei singoli chunk, anche in modo non bilanciato; questa operazione può risultare vantaggiosa nel caso il cluster sia composto da macchine dalle prestazioni molto diverse tra loro e che richiedono dunque carichi di diversa entità.

Un ultimo aspetto fondamentale da analizzare in un ambiente basato su cluster, è dato dalle modalità con cui il cluster server possa instradare le richieste ai vari shard.

I due scenari possibili sono:

- Instradamento broadcast
- Instradamento verso singolo shard o piccoli gruppi

Una query è inviata in modalità broadcast ogni volta che il cluster server non sia in grado di determinare quali shard contengano il dato richiesto. Alla ricezione di una query, il cluster server determina come prima cosa se è possibile limitare l'instradamento a un certo numero di shard, per poi inviare loro la query e stabilire per ognuno di essi un cursore. Quest'ultimo sarà usato successivamente per ricomporre i risultati ottenuti, con modalità diverse in base al tipo di dato risultante. In caso nell'interrogazione sia presente un ordinamento verrà eseguito un merge sort sui risultati prima di presentare il dato al client, mentre nel caso i dati non siano richiesti ordinati, viene ritornato un cursore che si occuperà di accedere ai frammenti di risultato secondo politiche di round robin.

Come ogni database NoSQL, MongoDB pone una forte enfasi alla distribuzione dei dati su più macchine. In un'ottica di lavoro su grande scala, vengono resi necessari meccanismi di automazione delle operazioni di manutenzione. Si è visto come MongoDB provveda automaticamente al rimpiazzo di server difettosi in scenari di replicazione, così come in un'ambiente cluster si faccia carico del bilanciamento automatico. Persino le operazioni che richiedono l'esplicito intervento dell'utente (come l'aggiunta di nuove macchine al cluster) sono risolvibili in pochi comandi. Seppur questo aspetto possa apparire limitante in ambienti di piccola scala dove è richiesta un'alta personalizzazione delle dinamiche operative, in una visione di distribuzione massiva l'automazione acquisisce un ruolo sempre più fondamentale.

2.8 Aggregation Framework e Map/Reduce

Il raggruppamento dei dati è un'operazione di utilizzo comune e MongoDB offre un supporto completo alle operazioni di aggregazione. Gli strumenti messi a disposizione sono i seguenti:

- Group (da non confondersi con l'operatore $group)
- Aggregation Framework
- Map Reduce

Il comando Group rappresenta un'implementazione rudimentale delle funzionalità di Map Reduce, utilizzata prima che quest'ultimo fosse implementato. Allo stato attuale non supporta l'esecuzione in ambiente distribuito e non può ritornare più di 20.000 risultati, pertanto si preferisce favorire altre forme di aggregazione dei dati.

L'Aggregation Framework è stato introdotto nella release 2.2 di MongoDB e permette di effettuare facilmente le operazioni di raggruppamento più comuni. Seppur non venga garantito il pieno controllo dei dati (come invece avviene attraverso Map Reduce) gli operatori disponibili permettono di definire le interrogazioni con un'espressività simile a quella del linguaggio SQL.

Tabella 2.3 Confronto operatori di aggregazione tra SQL e MongoDB

Operatore SQL	Corrispettivo in MongoDB
SELECT	$project
WHERE	$match
HAVING	$match
GROUP BY	$group
ORDER BY	$sort
LIMIT	$limit

Come si nota in tabella 2.3 esistono delle forti corrispondenze tra gli operatori messi a disposizione dal linguaggio SQL e quelli presenti in MongoDB, così come sussistono delle importanti differenze a livello di logica implementativa e sintassi, come, per esempio, il concetto di pipeline. Mentre in SQL l'ordine con cui gli operatori appaiono nella query è ininfluente, l'Aggregation Framework prevede la manipolazione del dataset iniziale tramite pipeline, ovvero una sequenza ordinata di comandi.

```
db.dimostratore_1x.aggregate
(
     {$match: {"MediatypeName":"blog"}},
     {$group: {_id: "$AuthorUID",
               avgsentiment:{$avg:"$SentimentCrawler"}
              }
     }
);
```

Figura 2.14 Esempio di raggruppamento con Aggregation Framework

Prendendo come oggetto di studio l'esempio mostrato in figura 2.14, all'esecuzione dell'interrogazione la collection "dimostratore_1x" viene passata in input al primo operatore, che si occupa di filtrare i dati in base alla condizione espressa. Successivamente, sul risultato ottenuto viene applicato l'operatore $group che effettua l'aggregazione sul campo "AuthorUID" e per ogni entry distinta calcola la media dei relativi campi "SentimentCrawler". Se si fosse effettuato il raggruppamento come prima operazione, l'input di $group sarebbe stato l'intera collection, con gli evidenti svantaggi prestazionali che un'organizzazione simile comporta.

L'uso dell'Aggregation Framework è sottoposto a certi limiti nel risultato, quali la possibilità di visualizzare i dati esclusivamente in-line e una dimensione massima del vettore risultato pari a 16 MB. Al tempo stesso, tra i metodi di aggregazione forniti da MongoDB, risulta essere quello con le prestazioni migliori e per operazioni semplici su dataset di piccole e medie dimensioni è la scelta preferibile.

L’ultimo strumento di aggregazione offerto da MongoDB è basato su meccanismi di Map Reduce e rappresenta la soluzione più completa se si vuole pieno controllo dei dati. Un’operazione di Map Reduce prevede due fasi principali, una di mappatura dei dati e una di manipolazione dei risultati aggregati. Nella fase di mappatura dei dati (*Map*) viene analizzo ogni documento del dataset in input, producendo un oggetto per ognuno di esso. L’oggetto così creato viene espresso nella forma chiave/valore, dove la chiave rappresenta il criterio su cui effettuare successivamente il raggruppamento, mentre il valore può consistere in un insieme di campi già esistenti nella collection o creati per l’occasione. Il mapping degli oggetti è un’operazione altamente parallelizzabile e dalla versione di MongoDB 2.4, può essere eseguita, a livello di singola macchina, da più thread concorrenti.

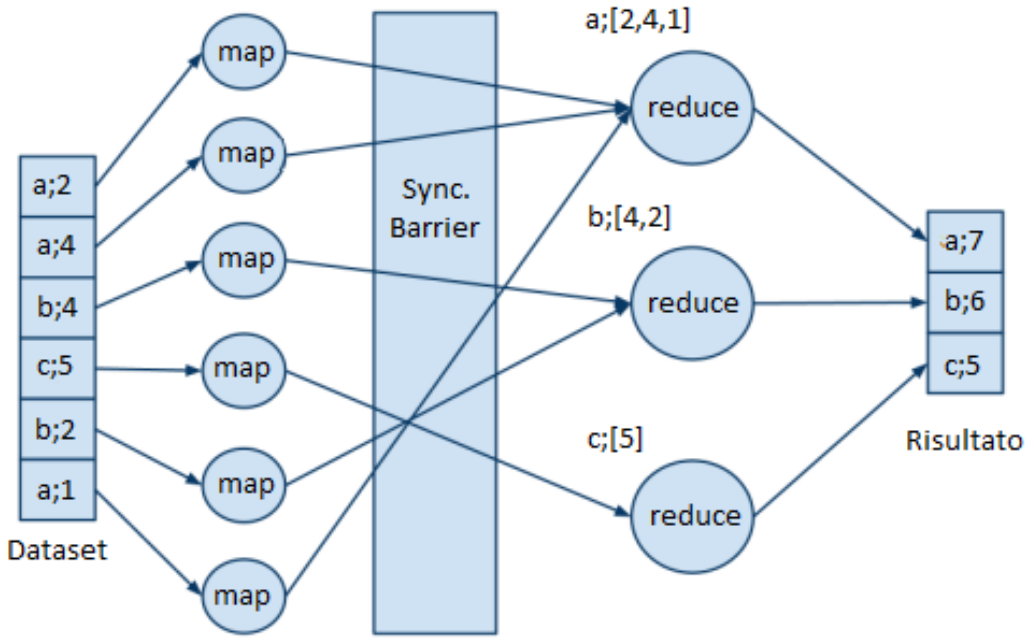

Figura 2.15 Map Reduce Workflow

Quando un thread conclude la mappatura dei dati a lui competenti, si arresta attendendo che tutti gli altri thread completino le rispettive operazioni (S*ync Barrier*); in questo modo si elimina la possibilità, nelle fasi successive dell'algoritmo, di lavorare con dati parziali non interamente elaborati. Terminata a livello globale la fase di mapping, ha inizio la riduzione dei dati (R*educe*), dove ogni oggetto creato precedentemente viene analizzato e raggruppato per valore di chiave. Durante questa fase è possibile la piena manipolazione degli oggetti tramite le funzioni e le strutture di controllo offerte da Javascript. Alla sua conclusione, l'operazione Reduce restituisce un nuovo vettore di coppie chiave/valore che è possibile processare ulteriormente attraverso la funzione opzionale di finalizzazione (*Finalize*); un esempio di utilizzo tipico può venire dal calcolo di una media, durante il quale la fase di Reduce si occupa di accumulare il valore degli elementi per una data chiave e memorizzare il loro numero in un contatore, mentre alla fase di finalizzazione viene lasciato il calcolo effettivo della media grazie ai due valori precedentemente ottenuti.

Analogamente alle altre operazioni di aggregazione anche tramite Map Reduce è possibile eseguire una selezione dei dati iniziali, in modo da avere un minor numero di dati da processare. Tra le caratteristiche uniche si segnala invece la mancanza di limiti nella dimensione del risultato (è possibile processare qualsiasi mole di dati e pertanto le funzioni di Map Reduce trovano impiego naturale nella gestione dei Big Data, con particolare attenzione all'ambiente cluster) e la possibilità di memorizzare quest'ultimo su apposite collection in modo persistente. Quest'ultima situazione permette l'utilizzo di Map Reduce incrementali, attraverso i quali è possibile processare solo una porzione di dati selezionati in base a certi criteri (ad esempio, per data recente) e aggiornare una collection precedentemente creata esclusivamente con le nuove informazioni prodotte. I Map Reduce incrementali trovano principale applicazione in job periodici e automatizzati, effettuabili nei momenti in cui il server sia inattivo o abbia un carico di lavoro minimo.

