DB NoSQL – Analisi prestazionale

1 I database NoSQL 2

1.1 Perché NoSQL? Il teorema di CAP e il No-SQL data model 2

1.2 Un confronto tra le famiglie di DB NoSQL 5

1.3 I database document-oriented e graph-oriented 7

2 Breve introduzione a MongoDB 8

2.1 Principali caratteristiche di MongoDB 9

2.2 Comparazione tra MongoDB e MySQL 10

2.3 Ambiti di utilizzo e realizzazione dei direct graph con MongoDB 13

2.4 Soluzioni ibride: MongoDB + database a grafo + RDF triple store 15

2.5 Analisi delle prestazioni di MongoDB all'aumentare dei documenti memorizzati 15

3 Confronto tra le performance di MongoDB e di altri database NoSQL 18

3.1 Confronto MongoDB con vari database NoSQL 18

3.2 Confronto HBase e MongoDB 20

4 Analisi di performance Graph Database 22

4.1 Elenco e caratteristiche di varie implementazioni di Graph Database 22

4.2 Analisi e confronto delle performance di varie implementazioni di Graph Database 24

4.3 Scalabilità 25

1 I database NoSQL

NO-SQL è un movimento che negli ultimi anni si è molto affermato, producendo dei risultati soddisfacenti con la creazione di progetti e iniziative utilizzate anche su larga scala. Tale movimento vuole "rompere" la storica linea dei database relazionali e definire delle nuove linee guida per l'implementazione di database che non utilizzano il linguaggio di interrogazione SQL e non siano strettamente legati ad una definizione "rigida" dello schema dati.

La **filosofia del NO-SQL** è descritta molto bene sul sito di **Carlo Strozzi** e si può riassumere nei seguenti punti, partendo dalla domanda "*Perchè avere altri DBMS se esistono quelli relazionali?*":

1. I database relazionali sono troppo costosi e spesso, quelli che svolgono bene il loro lavoro, sono commerciali. NO-SQL abbraccia totalmente la filosofia open-source;
2. NO-SQL è semplice da usare e non occorre uno specialista di DBMS. Il paradigma di programmazione è, infatti, ad oggetti;
3. I dati sono altamente portabili su sistemi differenti, da Macintosh a DOS;
4. Non definisce uno schema "rigido" (*schemaless*) e non occorre tipare i campi, per cui non esistono limiti o restrizioni ai dati memorizzati nei database NO-SQL
5. Velocità di esecuzione, di interrogazione di grosse quantità di dati e possibilità di distribuirli su più sistemi eterogenei (replicazione dei dati), con un meccanismo totalmente trasparente all'utilizzatore;
6. I DBMS NO-SQL si focalizzano su una scalabilità orizzontale e non verticale come quelli relazionali.

Dall'altro lato, NO-SQL non garantisce i requisiti **ACID** su cui si basano i sistemi relazionali, per cui si è ancora particolarmente scettici sull'utilizzo di questi nuovi DBMS.

1.1 Perché NoSQL? Il teorema di CAP e il No-SQL data model

Volendo continuare la trattazione della **"filosofia" NO-SQL** e sulle prerogative per cui è necessaria, si estrapolano qui le motivazioni e le caratteristiche lette negli articoli riportati su BigDataNERD:

- WHY NOSQL- PART 1 – CAP THEOREM
- WHY NOSQL- PART 2 – OVERVIEW OF DATA MODEL,RELATIONAL & NOSQL

Perché NO-SQL?

Per poter rispondere alla domanda occorre esaminare il "teorema di CAP" e ricordare che un concetto fondamentale degli RDBMS è quello delle "ACID transactions" (**A**=Atomicity, **C**=Consistency, **I**=Isolation, **D**=Durability).

- **A come Atomicità**. La transazione è indivisibile, ossia o tutti gli *statements* che la compongono vengono applicati ad un database o nessuno.
- **C come Consistenza.** Il database deve rimanere in uno stato consistente prima e dopo l'esecuzione di una transazione, quindi non devono verificarsi contraddizioni (*inconsistency*) tra i dati archiviati nel DB.
- **I come Isolamento**. Quando transazioni multiple vengono eseguite da uno o più utenti simultaneamente, una transazione non vede gli effetti delle altre transazioni concorrenti.
- **D come Durabilità.** Quando una transazione è stata completata (con un *commit*), i suoi cambiamenti diventano persistenti, ossia non vengono più persi.

I più comuni e diffusi *RDBMS* supportano le *transazioni ACID* e i principi su esposti sembrerebbero bastare. Ma dov'è il problema?

Nell'era del **Web 2.0**, le applicazioni devono lavorare su bilioni e trilioni di dati ogni giorno e la **scalabilità** è un concetto che in tale ambito ricopre un ruolo importantissimo. Per questo motivo i database hanno bisogno di essere distribuiti sulla rete per realizzare una *scalabilità orizzontale.* Esaminiamo qui il concetto di "CAP theorem" per valutare le caratteristiche di un siffatto sistema di storage distribuito dei dati.

Il teorema di CAP fu teorizzato da **Eric Brewer** nel 2000 (**C**=Consistency, **A**=Availability, **P**=Partition-Tolerance). Per maggiori informazioni si veda il seguente articolo: ACID vs. BASE il Teorema di CAP (di Stefano Pedone).

Nell'ambito di un sistema di storage distribuito, i 3 principi hanno la seguente definizione:

- **Consistency**: se viene scritto un dato in un nodo e viene letto da un altro nodo in un sistema distribuito, il sistema ritornerà l'ultimo valore scritto (quello *consistente*).

- **Availability**: Ogni nodo di un sistema distribuito deve sempre rispondere ad una query a meno che non sia indisponibile.
- **Partition-Tolerance:** è la capacità di un sistema di essere tollerante ad una aggiunta o una rimozione di un nodo nel sistema distribuito (*partizionamento*) o alla perdita di messaggi sulla rete.

Secondo la **teoria CAP**, è impossibile garantire tutti e tre i principi del teorema di CAP. Infatti, si consideri un sistema distribuito e si supponga di aggiornare un dato su un *nodo 1* e di leggerlo da un *nodo 2*, si verificheranno le seguenti conseguenze:

1. Il *nodo 2* deve ritornare l'ultima "miglior" versione del dato (quella consistente) per non violare il principio della *Consistenza*
2. Si potrebbe attendere la propagazione del dato modificato nel *nodo 2* e, quest'ultimo, potrebbe mettersi in attesa della versione più aggiornata, ma, in un sistema distribuito, si ha un'alta possibilità di perdita del messaggio di aggiornamento e il *nodo 2* potrebbe attenderlo a lungo. Così non risponderebbe alle query (*indisponibilità*), violando il principio dell'*Availability*
3. Se volessimo garantire sia la *Consistency* che l'*Availability*, non dovremmo partizionare la rete, violando il principio del *Partition-Tolerance*

Le web applications progettate in ottica **Web 2.0**, caratterizzate da architetture altamente scalabili e profondamente distribuite con politiche di prossimità geografica, puntano in maniera forte sulla garanzia di alte prestazioni ed estrema disponibilità dei dati. Si trovano, quindi, ad essere altamente scalabili e partizionate. La **filosofia NO-SQL** è sbilanciata verso la ridondanza dei nodi e la scalabilità.

Di seguito, si riporta una tabella dei maggiori **database NO-SQL** del momento:

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

Il *NoSQL data model* può essere implementato seguendo differenti approcci, a seconda delle strutture dati con cui si rappresentano i record di dato.

- **Famiglie di colonne o "wide column store"** (come Cassandra e HBase): le informazioni sono memorizzate in *colonne,* in coppie chiave/valore. Tipicamente sono usati nell'ambito della memorizzazione distribuita dei dati;
- **Document Store** (come CouchDB, MongoDB): le informazioni sono organizzate in "*Document*", rappresentati in XML, JSON o BSON, e l'accesso ai dati avviene attraverso API che interrogano veri e propri dizionari. Sono molto usati nello sviluppo di attuali web applications;

- **Key/Value Store** (come Membase, Redis): la struttura dati di memorizzazione è una *Hash Table* (coppia chiave-valore). Indicate per sistemi di caching o *Hash Table* distribuite;
- **Graph Data Store** (come Neo4J, InfoGrid, OrientDB). Il modello di programmazione è il *grafo* e l'accesso ad archi e nodi avviene attraverso algoritmi di ricerca su grafo. Tipicamente si usano nei *social network*.

