ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA
SCUOLA DI SCIENZE

CORSO DI LAUREA IN SCIENZE E TECNOLOGIE INFORMATICHE

MONGODB TRA I DATABASE NON RELAZIONALI
FOCUS SU MAP-REDUCE E GESTIONE DELLA MEMORIA

Relazione finale in
Basi di Dati

Relatore
Prof. Dario Maio

Presentata da
Sara Guerreri

Sessione I
Anno Accademico 2014/2015

Indice

Indice delle figure

1. Introduzione

1.1. Movimento NoSQL

L'espressione NoSQL fa riferimento al linguaggio SQL, che è il più comune linguaggio di interrogazione dei dati nei database relazionali, qui preso a simbolo dell'intero paradigma relazionale[1]. Il movimento NoSQL non è contrario all'utilizzo di database relazionali. Il termine NoSQL è infatti acronimo di Not Only SQL, a significare che esistono diversi scenari applicativi per i quali il modello relazionale non è appropriato, ma tanti altri per i quali esso rappresenta ancora la soluzione migliore.

Negli ultimi anni il movimento NoSQL ha attirato un gran numero di imprese e aziende, che sono passate dai database relazionali a quelli non relazionali. Ciò è motivato dal fatto che è cambiato il trend d'uso dei dati (Fig. 1), che si è enormemente ampliato per diversi fattori, tra i quali il più rilevante è rappresentato dall'aumento esponenziale degli utenti della rete, dovuto alla crescente disponibilità di dispositivi con accesso a Internet, come smartphone, tablet e altri dispositivi portatili. I sistemi relazionali risultano inadeguati, poiché non rispondono all'esigenze di gestire ingenti quantità di dati semi-strutturati o affatto strutturati (e.g. text, log files, click streams, blogs, tweets, audio, video, etc.), che devono essere sempre disponibili, mentre i

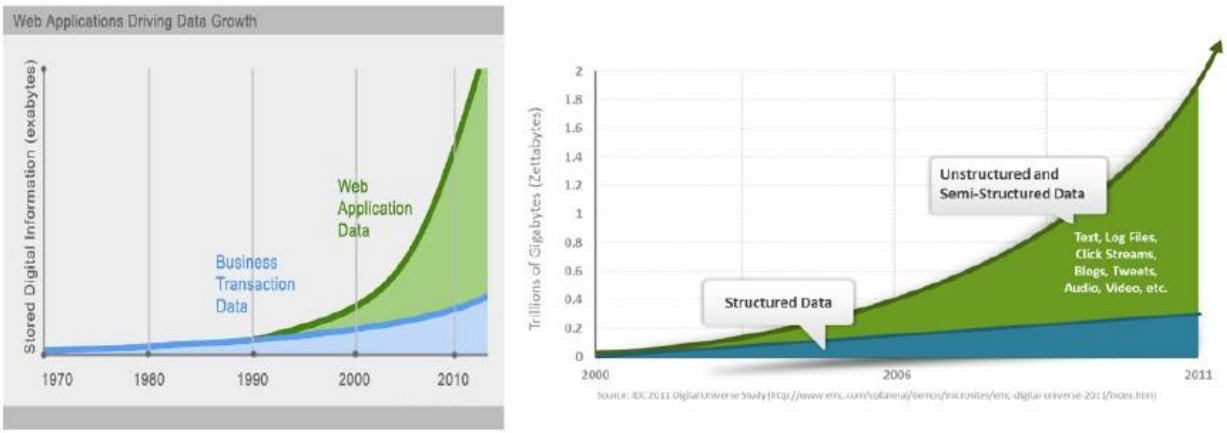

Fig. 1 Trend d'uso dei dati

sistemi non relazionali sono proprio una risposta a queste nuove esigenze.

La maggior parte dei database NoSQL ha adottato le idee di Bigtable di Google[3] e Dynamo di Amazon[4], che hanno fornito una serie di concetti che hanno ispirato molti degli attuali data store. I pionieri del movimento NoSQL sono grandi compagnie web o imprese che gestiscono importanti siti web, come Google, Facebook[5] e Amazon.

Per far fronte alla necessità di gestire quantità di dati sempre maggiori e a velocità sempre più elevate, i nuovi sistemi di memorizzazione si sono evoluti per essere più flessibili, facendo ricorso a modelli di dati meno complessi, per aumentare le prestazioni nella gestione e interrogazione dei dati. Le caratteristiche che accomunano i sistemi NoSQL possono essere riassunte con le seguenti parole chiave:

- ***horizontally scalable***: la scalabilità orizzontale, ossia l'abilità di distribuire i dati e il carico di lavoro su più server, così da poter gestire ingenti quantità di dati;
- ***distributed***: la flessibilità nel clustering e nella replicazione dei dati permette di distribuire su più nodi lo storage e il calcolo, in modo da realizzare potenti sistemi di fault-tolerance;
- ***non-relational***: la struttura di memorizzazione dei dati è differente dal modello relazionale, lo schema "rigido" dei database relazionali non permette di memorizzare dati fortemente dinamici, mentre i database NoSQL sono *schemaless* e consentono di memorizzare attributi, anche senza averli definiti a priori;
- ***open-source***: alla base del movimento NoSQL vi è la filosofia open-source, fondamentale per contribuire ad accrescere le potenzialità delle sue tecnologie;
- **BASE**: ***B****asically* ***A****vailable* ***S****oft state (services with)* ***E****ventual consistency*, ossia un modello di concorrenza delle transazioni non basato sulle proprietà ACID (*Atomicity, Consistency, Isolation, Durability*) della maggior parte dei database relazionali, poiché l'idea è che rinunciando alle proprietà ACID si possano ottenere prestazioni elevate.

1.2. Panoramica sui database NoSQL[2]

I database NoSQL vengono classificati in base al modello implementato per la memorizzazione dei dati, in particolare possono essere individuate quattro grandi famiglie:

1. Key-Value Pair database
2. Column Based database
3. Graph database
4. Document database

1.1.1 Key-Value Pair database

Il modello chiave-valore si basa su una API (*Application Programming Interface*) analoga a una mappa, dove il valore è un oggetto oscuro al sistema, cioè non è possibile fare *query* sui valori, ma solo sulle chiavi.

