POLITECNICO DI MILANO

Facoltà di Ingegneria dell'Informazione

Corso di Laurea Magistrale in Ingegneria Informatica
Dipartimento di Elettronica, Informazione e Bioingegneria

Modellazione e valutazione delle prestazioni di database NoSQL

Relatore: Marco Gribaudo
Correlatore: Pietro Piazzolla

Tesi di Laurea di:
Andrea Gandini, matricola 783184

Anno Accademico 2013-2014

alla mia famiglia

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Introduzione

Lo sviluppo delle tecnologie informatiche nell'ultimo decennio ha visto una crescita in termini di novità e soluzioni senza precedenti, diventando un settore ormai essenziale per qualunque attività e rendendosi sempre più presente anche nel privato. Le nuove infrastrutture informatiche hanno permesso, tra le altre cose, di memorizzare qualunque tipo di dato così creato, permettendo a sempre più aziende di avere accesso a database contenenti informazioni di ogni tipo. L'aumentare dei dati con un trend di crescita ben oltre le aspettative e la necessità di avere database in grado di gestire dati sempre meno strutturati, il tutto mantenendo ottime prestazioni, sono fattori che hanno messo in evidenza come i classici sistemi RDBMS presentassero delle lacune nel trattare questo tipo e mole di dati. E' per questi motivi che si è quindi andati a cercare una tecnologia che proponesse una soluzione ai problemi sopra descritti, trovando una plausibile risposta nei *database NoSQL*.

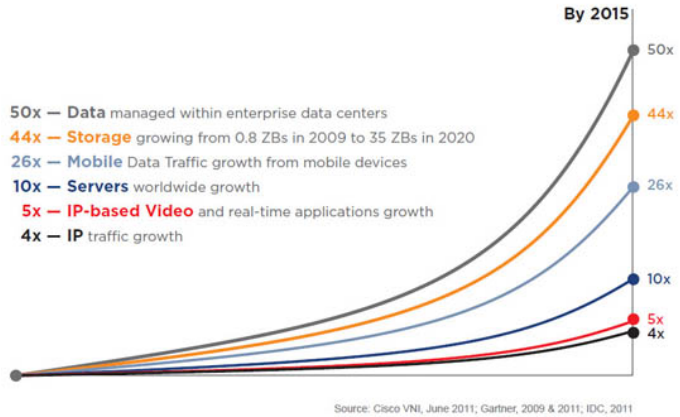

Figura 1.1: Crescita del traffico dati

I database di tipo NoSQL sono sempre più spesso utilizzati a causa delle loro caratteristiche, quali la scalabilità orizzontale e un modello dati free-schema, che combinati ad

un'ottima capacità nel gestire grandi quantità di dati e una particolare idoneità nell'essere implementati su Cloud, li sta rendendo sempre più popolari negli ultimi anni.

L'interesse per queste tecnologie continua a crescere nel tempo a causa della necessità di avere database in grado di trattare quelli che vengono chiamati BigData[31], enormi quantità di dati sempre meno strutturati e sempre più presenti in aziende di ormai qualunque settore o dimensione. Questi richiedono la progettazione di sistemi in grado di recuperare dati in un periodo di tempo sempre più corto, oltre che essere capaci di scalare proporzionalmente all'incremento dei dati stessi: competenze ormai necessarie in un mercato che richiede sempre maggiore competitività.

Database che presentino queste caratteristiche sono quindi generalmente chiamati col nome di database NoSQL: questo nome non significa che non possano essere interrogati utilizzando il linguaggio SQL, ma indica invece come essi non seguano il classico approccio relazionale. Non si hanno più quindi i classici schemi entità/relazione, dati rappresentati sotto forma di tabelle con rigidi vincoli di tipizzazione dei dati, ma si cerca invece di rendere più elastica la gestione dei dati per una maggiore velocità di esecuzione e una più libera schematizzazione dei dati. Grandi compagnie hanno investito nei precedenti anni molte risorse nel progettare le proprie tecnologie NoSQL, sperimentando diverse implementazioni alla ricerca della migliore soluzione. Google[10], Amazon[16], Facebook[32] e Oracle[43] sono solo alcune tra queste, ma negli anni recenti anche aziende di medie-piccole dimensioni si sono avvicinate a questa tipologia di database, portando alla creazione di un grande numero di soluzioni, ognuna con una sua diversa implementazione portando una diminuzione dei costi di produzione e manutenzione.

Come detto, la scalabilità orizzontale è una delle principali caratteristiche dei database NoSQL. Essa consente di essere in grado di aumentare la propria capacità computazionale e di immagazzinare un sempre maggior numero di dati aggiungendo macchine ad un sistema già funzionante. Questo porta alla facile conclusione che sistemi di questo tipo lavorano particolarmente bene quando ci sono più macchine che cooperano insieme, partizionando il lavoro tra i diversi nodi che compongono la rete che si viene così a formare. Un sistema di questo tipo deve essere capace, data una valida configurazione iniziale, di gestire autonomamente il partizionamento dei dati, la distribuzione del lavoro e una corretta comunicazione tra i diversi componenti da cui è formato, anche nel momento in cui questi vengano aggiunti o tolti dinamicamente. Questa proprietà ha come lato negativo la conseguenza, per le aziende che si avvicinano a queste soluzioni o hanno interesse nel creare nuove infrastrutture basate su questi database, di portare a perdere tempo e risorse nel cercare il più efficiente numero di macchine da assegnare al sistema, cercando di minimizzare errori dovuti a sovra stime o sotto stime di carichi di lavoro.