Capitolo 3

Un caso di studio reale

3.1 Modellazione del problema

Lo sviluppo della seguente applicazione nasce come approfondimento pratico delle funzionalità di MongoDB. Il problema analizzato è quello di fornire una soluzione web che permetta la ricerca full-text all'interno di un database contenente elementi presi da diversi social media. Il risultato dovrà essere presentato all'utente ordinato per "importanza", ovvero secondo modalità che diano priorità ai documenti più attinenti all'interrogazione eseguita (per esempio, cercando una data parola, si vogliono per primi i risultati che contengano il numero maggiore di occorrenze della keyword nel testo). La sorgente dati a cui si farà riferimento inizialmente è un database relazionale già esistente che sarà necessario analizzare e importare in ambiente MongoDB. Il modello di cui è richiesta la conversione è presentato in figura 3.1

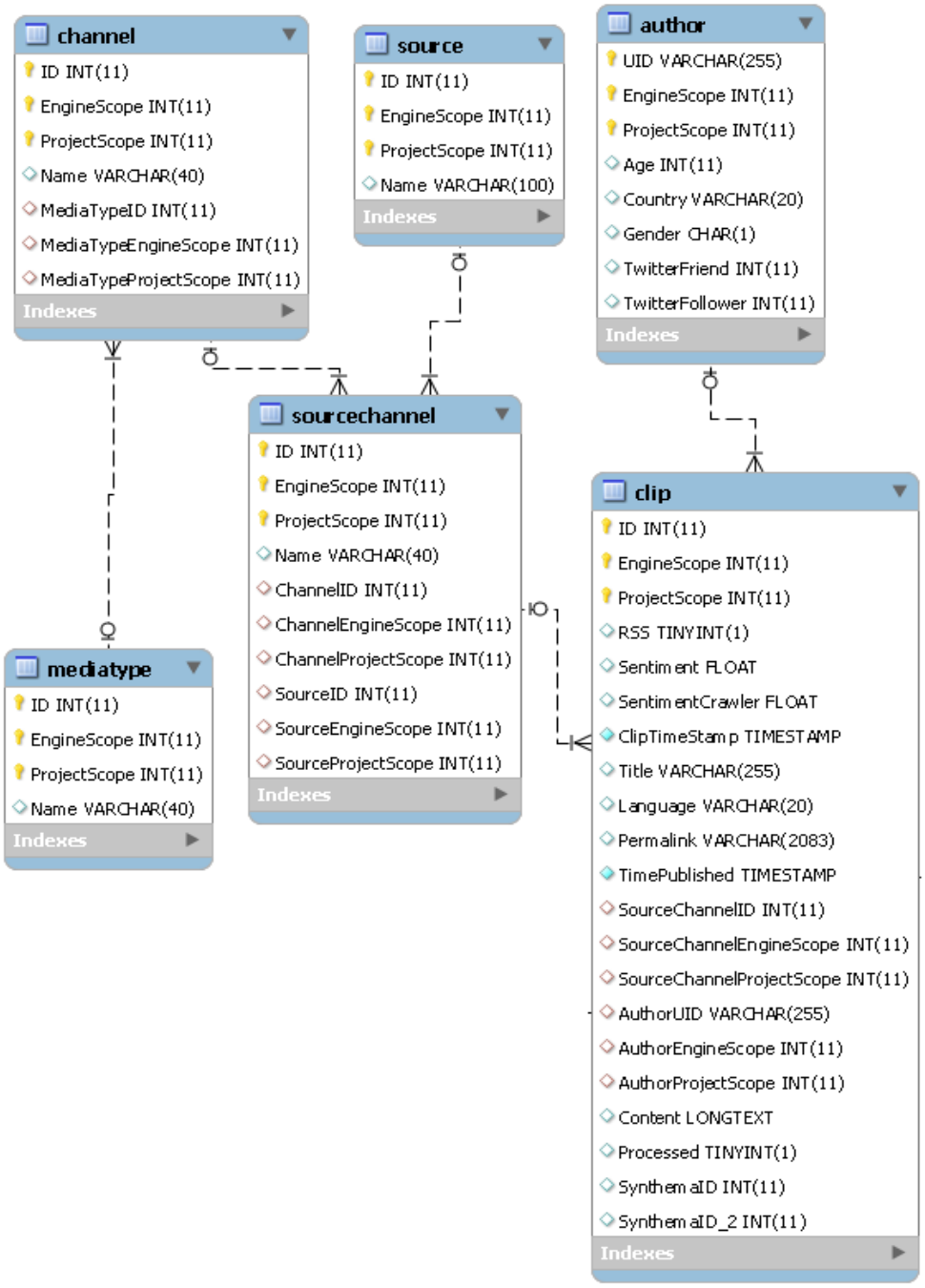

Figura 3.1 Schema relazionale di cui si richiede la conversione

L’impossibilità di eseguire join in MongoDB richiede una progettazione particolarmente consapevole delle interrogazioni che verranno successivamente svolte. Le principali metodologie di modellazione in MongoDB sono due:

- Accorpamento dei dati
- Suddivisione in collection distinte

L’accorpamento di informazioni in un'unica collection è generalmente la scelta preferibile ovunque sia possibile. La struttura non relazionale dei database NoSQL porta all’ovvia inadempienza alle forme normali, pertanto unire dati provenienti da fonti logicamente separate viene ritenuta pratica necessaria. L’entità principale su cui si opererà sarà la clip e il suo accesso è al centro delle scelte di modellazione. Le tabelle “mediatype”, “channel”, “sourcechannel” e “source” possiedono solo il campo “name” come elemento informativo. Non essendo un campo soggetto a cambiamenti frequenti, trova naturale modellazione nell’accorpamento all’interno delle clip; in questo modo a ogni clip sarà aggiunto un attributo “source_name”, “sourcechannel_name” e così via per le rimanenti tabelle. L’entità “Author” si presta invece a considerazioni che meritano approfondimenti. Campi quali “age”, “country” e “gender” sono considerabili statici o, al più, destinati a modifiche molto rare. Viceversa, gli attributi “twitterfriend” e “twitterfollower” sono campi potenzialmente soggetti a variazioni frequenti. Accorpando l’autore all’interno delle clip, a ogni cambiamento del valore di questi due ultimi campi si richiede l’analisi e la modifica di tutte le clip del database del relativo autore; se la sincronizzazione con le informazioni di twitter avvenisse in tempo reale, l’aggiornamento delle clip richiederebbe costi computazionali non trascurabili. Alternativamente è possibile considerare le entità “clip” e “author” come collection distinte, debolmente collegate tra di loro. A questo proposito si ricorda che MongoDB non dispone di controlli di integrità e coerenza sulle associazioni, pertanto la complessità nella gestione delle relazioni è a carico dell’applicazione o degli utenti, i quali dovranno accertare, in fase di inserimento o modifica, di non fare riferimento a

documenti inesistenti. Suddividendo le informazioni in collection distinte si risolve il problema dell'aggiornamento dei campi frequentemente variabili, ma si introduce una gestione delle interrogazioni più complessa. Ogni query che, per esempio, necessitasse di mostrare le clip di una serie di autori, si tradurrebbe in due interrogazioni distinte: una destinata a ottenere l'identificativo degli autori e una successiva rivolta alla ricerca delle corrispondenza tra author e clip, mentre una soluzione basata su accorpamento avrebbe richiesto un'unica query. Analizzando le operazioni eseguite sul database relazionale si trova che l'aggiornamento delle informazioni dinamiche enunciate precedentemente è un fenomeno non frequente, in quanto la sincronizzazione avviene su tempi maggiori di una settimana; a fronte di questa considerazione si è scelto di accorpare le informazioni dell'autore nelle clip. Il modello finale risultante è costituito dunque da un'unica collection comprendente tutti i campi delle tabelle in esame che non siano foreign key.

3.2 Analisi delle funzionalità di ricerca full-text

Il quantitativo di dati a cui si fa riferimento è sufficientemente grande per rendere sconveniente una ricerca basata su espressioni regolari e si necessitano strumenti alternativi quali la ricerca di tipo full-text. Un'operazione di questo tipo si avvale di un indice text (la cui definizione è data nel paragrafo 2.5), costruito come indicato nella figura 3.2.

```
db.dimostratore.ensureIndex(
    { Title: "text", Content: "text" },
    { weights: { Title: 2,Content: 1,},
    name: "TitleContentTextIndex",
    default_language: "italian" }
)
```

Figura 3.2 Query usata per la creazione dell'indice di testo

E’ possibile notare immediatamente come l’indice sia costruito su due campi, titolo e contenuto. Ad entrambi gli attributi è affidato un peso che sarà fondamentale in fase di interrogazione per determinare la priorità dei risultati; nel caso presentato l’occorrenza di una keyword nel titolo di una clip assume importanza maggiore della sua presenza nel contenuto.

Un secondo parametro della query che necessita di approfondimenti è il linguaggio default, una diretta conseguenza delle funzionalità di *stemming* offerte. Con tale termine si intende un processo che riduce una parola alla sua radice, per esempio entrambe le parole “andando” e “andare” hanno la stessa radice “and”. Bisogna notare come il risultato dello stemming non sia necessariamente la radice morfologica, ovvero il lemma (ad esempio, le parole “folla” e “follia”, hanno la stessa radice “foll” ma non sono riconducibili al medesimo lemma), pertanto è possibile la presenza di falsi positivi nel risultato. In MongoDB le funzionalità di stemming sono basate sul framework Snowball [**17**] dove, data una parola, la ricerca della radice non è effettuata attraverso l’ausilio di un dizionario ma tramite l’analisi delle regole di derivazione grammaticale. La scelta del linguaggio non è importante solo per la ricerca della radice, ma anche per la determinazione delle *stop words*, ovvero quello parole che non aggiungono contenuto particolarmente informativo alla frase e possono essere rimosse ai fini della ricerca. La fase di rimozione delle stop words è strettamente dipendente dal linguaggio esplicitato in fase di interrogazione, in quanto alcune parole sono significative in certi idiomi e non in altri; per esempio, impostando la lingua italiana, “and” è vista come testo da ricercare, mentre in lingua inglese viene identificata come congiunzione e rimossa. A differenza dello stemming, la ricerca delle stop words è un’operazione compiuta tramite dizionario. Una volta creato l’indice, è possibile effettuare ricerche di tipo full-text. A conclusione delle interrogazioni viene restituito all’utente un vettore contenente i risultati con il relativo punteggio di priorità, ordinato in modo discendente secondo quest’ultimo parametro.

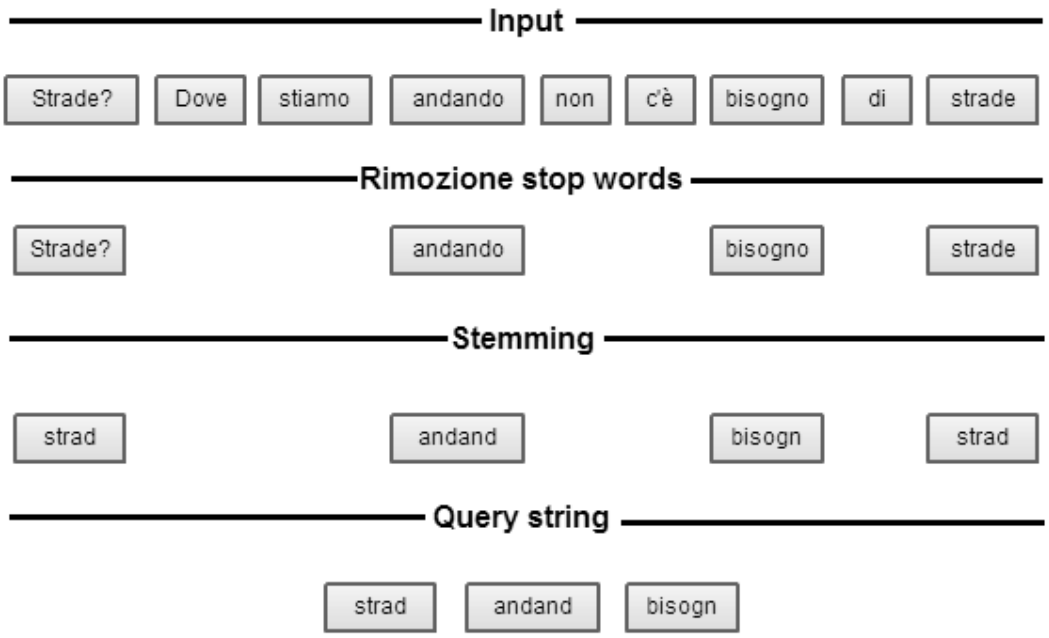

Figura 3.3 Esempio di formattazione input

Alla release attuale di MongoDB (v 2.4) le funzionalità di full-text search sono considerate sperimentali e l'uso in applicazioni in fase di produzione è sconsigliata. Una delle principali mancanze riguarda l'espressività della ricerca, che non dispone delle stesse potenzialità delle interrogazioni effettuate senza l'ausilio di indici testuali; a differenza di quest'ultime le query full-text non sono basate su cursori, pertanto alcune funzionalità quali Skip( ) e Sort( ) non sono disponibili. Un ulteriore difetto è dato dalle prestazioni non elevate che possono limitare l'utilizzo pratico su grandi quantità di dati; nel capitolo 4 si vedrà come si possono aumentare le performance distribuendo dati e operazioni su cluster e si provvederà a quantificare numericamente la differenza prestazionale.