Alcune implementazioni del modello chiave-valore permettono tipi di valore più complessi, come tabelle *hash* o liste; altri forniscono mezzi per scorrere le chiavi. Nonostante questi database siano molto performanti per certe applicazioni, generalmente non sono d'aiuto se necessitano operazioni complesse di interrogazione e aggregazione. Sebbene sia possibile estendere una tale API per permettere transazioni che coinvolgono più di una chiave, ciò sarebbe controproducente in un ambiente distribuito, dove l'utilità di avere coppie chiave-valore non legate fra loro permette di scalare orizzontalmente in modo molto semplice.

Ci sono molte opzioni disponibili *open-source*, tra le più popolari ci sono Redis e Riak.

1.1.2 Column Based database

I database *column based* memorizzano i dati utilizzando una mappa multidimensionale, ordinata e sparsa. Ogni record può variare nel numero di colonne che sono memorizzate e le colonne possono essere nidificate. Le colonne possono essere raggruppate in famiglie di colonne o possono essere distribuite su più famiglie. I dati vengono recuperati tramite chiave primaria sulle famiglie di colonne, ciò consente, per *query* che coinvolgono pochi attributi selezionati su un gran numero di *record*, di ottenere tempi di risposta migliori, in quanto si riduce, anche di diversi ordini di grandezza, il numero di accessi necessari alle memorie di massa.

È inoltre possibile applicare efficienti tecniche di compressione dei dati, in quanto ogni colonna è costituita da un unico tipo di dati. Riducendo lo spazio occupato, aggiungere una colonna risulta poco costoso, inoltre ogni riga può avere un diverso insieme di colonne, permettendo alle tabelle di rimanere piccole senza preoccuparsi di gestire valori nulli.

Tra i più noti *column based* database citiamo HBase e Cassandra.

1.1.3 Graph database

I graph database, sono costituiti da nodi e relazioni tra nodi (archi) e l'accesso avviene attraverso algoritmi di ricerca su grafo; nodi e relazioni hanno proprietà per la memorizzazione dei dati.

La forza di questo tipo di database consiste nel gestire dati fortemente interconnessi, proprietà che permette l'operazione di attraversamento (*graph traversal*) che rispetto a una normale *query* su database chiave-valore, stabilisce come passare da un nodo all'altro utilizzando le relazioni tra nodi. Questi database sono tipicamente usati nei *social network*.

Da citare Neo4j, InfoGrid, HyperGrapgDB e OrientdB.

1.1.4 Document database

Nei *document database*, come ClusterPoint, CouchDB, MarkLogic e MongoDB, le informazioni sono organizzate in *document*, rappresentati in YAML, XML, JSON o BSON. L'accesso ai dati avviene tramite API che interrogano veri e propri dizionari.

I documenti forniscono un modo naturale e intuitivo per modellare i dati, che è in linea con la programmazione a oggetti. I documenti contengono uno o più campi, dove ogni campo contiene un valore tipizzato, come `string`, `date`, `binary` o `array`. Invece di sparpagliare i dati su più tabelle, ogni record e i dati ad esso associati sono memorizzati insieme nello stesso documento. Questo semplifica l'accesso ai dati e riduce o addirittura elimina la necessità di operazioni di *join* e complesse transazioni.

Nei database orientati ai documenti, la nozione di schema è dinamica: ogni documento può contenere campi differenti. Questa flessibilità è particolarmente utile per modellare dati non strutturati o molto diversi tra loro. Rende, inoltre, più semplice l'evoluzione di un'applicazione durante lo sviluppo, poiché è possibile aggiungere a un documento un numero arbitrario di campi di qualsiasi lunghezza. Inoltre questo tipo di database è in grado di fornire la stessa robustezza nel linguaggio di interrogazione che è tipica dei database relazionali. In particolare è possibile interrogare i dati basandosi su qualsiasi campo in un documento.

Questo tipo di database è adatto a molti scopi diversi, per via della flessibilità del modello dei dati e della naturale maniera di mappare i dati dei documenti come oggetti nei moderni linguaggi di programmazione.

2. MongoDB

2.1. Filosofia di progettazione

Il database MongoDB deve il suo nome alla parola "humongous", un fantasioso neologismo[6] della lingua americana, derivato dalle parole "huge" (enorme) e "monstrous" (mostruoso), che si riferisce ovviamente alla capacità del database di immagazzinare grandi moli di dati. MongoDB, infatti, è stato progettato con un approccio non relazionale, per essere orizzontalmente scalabile su più macchine, al fine di poter memorizzare ordini di grandezza di dati maggiori rispetto al passato (Fig. 2).

Scritto in C++ e sviluppato dalla compagnia MongoDB Inc., inizialmente chiamata 10gen, è stato progettato per essere potente, flessibile e scalabile, combinando le migliori caratteristiche dei sistemi basati su chiave-valore, ossia semplicità, velocità e scalabilità, e dei database relazionali, ossia il ricco modello di dati e il potente linguaggio di interrogazione.

MongoDB si concentra principalmente su quattro dei vari aspetti importanti in un DBMS: flessibilità, potenza, velocità e facilità d'uso

Fig. 2 Collocazione di MongoDB nel panorama dei database

2.2. Teorema CAP e MongoDB[7]

Verso la metà degli anni ’90, con l’avvento dei grandi sistemi basati su Internet, si iniziò a considerare la disponibilità di servizio da parte di un sistema, come un requisito importante, mentre fino a quel momento la cosa che più contava era la coerenza dei dati.