Una soluzione che aiuta la progettazione e l'implementazione di questi database, ma anche di moltissime altre tecnologie, è la sempre più diffusa infrastruttura tecnologica

chiamata Cloud Computing. Questa è un insieme di tecnologie che consente di avere capacità computazionali e di archiviazione, anche molto elevate, offerte tramite una tipica architettura client/server, in cui le risorse fornite al cliente sono virtualizzate e distribuite. Ciò permette di avere elaborazione on-demand, accesso immediato a macchine create al momento grazie alla rete, evitando di dover comprare fisicamente le macchine necessarie a costruire il proprio sistema. Col termine Cloud Computing ci si vuole quindi riferire al fatto che la risorsa di cui il cliente usufruisce non è fisicamente l'hardware dislocato in un centro remoto, quanto il servizio offerto. La notazione Cloud Computing specifica un sotto insieme della tecnologia Cloud, la quale offre diverse tipologie di servizi e non solo quelle relative alla computazione.

Il Cloud Computing ha avuto sempre maggior successo fin dalle sue prime implementazioni, portando una grande innovazione nel settore informatico. Anche qui, come per i NoSQL, si sono sviluppate molte offerte, proponendo servizi competitivi e soluzioni ad hoc, come IAAS (Infrastructure As A Service), PAAS (Platform As A Service), SAAS (Software As A Service) e altri ancora: esempi di Cloud provider sono il già citato Amazon[2], Microsoft[36] o IBM[29], ognuna con la propria piattaforma Cloud.

Figura 1.2: Cloud providers

Lo scopo di questa tesi è cercare di caratterizzare tre[15] dei più usati database NoSQL, ossia *Cassandra*, *MongoDB* e *HBase*, rispetto alle loro principali caratteristiche, studiandone comportamenti e variazioni di prestazione al variare di alcuni parametri, come ad esempio la potenza della macchina su cui essi vengono eseguiti o il numero di nodi che vanno a comporre il sistema. Una volta analizzati i risultati si vorrebbe essere in grado di costruire dei modelli in grado di simulare al meglio i principali comportamenti di questi database, focalizzandosi sulle configurazioni più interessanti. Questi modelli potranno essere poi usati in futuro per verificare, al cambio delle specifiche delle risorse, le variazioni

subite dal sistema, permettendo studi su di essi senza dover costruire l'intera architettura fisicamente, risparmiando risorse quali tempo e denaro.

La tesi è organizzata come illustrato di seguito.

- Nel capitolo 2 verranno descritti i database NoSQL secondo le loro principali caratteristiche, fornendo anche una visione della loro divisione per tipologia. Verranno successivamente discussi ognuno dei database presi in considerazione, evidenziando le caratteristiche che li differenziano, per poi passare alla descrizione del Cloud Computing utilizzato e dei lavori precedenti esaminati.

- Nel capitolo 3 verranno fornite le metodologie con cui sono stati configurati i database e l'architettura cloud usata, oltre alla presentazione del software di benchmark utilizzato per svolgere i test necessari e il software utilizzato per la creazione dei modelli.

- Nel capitolo 4 verranno presentati in dettaglio i test svolti, analizzando i risultati ottenuti da questi anche in relazione allo studio dei database precedentemente effettuato.

- Nel capitolo 5 saranno mostrati i modelli progettati per riprodurre le caratteristiche dei database presi in analisi, verificando il comportamento di questi tramite simulazione, confrontandoli coi risultati ottenuti e presentati nel capitolo precedente.

- La tesi si conclude con il capitolo 6, dedicato alle conclusioni e alle prospettive future.

Capitolo 2

Stato dell'arte

Questo capitolo si vuole concentrare sulla caratterizzazione dei database e del cloud utilizzati, focalizzando l'attenzione sugli aspetti rilevanti per il lavoro di questa tesi.

In una prima sezione verranno descritte le principali proprietà dei database NoSQL, dando spazio ad una panoramica completa delle famiglie in cui vengono suddivisi, per permettere al lettore di comprendere in quale ambiente ci si sta muovendo.

Nella seconda sezione si illustreranno le peculiarità e le architetture dei tre database presi in considerazione, analizzandoli dettagliatamente e concentrandosi sugli aspetti che differenziano l'uno dagli altri.

Infine, nell'ultima sezione, sarà presentata una descrizione della piattaforma cloud usata, descrivendone le principali funzionalità e permettendo una maggiore comprensione delle metodologie mostrate nel capitolo successivo.

2.1 Database NoSQL

I database NoSQL si stanno affermando nel mondo delle basi di dati negli ultimi anni, a causa delle loro caratteristiche che ben si adattano alla direzione presa recentemente nel trattamento e memorizzazione dei dati: una tipizzazione meno severa dei dati, dovuta ad un modello dati molto più permissiva, una scalabilità orizzontale molto marcata ed una facile implementazione, sembrano essere le parole chiavi che hanno permesso una rapida diffusione sul mercato di questa tipologia di database.

Prima di proseguire si vuole dare una definizione di *database NoSQL*: si definisce di tipo *NoSQL* un database non basato su schema relazionale, fortemente improntato alla scalabilità orizzontale e dotato di un modello dati particolarmente flessibile.

2.1.1 Principali caratteristiche

Di seguito sono elencate le principali caratteristiche che contraddistinguono ed appartengono a tutti i database NoSQL.