1.2 Un confronto tra le famiglie di DB NoSQL

Ecco le keywork che riassumono le peculiarità dei **DB NoSQL**:

- **non-relational**: le struttura di memorizzazione dei dati è differente dal modello relazionale. L'approccio a "schema rigido" dei db relazionali non permette di memorizzare dati fortemente dinamici. I db NoSQL sono "*schemaless*" e consentono di memorizzare "on the fly" attributi, anche senza averli definiti a priori
- **distributed:** la flessibilità nella clusterizzazione e nella replicazione dei dati permette di distribuire su più nodi lo storage (e il calcolo), in modo da realizzare potenti sistemi *fault-tolerance*
- **open-source:** alla base del movimento NoSQL vi è la filosofia "open-source", fondamentale per contribuire ad accrescere le potenzialità delle sue tecnologie
- **horizontally scalable:** architetture enormemente scalabili, che consentono di memorizzare e gestire una grande quantità di informazioni

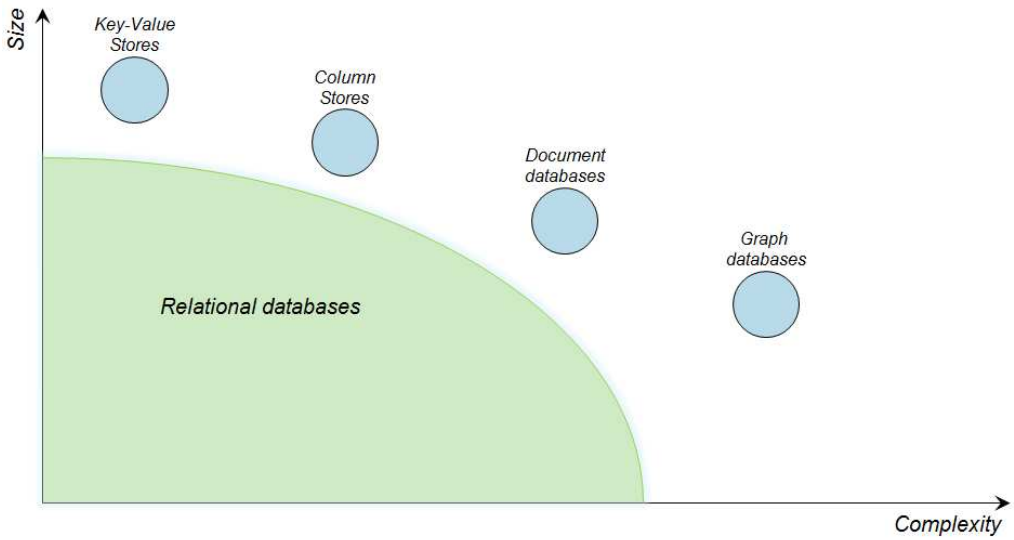
Al seguente link, si trova una lista aggiornata dei database NoSQL che rispettano i requisiti su citati: http://nosql-databases.org

Un whitepaper sulla tecnologia dei DB NoSQL è presente sul sito di CouchBase (un altro DB NoSql *document-oriented*): NoSQL Database Technology

E ancora, "**10 cosa da sapere sui DB NoSQL**": 10 things you should know about NoSQL databases

Una lista più ricca delle famiglie di DB NoSQL è la seguente:

- Wide Column Store / Column Families
- Document Store
- Key Value / Tuple Store
- Graph Databases
- Multimodel Databases
- Object Databases
- Grid & Cloud Database Solutions
- XML Databases
- Multidimensional Databases
- Multivalue Database



Dal precedente grafico, si possono fare delle considerazioni. Un parametro di valutazione è l'**operational complexity:** la struttura di memorizzazione dei DB a grafo è particolarmente complessa e anche gli algoritmi che ne permettono la navigazione (*traversal*) sono i classici algoritmi di routing, di una certa complessità computazione. Ovvio che all'aumentare della complessità nella struttura dati, diminuisce la capacità di memorizzazione dei dati stessi (parametro **size**). La famiglia dei **DB a grafo (graph databases),** come NEO4J e OrientDB, dispone di routine di accesso ai dati particolarmente performanti, ma non possono trattare troppi dati – [per una introduzione ben fatta sui **database a grafo**, ecco un buon riferimento: Graph Databases An Overview].

Al contrario, la famiglia dei **DB a chiave-valore (key-value databases),** come Redis**,** è particolarmente consigliata per memorizzare e gestire grosse quantità di dati. La famiglia dei DB a documenti (**document databases**), come MongoDB, si trovano in una posizione intermedia.

Un sondaggio aperto agli utilizzatori (previa registrazione) sui vari DB (sia relazionali che non) e che permette di comparare le varie caratteristiche, è presente a questo link (si possono inserire più colonne per confrontare più tecnologie tra loro): http://vschart.com/compare/mongodb/vs/redis-database

Una tesi di laurea in cui poter trovare un benchmark sui DB NoSQL e ho trovato quella ben fatta su Analisi delle performance dei database non relazionali: il caso di studio di MongoDB dell'Università di Padova, dove si compara **MongoDB** col db relazionale più diffuso, **MySQL**.

Altro articolo di comparazione tra i principali DB delle famiglie elencate:

Cassandra vs **MongoDB** vs **CouchDB** vs **Redis** vs **Riak** vs **HBase** vs **Couchbase** vs **Neo4j** vs **Hypertable** vs **ElasticSearch** vs **Accumulo** vs **VoltDB** vs **Scalaris** comparison

E ancora, un confronto tra alcuni DB NoSQL, dove Cassandra e HBase vincono sul throughput: A vendor-independent comparison of NoSQL databases: Cassandra, HBase, MongoDB, Riak.

1.3 I database document-oriented e graph-oriented

Se l'esigenza di progetto è quella di rappresentare un grafo di entità con relative relazioni, memorizzato in modo persistente su un database che possa garantire scalabilità e alte performance nell'accesso ai dati, definiamo le due famiglie di DB NoSQL che possono fare al caso nostro: i database *document-oriented* e *graph-oriented*.

I **database orientati ai documenti** sono simili a delle tabelle hash, con un unico campo di identificazione e valori che possono essere di qualsiasi tipo. I documenti possono contenere strutture nidificate, come ad esempio, documenti, liste o liste di documenti. A differenza dei database basati su chiave-valore, questi sistemi supportano indici secondari, replicazione e interrogazioni ad hoc. Ma come gli altri database NoSQL non sono supportate le proprietà ACID per le transazioni. Tra questi, il più famoso è **MongoDB.**

Un **database a grafo** è costituito da nodi e relazioni tra nodi, questo tipo di database può essere visto come un caso particolare di un database orientato ai documenti in cui i documenti rappresentano sia i nodi che che le relazioni che interconnettono i nodi. Pertanto nodi e relazioni hanno proprietà per la memorizzazione dei dati. La forza di questo tipo di database è di gestire dati fortemente interconnessi, proprietà che permette un'operazione molto interessante: l'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. Tra i più diffusi db a grafo, si annoverano **Neo4J** e **InfiniteGraph**.

2 Breve introduzione a MongoDB

MongoDB (il cui nome deriva da *"humongous*") è un database NOSQL open-source, scalabile e altamente performante.

Ecco il link al codice sorgente del progetto: http://www.mongodb.org/display/DOCS/Source+Code

Scritto in C++, ecco le sua caratteristiche distintive:

- **Document-oriented storage***:* i dati vengono archiviati sotto forma di *document* in stile **JSON** con schema dinamici, secondo una struttura molto semplice e potente;
- **Full Index Support:** indicizzazione di qualsiasi attributo
- **Replication & High Availability:** facilità nella replicazione dei dati attraverso la rete e alta scalabilità;
- **Auto-Sharding:** scalabilità orizzontale senza compromettere nessuna funzionalità;
- **Query** *document-based*
- **Fast In-Place Updates:** modifiche atomiche *in-place* con performance elevate
- **Map/Reduce:** aggregazione flessibile e data processing
- **GridFS:** memorizzazione di file di qualsiasi dimensione senza appesantimenti dello stack
- **Commercial Support:** Enterprise support, corsi di apprendimento e consulenze online.