Eric A. Brewer, presentò nel 2000, al Symposium on Principles of Distributed Computing, il Teorema CAP. Tale teorema esprime la legge che regola il compromesso tra disponibilità e consistenza dei dati, nell’ambito della progettazione di sistemi largamente distribuiti e ad alta scalabilità. In sostanza, un sistema di dati condivisi è generalmente caratterizzato da tre proprietà:

- ***C****onsistency* (Coerenza), proprietà per cui, dopo una modifica, tutti i nodi del sistema riflettono tale modifica, Brewer ha però chiarito che: “*In ACID, the C means that a transaction preserves all the database rules, such as unique keys. In contrast, the C in CAP refers only to single-copy consistency, a strict subset of ACID consistency*”;
- ***A****vailability* (Disponibilità), proprietà per cui, il sistema garantisce che ogni richiesta riceva una risposta su ciò che sia riuscito o fallito;
- ***P****artition Tolerance* (Tolleranza al Partizionamento), proprietà per cui, il sistema continua a funzionare nonostante arbitrarie perdite di messaggi;

e sarà in grado di soddisfare, allo stesso tempo, al massimo due di queste.

Ciò porta a tre possibili combinazioni (Fig. 3):

- **CA** (*Consistency & Availability*) sono garantite in sistemi localizzati, che non risentono di partizionamento di rete, come un database centralizzato o un *file system*, dove la consistenza è garantita da transazioni *2PC (Two-Phase Commit)*;
- **CP** (*Consistency & Partition Tolerance*) sono garantite in sistemi distribuiti, con un elevato partizionamento di rete, in cui il valore della consistenza dei dati è di primaria importanza, a scapito della eventuale elevata latenza e bassa disponibilità dovuta a pessimistiche politiche di *locking*;
- **AP** (*Availability & Partition Tolerance*) sono garantite, ad esempio, nei DNS (*Domain Name System*), dove la disponibilità è il valore principale, per questo utilizzano protocolli più ottimistici e un approccio *best-effort*.

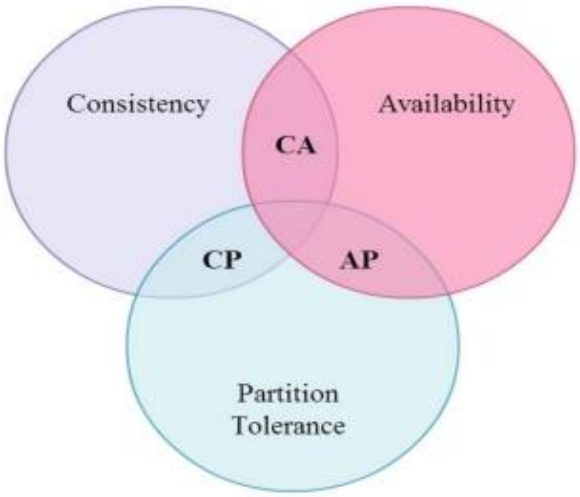

Fig. 3 Teorema CAP

Lo sforzo di garantire dati consistenti porta ad adottare approcci diversi per l'accesso e l'aggiornamento dei dati. Si passa dunque, dall'approccio transazionale ACID all'approccio BASE.

Una panoramica delle garanzie offerte dai sistemi attuali (Fig. 4)[8].

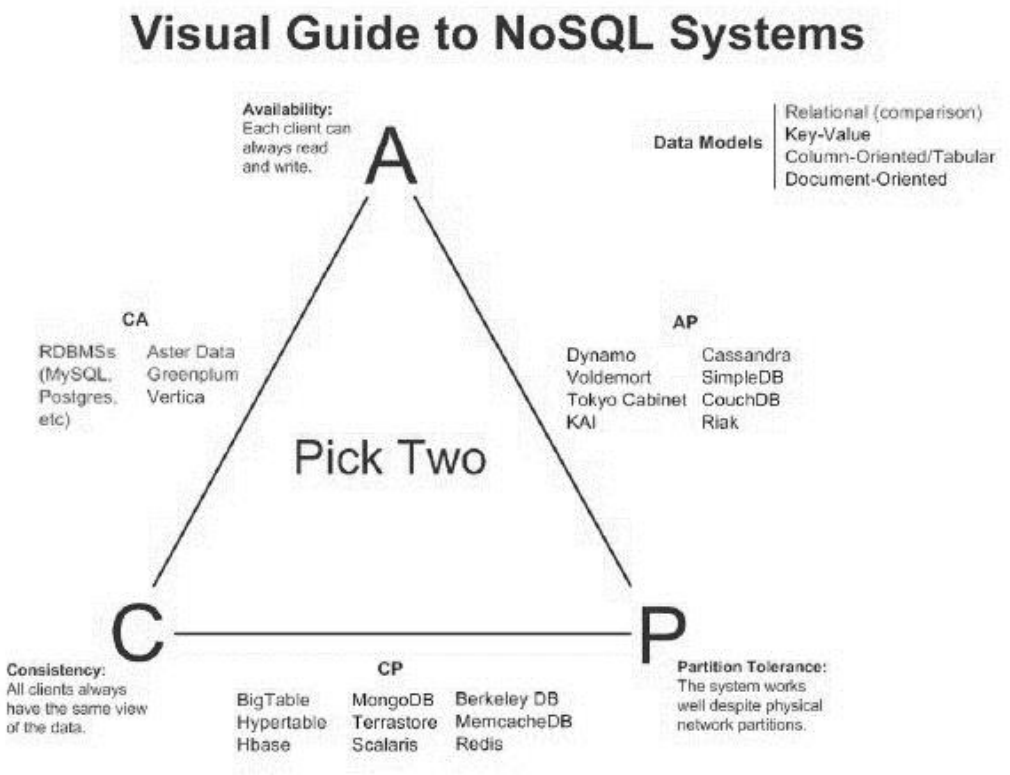

Fig. 4 Database e teorema CAP

Detto questo, MongoDB viene solitamente collocato tra i sistemi che garantiscono consistenza e tolleranza al partizionamento, però è piuttosto opportuno specificare che esistono due possibili scenari di applicazione del Teorema CAP a MongoDB[8], 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 al partizionamento**: anche nell'eventualità in cui si abbia una partizione nella rete, le interrogazioni non ritornano dati incorretti, gli *shard* continuano a lavorare in modo indipendente;
- **debole disponibilità dei dati**: letture e scritture su uno *shard* offline non sono disponibili 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 al partizionamento**: se un numero sufficiente di nodi non è più raggiungibile, viene, automaticamente eletto un nuovo server primario, garantendo sempre la possibilità di ricezione delle richieste;
- **debole disponibilità dei dati**: quando il server primario non è raggiungibile non è possibile accedere ad alcun dato.