Distributed

Questi database lavorano molto bene in un ambiente distribuito[9], essendo capaci di gestire autonomamente la comunicazione tra i vari nodi. Distribuzione dei dati, gestione della loro replicazione e bilanciamento del carico di lavoro sono tutte operazioni che vengono svolte dai nodi che, cooperando direttamente tra loro o tramite un nodo gestore, appaiono all'utente finale come se il tutto venisse eseguito da una sola entità.

Elastic Scalability

La scalabilità è una proprietà architetturale che permette ad un sistema di continuare a servire un sempre maggior numero di richieste subendo delle lievi ripercussioni sulle prestazioni. La strada più semplice per avere questo comportamento è quello di aumentare le prestazioni dell'hardware della macchina corrente, metodo detto *scalabilità verticale*. La *scalabilità orizzontale* invece, indica l'aggiungere macchine al sistema corrente, condividendone dati e carico di lavoro, portando però alla necessità di avere software in grado di gestire sincronizzazione dei dati e gestione ottimale della rete che si viene così a creare.

Per *elastic scalability*[8] si vuole però indicare una particolare forma di scalabilità orizzontale, capace di scalare sia in numero positivo che in numero negativo.

Nel primo caso si vuole quindi aggiungere nuove macchine al sistema corrente, senza dover riavviare il sistema, rendendo questa operazione trasparente all'utente finale e senza configurare manualmente il bilanciamento della rete.

Nel secondo caso invece, per motivi che possono andare da un mero risparmio economico allo spostamento di intere applicazioni/dati verso un altro centro, si vuole eliminare una o più macchine dal sistema. Questo comporta il fatto di dover gestire la migrazione dei dati dalle macchine così rimosse, lasciando ovviamente al sistema la completa gestione automatica di tutte queste operazioni.

Si noti che questa proprietà si adatta estremamente bene all'ambiente fornito dal cloud, in cui l'avviamento e lo spegnimento delle singole macchine virtuali può avvenire spesso e rapidamente.

Replication

Avere una grande mole di dati distribuiti su una grande quantità di macchine, porta alla necessità di voler replicare i propri dati, permettendo alla rete di aver sempre una copia del dato qualora un nodo dovesse avere un malfunzionamento o non essere più raggiungibile dagli altri nodi. Questa funzione viene supportata da tutti i database NoSQL in maniera anche molto differente tra loro. Questo metodo verrà analizzato meglio nella descrizione dei singoli database nella sezione successiva, permettendo una visione più approfondita e personalizzata di questa tecnica.

CAP Theorem

Il CAP theorem è un teorema sviluppato da Eric Brewer nel 2000[6] che indica come in un sistema distribuito[27] solo due tra le tre seguenti proprietà possano essere supportate allo stesso tempo:

- **C***onsistency:* tutti i client ricevono sempre e solo l'ultimo valore valido rispetto alla stessa interrogazione, anche in caso di scritture concorrenti.
- **A***vailability:* tutti i client sono sempre in grado di leggere e scrivere dati.
- **P***artition:* il database è partizionato su diverse macchine, potendo continuare a funzionare anche in seguito ad un malfunzionamento della rete.

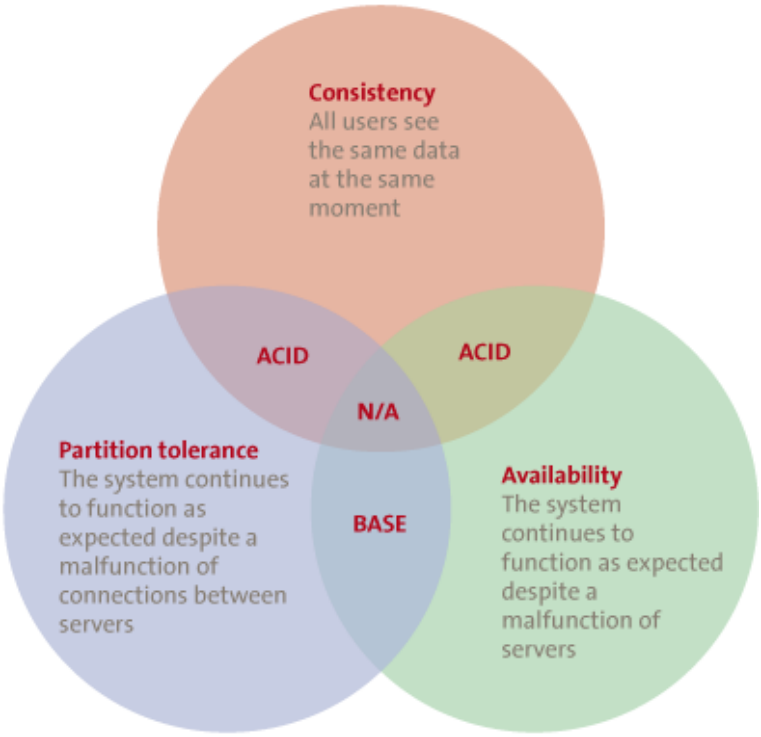

Figura 2.1: Cap theorem

Sarebbe comunque più corretto dire che il CAP theorem indica come, nel momento in cui si vuole avere la proprietà del partizionamento, si è obbligati a dover scegliere o la proprietà di availability o la proprietà di consistency. Questo teorema è stato qui illustrato per poter verificare, nella prossima sezione, quali tra le tre proprietà vengano scelte dai database presi in esame.

BASE

L'acronimo *ACID* (**A***tomicità*, **C***onsistenza*, **I***solamento*, **D***urabilità*) indica le proprietà logiche che le transazioni devono avere e che devono essere soddisfatte attraverso diversi meccanismi implementati nei database relazionali. Nei database NoSQL, invece, queste proprietà non possono essere sempre garantite per via della loro natura.