Concetti e definizioni

- **Collection** (Tabella) – **Document** (Tupla o Oggetto) – **Proprietà** (attributo-colonna tabella)
- Una relazione **1-to-Many** si mappa inserendo un Document in un altro Document.
- Una relazione **Many-to-Many** si mappa inserendo programmaticamente una join tra gli elementi di una collection.

Perchè MongoDB?

- Innanzitutto, perchè è *document-oriented* ed è *schemaless* (dinamicamente tipato per una facile evoluzione dello schema). MongoDB memorizza dati in documenti JSON, che facilita la mappatura dei tipi nativi dei linguaggi di programmazione ed è *schemaless*, con semplicità nell'evoluzione del data model;
- Alte performance: non ci sono join che possono rallentare le operazioni di lettura o scrittura; l'indicizzazione include gli indici di chiave sugli *embedded documents* e gli array;
- Alta disponibilità: alto meccanismo di replicazione su server
- Alta scalabilità: *automatic sharding* – autopartizionamento di dati attraverso i servers, con operazioni di lettura e scrittura distribuite sugli *shards*
- Rich query-language

Mongo Data Model

Un *Mongo system* contiene un set di database. Un **database** contiene un set di **collections**, le quali hanno al loro interno un set di **documents**. Un documento è un set di campi (*fields*), ognuno dei quali è una coppia chiave-valore (dove chiave è un nome stringa, un valore è un tipo di base come *string, integer, float, timestamp, binary, ecc.* o un documento o un array di valori).

2.1 Principali caratteristiche di MongoDB

Potenza

MongoDB fornisce molte proprietà dei database relazionali come gli indici secondari, interrogazioni dinamiche, sorting, ricchi aggiornamenti, upserts, cioè aggiornamenti che avvengono solo se il documento esiste e viene inserito se non esiste e aggregazioni. Questo fornisce un'ampia gamma di funzionalità dei database relazionali assieme alla flessibilità e la capacità di scalare propria di un modello non relazionale.

Velocità e scalabilità

Tenendo dati riferiti assieme in un documento, le interrogazioni sono più veloci rispetto a quelle dei database relazionali dove i dati riferiti sono separati in tabelle multiple che necessitano di essere unite tramite join. MongoDB inoltre permette di scalare i dati del database, infatti, è stato progettato per essere scalabile orizzontalmente. In **MongoDB** lo sharding è l'approccio per scalare orizzontalmente, che permette di dividere i dati e memorizzare partizioni di essi su macchine differenti, questo consente di memorizzare più dati e gestire più carico di lavoro senza richiedere l'utilizzo di macchine più grandi e potenti.

L'**auto sharding** permette di scalare i cluster linearmente aggiungendo più macchine. È possibile aumentarne il numero e la capacità senza tempi di inattività, che è molto importante nel web quando il carico di dati può aumentare improvvisamente e disattivare il sito web per lunghi periodi di manutenzione.

I dati vengono distribuiti e bilanciati in modo automatico sui diversi *shard*, che sono dei contenitori che gestiscono un sottoinsieme di una collezione di dati, uno **shard** è visto anche come un singolo server o un insieme di repliche del database. L'auto-sharding divide le collezioni in partizioni più piccole e le distribuisce tra i diversi shard, così che ogni shard diventa responsabile di un sottoinsieme dei dati totali.

Facilità d'uso

MongoDB è stato creato per essere facile da installare, configurare, matenere e usare. A tal fine, MongoDB fornisce poche opzioni di configurazione e cerca automaticamente, quando possibile, di fare la "cosa giusta" permettendo agli sviluppatori di concentrarsi sulla creazione dell'applicazione piuttosto che perdere tempo in oscure configurazioni di sistema.

Indici geospaziali bidimensionali

Caratteristica peculiare di MongoDB è la possibilità di utilizzare **indici geospaziali bidimensionali**. Una volta definito un indice geospaziale sarà possibile interrogare la collezioni con query basate sulla posizione geospaziale del tipo trova gli N elementi più vicini ad una dataposizione o criteri più specifici del tipo trovami gli N musei più vicini ad una dataposizione.

2.2 Comparazione tra MongoDB e MySQL

In questo paragrafo, si fornisce una comparazione tra **MongoDB** e il database relazionale **MySQL**. Vengono analizzate le prestazioni di inserimento e interrogazione dei due database per fornire una serie di risultati che andranno a svelare eventuali punti di forza e debolezza di entrambi. L'analisi è stata estratta da una tesi di laurea dell'Università di Padova: Analisi delle performance dei database non relazionali

L'analisi di benchmarking è stata condotta esaminando, prima di tutto, le prestazioni di inserimento attraverso di diversi metodi di inserimento: ad esempio, utilizzando i *prepared statement* per MySQL o i *bulk insert* per MongoDB. In secondo luogo, si sono testate le prestazioni dei due metodi interrogazione. In particolare, si sono testate le prestazioni del **map-reduce** di MongoDB e poi è stata scritta l'interrogazione speculare in SQL da essere applicata a MySQL.

NOTA. La macchina su cui sono stati eseguiti i test è un portatile da 4 Giga di RAM con processore Duo 2 Core.

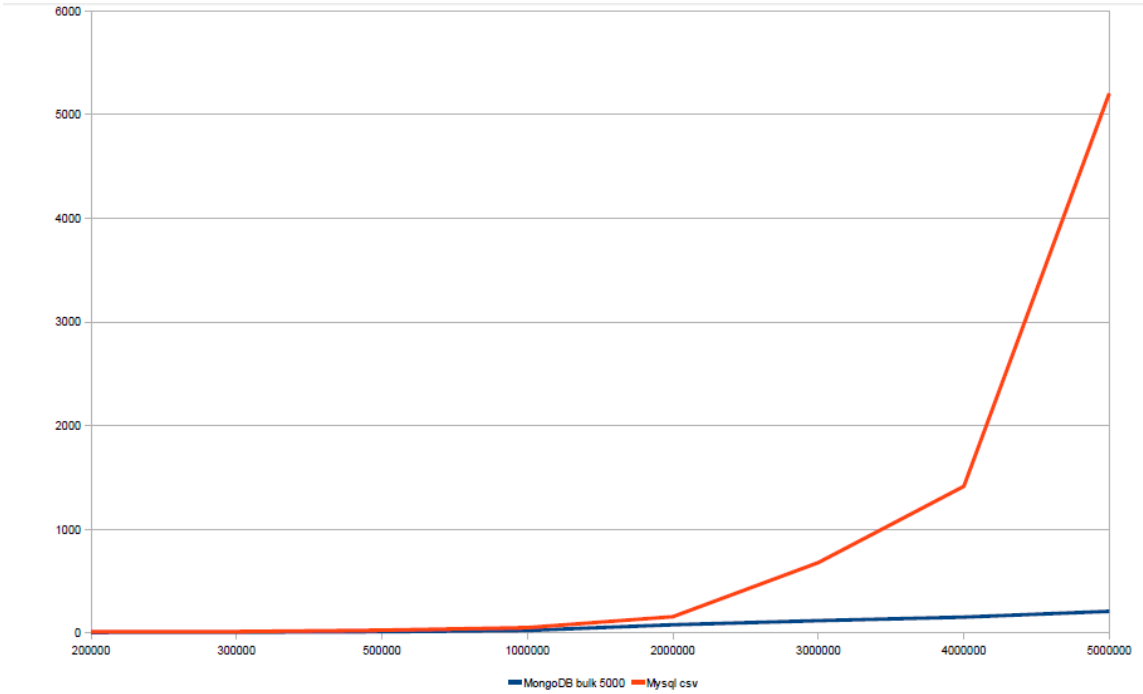
Su un **pc a 32 bit**, ecco i tempi di risposta delle query di inserimento su **MongoDB** e **MySQL**, a parità di dati.