Bisogna dunque notare come MongoDB sia un caso particolare di database NoSQL, dove non è possibile definire a priori il grado di fedeltà al Teorema CAP. Infatti, è possibile ottenere, attraverso opportune configurazioni e richieste da parte dei *client*, una maggior disponibilità dei dati a fronte di una minore consistenza degli stessi.

2.3. Principali Caratteristiche[10]

2.3.1 Modello dei Dati

Il modello dei dati di MongoDB si basa su documenti memorizzati in formato BSON (*Binary* JSON). La codifica BSON estende la più nota rappresentazione JSON (*Javascript Object Notation*) per includere tipi aggiuntivi come `int`, `long` e `floating point`. I documenti BSON contengono uno o più campi e ogni campo contiene un valore di uno specifico tipo di dato, compresi *array*, dati binari e sotto-documenti.

Documenti che tendono ad avere la stessa struttura sono organizzati in collezioni (*collections*). Le collezioni sono paragonabili alle tabelle di un sistema relazionale, dove i documenti sarebbero le righe e i campi le colonne.

I documenti di MongoDB tendono a raggruppare tutti i dati relativi a un record, mentre in un database relazionale le informazioni relative a un record sono sparse su più tabelle. Il risultato è dato da scalabilità e performance altamente migliori nel *commodity hardware* dal momento che con un'unica lettura si ottiene l'intero documento che contiene tutte le informazioni relative al dato cercato.

Inoltre, i documenti BSON di MongoDB si avvicinano maggiormente alla struttura degli oggetti nel linguaggio di programmazione. Questo rende più semplice e più veloce, per gli sviluppatori, la modellazione di come i dati nell'applicazione saranno associati ai dati archiviati nel database.

I documenti di MongoDB possono variare struttura. I campi possono cambiare da documento a documento, non vi è necessità di dichiarare a priori la struttura dei documenti al sistema, i documenti sono *self describing* ("autodescrittivi"). Se si vuole aggiungere un nuovo campo a un documento, il campo può essere creato senza interessare tutti gli altri documenti nel sistema, senza la necessità di aggiornare un catalogo del sistema centrale e senza spegnere il sistema.

Anche se MongoDB è *schemaless* e permette quindi flessibilità, la progettazione di uno schema rimane importante. È buona pratica considerare a priori alcuni aspetti quali, i tipi di interrogazioni che l'applicazione dovrà eseguire e come i documenti cambieranno nel corso del tempo.

2.3.2 Modello di Interrogazione

MongoDB fornisce *driver* nativi per tutti i linguaggi di programmazione più diffusi e *framework* per rendere agevole lo sviluppo. I *driver* supportati comprendono Java, .Net, Ruby, PHP, JavaScript, node.js, Python, Perl e altri ancora. I *driver* di MongoDB sono progettati per essere idiomatici per il linguaggio in questione.

Una differenza fondamentale con i database relazionali è che il modello di interrogazione di MongoDB è implementato per mezzo di metodi o funzioni all'interno dell'API di uno specifico linguaggio di programmazione, in opposizione a un linguaggio completamente separato come SQL. Questo, insieme all'affinità tra il modello JSON dei documenti di MongoDB e le strutture dati usate nella programmazione a oggetti, rende semplice l'integrazione con l'applicazione.

Diversamente da altri database NoSQL, MongoDB non si limita a semplici operazioni chiave-valore. È possibile sviluppare ricche applicazioni usando interrogazioni complesse e indici secondari che sbloccano il valore in dati strutturati, semi-strutturati e non strutturati.

Un elemento chiave di questa flessibilità è il supporto di MongoDB per molti tipi di interrogazioni. Una *query* può restituire un documento, un sottoinsieme di campi specifici del documento o complesse aggregazioni tra più documenti:

- **Key-Value query** restituiscono risultati basati su un qualsiasi campo del documento, spesso la chiave primaria;
- **Range query** restituiscono risultati basati su valori definiti come disuguaglianze (e.g. maggiore di, minore o uguale a, nell'intervallo);

- **Geospatial query** restituiscono risultati basati su criteri di prossimità, intersezione e inclusione come specificato da un punto, una linea, un cerchio o un poligono;
- **Text Search query** restituiscono risultati in ordine di rilevanza basati su argomenti di testo, usando operatori booleani (e.g. AND, OR, NOT);
- **Aggragation Framework query** restituiscono aggregazioni di valori restituiti dalla *query* (e.g. count, min, max, average) simili al GROUP BY di SQL;
- **MapReduce query** eseguono complesse operazioni di processo dei dati, che sono espresse in JavaScript ed eseguite sui dati contenuti in parti diverse del database.

2.3.3 Indicizzazione

Gli indici sono un meccanismo cruciale per ottimizzare le prestazioni e la scalabilità del sistema fornendo allo stesso tempo un accesso flessibile ai dati. Come nella maggior parte dei DBMS (*Database Management System*), mentre gli indici migliorano le prestazioni di alcune operazioni in termini di ordini di grandezza, provocano *overhead* nelle operazioni di scrittura, uso del disco e consumo di memoria. Per *default*, lo *storage engine* WiredTiger comprime gli indici nella RAM, liberando più *working set* per i documenti. MongoDB comprende il supporto per molti tipi di indici, che possono essere dichiarati su qualsiasi campo del documento, compresi i campi dentro agli *array*. Gli indici di cui dispone MongoDB sono:

- **Unique Index**: specificando un indice come univoco, MongoDB rifiuterà inserimenti di nuovi documenti o l'aggiornamento di un documento con un valore esistente per il campo su cui l'indice univoco è stato creato. Per default gli indici non sono dichiarati come univoci.
- **Compound Index**: possono essere utili per interrogazioni che specificano predicati multipli, inoltre qualsiasi campo principale all'interno del *compound index* può essere utilizzato, così saranno necessari meno indici su singoli campi.