Essendo i database NoSQL fortemente improntati sulla distribuzione delle query su un elevato numero di nodi, la consistenza, così come descritta dal CAP theorem, non può essere garantita in modo così rigido come invece descritto attraverso le proprietà ACID. Un differente insieme di principi, detto *BASE*[17], viene introdotto in aiuto di questa tipologia di database, diventando particolarmente rilevante soprattutto per i database che optano per una scelta di partitioning e availability tra le proprietà offerte dal CAP theorem.

Questi sistemi sono allora detti **E***ventually consistent*, poichè assicurano la consistenza dei dati dopo un ragionevole lasso di tempo. La consistenza è quindi garantita, ma non immediatamente. Se questa non è assicurata dopo ogni singola transazione, il sistema è detto stare in un **S***oft state*, cioè in uno stato per cui il database può cambiare nel tempo a causa del principio appena spiegato: a causa della propagazione della consistenza tra i dati in modo non perfettamente sincrono, lo stato del database può variare anche in assenza di input. Le proprietà appena descritte hanno però l'effetto di portare un forte incremento della disponibilità generale dei dati, rendendo il sistema **B***asically* **A***vailable*, cioè interrogabile sempre.

Schema Free

Mentre i database relazionali presentano un modello dati basato, come suggerito dal nome, sul *modello relazionale*, in cui ogni tabella e attributi vengono definiti a priori e imponendo vincoli di tipizzazione dei dati, nei databaes NoSQL questo modello non è più valido. Per *schema free* si vuole proprio intendere la caratteristica per cui il modello dati di questi database permette una forte libertà di azione: si possono quindi aggiungere attributi solo a determinate righe, senza specificare il tipo di dato accettato da esso, oltre al fatto di non dover nemmeno creare a priori uno schema. Ma i database NoSQL vanno ben oltre a questo, poichè esistono molte variazioni nel modello dati, decidendo anche di escludere del tutto l'idea di tabella e arrivare alla rappresentazione dei dati attraverso grafi, come illustrato di seguito.

2.1.2 Tassonomia

Di seguito vengono definite le quattro famiglie di modelli dati in cui i database NoSQL vengono suddivisi, elencate in ordine di complessità.

Key-Value

La migliore opzione nel caso si voglia un modello dati estremamente semplice e scalabile. I dati inseriti in un database di questo tipo sono rappresentati in forma di *chiave-valore*, come mostrato in Fig.2.2, in cui il tipo di quest'ultimo non è specificato in nessun modo al database: esso può essere un documento, un'immagine o un semplice numero, senza doverne esplicitare a priori il tipo. I dati su questi sistemi sono accessibili esclusivamente per chiave, con la possibilità di recuperare valori per un certo range di queste. Proprio per la loro naturale semplicità, questi database si adattano particolarmente bene alla scalabilità orizzontale, oltre ad offrire performance particolarmente elevate, soprattutto in lettura, e ad avere un bilanciamento del lavoro distribuito quasi lineare.

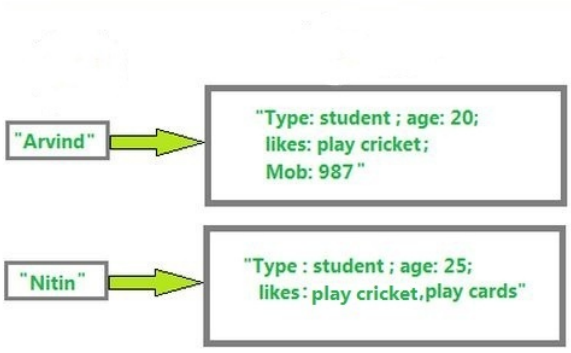

Figura 2.2: Esempio di key-value data model

Particolare fondamentale da tenere in considerazione è la progettazione della chiave primaria[1], visto che questi database possono essere considerati come delle grosse hash table e che sono quindi basati su indirizzamento diretto, rischiando di perdere in prestazioni nell'effettuare ricerche ordinate o ordinamenti. Conoscere a priori le chiavi che si vogliono utilizzare permette quindi di scegliere adeguatamente una codifica di generazione delle stesse, senza dipendere da un sistema automatico di generazione casuale.

Molti dei database appartenenti a questa famiglia sono soluzioni basate sul solo utilizzo della memoria: si sceglie quindi di sacrificare la persistenza per avere prestazioni migliori. Ovviamente queste soluzioni devono essere usati per applicazioni particolari, essendo consapevoli del trade-off a cui si va incontro. Un database di questo tipo che sta riscuotendo un discreto successo negli ultimi tempi è *Redis*[12], un database appunto *memory-based*, con possibilità di esplicitare la volontà di eseguire backup su disco fisso, riducendo ovviamente le prestazioni per la durata di questa operazione.

Column-oriented

Questo modello permette di memorizzare e raggruppare dati per colonne invece che per righe. Questa particolarità permette a questa famiglia di modelli di essere particolarmente adatta alla elastic-scalability, potendo scalare orizzontalmente per righe quanto verticalmente per colonne, ottenendo un modello composto dagli elementi che seguono.

Column families contengono righe e colonne dove ogni riga è identificata da una *Row Key* mentre ogni colonna è una tripletta, consistente in un *column name*, un *column value* e un *column timestamp*. Ogni riga non deve necessariamente condividere il set delle proprie colonne con altre righe, permettendo al modello di poter aggiungere colonne solo su determinate row-key ed arrivando quindi ad avere una struttura di tipo array multidimensionale sparso. L'idea è quella di raggruppare sotto una stessa column family diverse row keys che hanno simili, ma non per forza identiche , colonne.