Dati inseriti	Query MongoDB	Query MySQL
100	78 ms	0.05 sec
500	187 ms	0.08 sec
1.000	280 ms	0.22 sec
5.000	1.19 sec	0.91 sec
10.000	2.28 sec	1.75 sec
50.000	11.23 sec	35.46 sec
100.000	22.35 sec	1 min 42.01 sec
200.000	45.97 sec	3 min 12.33 sec

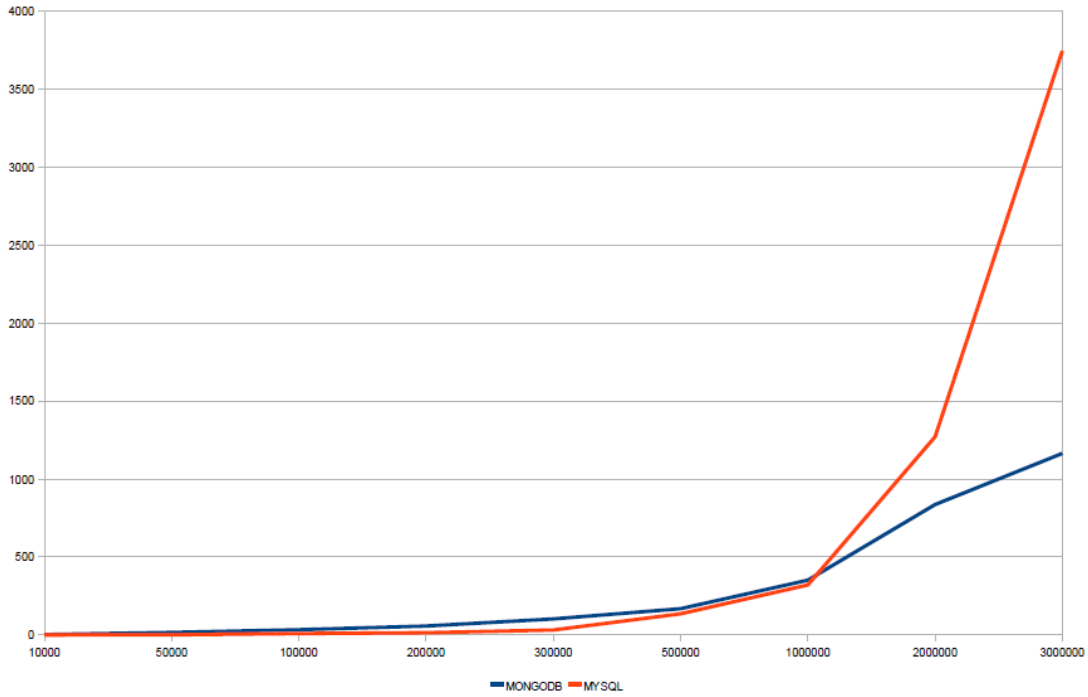
Su un **pc a 64 bit**, ecco i tempi di risposta delle query di inserimento su **MongoDB** e **MySQL**, a parità di dati.

Dati inseriti	Query MongoDB	Query MySQL
100	109 ms	0.06 sec
500	202 ms	0.09 sec
1.000	374 ms	0.12 sec
5.000	1.79 sec	0.23 sec
10.000	3.48 sec	0.41 sec
50.000	17.52 sec	2.01 sec
100.000	34.30 sec	9.73 sec
200.000	58.12 sec	14.76 sec
300.000	1 min 43.12 sec	32.23 sec
500.000	2 min 49.53 sec	2 min 16.33 sec
1.000.000	5 min 51.56 sec	5 min 21.05 sec
2.000.000	13 min 56.06 sec	21 min 10.33 sec
3.000.000	19 min 24.55 sec	1 h 2 min 23.69 sec

Nel seguente grafico, vengono confrontati i due database per quanto riguarda le operazioni di inserimento:



Invece, nel seguente grafico, si confrontano le prestazioni sulle operazioni di interrogazione:



Considerazioni sull'analisi delle performance tra MongoDB e MySQL

Dai risultati ottenuti evince che **MongoDB** è sicuramente più performante di MySQL per l'inserimento e l'interrogazione di grosse quantità di dati. Su piccole quantità di dati, presenta una latenza trascurabile, guadagnando esponenzialmente nei tempi quando si interrogano milioni di documenti/record.
Tuttavia, si nota anche che al crescere dei dati, le operazioni di lettura e inserimento di **MongoDB** subiscono un notevole peggioramento delle prestazioni.
Si può concludere che **MongoDB** risulta essere una valida alternativa ai database relazionali grazie all'innovativo modello dei dati basato sui documenti che fornisce una maggiore interazione con i linguaggi di programmazione orientati agli oggetti, all'agile e semplice linguaggio di interrogazione e alla possibilità di essere facilmente scalabile. **MongoDB** è, inoltre, in grado di fornire ottime prestazioni in scrittura e in lettura addirittura migliori rispetto ai database relazionali.

2.3 Ambiti di utilizzo e realizzazione dei direct graph con MongoDB

MongoDB è adatto in svariati ambiti di utilizzo, come ad esempio:

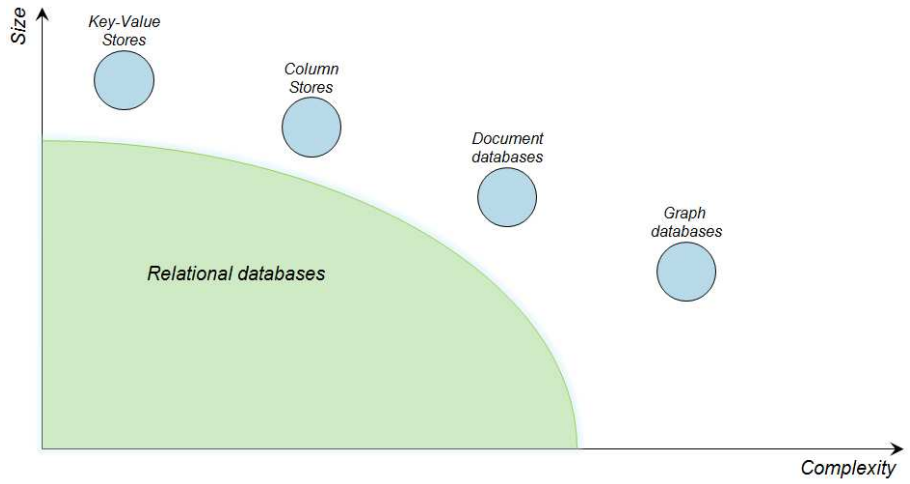
- archiviazione ed elaborazione di grosse quantità di dati
- sistemi di gestione di documenti e contenuti
- E-commerce. Molti siti usano MongoDB come nucleo per l'infrastruttura ecommerce spesso affiancato con un RDBMS per la fase finale di eleborazione e contabilità degli ordini.
- Gaming
- Le infrastrutture lato server per sistemi mobili

Sul sito ufficiale della società **10gen**, che ha realizzato e promuove **MongoDB**, compaiono i seguenti ambiti applicativi:

- **Big Data**
- **Content Management & Delivery**
- **Data Hub**
- **Social & Mobile Infrastructure**
- **User Data Management**

MongoDB, tuttavia, non è particolarmente indicato quando le informazioni memorizzate hanno una struttura complessa, fatta da troppi riferimenti tra i nodi, come si ha per esempio nei **grafi diretti.**

Volendo riprendere un grafico già allegato precedentemente, possiamo vedere che all'aumentare della complessità della struttura dati, diminuisce il numero di informazioni (nodi, record, documenti) che è possibile memorizzare in una struttura dati non relazionale:



Per la gestione e il traversamento di una struttura a grafo, sono stati realizzati i **db "graph-oriented"** che, tuttavia, all'aumentare del numero di nodi e relazioni memorizzate, possono avere problemi nell'esecuzione degli algoritmi di routing che vengono utilizzati per la loro interrogazione. Nei database a documenti, come **MongoDB**, si possono relazionare i documenti tra di loro, per formare una struttura a

grafo, ma non è una soluzione nativa. Infatti, il traversamento dei nodi viene fatto con le "normali" query di interrogazione che non implementano delle routine di calcolo del percorso su grafo.

Dall'altro lato, i database "a grafo" (come **Neo4J** ed **OrientDB**), sono difficilmente scalabili o presentano comunque una certa complessità nella configurazione di una scalabilità orizzontale, che invece in MongoDB è nativa.