- **Array Index**: per i campi che contengono un array, ogni valore dell'array è immagazzinato come un'entrata separata dell'indice. Non c'è una sintassi particolare per gli *array index*, se il campo contiene un *array*, sarà indicizzato come *array index*.
- **TTL Index**: gli indici TTL (*Time To Live*) permettono all'utente di specificare un periodo di tempo dopo il quale i dati saranno automaticamente eliminati dal database.
- **Geospatial Index**: permettono di ottimizzare le interrogazioni relative alla localizzazione in uno spazio bidimensionale. Questi indici consentono anche l'ottimizzazione delle interrogazioni su documenti che contengono punti o un poligono che sono più vicini a un dato punto o linea, o che si trovano dentro un cerchio o un poligono, o che intersecano un cerchio o un poligono.
- **Sparse Index**: contengono entrate solamente per i documenti che contengono il campo specificato. Poiché MongoDB permette flessibilità nella struttura dei documenti, è usuale che alcuni campi siano presenti solo in un sottoinsieme di documenti; gli *sparse index* consentono di avere indici più efficienti quando determinati campi non sono presenti in tutti i documenti.
- **Text Search Index**: MongoDB fornisce un indice specializzato per la ricerca testuale, che utilizza regole linguistiche specifiche per ricercare la forma base di una parola e riconoscere i simboli. Le interrogazioni che usano *text search index* restituiscono i documenti in ordine di rilevanza, inoltre più campi possono essere inclusi nel *text index*.

2.3.4 Scalabilità

MongoDB fornisce scalabilità orizzontale a basso costo, su commodity hardware o infrastrutture *cloud*, grazie al meccanismo di *sharding*, che è trasparente alle applicazioni. Lo *sharding* permette di distribuire i dati su più partizioni fisiche chiamate *shards* e permette ai dispiegamenti di MongoDB di far fronte alle limitazioni hardware di un singolo server, come colli di bottiglia

nella RAM o le operazioni di I/O su disco, senza aggiungere complessità all'applicazione.

Diversamente dai database relazionali, il meccanismo di *sharding* di MongoDB è automatico e *built in*. Gli sviluppatori non affrontano la complessità di costruire una logica di *sharding* nel loro codice dell'applicazione, che poi necessita di essere aggiornato quando gli *shards* vengono migrati. Inoltre sono disponibili diverse politiche di sharding per consentire a sviluppatori e amministratori di distribuire i dati su cluster secondo modelli di interrogazione o localizzazione dei dati. Di conseguenza MongoDB consente un'alta scalabilità su diversi carichi di lavoro:

- **Range-based Sharding**: i documenti sono partizionati sugli *shard* sulla base del valore della *shard key*, documenti con valore di *shard key* vicini tra loro saranno probabilmente collocati nello stesso *shard*; questo approccio è adatto per applicazioni che hanno bisogno di ottimizzare le interrogazioni basate su intervalli;
- **Hash-based Sharding**: i documenti sono uniformemente distribuiti sulla base del valore della *shard key* restituito da una funzione *hash*, diversamente dal caso precedente, in documenti con valori di *shard key* vicini tra loro difficilmente saranno collocati nello stesso *shard*; questo approccio garantisce una distribuzione uniforme delle scritture sugli *shard*;
- **Location-aware sharding**: i documenti sono partizionati secondo una configurazione specificata dall'utente, che associa intervalli di valori di *shard key* a specifici *shard*. Gli utenti possono continuamente raffinare la posizione fisica dei documenti, per esigenza delle applicazioni, come localizzare i dati in specifici *data center*.

2.3.5 Replicazione

MongoDB mantiene copie multiple dei dati, chiamate *replica set*, usando la nativa capacità di replicazione. Un *replica set* è uno *shard* completamente auto-riparante, che aiuta a prevenire il tempo di inattività del database. Il *failover* di una replica è completamente automatico, eliminando la necessità degli amministratori di intervenire manualmente.

Un *replica set* è composto da più repliche. In ogni momento un solo membro agisce come membro primario del *replica set* e gli altri membri agiscono come secondari (Fig. 5). MongoDB è fortemente consistente per default: letture e scritture avvengono su una copia primaria dei dati. Se il membro primario fallisce per qualsiasi motivo (e.g. *hardware failure*, *network partition*) uno dei membri secondari è automaticamente eletto a primario e comincia a gestire tutte le scritture.

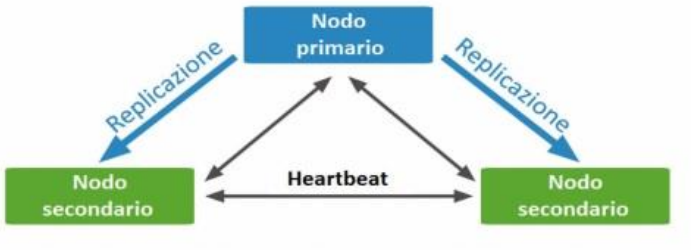

Fig. 5 Membri di un replica set

Le applicazioni possono scegliere di leggere dalle repliche secondarie, dove i dati sono alla fine consistenti per default. Le letture dai membri secondari possono essere utili in scenari in cui è accettabile che i dati siano leggermente datati. Le applicazioni possono leggere anche dalla replica più vicina, misurando la distanza di *ping* quando la latenza geografica è più importante della consistenza.

I *replica set* riducono l'*overhead* operazionale e aumentano la disponibilità del sistema. Se la replica primaria fallisce, le repliche secondarie decidono insieme quale replica deve diventare

la nuova replica primaria, con un processo chiamato “elezione” (Fig. 6). Una volta che la nuova replica primaria è stata determinata, le restanti repliche secondarie vengono configurate per ricevere aggiornamenti dalla nuova primaria. Se la replica primaria originale torna online, riconoscerà che non è più la primaria e configurerà sé stessa come secondaria.