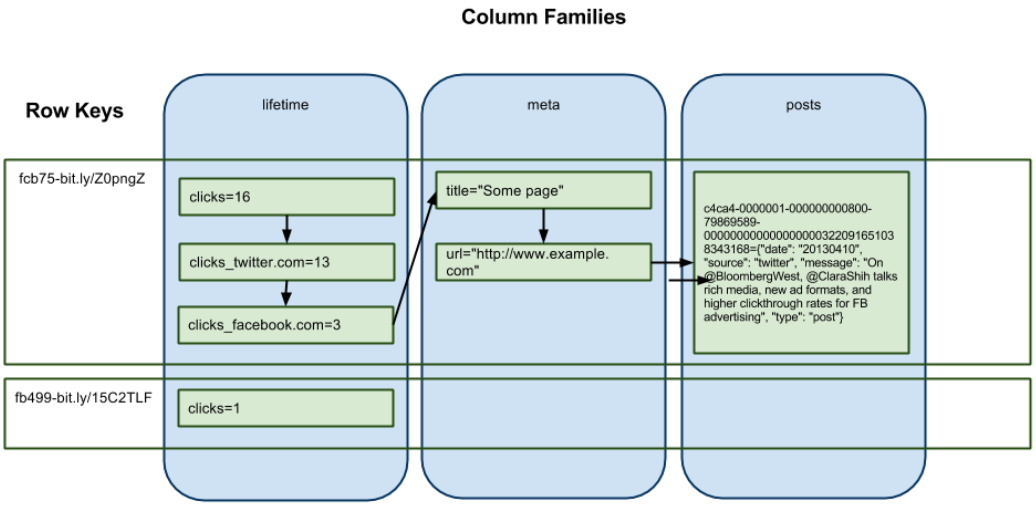

Figura 2.3: Rappresentazione di column families

La Fig.2.3 è un esempio pratico di quanto detto. In un modello del genere le interrogazioni vengono solitamente eseguite svolgendo raggruppamenti per column families, andando poi ad analizzare le row key e i relativi attributi appartenenti ad esse.

Cassandra ed HBase, due dei tre database scelti ai fini di questa tesi, appartengono a questa categoria. Cassandra in particolare offre una soluzione ancora più personalizzata, introducendo il concetto di *Super Column Family* mostrato in Fig.2.3, che permette di aumentare di un livello l'indicizzazione tra row key e column families. Queste colonne sono rappresentate da un nome e da un insieme di attributi che vengono quindi raggruppati al suo interno, fungendo da una sorta di view.

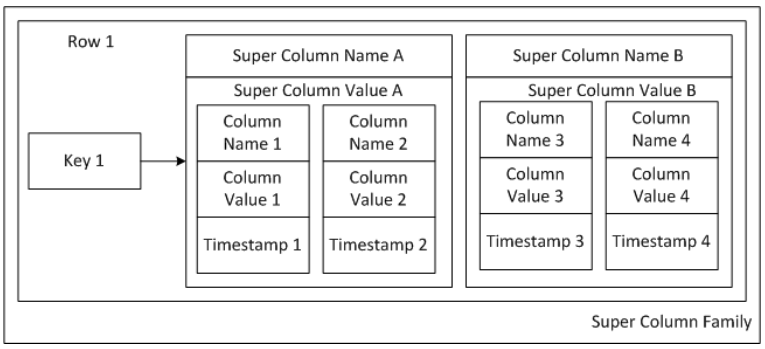

Figura 2.4: Esempio di Super Column Family

Document-oriented

Questo modello dati vede il database come uno storage di *documenti*, dove ognuno di essi è una complessa struttura dati. Generalmente essa viene rappresentata tramite una codifica in XML o ancora più comunemente in JSON: si ha quindi che ogni documento è un insieme di chiavi a cui è associato un valore, dove quest'ultimo può essere una stringa, una lista o altro ancora senza nessun vincolo, potendo anche inserire un riferimento ad un altro documento. Questa ultima possibilità permette una particolare flessibilità nella gestione dei dati, che diventa in qualche modo simile ad un database ad oggetti. La grande differenza tra questo modello e quello key-value è dovuta dalla maggiore trasparenza del documento, dove ogni attributo può essere utilizzato per eseguire query più specifiche.

```
doc =
{ _id: new ObjectId("4c4b1476238d3b4dd5003981"),
  slug: "wheel-barrow-9092",
  sku: "9092",
  name: "Extra Large Wheel Barrow",
  description: "Heavy duty wheel barrow...",

  details: {
    weight: 47,
    weight_units: "lbs",
    model_num: 4039283402,
    manufacturer: "Acme",
    color: "Green"
  },

  total_reviews: 4,
  average_review: 4.5,
```

Figura 2.5: Un documento di MongoDB

Dalla Fig.2.5 si intuisce la facile comprensione e il semplice controllo che si ha sui dati con questo modello, anche se la gestione di documenti particolarmente grossi diventa di

più difficile controllo, richiedendo anche una computazione maggiore. Soluzioni a questo problema sono per esempio il limitare la dimensione della grandezza di ogni singolo documento, come fatto da molti database di questo tipo.

Le query sul documento permettono allora di recuperare interi documenti (ognuno di essi è sempre caratterizzato da una chiave univoca `_id`) o porzioni di essi che presentino particolari valori sulle chiavi ricercate.