Tuttavia, vi sono esempi e progetti in esercizio, che sfruttano **MongoDB** per la memorizzazione e gestione di una struttura **grafo**, come dimostra questo articolo: Building a Directed Graph with MongoDB

Anche sul sito ufficiale di **MongoDB**, si possono trovare vari progetti in cui viene impiegato in scala enterprise e, per alcuni, nella implementazione di una struttura a grafo:

http://www.mongodb.org/about/production-deployments/

Tra questi, particolarmente interessante è il progetto **Wordnik**: si memorizzano **3.5 Terabytes** di dati costituiti da **20 miliardi** di record. Quindi, scalando oppurtamente MongoDB, si riescono comunque a gestire grandi moli di record, anche se le prestazioni nell'esecuzione delle operazioni CRUD potrebbero subire un drastico peggioramento, se non si progetta l'architettura in termini di **Big Data**.

Per quanto riguarda la soluzione proposta ed implementata da **Wordnik**, il cui obiettivo è stato quello di memorizzare e gestire una struttura a grafo in cui visualizzare le correlazioni tra parole di un dizionario (**Word Graph**), ecco un po' di numeri:

- Applicazione in Cluster. Tecnologie usate: Java, Scala e Jetty
- Interrogazioni via REST: 19 milioni di chiamate al giorno (7msec/query in media)
- Server Fisici: 72 GB RAM, 8 core - 4.3 Terabyte DAS
- Decine di miliardi di documenti memorizzati

[**Fonte**: http://www.10gen.com/presentations/mongosf2011/wordnik]

Altro esempio di utilizzo di realizzazione di un social network con MongoDB è dato dal progetto **EventBrite**:

http://www.10gen.com/presentations/mongosv-2010/building-social-graph-mongodb-eventbrite

e sul sito ufficiale di **MongoDB** compare anche un tutorial che spiega come utilizzare il **modello Tree RB** per navigare un grafo/albero di documenti in esso memorizzati:

http://docs.mongodb.org/manual/tutorial/model-tree-structures/

Dagli esempi precedenti, dunque, possiamo osservare che MongoDB si può adattare alla gestione e memorizzazione di un grafo, ma non disponendo di algoritmi di routing (che invece sono nativamente presenti nei database a grafo), occorre interrogare con un approccio iterativo le coppie di nodi del grafo. Questo approccio va bene quando il numero di hop da traversare non è particolarmente elevato. Inoltre,

non dimentichiamo che MongoDB è una soluzione scalabile, rispetto ai database a grafo (che esamineremo in un capitolo successivo) ,con tutti i vantaggi che abbiamo elencato precedentemente.

2.4 Soluzioni ibride: MongoDB + database a grafo + RDF triple store

Esistono anche **soluzioni ibride** che combinano l'utilizzo di **MongoDB,** per memorizzare le informazioni di entità (attributi), e di **database a grafo**, come **Neo4J,** dove si mantengono le relazioni tra i nodi/entità e si sfruttano gli algoritmi di routing per il traversamento del grafo. Ciascun nodo del grafo mantiene un riferimento (*NodeID*) al corrispettivo documento memorizzato in *MongoDB*.

Per quanto riguarda la memorizzazione delle triple <Soggetto>, <Predicato> e <Oggetto> è tradizionalmente diffuso l'utilizzo di **triple Store** che utilizzano il formato standard **RDF (Resource Description Framework)** e che permettono una navigazione tra le informazioni grazie al linguaggio **SPARQL**.

Esaminando alcuni articoli scientifici, in cui si propone di memorizzare un **triple store** su *MongoDB*, si notano poco soddisfacenti: Document Oriented Triple Store based on RDF/JSON e Experimenting with MongoDB as an RDF Store. La difficoltà è dovuta al fatto che le triple devono essere mappate in una struttura a documenti con coppie chiave-valore (in formato JSON) e occorre, dunque, una operazione preliminare di *mapping* (da RDF a JSON). Inoltre, occorre uno strato software in grado di tradurre le query *SPARQL* in *MongoDB Query Language*. In sintesi, per piccole strutture dati, la realizzazione di un *triple store* su *MongoDB* si può implementare, ma al crescere del numero di triple, le performance subiscono un netto calo di prestazioni sulle operazioni di lettura/scrittura. Per realizzare un **Triple Store** esistono delle apposite tecnologie che utilizzano *RDF* e *SPARQL*, come il framework **JENA** in tecnologia JAVA, molto utilizzato nell'ambito del Semantic Web.

2.5 Analisi delle prestazioni di MongoDB all'aumentare dei documenti memorizzati

Come si legge in questo articolo, MongoDB Schema Design at Scale, le prestazioni nell'esecuzione delle operazioni CRUD su MongoDB subiscono una notevole latenza all'aumentare dei dati. Esistono però varie tecniche per risolvere il problema: con una tecnica detta di **preallocazione**, dove si configura una *memcache* per allocare "in-memory" blocchi di record ed effettuando una configurazione su più nodi *slave* (con **sharding** e scalabilità orizzontale). **MongoDB**, infatti, è semplice da configurare per garantire una alta affidabilità e scalabilità e, dunque, migliorare i tempi di accesso alle informazioni.

MongoDB si affida a **MapReduce** per la gran parte dei compiti di ricerca e aggregazione dei dati. In linea teorica, **MapReduce** può operare in parallelo elaborando grandi set di dati contemporaneamente su più core/CPU/computer.

Tuttavia, l'implementazione di MongoDB si affida a **JavaScript**, che è single-threaded. Per elaborare una grande mole di dati, entrando dunque nell'ambito del Big Data, è possibile integrare **MongoDB** con **Hadoop** (grazie ad un *adapter MongoDB* per Hadoop).

Limiti di storage di MongoDB

Esaminiamo il caso d'uso citato in questo articolo: A Year with MongoDB. Qui vengono evidenziati gli aspetti negativi nell'utilizzo di MongoDB per quanto riguarda il degrado prestazionale che è stato riscontrato all'aumentare dei dati. Tuttavia, non è stata effettuata nessuna scalabilità orizzontale, ma MongoDB è stato installato nella sua configurazione di base.

Il **limite della configurazione di base** riscontrato è il seguente:

- Dimensione su disco: 240 GB
- Numero totale di documenti: 85.000.000
- Operazioni al secondo: 520 (operazioni CRUD)

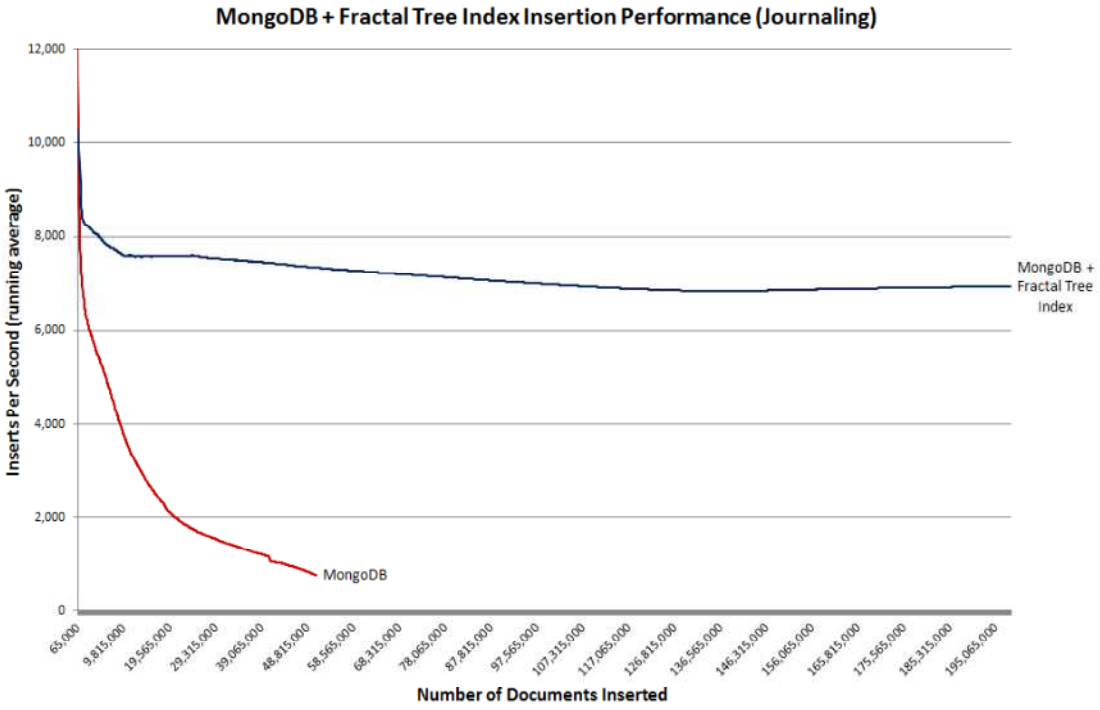
Aldilà di questa soglia, senza adottare nessuna delle tecniche su menzionate, è possibile riscontrare un aumento della latenza media nell'esecuzione delle query su MongoDB.