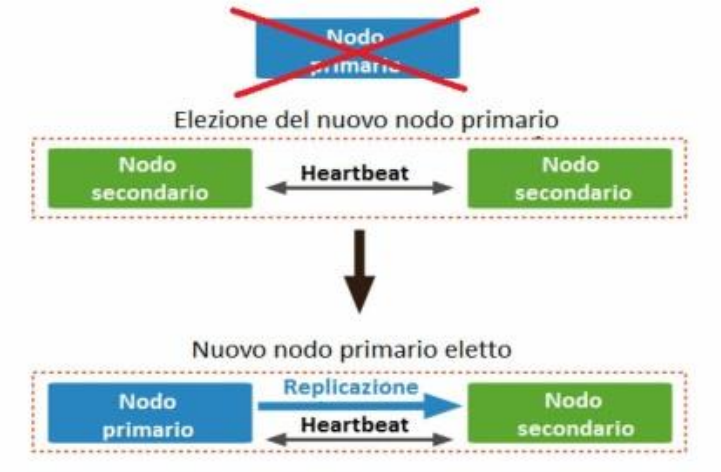

Fig. 6 Elezione di un nuovo nodo primario

3. Algoritmo Map-Reduce

3.1. Aggregazione[11]

Le aggregazioni sono le operazioni che processano i *record* di dati e restituiscono un risultato. MongoDb fornisce una vasta gamma di operazioni di aggregazione. Eseguire aggregazioni sui dati sull'istanza `mongod`, semplifica il codice dell'applicazione e limita la necessità di risorse.

Come le *query*, le operazioni di aggregazione in MongoDB prendono collezioni di documenti come input e restituiscono un risultato sotto forma di uno o più documenti.

Le modalità di aggregazione comprendono:

- Aggregation pipelines;
- Operazioni di aggregazione single purpose;
- Map-Reduce.

MongoDB 2.2 introdusse un nuovo *aggregation framework*, modellato sul concetto di *data processing pipeline*. I documenti vengono introdotti in una *multi-stage pipeline* che li trasforma in risultati aggregati. Le fasi più basilari di una *pipeline* forniscono filtri che operano come *query* e trasformazioni dei documenti che modificano la forma del documento di output. Altre operazioni di *pipeline* forniscono strumenti per raggruppare e ordinare i documenti per campi specificati, come anche strumenti per aggregare il contenuto di *array*, compresi array di documenti. Inoltre, nelle fasi di *pipeline* è possibile usare operatori che svolgono compiti quali, il calcolo della media o la concatenazione stringhe. La *pipeline* fornisce un'efficiente aggregazione dei dati, utilizzando operazioni native di MongoDB, ed è il metodo preferito per l'aggregazione dei dati in MongoDB.

Per un certo numero di operazioni di aggregazione comuni, MongoDB fornisce appositi comandi. Queste operazioni di

aggregazione comprendono: restituire il conteggio di documenti corrispondenti a una data selezione, la restituzione dei valori di un certo campo e raggruppamento dei dati basato sui valori di un determinato campo. Tutte queste operazioni aggregano documenti di una sola collezione. Se da un lato queste operazioni forniscono un accesso semplice a comuni processi di aggregazione, dall'altro mancano della flessibilità e della potenza proprie delle *pipeline* e di Map-Reduce.

3.2. Dettaglio Map-Reduce[12]

Map-Reduce (Fig. 7) è un paradigma di processo dei dati ideato per ottenere risultati di aggregazione su grandi volumi di dati. Per le operazioni di Map-Reduce MongoDB fornisce il comando `mapReduce`.

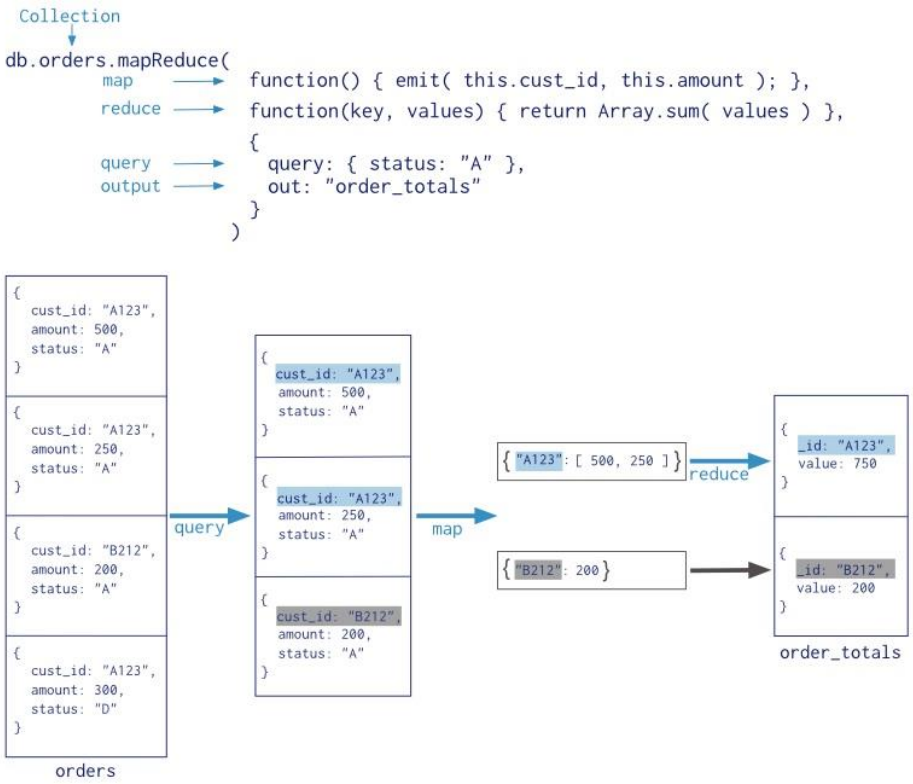

Fig. 7 Esempio di Map-Reduce

Considerando l'esempio in figura, possiamo osservare che MongoDB applica la fase *map* a ogni documento in input (i.e. i documenti che della collezione che soddisfano le condizioni della

query). La funzione `map` restituisce coppie chiave-valore. Per quelle chiavi che hanno valori multipli, MongoDB applica la fase *reduce*, che raccoglie e condensa i dati aggregati (Fig. 8). MongoDB poi archivia i dati in una collezione. Opzionalmente, l'output della funzione `reduce`, può passare alla funzione `finalize` per condensare o processare ulteriormente i dati di aggregazione.