Un esempio di questi database è MongoDB, database scelto per la stesura di questa tesi, che ha ricevuto particolare attenzione da parte delle aziende forte di un vasto supporto per molti linguaggi di programmazione e di una facile installazione.

Graph-oriented

L'ultima famiglia di database NoSQL è quella che usa la struttura dei grafi, come nodi e archi, per rappresentare e memorizzare i propri dati. Qui i nodi rappresentano le singole entità a cui è possibile assegnare attributi, mentre gli archi rappresentano le relazioni tra i vari nodi. Anche a loro è possibile assegnare attributi, considerando che in questi modelli l'informazione principale è proprio contenuta sulle relazioni.

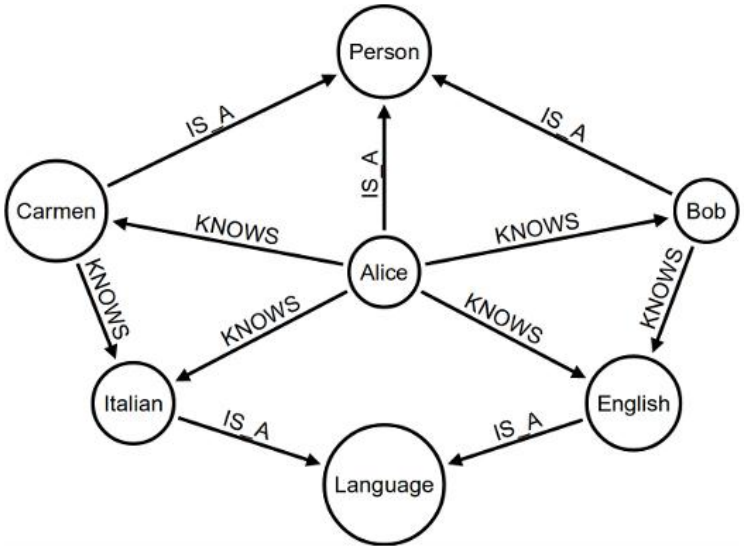

Figura 2.6: Rappresentazione di dati tramite un graph-oriented database

Questo modello dati sta ricevendo molti consensi negli ultimi anni, vista la particolare predisposizione a trattare dati ottenuti da applicazioni sempre più usate, come social network o software di analisi di flusso. Un database di questo tipo che sta riscuotendo molto successo è *Neo4j*[42], estremamente performante nella rappresentazione di dati in XML, file system e reti.

Va notato che mentre le tipologie di NoSQL viste precedentemente sono *aggregated oriented*, cioè orientate verso un partizionamento efficace dei dati, questo modello tende invece a focalizzarsi quasi esclusivamente sulla rappresentazione e manipolazione delle relazioni.

Le soluzioni appena viste rendono i database NoSQL, se considerati nel loro complesso, potenzialmente utili per una enorme quantità di utilizzi. I modelli appena elencati mettono in evidenza come questi database siano evidentemente indicati per applicazioni che necessitano una scalabilità particolarmente forte, oltre ad una gestione dei dati leggera e dinamica che non vincoli troppo la modifica del database stesso. Se questo è sicuramente un aspetto positivo, bisogna però tenere conto che una troppa libertà nella creazione di indici, attributi e tabelle può avere lati negativi. Se non si ha un livello applicativo soprastante al database che controlli la gestione dello stesso, è facile ritrovarsi dopo poco tempo di utilizzo un modello estremamente sparso e ridondante. Si pensi per esempio ad un insieme di utenti che inseriscano ciascuno una nuova entità persona, ognuna composta da una diversa chiave rappresentante il suo indirizzo: *address*, *via*, *road*, semanticamente uguali ma che creano confusione nell'interrogazione del database, senza considerare il fatto che il valore può essere espresso attraverso una stringa, un numero o ancora un riferimento ad un altro documento.

Nella prossima sezione verranno analizzati in dettaglio i tre database presi in analisi, principalmente dai punti di vista architetturale e di gestione di sistema, utili per i fini di questa tesi.

2.2 Cassandra, MongoDB e HBase

In questa sezione verranno presentati i tre database analizzati, concentrandosi sulla loro architettura e la loro tipologia, il loro modo di gestire la distribuzione dei dati o ancora come differiscono nel fornire replicazione dell'informazione.

I database sono stati scelti perchè considerati i più comuni nell'essere utilizzati, oltre che presentare caratteristiche diverse tra loro che hanno permesso un confronto valido tra diverse scelte architetturali e metodologiche. Un quarto database fu preso in considerazione all'inizio di questo lavoro, Redis, che come già detto vede la sua peculiarità nell'utilizzare esclusivamente la memoria per ogni operazione, presentando quindi una persistenza facoltativa. Poichè questa peculiarità rappresenta casi applicativi molto specifici, è stato scelto di concentrarci su database più ordinari.

2.2.1 Cassandra

Cassandra[23] è un data storage open source nato inizialmente come progetto interno a Facebook, per poi diventare nel marzo 2009 parte del progetto Incubator[21] di Apache Software Foundation[20]. Ancora prima di essere rilasciata la versione 1.0, questo database contava già importanti utilizzi in produzione per grandi compagnie quali Facebook, Twitter, e Cisco.

Tipologia

Appartiene alla famiglia dei column-oriented database, fornendo tramite il concetto di Column Family una versione estesa del metodo key-value store costruito su una struttura column-oriented, oltre che introdurre una struttura aggiuntiva, le Super Column Families, descritta nel paragrafo 2.1.2.