Altro limite di **MongoDB** è la dimensione occupata su disco dai singoli **documenti** (dove i documenti sono i record memorizzati in *BSON – Binary JSON*): ogni documento, sull'ultima release di MongoDB, non può superare i **16 MB**.

Pochi sono i benchmark di progetti in esercizio che impiegano **MongoDB** e che è possibile reperire online, per poter condurre un'analisi dei limiti di storage della soluzione.

Tra quelli esaminati, citiamo la soluzione di **Tokutek**, società che ha customizzato la distribuzione di **MongoDB** adottando degli indici proprietari, per migliorare le prestazioni del database e poter gestire centinaia di milioni di documenti: http://odbms.org/download/OptimizingMongoDBWithFTI.pdf

Dal benchmark, la velocità di uscita di una installazione standard di **MongoDB** è di 1,045 inserimenti di documenti al secondo su un corpus documentale di circa 54 milioni di documenti già memorizzati. All'aumentare del numero di documenti memorizzati, dunque, la velocità di nuovi inserimenti diminuisce.


MongoDB + Fractal Tree Index Insertion Performance (Journaling)
Inserts Per Second (running average)
12,000
10,000
8,000
6,000
4,000
2,000
0
MongoDB + Fractal Tree Index
MongoDB
65,000
9,815,000
19,565,000
29,315,000
39,065,000
48,815,000
58,565,000
68,315,000
78,065,000
87,815,000
97,565,000
107,315,000
117,065,000
126,815,000
136,565,000
146,315,000
156,065,000
165,815,000
175,565,000
185,315,000
195,065,000
Number of Documents Inserted

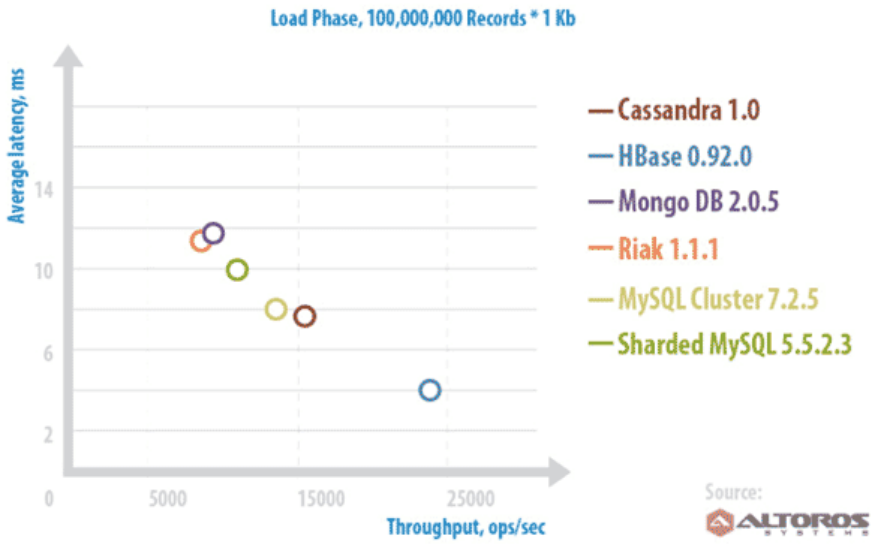
3 Confronto tra le performance di MongoDB e di altri database NoSQL

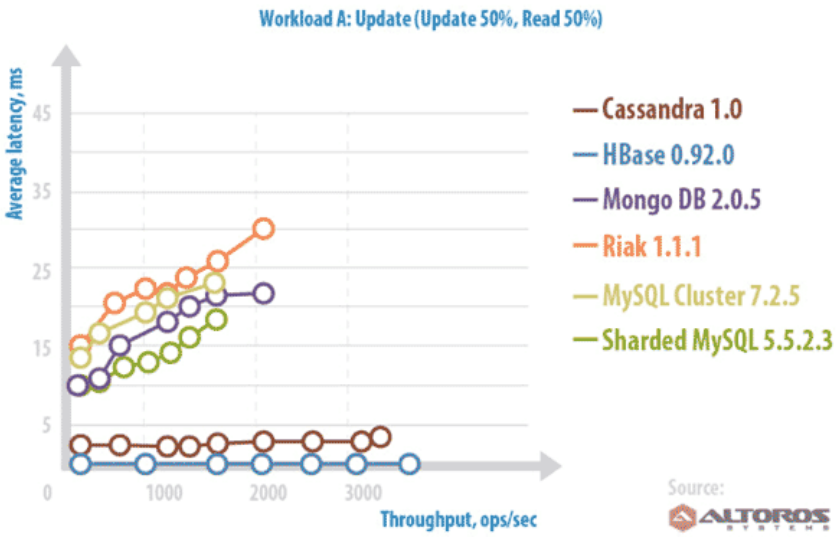
3.1 Confronto MongoDB con vari database NoSQL

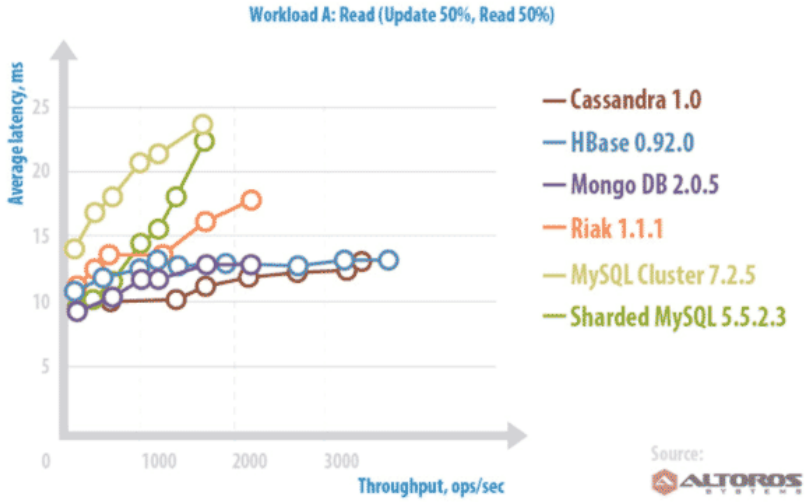
Ecco di seguito, un'analisi di bechmark tra i maggiori DB NoSQL, dove si analizzano le prestazioni (al variare dei workload, ossia delle interrogazioni e del carico impostato) in termini di *throughput* (numero di operazioni al secondo) e di latenza media nell'esecuzione.