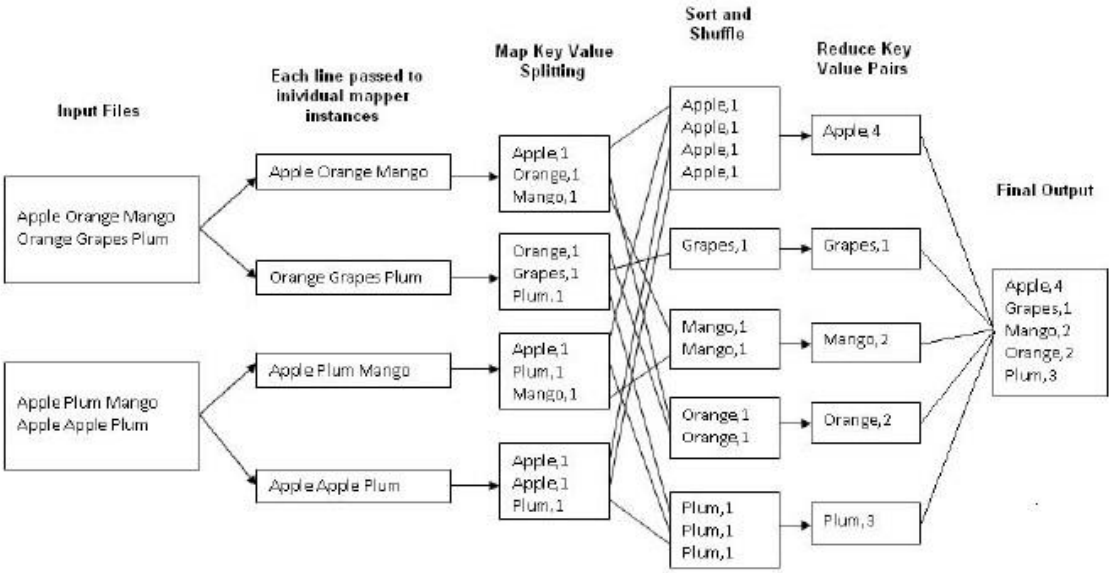

Fig. 8 Come funziona Map-Reduce

Tutte le funzioni Map-Reduce in MongoDB sono scritte in Javascript e sono eseguite all'interno del processo `mongod`. Le operazioni di Map-Reduce prendono i documenti di una singola collezione come input e possono eseguire qualsiasi ordinamento o restringimento prima di cominciare la fase di `map`. `mapReduce` può restituire il risultato di un'aggregazione sotto forma di documento oppure scrivere il risultato alle collezioni. Le collezioni in input e in output possono essere frammentate. Per la maggior parte delle operazioni di aggregazione la *pipeline* fornisce prestazioni migliori e un'interfaccia più coerente, però le operazioni Map-Reduce forniscono una flessibilità che non è attualmente disponibile nella *pipeline*.

In MongoDB, l'algoritmo Map-Reduce usa funzioni Javascript personalizzate per mappare o associare valori a una chiave. Se una chiave ha valori multipli, l'operazione *reduce* riduce tali valori in un singolo oggetto associato alla chiave.

L'utilizzo di funzioni Javascript personalizzate fornisce flessibilità per le operazioni di Map-Reduce. Per esempio, quando la funzione *map* processa un documento, può creare più di una mappatura chiave-valore o nessuna mappatura. Inoltre Map-Reduce può utilizzare le funzioni personalizzate di Javascript per apportare ulteriori modifiche al risultato di un'operazione di Map-Reduce al termine dell'algoritmo, come effettuare ulteriori calcoli.

In MongoDB, Map-Reduce può scrivere i risultati a una collezione o restituirli *inline*. Se il risultato di Map-Reduce viene inviato a una collezione, è possibile effettuare successive operazioni di Map-Reduce sulle stessa collezione in input, che fonde, rimpiazza, fonde o riduce i nuovi risultati con quelli precedenti.

La potenza di Map-Reduce sta nel fatto che è in grado di operare su collezioni frammentate e che quindi risiedono su diversi *shard*.

4. Gestione della memoria

4.1. Memory Mapping

La gestione della memoria di MongoDB è orientata alla massima semplificazione e sfrutta il concetto di file mappato in memoria in modo persistente, per il quale ai segmenti della memoria virtuale viene assegnata una correlazione byte a byte con una porzione del file in cui risiedono i dati (Fig. 9).

La memoria virtuale è una rappresentazione, a livello di processo, della memoria fisica, per permettere di astrarre l'indirizzamento su di essa.