Questo database rispetta le proprietà di partitioning e availability rispetto a quanto detto nella sezione precedente riguardo al CAP theorem. I nodi, anche se non comunicanti tra loro, rendono sempre disponibile la possibilità di scrivere e leggere i propri dati, lasciando poi decidere all'utente quali politiche di merge adottare nel momento in cui la comunicazione tra nodi torni stabile.

Architettura

La sua architettura è basata sulla costruzione di una *ring network*[28], una rete ad anello in cui ogni nodo è gerarchicamente uguale agli altri, come mostrato in Fig.2.7. Non ci sono configurazioni master-slave o simili, permettendo di avere un sistema *no-single-point-of-failure*, cioè in grado di essere disponibile anche dopo la caduta di un qualsiasi nodo della rete. Il protocollo peer-to-peer permette comunicazione tramite i vari nodi della ring.

Ad ogni nodo viene assegnato un determinato range di *token*, valore attribuito poi ad ogni dato inserito per permettere di determinare su quale nodo della rete andrà collocato, permettendo una facile distribuzione dei dati su tutta la rete con una funzione di hash. I nodi che si prendono carico di gestire le richieste dei clients sono detti *entry point* e questo ruolo può ovviamente essere ricoperto da un numero desiderato di nodi, senza dover essere preventivamente specificato, dipendendo solamente dalle impostazioni del client.

Clusters

Creare un singolo anello è una soluzione comunemente usata, soprattutto per la sua semplicità implementativa. E' possibile comunque disporre, in caso si voglia riservare particolare attenzione alla distribuzione fisica dei dati, di più anelli cooperanti tra loro, soluzione in cui ovviamente tutti i nodi sono considerati ancora uguali. La possibilità di avere una rete multi clusters viene utilizzata spesso per la replicazione dei dati, aumentando la reliability dell'intero sistema.

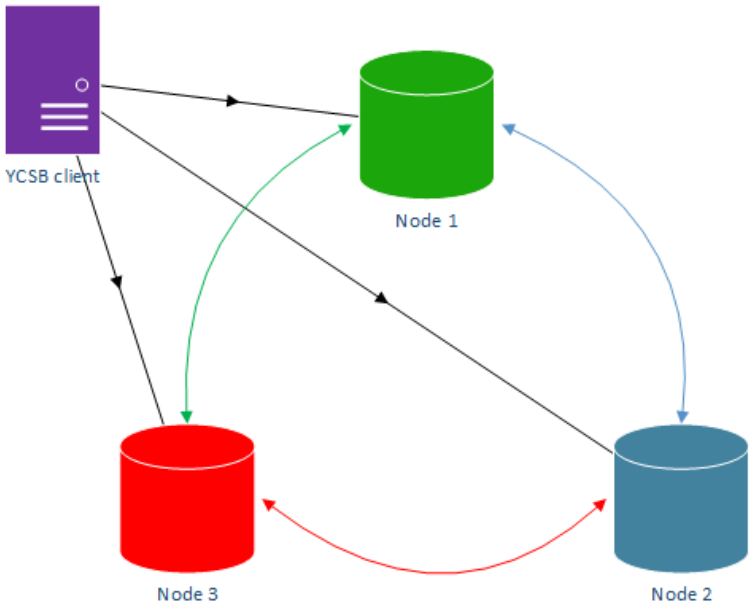

Figura 2.7: Implementazione di una ring di Cassandra in cui ogni nodo è un entry point

Gossip protocol

Per verificare il corretto funzionamento della rete e controllarne l'integrità, come mostrato in Fig.2.8, Cassandra utilizza un protocollo di tipo *gossip*, protocollo basato sulla distribuzione casuale e continua dell'informazione da un nodo all'altro, quest'ultimo scelto casualmente dal primo. In Cassandra il gossip protocol implementato funziona proprio come detto: periodicamente, un primo nodo ne sceglie un secondo a caso nella rete ed inizia con lui una *gossip session*, composta da tre scambi di messaggi simile a quanto avviene nella procedura *three-way handshake*[4] nel protocollo TCP. Colui che vuole iniziare una sessione manda al nodo scelto un messaggio contenente le informazioni di cui dispone, quali la sua conoscenza della rete e lo stato attuale dei dati. Il secondo nodo riceve il messaggio, modifica il proprio stato nel caso ce ne fosse bisogno, e invia le proprie informazioni al primo nodo, il quale utilizza l'informazione ricevuta per modificare la sua conoscenza, prima di terminare la comunicazione instaurata tramite un messaggio di chiusura.

Elastic scalability

Proprio per la semplicità strutturale della ring, l'elasticità dell'intero anello viene gestita in modo facile e completamento automatico. L'aggiunta di un nodo prevede il suo inserimento all'interno dell'anello, la ridistribuzione dei token per permettere un bilanciamento del carico dei dati e del lavoro, quindi la riassegnazione/migrazione dei dati da un nodo