3.3 Sviluppo dell'applicazione

L'applicazione che si intende sviluppare ha come centro delle funzionalità la ricerca di parole o frasi all'interno dei campi precedentemente indicizzati. Si è discusso come lato database sia possibile effettuare l'analisi attraverso funzionalità di ricerca full-text. Lato applicazione si deve invece garantire l'interazione con l'utente, la creazione della query e la stampa a video degli eventuali risultati ottenuti. L'interfaccia grafica creata per l'inserimento dei dati da parte dell'utente è riportata in figura 3.4

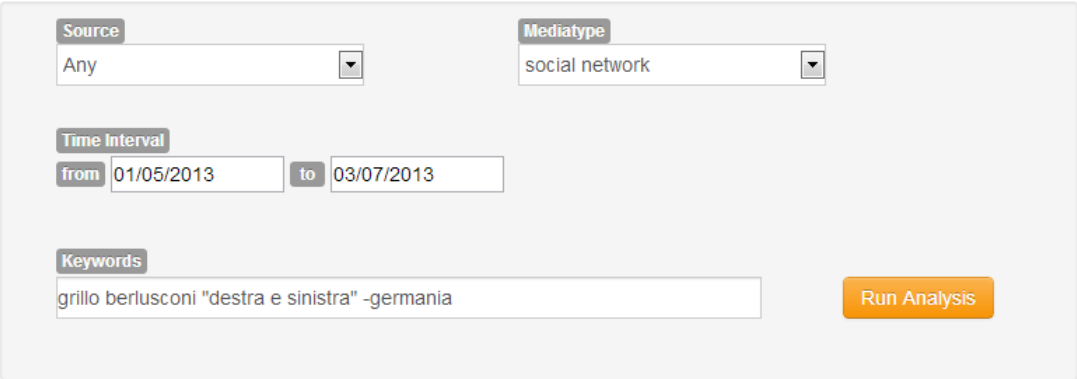

Figura 3.4 Interfaccia per l'inserimento dei dati

I campi "source" e "mediatype" permettono una prima selezione degli elementi su cui effettuare la query; il secondo, in particolar modo, garantisce l'interrogazione solo su clip di una particolare tipologia, quale blog, social network, sito web o microblog. Entrambi i campi sono valorizzati tramite due query al database, effettuate all'avvio dell'applicazione. L'inserimento delle keyword ricopre l'aspetto più importante, in quanto la ricerca full-text verrà eseguita su di esse. Sono garantiti gli operatori logici OR e NOT, così come la ricerca di frasi esatte. La disgiunzione logica è implicita, pertanto le parole chiave separate da spazi verranno considerate in OR tra loro. La ricerca di frasi esatte (proposizioni in cui l'ordine delle parole debba essere rispettato) viene eseguita racchiudendo la frase tra doppi apici, mentre l'esclusione di determinate

parole viene effettuata anteponendo il simbolo "-" alla keyword. Una volta completati i campi richiesti, è possibile eseguire l'interrogazione.

In caso di successo della query, si richiede la paginazione dei risultati da mostrare all'utente. In fase di analisi sono state identificate 3 diverse scelte implementative:

- Paginazione lato database: garantisce il trasferimento minimo dei dati, seppur un carico di lavoro maggiore per il database server, in quanto ogni pagina di risultati richiede una query. Secondo questa metodologia la gestione del numero di risultati viene effettuata tramite operatori LIMIT e SKIP in fase di query. Purtroppo, alla release attuale 2.4 di MongoDB questa scelta non è possibile in quanto per le ricerche full-text non sussiste una logica a cursori e la funzione SKIP non è supportata. Queste funzionalità sono previste nelle prossime versioni del DBMS e, qualora siano implementate, renderanno la paginazione lato database la scelta ottimale.

- Paginazione lato server web: se il database server ha tempi di trasferimento dati bassi verso il web server (ad esempio se sono fisicamente collegati) questa scelta permette di memorizzare il risultato complessivo lato web server per poi fornire all'utente finale solo la pagina di risultati richiesta. In questo caso i tempi di attesa dell'utente sono minimi, ma l'accumulo dei dati sul server web può provocare in breve tempo la saturazione delle risorse. Questa soluzione è apprezzabile in termini di reattività, ma realizzabile solo se il numero di utenti contemporanei è molto basso.

- Paginazione lato utente: sia il database server che il web server restituiscono al client l'intero set di risultati, che provvederà poi a gestirlo attraverso controlli Javascript. La mole di dati da trasferire può raggiungere la dimensione massima di 16 MB (limitazione causata da MongoDB nel restituire dati in formato BSON). In caso di connessioni lente questo fattore può rappresentare penalizzante, così come computer client dalle scarse specifiche hardware

potrebbero subire rallentamenti nel gestire 16 MB di dati tramite Javascript; al tempo stesso, questa soluzione necessita di poche risorse lato server e permette di servire diversi utenti contemporaneamente.

Valutando le caratteristiche dei metodi proposti, si è scelto di fornire una paginazione lato utente, in quanto connessioni veloci e computer dalle prestazioni adeguate sono aspetti piuttosto comuni. In questo modo, il massimo quantitativo di dati da gestire è pari a 16 MB, valore facilmente ridimensionabile con appositi filtri sui dati (in particolare sull'intervallo temporale di ricerca).

La stampa a video dei risultati delle query effettuate avviene nel corpo della pagina, dove vengono visualizzati, in ordine di punteggio (e quindi di importanza), data, titolo, contenuto e un'indicazione dell'apprezzamento delle singole notizie (campo "sentiment", già presente nel database di riferimento). Un'ultima sezione caratteristica è quella relativa alle statistiche, dove vengono mostrate informazioni sulla query eseguita. Tra i parametri visualizzati si trova il numero di documenti restituiti dall'interrogazione, il valore di riscontri delle keyword sull'indice, il tempo di esecuzione della query da parte del DBMS e il sentiment medio calcolato tra tutti i risultati ottenuti (grado di apprezzamento a livello di argomento, non più di singolo articolo). E' necessario specificare come il quantitativo di documenti restituiti non sia necessariamente pari al numero di match sull'indice. Questo comportamento è conseguenza del limite di 16MB che MongoDB pone sul risultato: qualora una query restituisse un risultato superiore a tale dimensione, verrebbe inviata al client solo la porzione di dati più significativa in termini di punteggio. In tale situazione si ha un numero di documenti restituiti inferiore a quello effettivamente riscontrabile nel database, mentre in ogni altro caso i due campi assumono lo stesso valore.

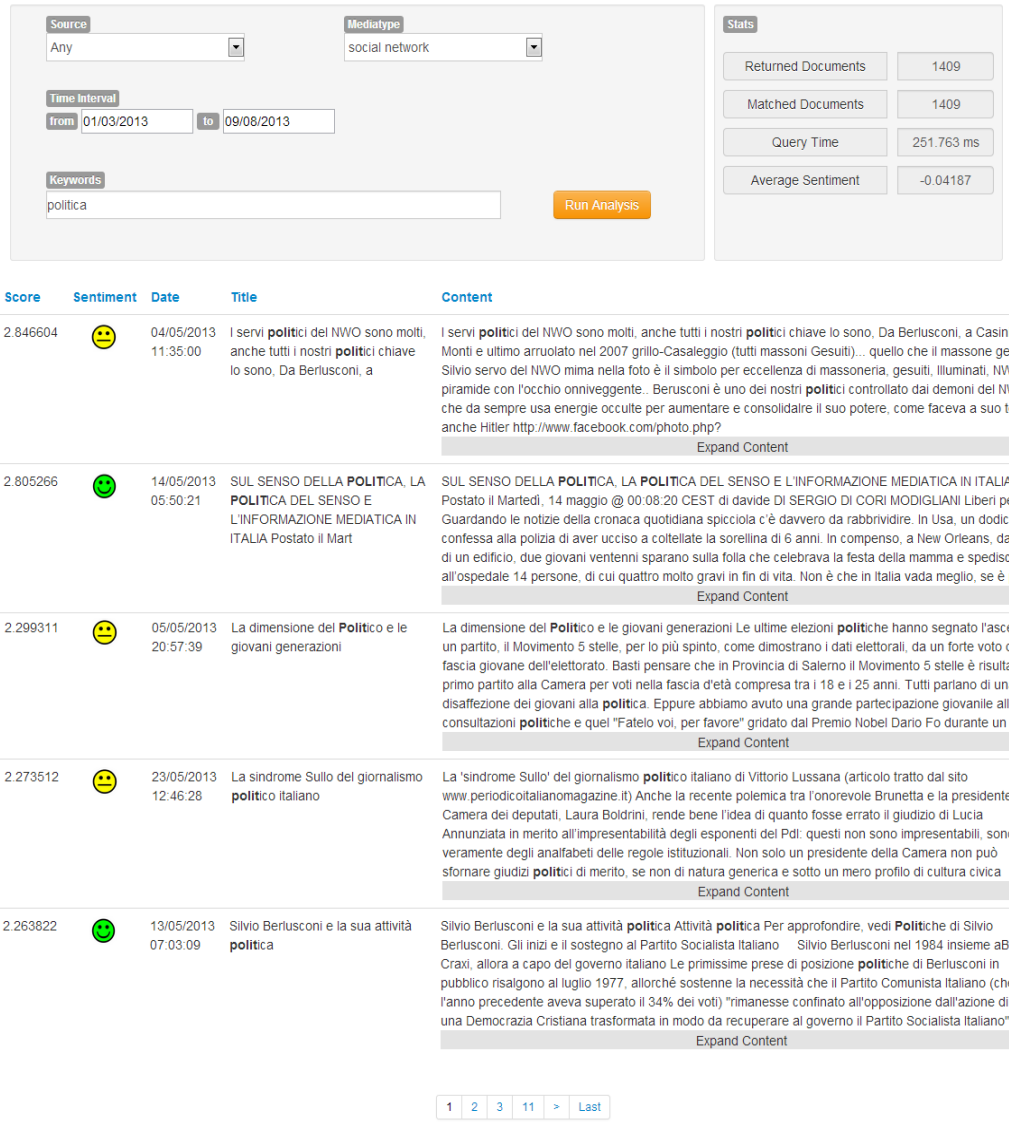

Figura 3.5 Applicazione web sviluppata

L'applicazione sviluppata è un prototipo esemplificativo di come si possano sfruttare le funzionalità offerte da MongoDB in contesti di analisi dei dati, come quelli risultanti dalle necessità della Social Business Intelligence.

Capitolo 4

Studio della scalabilità su cluster

4.1 Strumenti e metodologie

I sistemi NoSQL, tra cui MongoDB, fanno della distribuzione su cluster uno dei loro punti forti. In scenari distribuiti, una prima, semplice previsione del guadagno prestazionale che è possibile ottenere è legata linearmente alla crescita del numero di macchine presenti; per esempio, se su singolo server un'interrogazione impiega 100ms, si può supporre che un cluster di 2 server raddoppi le prestazioni risultando in una query di 50ms, 4 server dimezzino i tempi a 25ms e così via fino a raggiungere i limiti fisici offerti dall'infrastruttura di rete. In questo capitolo si vuole quantificare il guadagno prestazionale che si ottiene in ambiente cluster e identificare quali operazioni sui dati traggono il maggior beneficio in scenari distribuiti. I computer su cui sono stati eseguiti i test sono server fisici (non macchine virtuali) e possiedono tutti le medesime caratteristiche, illustrate nella tabella 4.1

Tabella 4.1 Caratteristiche hardware e software dei server oggetto di test

Componente	Valore
Processore	Pentium D 945 3.40 GHZ (dual core)
Memoria central	4GB DDR2 333Mhz
Hard Disk	Hitachi HDT725025VLA380 (7200 RPM)
Sistema operative	Windows 7 SP1 (64bit)
Versione MongoDB	v2.4.4 (64bit)

I test sono stati eseguiti prima su singolo server, poi su cluster di 2, 4 e 8 macchine. Mentre per un database in ambiente di produzione viene considerato necessario far corrispondere un replica set a ogni shard come misura di tolleranza ai guasti, al fine dei benchmark si è preferito utilizzare shard composti da un unico server senza replicazione dei dati. Permettendo le letture dei dati dai membri secondari dei replica set è possibile aumentare le prestazioni in lettura, ma solo in caso che il server primario sia congestionato o abbia una configurazione hardware meno potente, entrambe caratteristiche non presenti nei test effettuati (viene eseguita una query per volta e la dotazione fisica delle macchine è identica per tutti i membri del cluster). Si conclude che una soluzione basata su replica set non avrebbe portato vantaggi nei test eseguiti.