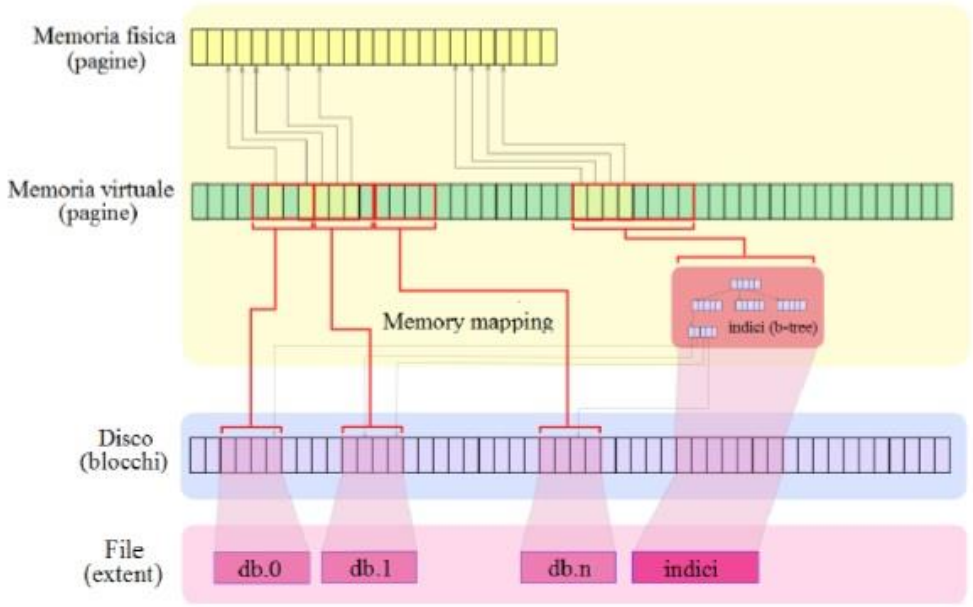

Fig. 9 Memory mapping

Se osserviamo la memoria 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 compiuto in maniera 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. Quando si conclude questa operazione si possono presentare due casi:

1. se la memoria fisica non è interamente occupata, la nuova pagina viene semplicemente carica nella RAM;
2. 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 di 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 fosse superiore al quantitativo di RAM disponibile. Infatti all'avvio dell'istanza di MongoDB, i data file vengono mappati in memoria attraverso la *system call* `mmap()`. Una volta caricato 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 apportate al database, che è a sua volta suddiviso in *extent*, ovvero porzioni fisiche di dati di dimensione preallocata, operazione che è eseguita in piena autonomia da quest'ultimo a intervalli regolari.

E' grazie al meccanismo dei file mappati in memoria che MongoDB non necessita di un uno strato di *caching* separato per il *caching* a livello di applicazione, perché il database già mette a disposizione la versione manipolabile dei dati, rendendoli utilizzabili sia nell'archivio persistente che nella cache dell'applicazione. Tutto ciò è molto diverso dai database relazionali, dove fare il *caching* dei dati è molto più costoso. I database relazionali devono trasformare i dati in rappresentazioni di oggetti, che le applicazioni possono leggere e devono poi immagazzinare in una *cache* separata: se questa trasformazione da dati a oggetti per le applicazioni richiede operazioni di *join*, il processo aumenta l'*overhead* relativo all'utilizzo del database aumentando così l'importanza di avere una *cache* dedicata.

4.2. Storage Engine[13]

MongoDB supporta due *storage engine*: **MMAPV1** (*Memory Mapped Version 1*) ***engine***, una versione migliorata dell'*engine* usato nelle precedenti *release* di MongoDB, e il nuovo **WiredTiger *storage engine***, che fornisce maggiore concorrenza e compressione. Entrambi gli *engine* possono coesistere all'interno dello stesso *replica set*, semplificando la valutazione e la migrazione dei dati tra loro. Il nuovo MMAPV1 implementa controllo della concorrenza a livello delle collezioni, mentre WiredTiger migliora ulteriormente le prestazioni per molti carichi di lavoro implementando controllo della concorrenza a livello dei documenti. MongoDB e la sua community stanno sviluppando altri *engine*, tre cui RocksDB *Key-Value engine*, HDFS *storage engine* e FusionIO *engine* che aggira il *filesystem*. Questi e altri *engine* potrebbero essere supportati in futuro, su richiesta dei clienti.

I file mappati in memoria sono il cuore di MMAPV1 in MongoDB, poiché utilizzando i file mappati MongoDB può trattare i dati come se fossero in memoria, questo fornisce metodi estremamente semplici e veloci per accede ai dati e manipolarli.

WiredTiger è un nuovo *engine* per MongoDB, esso scala su moderne architetture multi-CPU. Utilizzando una varietà di tecniche di programmazione diverse, come *hazard pointers* algoritmi *lock-free* e passaggio di messaggi, WiredTiger effettua più lavoro per CPU core di altri *storage engine*. Per minimizzare l'*overhead* su disco e I/O, WiredTIger usa formati compatti di file e opzionalmente la compressione. Per molte applicazioni WiredTiger fornisce miglioramenti significativi per quanto riguarda i costi di archiviazione, utilizzo dell'*hardware* e prevedibilità delle prestazioni.

5. Conclusioni

MongoDB 3.0 ad oggi è la *release* più significativa del *database* che sta crescendo più velocemente di ogni altro al mondo. Quest'ultima *release* espande significativamente la varietà di applicazioni che trovano in MongoDB la soluzione più adatta alle loro esigenze, poiché MongodB ha introdotto una nuova architettura di archiviazione altamente flessibile, incorporando la tecnologia di WiredTiger acquisita nel 2014. I miglioramenti delle prestazioni e della scalabilità inclusi in questa release, ora piazzano MongoDB in prima linea nel mercato dei *database*, eliminando il bisogno di alternative di nicchia e rendendo MongoDB il DBMS standard per le moderne applicazioni.

Bibliografia

[1] “http://it.wikipedia.org/wiki/Modello_relazionale”

[2] Top 5 Considerations When Evaluating NoSQL Databases - A MongoDB White Paper February 2015

[3] “http://www.google.com/”

[4] “http://www.amazon.com/”

[5] “https://www.facebook.com/”

[6] “http://dictionary.reference.com/browse/humongous?s=t”

[7] “https://stefanopedone.wordpress.com/2008/06/16/acid-vs base-il-teorema-di-cap/”

[8] “http://jandiandme.blogspot.de/2013/06/mongodb-and-cap-theorem.html”

[9] “http://blog.nahurst.com/visual-guide-to-nosql-systems”

[10] MongoDB Architecture Guide MongoDB 3.0 - A MongoDB White Paper 2015

[11] “http://docs.mongodb.org/manual/core/aggregation-introduction/”

[12] “http://docs.mongodb.org/manual/core/map-reduce/”

[13] What’s New in MongoDB 3.0 - A MongoDB White Paper February 2015