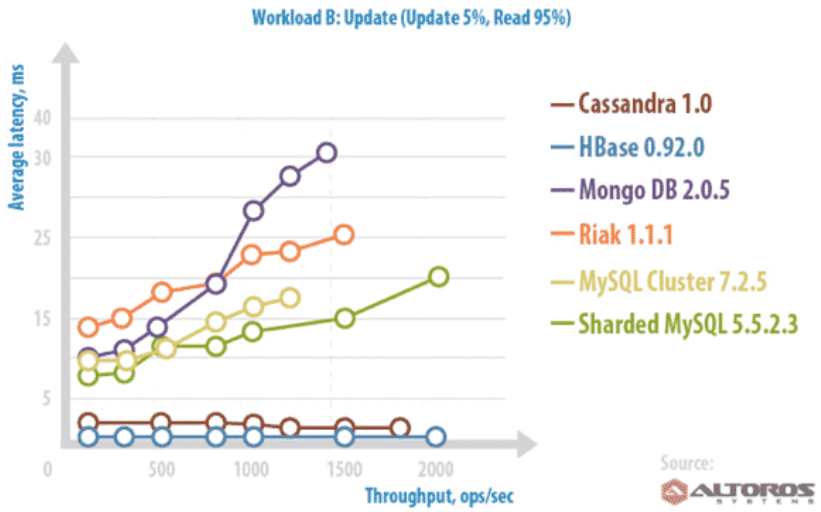
L'analisi è stata effettuata su 4 DB NoSQL più diffusi, di diversa famiglia: **Cassandra** (database "column based"), **HTable** (database "key-value"), **MongoDB** (database "document-oriented") e **Riak** (key-value store). Il confronto avviene anche con il più diffuso database relazionale **MySQL** nelle versioni **Cluster** e **Sharded** (soluzioni a pagamento).

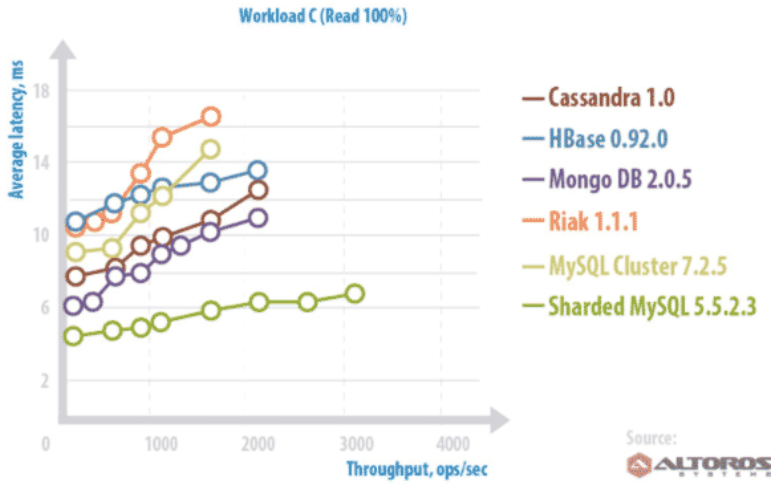
Fonte: http://www.networkworld.com/news/tech/2012/102212-nosql-263595.html?page=5




Workload A: Update (Update 50%, Read 50%)
Average latency, ms
45
35
25
15
5
0
1000
2000
3000
Throughput, ops/sec
Cassandra 1.0
HBase 0.92.0
Mongo DB 2.0.5
Riak 1.1.1
MySQL Cluster 7.2.5
Sharded MySQL 5.5.2.3
Source:
ALTOROS


Workload A: Read (Update 50%, Read 50%)
Average latency, ms
25
20
15
10
5
0
1000
2000
3000
Throughput, ops/sec
Cassandra 1.0
HBase 0.92.0
Mongo DB 2.0.5
Riak 1.1.1
MySQL Cluster 7.2.5
Sharded MySQL 5.5.2.3
Source:
ALTOROS


Workload B: Update (Update 5%, Read 95%)
Average latency, ms
40
30
25
15
5
0
500
1000
1500
2000
Throughput, ops/sec
Cassandra 1.0
HBase 0.92.0
Mongo DB 2.0.5
Riak 1.1.1
MySQL Cluster 7.2.5
Sharded MySQL 5.5.2.3
Source:
ALTOROS



Nelle operazioni di inserimento e aggiornamento, **HBase** e **Cassandra** si dimostrano più veloci ed elaborano un numero maggiore di documenti rispetto a **MongoDB**. Tuttavia, la struttura a "colonne" e "chiave-valore" è meno adatta a memorizzare strutture dati complesse, come i grafi, sebbene possano trattare una quantità di informazioni superiore rispetto ai database "document oriented".

MongoDB si dimostra più performante nel Workload C, dove è stato configurato con un sistema di caching. In sintesi, se MongoDB viene configurato in modo scalabile e in ambiente cluster, è possibile aumentare notevolmente il throughput e far aumentare la velocità di accesso all'informazione.

3.2 Confronto HBase e MongoDB

HBase è una soluzione NoSQL nata e maturata nel progetto **Hadoop** per la memorizzazione dei dati su nodi distribuiti. Presenta caratteristiche interessanti e, per certi aspetti è superiore a MongoDB. Per cui analizziamo la seguente tabella di confronto per trarne delle considerazioni:

Fonte: http://hammerprinciple.com/databases/items/hbase/mongodb

Features	Hbase	MongoDB
Affidabilità	14%	86%
Bassa latenza nelle letture concorrenti	0%	100%
Velocità nell'esecuzione di operazioni matematiche sui dati	75%	25%
Performance nell'esecuzione di complesse operazioni su grosse quantità di dati	50%	50%

Database altamente scalabile	33%	67%
Database con un modello dei dati più restrittivo	33%	67%
Scritture concorrenti	80%	20%
Performance nell'esecuzione di piccole operazioni sui dati	66%	34%
Flessibilità del data model	16%	84%
Facilità di installazione	60%	40%
Interrogazione della struttura a grafo	60%	40%
Facilità nell'implementazione di query ad hoc	42%	58%
Presenza di strumenti open-source per interfacciarsi con il DB	62%	38%
Presenza di strumenti commerciali per interfacciarsi con il DB	19%	81%
Presenza di diversi linguaggi per poter interrogare/integrare il DB	40%	60%
Supporto della Community	33%	67%
Performance in scrittura	33%	67%
Performance in lettura	33%	67%
Replicazione	33%	67%
Semplicità di integrazione in altre piattaforme (embedding)	28%	72%

Aspetti sicuramente importanti, su cui vince MongoDB, sono l'alta affidabilità, replicazione dei dati, accesso concorrente ai dati, semplicità di integrazione, alta flessibilità del data model (che consente di memorizzare strutture dati fortemente dinamiche) e presenza di diversi linguaggi (driver), tra cui Java, per poter interrogare la basedati.

Punto sicuramente non trascurabile è quello del supporto di una Community: gli sviluppatori e i progetti che utilizzano MongoDB sono attualmente i più diffusi rispetto alle altre tecnologie NoSQL.

4 Analisi di performance Graph Database

4.1 Elenco e caratteristiche di varie implementazioni di Graph Database

In generale, i database a grafo hanno una struttura che li rende idonei alla rappresentazione di dati fortemente interconnessi tra loro, creando strutture complesse da gestire. Questa tipologia, infatti, è particolarmente adatta per rappresentare i dati in progetti di tipo chimico, biologico, web mining, web semantico, ecc.

I benefici nell'usare i database a grafo sono dati da un più alto livello di astrazione dei dati che consentono di modellare meglio le informazioni e da un linguaggio di interrogazione ad-hoc potente che utilizza specifici algoritmi di routing.

Nella scelta delle varie implementazioni, bisogna tener presente soprattutto la struttura del grafo usata perché da questa dipendono le operazioni di query ed aggiornamento. Le possibilità sono:

- **Simple Graph**: una serie di nodi e di relazioni tra essi
- **Hypergraph**: estensione del simple graph con la possibilità che un arco può essere associato ad un insieme di nodi
- **Nested Graph**: estende quello precedente, dove ogni nodo può essere un grafo
- **Attributed Graph**: i nodi e gli archi possono contenere degli attributi per descrivere le loro proprietà

Altre importanti caratteristiche da tener presente riguardano i **vincoli di integrità** dei dati, attualmente presa in considerazione in poche implementazioni, e la **scalabilità**. Quest'ultimo è un aspetto cruciale che poche tecnologie a grafo riescono attualmente ad offrire e su cui ci sarà da lavorare nell'immediato futuro.