I test sono stati condotti su 3 database di uguale struttura ma differente volume dati. Il primo, denominato 1x, è il database usato per l'applicazione descritta nel capitolo 3 mentre i successivi due database, rispettivamente 10x e 100x, rappresentano una copia dei dati espansa secondo un fattore 10 e 100; in questi casi ogni documento originario viene inserito nel database un numero di volte pari al fattore moltiplicativo, con l'unica differenza dei campi "_id" e "data", impostati in modo da differenziare le singole copie tra loro.

La ripartizione dei dati nei diversi shard è stata eseguita utilizzando come shard key i campi “data” e “_id”. In MongoDB le query più rapide sono quelle che il cluster server riesce a indirizzare a target specifici piuttosto che a tutto il cluster, pertanto per ottenere buone prestazioni si preferisce usare una shard key composta da campi su cui viene imposta una condizione in fase di interrogazione. Nel prototipo di applicazione sviluppato, i campi nei quali si applica un filtro sui dati sono “source”, “mediatype” e “data”. I primi due non rappresentano una valida chiave di shard in quanto “source” è valorizzato in pochi documenti, mentre il campo “mediatype” presenta una concentrazione di clip associate distribuita non omogeneamente tra i valori di mediatype disponibili, dunque tale campo non può essere usato per distribuire il carico equamente. Al contrario, il campo data è presente in tutti i documenti e ha una distribuzione di valori più omogenea. Si è inoltre scelto di includere il campo “_id” nella chiave di shard in modo da garantire l’unicità nel probabile caso ci siano documenti con lo stesso timestamp.

Nelle successive pagine, una prima analisi riguarderà la ripartizione dei volumi dati e le prestazioni delle operazioni di setup database, quali inserimento dei record e analisi dei tempi di costruzione dell’indice testuale. Successivamente verranno testate le funzionalità d’uso più comune quali conteggio di elementi, uso dell’operatore distinct e operazioni di aggregazione. Quest’ultima caratteristica sarà particolarmente approfondita, mostrando le differenza prestazionali tra le diverse funzionalità che MongoDB mette a disposizione per raggruppare dati, così come l’incidenza nei tempi di calcolo che si ha ponendo una condizione sul dataset dato in input a tali operazioni. In ultima analisi verrà dato spazio allo studio delle prestazioni con query full-text, operazione di particolare importanza in quanto elemento fondamentale dell’applicazione destinata al social BI descritta precedentemente.

4.2 Analisi operazioni preliminari sul database

I dati sono stati caricati nel database attraverso un'applicazione scritta in linguaggio c# appositamente progettata. Le informazioni vengono lette da un file .csv (comma-separated values) e caricate in memoria centrale; una volta conclusa l'operazione viene effettuata una query di inserimento per ogni documento presente in memoria fino all'esaurimento dei dati. Al fine di massimizzare le prestazioni non sono state adottate politiche di *write concern*, ovvero controlli di correttezza sui dati al momento dell'inserimento, preferendo un'unica verifica eseguita a conclusione del popolamento del database. I tempi rilevati sono presentati in tabella 4.2

Tabella 4.2 Tempo popolamento database

	Database 1x (158.118 doc.)	**Database 10x (1.581.180 doc.)**	**Database 100x (15.811.800 doc.)**
Server singolo	1m 37s	20m	5h 30m
2 shard	3m 43s	32m 40s	6h 10m
4 shard	2m 55s	38m 3s	5h 49m
8 shard	3m 1s	28m 48s	5h 26m

Come è logico immaginare, se l'hardware dei dispositivi è identico e l'infrastruttura di rete comune, i tempi di popolamento del database non differiscono in modo significativo tra le soluzioni proposte. La richiesta di inserimento da parte dell'applicazione avviene sempre e solo a un singolo server; nel caso di ambiente cluster, l'attesa è data principalmente dal tempo con cui il cluster server evade la

richiesta allo shard di competenza, mentre in un ambiente single-server l'attesa è determinata dall'inserimento effettivo. I tempi di queste due operazioni sono simili e l'inserimento non è dunque un'operazione che trae particolari vantaggi prestazionali in ambiente cluster; in scenari in cui l'inserimento di record ricopre l'aspetto principale, si può preferire una soluzione con più cluster server operanti distintamente.

In figura 4.1 è riportato un esempio grafico di come i dati siano stati ripartiti nei vari shard da parte del balancer. Si è preferito mostrare la soluzione con il maggior numero di shard in quanto considerata quella più rappresentativa.

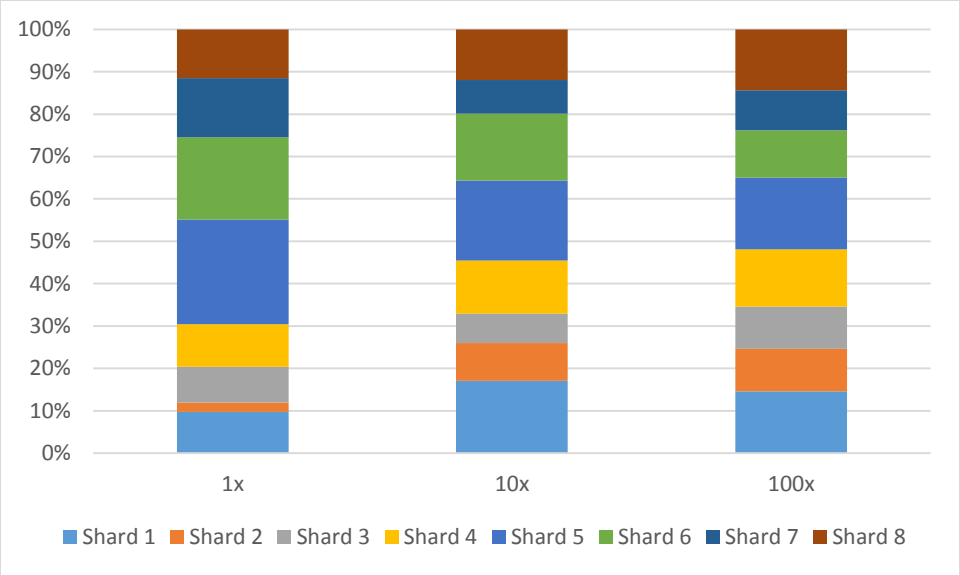

Figura 4.1 Ripartizione dati su 8 shard

E' immediato notare come il database 1x non permetta una distribuzione equa, mentre al crescere delle dimensioni si ha una ripartizione più soddisfacente. Le motivazioni di questo comportamento sono facilmente individuabili osservando le modalità con cui agisce il balancer automatico. Il bilanciamento dei dati viene avviato solo quando la differenza in numero di chunk tra due shard superi un certo limite; per database di piccole dimensioni come quello 1x la differenza limite è pari a 2 chunk ed essendo

ognuno di questi frammenti grande 64 MB a fronte di un database complessivamente pari a 380 MB, si può concludere che il balancer non venga attivato e i documenti restino ripartiti secondo la distribuzione eseguita al momento del popolamento. Nel caso del database 100x si assiste invece a un bilanciamento maggiormente equo, grazie al ruolo attivo del balancer.

Un'ulteriore valutazione che è possibile effettuare in fase di setup dei database è data dal tempo di costruzione dell'indice text. Tale operazione è molto dispendiosa in termini di risorse, sia come potenza di calcolo, sia come spazio occupato su disco; per poterne valutare le prestazioni si è deciso di avviarla solo a conclusione del popolamento del database. In figura 4.2 sono mostrati i tempi richiesti dall'operazione.

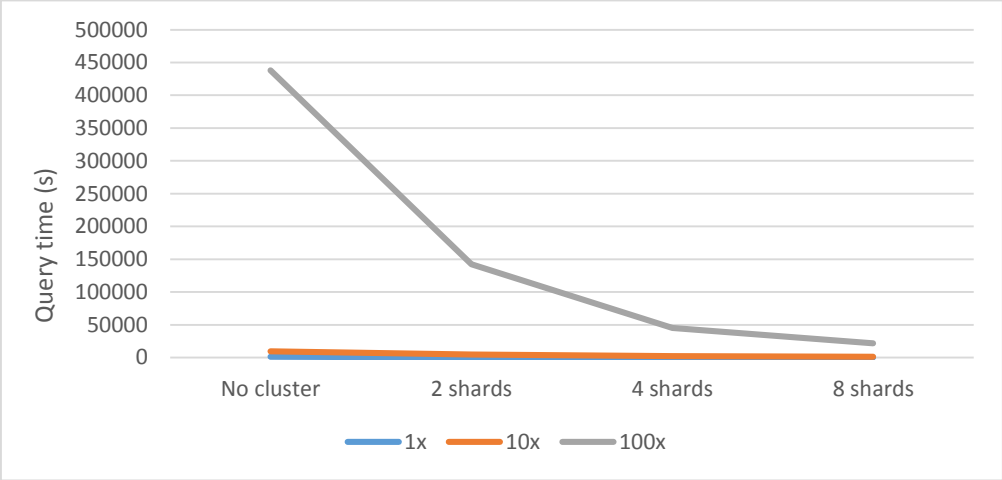

Figura 4.2 Grafico dei tempi di costruzione indice text

E' possibile notare come nel caso del database 1x e 10x i tempi siano in linea con i risultati che ci si aspetta nella distribuzione su multiple macchine, mentre per il database 100x si hanno tempi molto più elevati e variabili tra le varie configurazioni, in particolar modo per quanto riguarda la costruzione dell'indice da parte di un solo server. Le motivazioni di questo risultato si possono far risalire nelle modalità con cui viene creato l'indice testuale. Tale operazione è suddivisa in due fasi: nella prima viene

effettuato un ordinamento sui campi relativi all'indice, per poi memorizzare il risultato in una collection temporanea; successivamente, nella seconda fase, si ha la creazione vera e propria dell'indice. Durante quest'ultima operazione si ha la consultazione continua della collection temporanea, la quale, avendo nel caso del database 100x dimensione molto maggiore della RAM disponibile, causa il verificarsi di continui fenomeni di swapping su disco. Aumentando il numero di server si hanno invece collection temporanee sempre più piccole che portano a un numero minore di operazioni di swap.

Concluse le operazioni di setup, si vogliono riepilogare le caratteristiche principali dei 3 database. I risultati sono presentati in tabella 4.3

Tabella 4.3 Statistiche database

	Database 1x	**Database 10x**	**Database 100x**
# documenti	158.118	1.581.180	15.811.800
Dimensione dati	≈ 380 MB	≈ 3413 MB	≈ 34147 MB
Dimensione indice text	≈ 420 MB	≈ 4200 MB	≈ 42050 MB

E' importante osservare come lo spazio occupato su disco da dati e indice sia una stima media di quello osservato nei vari setup di macchine del cluster; MongoDB prealloca i file di sistema, pertanto da una configurazione all'altra la dimensione occupata può cambiare in base a come i dati vengano ripartiti e dunque dalle necessità di preallocare su un server più o meno datafile.

4.3 Analisi operazioni comuni

L'operazione più comune che si ha lavorando con MongoDB è la ricerca delle informazioni tramite funzione find( ). Attraverso questa istruzione è possibile effettuare selezioni sui dati, così come proiezioni dei risultati; i dati elaborati verranno presentati all'utente grazie a un cursore, il quale può essere sfruttato per la semplice lettura delle informazioni o come metodo di accesso ai dati per ulteriori operazioni quali conteggio dei risultati e loro ordinamento. Tra le funzioni disponibili si è scelto di analizzare le prestazioni dell'operazione di conteggio con condizione su campo non indicizzato, in modo da rendere necessario l'accesso fisico alla collection. Come in molte delle funzioni di MongoDB, è possibile esprimere il conteggio degli elementi in diversi modi, per esempio tramite l'uso della già citata find( ) oppure attraverso la specifica istruzione "count", entrambi i risultati sono equivalenti. La query usata nel test permette di ottenere il numero di documenti il cui sentiment sia strettamente positivo e la rappresentazione grafica dei relativi tempi di calcolo è data in figura 4.3

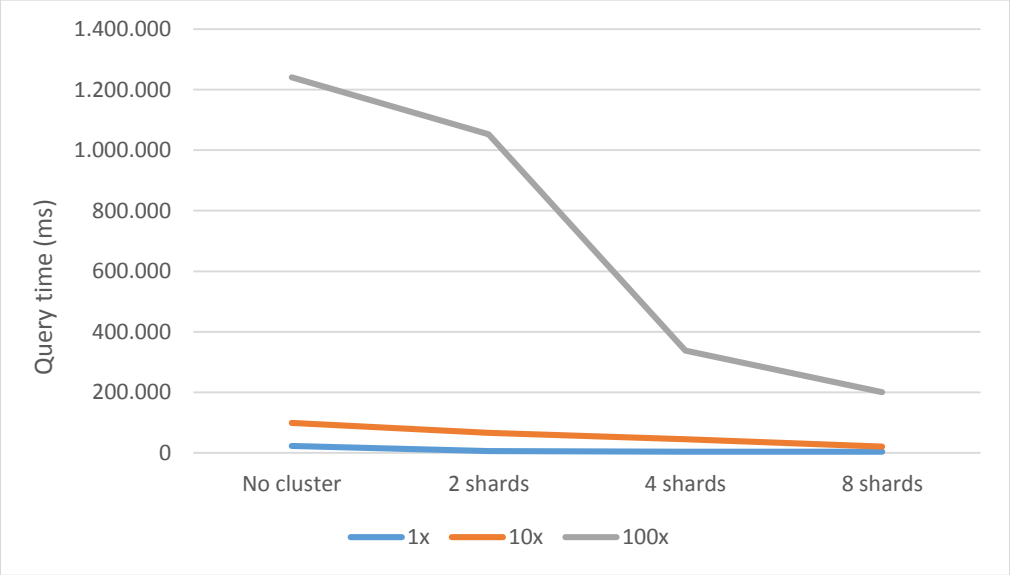

Figura 4.3 Grafico dei tempi di calcolo con operatore count

Dal grafico si nota subito come il conteggio degli elementi sia un'operazione che trae grande vantaggio dall'esecuzione in ambiente cluster. E' necessario notare come i tempi rilevati debbano essere analizzati solo in comparazione tra loro; in termini assoluti assumono valori piuttosto elevati per un database in ambiente di produzione. In tal contesto, se l'operazione di conteggio è frequente, è possibile e consigliabile creare un indice sui campi oggetto della query, al fine di minimizzare i tempi di calcolo.

Una seconda funzionalità di MongoDB che merita analisi è quella relativa alla selezione dei valori distinti di un determinato campo. I risultati del test condotto sono presentati in figura 4.4

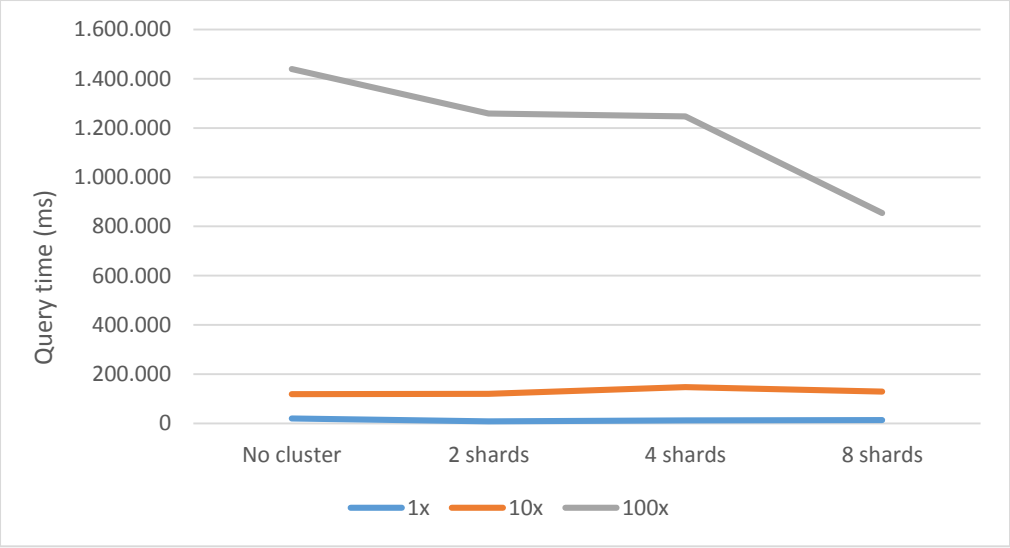

Figura 4.4 Grafico dei tempi di calcolo con operatore distinct

Il comportamento dell'operatore distinct è notevolmente diverso da quello riscontrato nel conteggio degli elementi eseguito in precedenza. Il database 100x mostra un guadagno prestazionale non elevato ma pur sempre presente, mentre nel caso dei due database 1x e 10x si ha addirittura un calo di prestazioni nell'esecuzione su cluster. In questi due casi, l'aumento dei tempi alla crescita del numero di macchine può essere

fatto risalire all'inevitabile varianza che si ha lavorando con tempi tra loro simili in modulo. Analizzando i dati si può concludere come l'operatore distinct non tragga particolare vantaggio dall'esecuzione in ambiente cluster, qualora venga applicato a piccole quantità di dati.

Le operazioni di aggregazione hanno tipicamente un impatto significativo sulle prestazioni di un sistema di basi di dati e pertanto lo studio delle loro performance in ambiente cluster può essere significativo per valutare la convenienza nell'applicare un approccio distribuito o meno. Come descritto nel paragrafo 2.8, i comandi di aggregazione messi a disposizione da MongoDB sono 3, ma in questa sede possono essere trattati solo Map Reduce e Aggregation Framework, in quanto l'uso del comando "Group" non è possibile in ambiente cluster. La query eseguita aggrega i dati per autore, esprimendo per ognuno di essi il sentiment medio delle relative clip; il risultato ottenuto tramite Aggregation Framework è fornito in figura 4.5

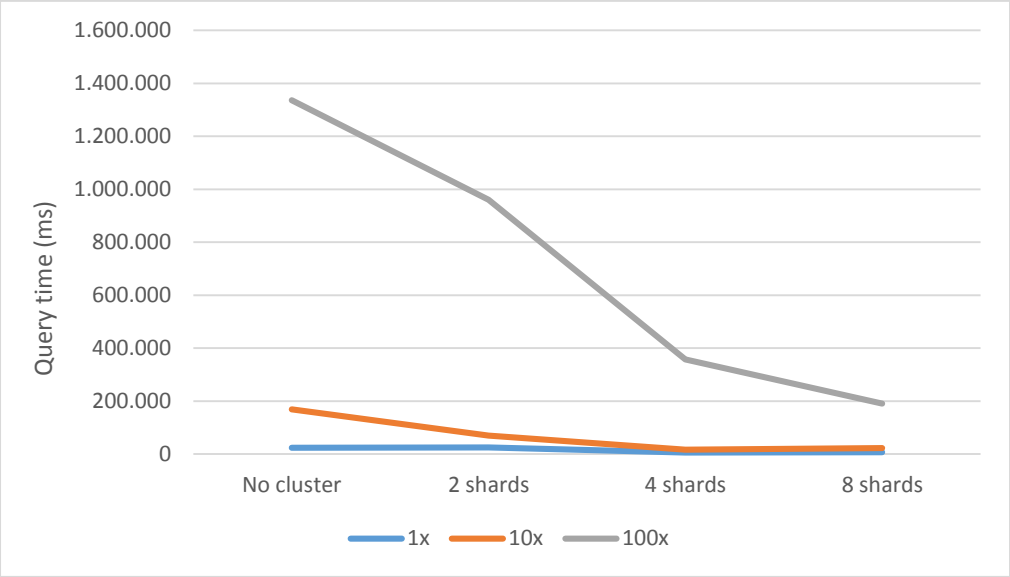

Figura 4.5 Grafico dei tempi di calcolo con Aggregation Framework

L'uso dei metodi offerti dall'Aggregation Framework porta a buoni risultati prestazionali, in particolar modo nel database 100x, con un decremento dei tempi di calcolo tra la soluzione single server e quella con 8 macchine per un fattore pari a 6,9 volte. Valutando l'operazione in termini assoluti, l'Aggregation Framework si dimostra una buona soluzione con piccole quantità di dati, richiedendo, su cluster di 8 macchine, solo 7 ms per eseguire l'interrogazione sul database 1x. In figura 4.6 sono invece esposti i risultati ottenuti effettuando l'interrogazione con Map Reduce.

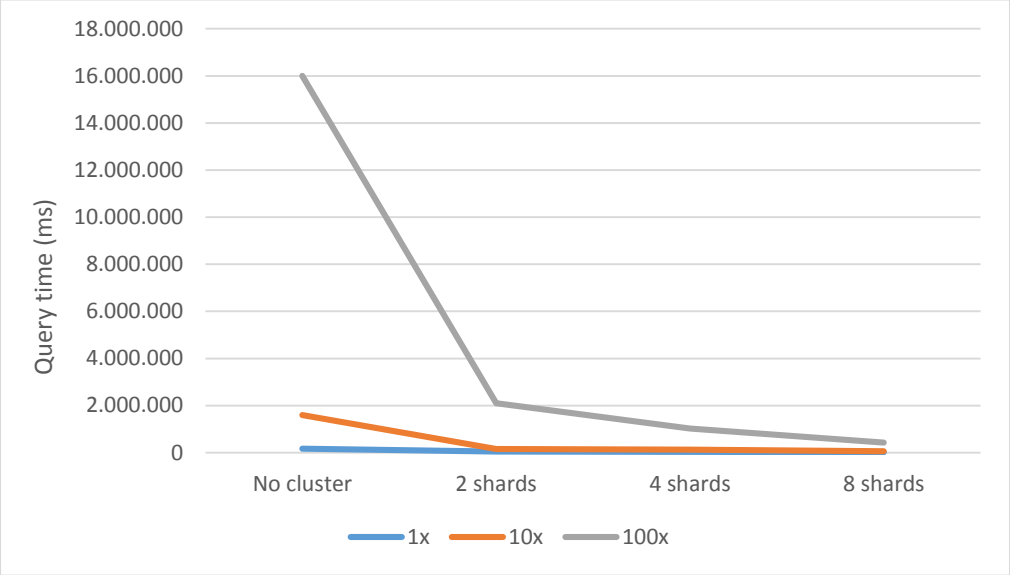

Figura 4.6 Grafico dei tempi di calcolo con Map Reduce

Il Map Reduce presenta un comportamento notevolmente differente rispetto a quello riscontrato tramite Aggregation Framework. Su singolo server si assistono a tempi computazionali di notevole entità, incomparabili con quelli offerti dalla precedente soluzione. Le motivazioni di questo risultato sono analoghe a quelle evidenziate durante la costruzione dell'indice text: il Map Reduce, pe aggregare i dati, utilizza una collection temporanea memorizzata su disco in fase di esecuzione della query. Avendo dimensioni troppo elevate per essere memorizzata in RAM, si hanno continui swap su

memoria di massa. Al diminuire della dimensione dei dati di ogni shard, si hanno prestazioni migliori, al punto che considerando i valori in termini relativi, il miglioramento che si ha raddoppiano il numero di macchine è maggiore rispetto a quello evidenziato tramite Aggregation Framework. Il Map Reduce è un framework sviluppato per supportare la computazione di grandi quantità di dati in ambiente cluster e i risultati ottenuti confermano questa specializzazione. Osservando i valori in termini assoluti, l'Aggregation Framework risulta sempre una scelta vincente nei test effettuati, ma dato il maggior incremento prestazionale al crescere del numero di shard offerto dal Map Reduce, è ragionevole supporre che in distribuzioni di larga scala quest'ultima sia la scelta migliore.