Qui un breve confronto tra i database a grafo più importanti:

	NEO4J	**Infinite Graph**	**DEX**	**INFO GRID**	**Hypergraph**	**Trinity**	**AllegroGraph**
Documentation?	Good	Good	Fair	Bad	Good	Bad	Good
Portable?	Y	N	Y	Y	Y	N	N
Java?	Y	Y	Y	Y	Y	N	Y
Free?	Y	< 1M	< 1M	Y	Y	N	< 50 M
Property Graph?	Y	Y	Y	Y	Y	Y	RDF
Hypergraph?	N	N	N	N	Y	Y	N

Un confronto delle caratteristiche principali di database a grafo :

TABLE I

DATA STORING FEATURES

Graph Database	Main memory	External memory	Backend Storage	Indexes
AllegroGraph	•	•		•
DEX	•	•		•
Filament	•		•	
G-Store		•		
HyperGraphDB	•	•	•	•
InfiniteGraph		•		•
Neo4j	•	•		•
Sones	•			•
vertexDB		•	•	

GRAPH DATA STRUCTURES

	Graphs				Nodes		Edges		
Graph Database	Simple graphs	Hypergraphs	Nested graphs	Attributed graphs	Node labeled	Node attribution	Directed	Edge labeled	Edge attribution
AllegroGraph	•				•		•	•	
DEX				•	•	•	•	•	•
Filament	•				•		•	•	
G-Store	•				•		•	•	
HyperGraphDB		•			•		•	•	
InfiniteGraph				•	•	•	•	•	•
Neo4j				•	•	•	•	•	•
Sones		•		•	•	•	•	•	•
vertexDB	•				•		•	•	

REPRESENTATION OF ENTITIES AND RELATIONS

	Schema			Instance					
Graph Database	Node types	Property types	Relation types	Object nodes	Value nodes	Complex nodes	Object relations	Simple relations	Complex relations
AllegroGraph					•			•	
DEX	•		•	•	•		•	•	
Filament					•			•	
G-Store					•			•	
HyperGraphDB	•		•		•			•	•
InfiniteGraph	•		•	•	•		•	•	
Neo4j				•	•		•	•	
Sones					•			•	•
vertexDB					•			•	

COMPARISON OF INTEGRITY CONSTRAINTS

Graph Database	Types checking	Node/edge identity	Referential integrity	Cardinality checking	Functional dependency	Graph pattern constrains
DEX	•	•	•			
HyperGraphDB	•	•				
InfiniteGraph	•	•				
Sones		•		•		

CURRENT GRAPH DATABASES AND THEIR SUPPORT FOR ESSENTIAL GRAPH QUERIES

	Adjacency		Reachability				
Graph Database	Node/edge adjacency	k-neighborhood	Fixed-length paths	Regular simple paths	Shortest path	Pattern matching	Summarization
Allegro	•		•			•	
DEX	•		•	•	•	•	
Filament	•		•			•	
G-Store	•		•	•	•	•	
HyperGraph	•					•	
Infinite	•		•	•	•	•	
Neo4j	•		•	•	•	•	
Sones	•					•	
vertexDB	•		•	•		•	

4.2 Analisi e confronto delle performance di varie implementazioni di Graph Database

In questo paragrafo, vediamo dei benchmark su alcuni database a grafo. Si misura la scalabilità delle varie soluzioni all'aumentare del numero dei nodi ed archi del grafo.

Il test è stato fatto su un Quad Core Intel Xeon E5410 a 2,33 GHz, 11 GB di RAM e LFF 2.25Tb su disco. Impostazioni di default della Java Virtual Machine, tranne per i casi di test più pesanti: 2GB per scale 20, e 10GB per scale 22.

Lo scale è il seguente:

- Scale 10: 1000 nodi
- Scale 15: 32000 nodi
- Scale 20: 1000000 nodi

I casi di test sono:

- K1: lettura da in file e caricamento nel database
- K2: lettura di archi con un peso maggiore
- K3: creazione di un sottografo a partire dai nodi adiacenti ad un nodo.
- K4: tempi di attraversamento del grafo a partire da due nodi distanti tra loro

Table 1. Scale factor 10

Kernel	DEX	Neo4j	Jena	HypergraphDB
K1 Load (s)	**0.316**	19.30	7.895	376.9
K2 Scan edges (s)	**0.001**	0.131	0.090	0.052
K3 2-hops (s)	**0.003**	0.006	0.245	0.015
K4 BC (s)	0.512	**0.393**	5.064	1.242
Db size (MB)	2.1	**0.6**	6.6	26.0

Table 2. Scale factor 15

Kernel	DEX	Neo4j	Jena	HypergraphDB
K1 Load (s)	**7.44**	697	141	+24h
K2 Scan edges (s)	**0.001**	2.71	0.689	N/A
K3 2-hops (s)	**0.012**	0.026	0.443	N/A
K4 BC (s)	14.8	**8.24**	138	N/A
Db size (MB)	30	**17**	207	N/A

Table 3. Scale factor 20

Kernel	DEX	Neo4j	Jena	HypergraphDB
K1 Load (s)	**317**	32094	4560	+24h
K2 Scan edges (s)	**0.005**	751	18.60	N/A
K3 2-hops (s)	0.033	**0.023**	0.458	N/A
K4 BC (s)	**617**	7027	59512	N/A
Db size (MB)	893	**539**	6656	N/A

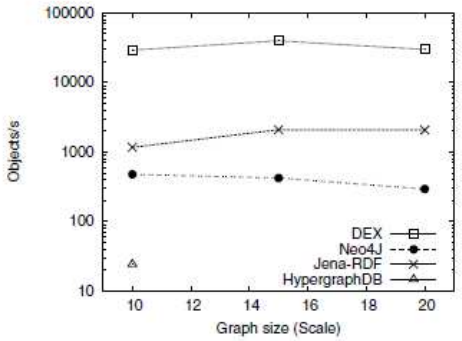


Fig. 1. Objects loaded per sec. (Kernel 1)

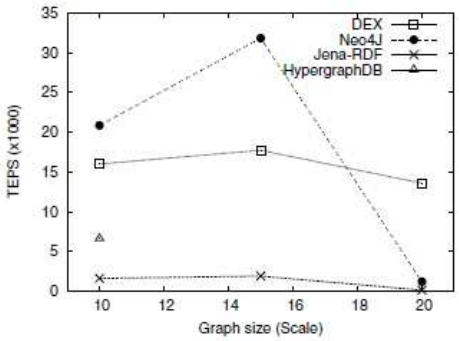


Fig. 2. TEPS (Kernel 4)

In definitiva, il test ha dimostrato come DEX e Neo4J sono le soluzioni più scalabili. Le performance all'aumentare dei nodi sono ancora accettabili.

DEX ha performance superiori rispetto a Neo4j.

[**Fonte Sparsify**: http://sparsity-technologies.com/blog/?p=228]

4.3 Scalabilità

Uno delle problematichedei database a grafo è la scalabilità.

La difficoltà nel garantire la scalabilità di un database a grafo è un problema non facilmente risolvibile, poiché occorre distribuire su più server i nodi del grafo, garantendo la loro raggiungibilità e senza far decadere le performance delle operazioni di attraversamento dei percorsi.

Dex sembra essere la soluzione migliore in termini di scalabilità.

La soluzione trovata da **Infinity Graph** è quella di impostare varie directory in cui memorizzare gruppi di nodi del grafo (partizionamento su file system). Quando termina lo spazio di una directory si può far continuare la memorizzazione dei nodi su quella immediatamente successiva. Il numero delle directory in cui partizionare il grafo è configurabile.

Neo4j non adotta una tecnica per la scalabilità orizzontale (sarà disponibile nella versione 2.0). Attualmente alcuni affermano che la scalabilità di Neo4J possa avvenire tramite lo *sharding domain* (di cui esiste poca documentazione a riguardo). Il vantaggio che si ha nell'utilizzo di **Neo4j** è che, con un solo server, è possibile gestire un grafo di grandi dimensioni:

- 2^35 (~ 34 miliardi) nodi
- 2^35 (~ 34 miliardi) relazioni
- 2^36 (~ 68 miliardi) proprietà
- 2^15 (~ 32 000) tipi di relazioni

Altre tecnologie di recente diffusione, nativamente già scalabili, sono **Titan** o **Giraph,** che implementano anche potenti algoritmi di attraversamento del grafo. Sono ancora però poco mature.

Di seguito, un grafico che rappresenta il posizionamento delle varie tecnologie NoSQL "graph- oriented". Si nota che al crescere del numero dei nodi, peggiorano i tempi di processamento dei percorsi.