Sui metodi di aggregazione è stata condotta un'ulteriore analisi, per verificare il comportamento dei due framework in presenza di condizioni sui dati. Il test è stato effettuato solo su singolo server e si sono voluti recuperare solo i documenti il cui campo "Mediatype" fosse pari a "blog", per un risultato di dimensione pari a circa 1/7 di quella originale.

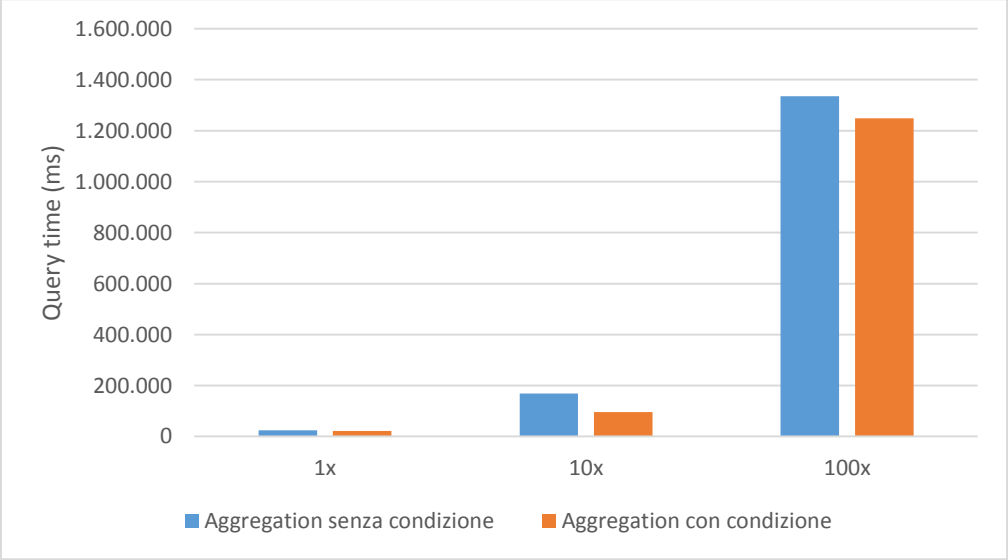

Figura 4.7 Comparazione Aggregation Framework con e senza condizione

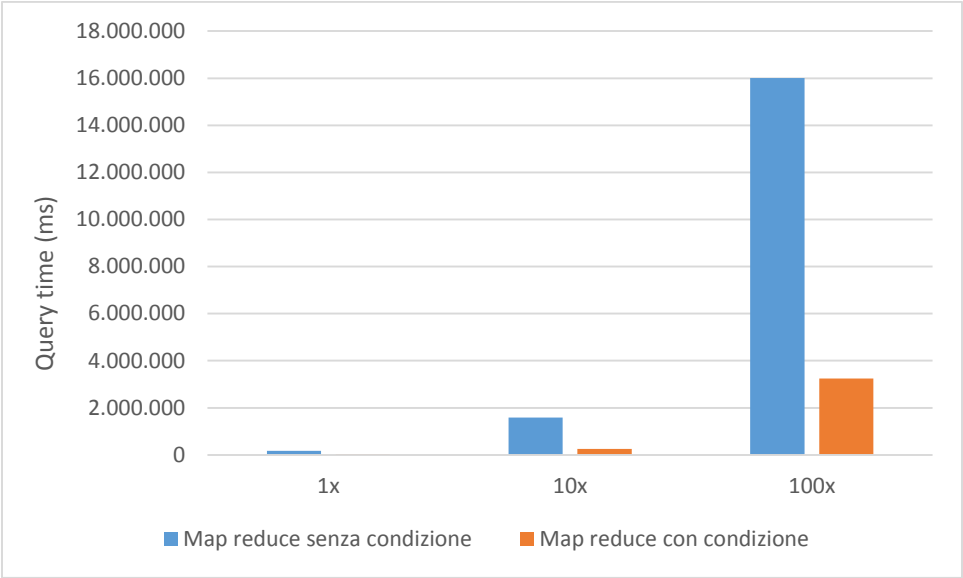

Figura 4.8 Comparazione Map Reduce con e senza condizione

Come previsto, l'esecuzione di Map Reduce risulta molto più rapida se viene posta una condizione sui dati in input; in ambiente single server un dataset iniziale più ristretto significa una collection temporanea di minore dimensione, il cui vantaggio è stato descritto precedentemente. Al contrario, l'Aggregation Framework non sembra trarre grande beneficio dal filtraggio iniziale dei dati, pur organizzando secondo le giuste priorità la pipeline della query (match condizionale eseguito prima dell'operazione di raggruppamento). Nonostante la poca differenza che comporta la condizione, se comparato al Map Reduce, l'Aggregation Framework risulta ancora una volta più performante in un ambiente non distribuito.

4.4 Analisi operazioni su indice text

L'uso degli indici text rappresenta, al momento attuale, una feature sperimentale e non è consigliata l'esecuzione in ambienti di produzione. In particolar modo le prestazioni non sono ottimizzate, così come risulta evidente la mancanza di alcune funzionalità come l'ordinamento dei risultati, caratteristiche presenti invece nelle query eseguite tramite operatore find( ). Nel capitolo 3 si è visto come sia possibile aggirare tali mancanze con un'opportuna programmazione lato client, mentre in questa sede si vogliono quantificare le prestazioni. Seppur i tempi rilevati siano elevati per essere considerati in termini assoluti, posti in relazione tra loro, nelle diverse configurazioni del cluster, assumono maggiore espressività. Le operazioni su indice text permesse da MongoDB e presenti nella seguente analisi riguardano la ricerca di parole chiave singole, multiple, negate (dunque non presenti) e frasi esatte. I test sono stati effettuati su tutti i 3 database descritti in precedenza, ma qui verranno riportati, salvo nel caso della ricerca di singole keyword, solo i risultati relativi al database 100x, in quanto maggiormente espressivi e più idonei per valutare il comportamento del cluster. Il tipo di contenuto maggiormente presente nei database in esame è l'articolo di giornale, specificatamente orientato all'argomento della politica. Le query rifletteranno questo comportamento ricercando keyword che generino risultati significativi e numerosi. In tutte le interrogazioni, ad eccezione di quella con negazioni multiple, si è applicato un limite nel numero di risultati pari a 100.

Il primo test preso in esame riguarda la ricerca delle singole parole chiave. A seconda della keyword usata si hanno risultati molto differenti tra loro; la velocità di ricerca è strettamente legata al numero di occorrenze e nel caso di parole frequenti si arriva a notare tempi di un ordine di grandezza superiore. Si è dunque deciso, allo scopo di ottenere risultati meglio rappresentativi, di ricercare keyword che generassero un numero variabile di risultati (da un minimo di 1762 elementi a un massimo di 44014,

per quanto riguarda il database 1x) e di presentare i dati come media dei tempi di tutte le interrogazioni eseguite. I risultati sono mostrati in figura 4.9

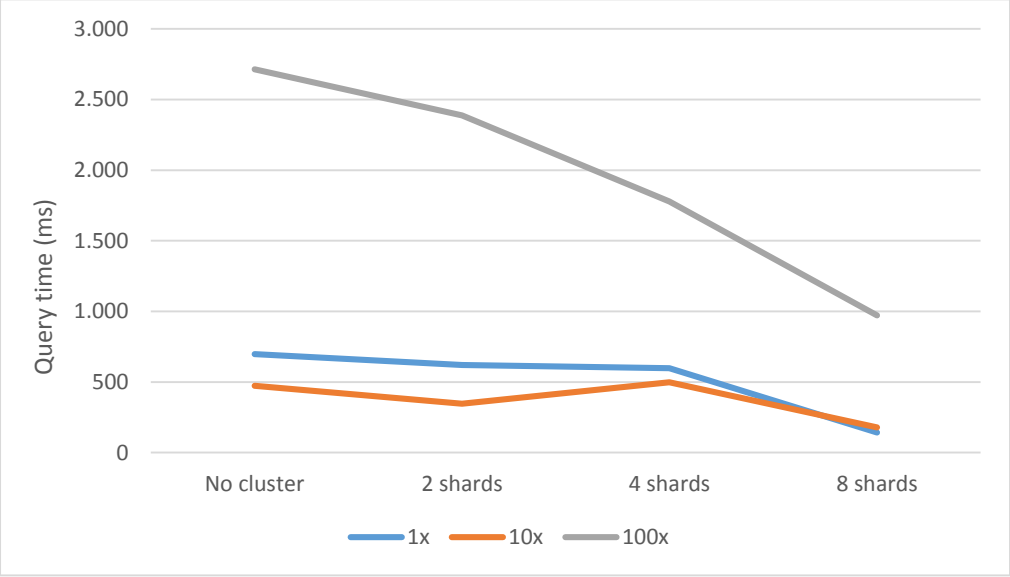

Figura 4.9 Tempi ricerca text di keyword singola

La prima caratteristica che si può riscontare è una maggiore lentezza nell'esecuzione dell'operazione sul database 1x rispetto a quello 10x. I test sono stati ripetuti più volte, assicurandosi di azzerare la cache tra le diverse prove, ma tale anomalia si è sempre presentata. Il database 100x presenta invece un risultato più consono alle aspettative, seppur non benefici in modo determinante dell'ambiente distribuito.

Il secondo tipo di esame prestazionale riguarda la ricerca di parole multiple. A differenza del precedente test, in questa sede si vogliono mostrare le differenze prestazionali che si hanno al crescere del numero di parole cercate. I risultati proposti in figura 4.10 sono quelli ottenuti su database 100x.

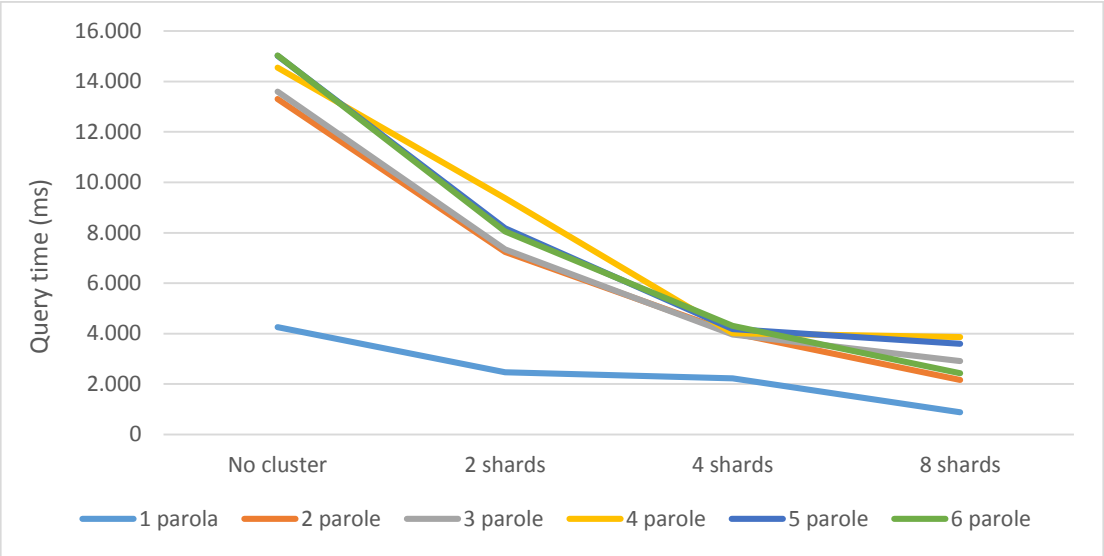

Figura 4.10 Tempi ricerca text di keyword multiple

Le motivazioni dei risultati ottenuti sono identificabili nelle modalità con cui è stata eseguita la ricerca. La prima parola cercata prevede un numero di risultati pari a 1.209.900 nel caso del database 100x, mentre aggiungendo la seconda parola si hanno 5.611.300 elementi. Dal grafico è possibile notare la corrispondenza nei tempi di ricerca, con valori bassi per la prima interrogazione e tempi molto maggiori nel secondo caso. L'aggiunta di nuove parole comporta piccoli aumenti nel numero di valori restituiti e tale situazione è nuovamente riscontrabile analizzando i tempi nel grafico. Si conclude che i tempi di ricerca di parole multiple non siano legati direttamente al numero di keyword presenti nell'interrogazione, ma bensì al quantitativo di occorrenze identificate e dunque al tempo richiesto dalla lettura delle informazioni nella collection fisica (per esempio è possibile che la ricerca di due parole molto occorrenti richieda più tempo della ricerca di 10 keyword scarsamente presenti nel testo). Dal punto di vista del cluster si ha invece un maggior vantaggio prestazionale nella distribuzione su più macchine, rispetto alla ricerca della keyword singola.

La ricerca di keyword negate è normalmente un'operazione meno frequente ma altrettanto meritevole di considerazione. La query composta in fase di interrogazione prevede una parola, sempre uguale, da cercare e un numero di negazioni crescenti, sino al massimo di 6 parole negate. Il risultato che si desidera è dato da tutti gli elementi che contengono la prima parola ma non le successive keyword negate. Il test effettuato presenta un'importante differenza rispetto a quelli eseguiti precedentemente e si è preferito rimuovere la limitazione sul numero di risultati. Questa scelta è stata fatta in seguito all'analisi dei risultati ottenuti con la presenza dell'operatore limit; in tal caso si sono osservati tempi sempre uguali, indipendentemente dal numero di keyword e, soprattutto, dal database usato. In questo test l'imposizione del limit non permette di esprimere le prestazioni dell'operatore di negazione, pertanto si è deciso di rimuoverlo. E' importante notare come questo fattore implica tempi di query molto più elevati di quelli evidenziati nei test con singole e multiple keyword, rendendo erroneo il confronto di questo test con i precedenti. Il risultato dell'analisi condotta è mostrato in figura 4.11

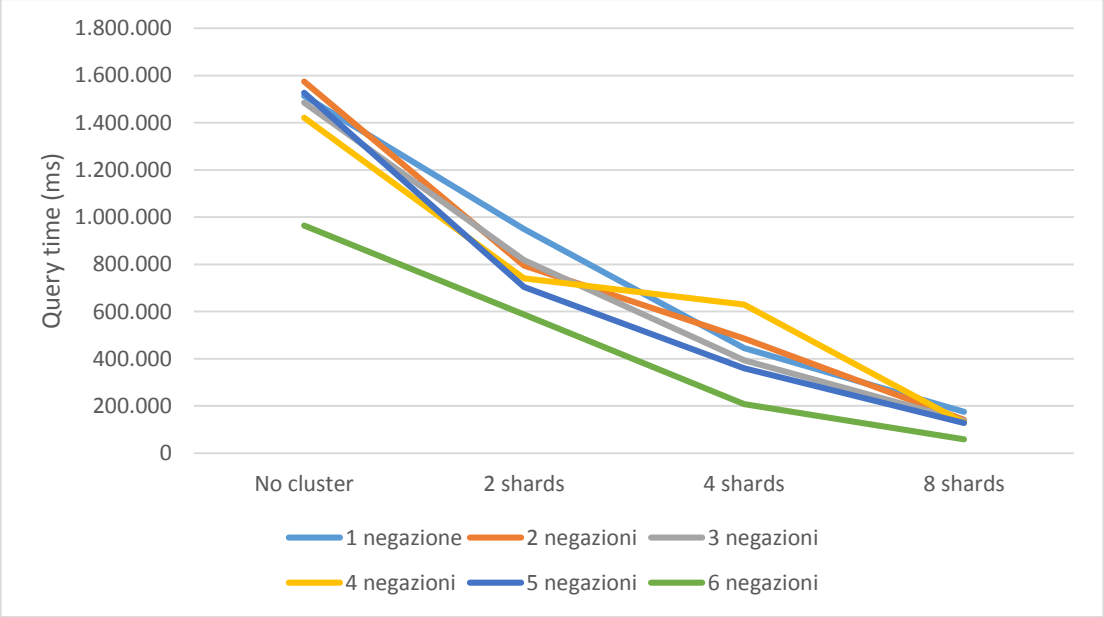

Figura 4.11 Tempi ricerca text di keyword multiple negate

I risultati evidenziati presentano importanti differenze rispetto a quelli ottenuti nei test precedenti. Innanzitutto si nota che i tempi, presi in termini assoluti, sono di notevole entità, come diretta conseguenza della mancanza dell'operatore limit. In secondo luogo si può osservare come generalmente i tempi di query diminuiscano all'aumentare del numero di negazioni. Ad un maggior numero di parole escluse si ha un quantitativo minore di match sull'indice e dunque un numero inferiore di documenti da leggere nel database. Essendo l'accesso al disco il principale fattore di diminuzione delle prestazioni, quanto più si riesce a ridurre il numero di documenti letti, sia attraverso condizioni sui dati, sia attraverso raffinamenti della ricerca sull'indice, tanto più si ottengono performance migliori. Dal punto di vista del comportamento dell'operatore di esclusione in ambiente cluster, si hanno ottimi risultati, con un aumento delle prestazioni crescente al pari del numero di macchine presenti.

L'ultima analisi effettuata in presenza di indici di testo riguarda la ricerca di frasi esatte, ovvero combinazioni di keyword in cui l'ordine di apparizione debba essere rispettato. I risultati sono presentati in figura 4.12

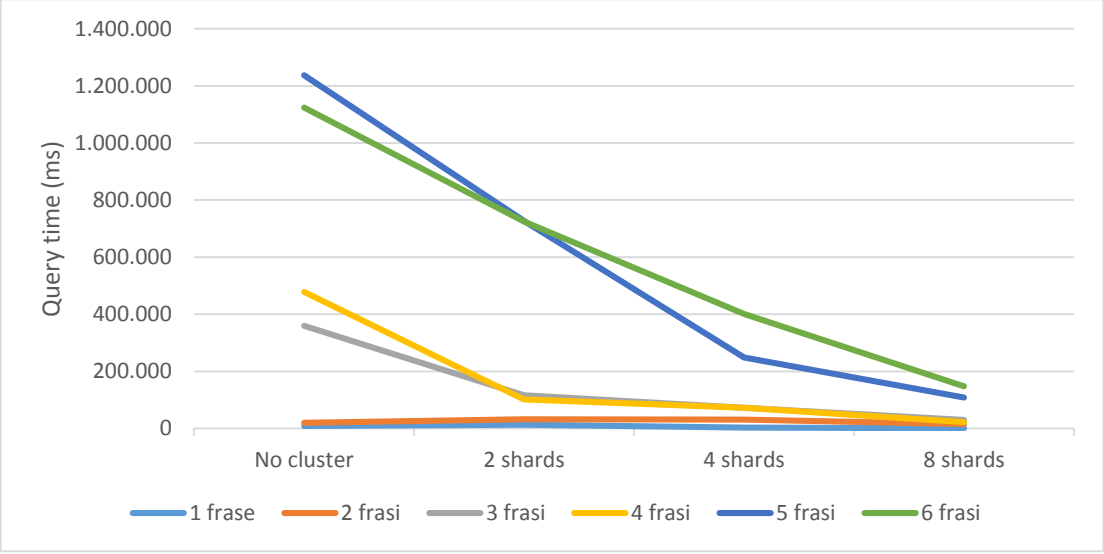

Figura 4.12 Tempi ricerca text di frasi esatte

La ricerca di frasi esatte ha un impatto sulle prestazioni notevole, anche in presenza del limite sul numero di risultati restituiti. Su architettura single server la ricerca di 6 frasi richiede circa 18 minuti, operazione inammissibile in molti ambienti rivolti alla produzione. Lo scaling su cluster è elevato e considerando l'operazione in termini relativi, il guadagno di prestazioni è considerevole. Le motivazioni per cui i tempi delle singole query siano così elevati in senso assoluto si possono ricondurre alle modalità di ricerca che MongoDB impiega in caso di frasi esatte. Ogni proposizione viene separata nelle parole che la compongono e per ognuna di esse viene calcolato lo stem. Successivamente vengono identificati tutti i documenti in cui compaiono almeno una volta le radici delle keyword e, per ogni risultato ottenuto, viene osservato in quali casi si presenta la specifica frase richiesta. Come è intuibile questa concatenazione di operazioni ha pesanti ripercussioni sui tempi di riposta e, seppur l'operazione benefici notevolmente dell'ambiente cluster, per una buona reattività del sistema è consigliato minimizzare la ricerca di frasi esatte.

Conclusioni

Durante questo studio di tesi sono state affrontate le tematiche del movimento NoSQL e il caso specifico di MongoDB da diversi punti di vista: in un primo momento si è svolta un'analisi funzionale volta a determinare le caratteristiche messe a disposizione da questi strumenti, per poi proseguire con la dimostrazione pratica di una delle funzionalità più peculiari di MongoDB quale la ricerca full-text e infine, ottenuto un prototipo applicativo, si è discusso come poterne migliorare le prestazioni attraverso l'ausilio di un cluster. Lo scopo di questo percorso è stato quello di fornire gli strumenti necessari per valutare il possibile utilizzo di MongoDB come alternativa, in certi settori, alle soluzioni relazionali. L'osservazione più evidente che si ha analizzando il panorama dei DBMS NoSQL, è quella relativa alla specializzazione di tali soluzioni. Caratteristiche quali il mancato supporto alle transazioni, rendono questi prodotti inadeguati in contesti in cui si richiede la completa esecuzione di una serie di operazioni, come una transazione finanziaria o la gestione di un magazzino, entrambi aspetti tipici, ad esempio, di un sistema di e-commerce. In questi scenari, un prodotto come MongoDB può risultare vantaggioso solo se utilizzato come supporto secondario;

applicando una simile soluzione al concetto di e-commerce possiamo ottenere un catalogo dei prodotti basato su MongoDB, in modo da beneficiare del modello dati flessibile, relegando transazione economica e gestione delle giacenze ad un sistema relazionale. Nonostante la possibilità di avere sistemi ibridi, le soluzioni NoSQL trovano naturale collocazione in ambienti massivamente distribuiti. A questo proposito si è visto, illustrando il teorema di Brewer, che per garantire la tolleranza alle partizioni di rete si deve rinunciare a garanzie di consistenza o disponibilità dei dati; la scelta intrapresa dal team di MongoDB sacrifica quest'ultima caratteristica in favore degli altri due aspetti. Tale soluzione ha forti ripercussioni sugli ambiti ai quali può essere applicato con successo MongoDB, pertanto in fase decisionale si deve valutare il fatto che, in caso il nodo primario di uno shard vada offline, i dati non sono disponibili da altre fonti; la consistenza viene garantita (quando è possibile leggere un dato lo si può definire attendibile), ma bisogna considerare una manutenzione rapida e costante dei server per garantire la continuità del servizio.

Uno degli aspetti di MongoDB più criticati è la scarsa specializzazione del prodotto, ovvero la disponibilità di numerose funzioni, appartenenti agli ambiti più diversi, ma sviluppate in modo poco approfondito e, dunque, non sempre capaci di modellare un problema reale. Un tipico esempio sono gli strumenti di ricerca full-text, caratteristica non comune nella maggioranza dei DBMS, presenti invece in MongoDB ma dalle funzionalità ridotte rispetto a soluzioni esplicitamente dedicate a questi ambiti quali DBSight e Clusterpoint. E' possibile dunque definire MongoDB come uno dei prodotti più *general purpose* tra le soluzioni NoSQL, pur sempre rimanendo nel settore di nicchia tipico dei DBMS non relazionali. Tuttavia, tale mancanza di specializzazione non rappresenta necessariamente una caratteristica negativa e la capacità di modellare un gran numero di situazioni differenti presenta importanti vantaggi in applicazioni dedicate all'analisi di dati, il cui esempio fornito in questo studio di tesi è dato dalla Social Business Intelligence, ambito nel quale la grande variabilità dei dati forniti dai social network ha nella flessibilità degli strumenti d'analisi un requisito fondamentale.

Sempre in funzione delle operazioni su grandi quantità di dati, si è dimostrato come la distribuzione su cluster sia una scelta tipicamente molto vantaggiosa. In particolar modo per le operazioni più computazionalmente dispendiose quali le aggregazioni, l'ambiente distribuito ha presentato notevoli vantaggi in termini prestazionali, arrivando ad un incremento di performance, tra la soluzione single server e quella con 8 shard, di circa 32 volte. Nonostante gli ottimi risultati, bisogna notare come non tutte le operazioni effettuate traggano un vantaggio significativo dall'ambiente cluster; tra di esse figurano l'inserimento dei record e l'uso dell'operatore DISTINCT. Uno scenario che faccia uso intensivo della creazione di nuovi documenti è quello dei server di log, caso in cui, qualora siano richieste prestazioni notevoli, si può preferire una soluzione di scaling verticale. Seppure sia raro che un'applicazione ponga l'operazione di distinzione dei documenti al centro delle sue funzionalità, anche in tal caso può essere conveniente considerare politiche di aggiornamento dell'hardware esistente, piuttosto che scenari di distribuzione.

E' necessario far notare come un'analisi delle caratteristiche come quella proposta in questo studio non possa essere completa, soprattutto a fronte del rapido ciclo di sviluppo a cui è sottoposto MongoDB. Un prodotto simile richiede una valutazione costante, soprattutto se relazionata alla prestazioni, lasciando dunque aperte diverse possibilità nell'espansione del materiale. In particolar modo, l'applicazione sviluppata si presta all'aggiunta di nuove funzionalità qualora nelle successive versioni di MongoDB vengano ampliate le possibilità di ricerca full-text.

A conclusione di questo studio di tesi si può definire MongoDB un prodotto soddisfacente, dalle diverse funzionalità e capace di garantire una modellazione dei dati facile e veloce, il cui uso in scenari reali destinati alla produzione merita di essere preso in considerazione in fase decisionale.

Appendice

In questa sezione sono riportate le query e le tabelle da cui sono stati ricavati i grafici presentati nell'analisi delle prestazioni effettuata nel capitolo 4.

Ripartizione dei dati su 8 shard

La tabella mostra il numero di documenti per shard

	Database 1x	**Database 10x**	**Database 100x**
Shard n.1	15.411	269.715	2.304.919
Shard n.2	3.494	140.695	1.588.395
Shard n.3	13.282	109.674	1.577.548
Shard n.4	15.943	198.164	2.129.632
Shard n.5	39.040	299.478	2.676.475
Shard n.6	30.638	249.231	1.764.174
Shard n.7	22.062	125.156	1.496.111
Shard n.8	18.248	189.067	2.274.546

Costruzione indice text

La tabella mostra i tempi rilevati in secondi

	Database 1x	Database 10x	Database 100x
Server singolo	1.005	9.630	438.060
2 shard	360	4.740	142.200
4 shard	200	2.000	45.000
8 shard	125	1.080	21.960

Conteggio degli elementi con condizione sui dati

```
db.runCommand( { count:'dimostratore_1x',
                 query: { SentimentCrawler: { $gt: 0 } }
               } )
```

La tabella mostra i tempi rilevati in millisecondi

	Database 1x	Database 10x	Database 100x
Server singolo	22.908	99.090	1.240.400
2 shard	6.302	66.347	1.052.820
4 shard	3.855	44.627	337.662
8 shard	3.861	20.550	200.420

Operatore Distinct

```
db.dimostratore_1x.distinct("SourceName");
```

La tabella mostra i tempi rilevati in millisecondi

	Database 1x	Database 10x	Database 100x
Server singolo	20.126	118.183	1.439.290
2 shard	8.461	120.567	1.258.240
4 shard	11.961	147.199	1.247.500
8 shard	13.254	129.238	855.124

Aggregation Framework senza condizione sui dati

```
db.dimostratore_1x.aggregate
(
    { $group: { _id: "$AuthorUID", avgsentiment: { $avg: "$SentimentCrawler" } } }
)
```

La tabella mostra i tempi rilevati in millisecondi

	Database 1x	Database 10x	Database 100x
Server singolo	24.188	168.729	1.335.690
2 shard	25.520	70.439	959.859
4 shard	5.909	17.114	357.433
8 shard	7.432	22.818	191.395

Map Reduce senza condizione sui dati

```
var mapFunction = function() {
    var key = this.AuthorUID;
    var value = {
        count: 1,
        sentiment: this.Sentiment
    };
    emit(key, value);
};
var reduceFunction = function(keyChannelName, valuesSentiment) {
                     reducedVal = { count: 0, sentiment: 0 };
                     for (var idx = 0; idx < valuesSentiment.length; idx++) {
                         reducedVal.count += valuesSentiment[idx].count;
                         reducedVal.sentiment += valuesSentiment[idx].sentiment;
                     }
                     return reducedVal;
                     };
var finalizeFunction = function (key, reducedVal) {
                       reducedVal.avg = reducedVal.sentiment/reducedVal.count;
                       return reducedVal;
                     };
db.dimostratore_1x.mapReduce(
                     mapFunction,
                     reduceFunction,
                     {

                       out : {inline : 1}, //Stampa a console
                       finalize: finalizeFunction
                     }
                   )
```

La tabella mostra i tempi rilevati in millisecondi

	Database 1x	Database 10x	Database 100x
Server singolo	169.522	1.591.720	16.006.200
2 shard	45.012	155.516	2.096.870
4 shard	38.636	128.506	1.017.600
8 shard	16.266	55.875	432.026

Comparazione Aggregation Framework con/senza condizione

La tabella mostra i tempi rilevati in millisecondi

	Database 1x	Database 10x	Database 100x
Senza condizione	24.188	168.729	1.335.690
Con condizione	21.455	96.288	1.248.600

Comparazione Map Reduce con/senza condizione

La tabella mostra i tempi rilevati in millisecondi

	Database 1x	Database 10x	Database 100x
Senza condizione	169.522	1.591.720	16.006.200
Con condizione	39.224	254.257	3.239.460

Ricerca keyword singola con indice text

Le parole ricercate sono: grillo, berlusconi, crimi, bossi, casini, vendola.

I tempi visualizzati sono una media tra i tempi di ricerca di ogni keyword.

```
db.dimostratore_1x.runCommand( "text", {
search: "grillo"
,limit: 100
,filter: {  }
,language: "italian"
,project: {_id: 1 }
})
```

La tabella mostra i tempi rilevati in millisecondi

	Database 1x	Database 10x	Database 100x
Server singolo	698	473	2.713
2 shard	620	346	2.387
4 shard	598	498	1.778
8 shard	142	178	972

Ricerca keyword multiple con indice text (db 100x)

```
db.dimostratore_1x.runCommand( "text", {
    search: "grillo berlusconi crimi bossi casini vendola"
    ,limit: 100
    ,filter: {  }
    ,language: "italian"
    ,project: {_id: 1 }
})
```

La tabella mostra i tempi rilevati in millisecondi

	1 parola	2 parole	3 parole	4 parole	5 parole	6 parole
Server singolo	4.251	13.310	13.603	14.553	15.027	15.032
2 shard	2.467	7.236	7.346	9.367	8.178	8.045
4 shard	2.218	4.021	3.963	4.014	4.184	4.311
8 shard	876	2.157	2.914	3.864	3.598	2.422

Ricerca keyword negate con indice text (db 100x)

```
db.dimostratore_1x.runCommand( "text", {
    search: "grillo -berlusconi -crimi -bossi -casini -vendola -fini"
    ,limit: 20000000
    ,filter: {  }
    ,language: "italian"
    ,project: {_id: 1 }
})
```

La tabella mostra i tempi rilevati in millisecondi

	1 negaz.	2 negaz.	3 negaz.	4 negaz.	5 negaz.	6 negaz.
Server singolo	1.514.323	1.574.444	1.485.539	1.421.222	1.527.853	963.944
2 shard	949.484	795.713	817.505	741.496	704.191	587.519
4 shard	445.122	486.041	394.099	630.219	360.007	208.308
8 shard	175.799	141.438	138.040	129.482	127.947	59.161

Ricerca frasi esatte con indice text (db 100x)

```
db.dimostratore_1x.runCommand( "text", {
    search: "\"beppe grillo\" \"pd e pdl\" \"il partito\" \"movimento 5\"
                \"destra e sinistra\" \"il parlamento\" "
    ,limit: 100
    ,filter: {  }
    ,language: "italian"
    ,project: {_id: 1 }
})
```

La tabella mostra i tempi rilevati in millisecondi

	1 frase	**2 frasi**	**3 frasi**	**4 frasi**	**5 frasi**	**6 frasi**
Server singolo	8.472	19.750	359.676	478.323	1.237.025	1.123.474
2 shard	13.161	32.887	115.986	101.755	727.436	724.902
4 shard	3.314	30.862	72.894	72.532	248.828	400.905
8 shard	1.173	15.052	30.519	22.606	108.111	148.320

Bibliografia e riferimenti

1. Codd, E.: A Relational Model of Data for Large Shared Data Banks. (1970)
2. Codd, E.: Is Your DBMS Really Relational? (1985)
3. Solid IT In: DB-Engines. (Accessed 2013) Available at: www.db-engines.com
4. Couchbase Inc. In: Couchbase.com. (Accessed 2011) Available at: http://www.couchbase.com/press-releases/couchbase-survey-shows-accelerated-adoption-nosql-2012
5. McKinsey Global Institute: Big data: The next frontier. (2011)
6. Cattell, R.: Scalable SQL and NoSQL Data Stores. (2010)
7. Brewer, E.: Towards robust distributed systems. (2000)
8. Gilbert, S., Lynch, N.: Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services. (2002)

9. Kumar, A., Rugg, K.: Brewer's Conjecture and a characterization of the limits, and relationships between Consistency, Availability and Partition Tolerance in a distributed service. (2011)

10. Thant, P., Nain, T.: Improving the Availability of NoSQL Databases for Cloud Storage., University of Computer Studies, Yangon

11. DataStax: Cassandra Consistency. (Accessed 2011) Available at: http://www.datastax.com/docs/0.8/dml/data_consistency

12. Yen, S.: NoSQL is a horseless carriage. (2009)

13. North, K.: Databases in the cloud. Dr. Drobb's Magazine (2009)

14. Chang, F., Dean, J., Ghemawat, S., Hsieh, W., Wallach, D., Burrows, M., Chandra, T., Fikes, A., Gruber, R.: Bigtable: A Distributed Storage System for Structured Data. Proceedings of the 7th symposium on Operating systems design and implementation, 205-218 (2006)

15. AGPL v.3. (Accessed 2007) Available at: https://www.gnu.org/licenses/agpl-3.0.html

16. Apache License v.2.0. (Accessed 2004) Available at: https://www.apache.org/licenses/LICENSE-2.0.txt

17. Porter, M.: Snowball Framework. (Accessed 2001) Available at: http://snowball.tartarus.org/

18. Laurent Bonnet, A.: Reduce, you say: what NoSQL can do for Data Aggregation and BI in Large Repositories. (2011)

19. Kristina, C., Michael, D.: MongoDB - The Definitive Guide. O'Reilly Media (2010)

20. Kyle, B.: MongoDB in Action. Manning Pubns Co (2011)

21. Kristina, C.: Scaling MongoDB. O'Reilly Media (2011)