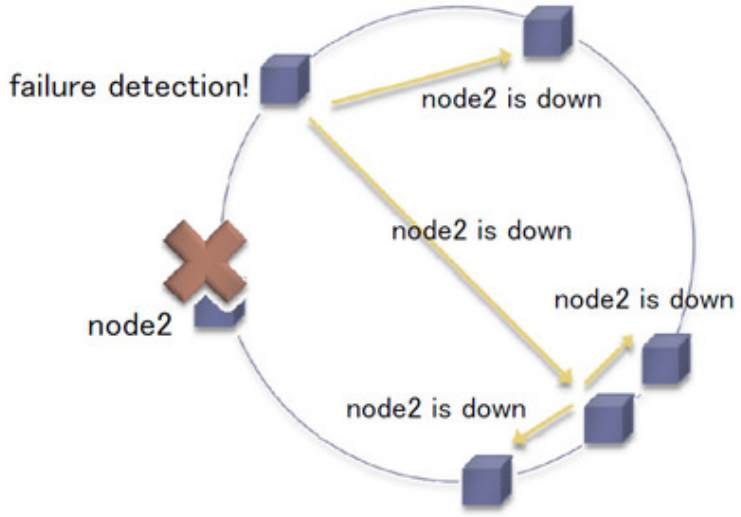

Figura 2.8: Gossip protocol applicato ad un cluster di Cassandra con sei nodi

all'altro in base al nuovo assegnamento dei token. Stessa cosa, al contrario, avviene nel caso un nodo venga tolto dalla rete o non riesca ad essere più raggiungibile, caso in cui i nodi rimanenti si occuperanno di distribuire tra loro i dati del nodo rimosso, riassegnando anche in questo caso i token per un più equilibrato bilanciamento del database.

Keyspace

Si vuole ora introdurre il concetto di *Keyspace*. Questo è l'unità informativa di più alto livello per Cassandra, associabile ad un intero database di un modello relazionale, che deve essere creato prima di poter effettuare qualsiasi operazione. Esso conterrà quindi tutte le singole tabelle, le quali erediteranno gli attributi definiti sul Keyspace che le contiene. Questi attributi sono essenzialmente tre: *Replication factor* e *Replica placement strategy*, i cui funzionamenti verranno illustrati a breve, indicanti quante copie del dato si vuole tenere nella rete e il modo in cui si voglia distribuire queste copie, e le *Column families*, introdotte già nelle sezioni precedenti, paragonabili alle tabelle di un database relazionale ed appartenenti ad uno e un solo keyspace[18].

L'ordine di importanza riguardo ai singoli elementi che compongono Cassandra è indicato tramite una classica interrogazione, generalmente definita come *get(keyspace, column family, row key)*.

Replicazione: Replication Factor e Replica placement strategy

La replicazione in un database consiste nel creare una copia del dato originale, permettendo di disporre dei dati anche nel momento in cui un nodo presenti degli errori. Cassandra crea tante copie del dato originale quante ne sono state definite tramite l'attributo Replication factor del relativo Keyspace nel quale si sta lavorando. Le copie sono recuperabili in qualunque momento dal database, permettendo la disponibilità dei dati anche nel caso di malfunzionamento del nodo contenente i dati originali. Le copie ricoprono inoltre un ruolo di controllo di correttezza e consistenza, illustrato a breve, che può avere impatti anche molto rilevanti riguardo alle prestazioni.

Le copie così create possono essere distribuite sui vari nodi in accordo a due principali strategie tra cui scegliere: *Simple Strategy* o *Network Topology Strategy*. Il primo, come mostrato in Fig.2.9, distribuisce i dati seguendo l'ordine dell'anello: se si vuole creare una singola copia del dato originale, memorizzato nel nodo numero due, la copia andrà collocata nel prossimo nodo seguendo la rete, vale a dire nel nodo numero tre. Si segue, all'aumentare del replication factor, la circolarità dell'anello fino alla memorizzazione di tutte le copie. Il secondo distribuisce i dati in base alla propria conoscenza della rete, dando priorità alla distribuzione su vari anelli poichè ci si aspetta che nodi appartenenti alla stessa ring abbiano più probabilità di fallire piuttosto che nodi inseriti in anelli diversi, tenendo anche in considerazione la vicinanza fisica delle macchine. Si noti che Cassandra permette di avere un replication factor maggiore del numero dei nodi che compongono il database.

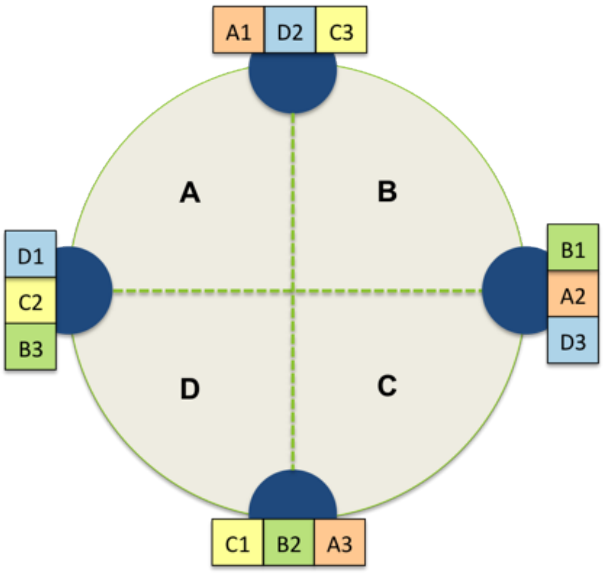

Figura 2.9: Simple Strategy replication applicata ad una ring di Cassandra

Eventual consistency

Cassandra sceglie un modello *eventual consistency*, facendo decidere al client che tipo di consistenza voglia essere utilizzato. Questa indica quanti nodi devono confermare l'avvenuta operazione prima che questa possa essere considerata conclusa, permettendo inoltre di controllare periodicamente la consistenza tra tutte le copie del dato richiesto presenti nella rete. Si può allora attribuire a questo attributo il valore `ONE`, in cui la conferma da parte di un solo nodo è necessario per considerare la transazione riuscita, `TWO`, in cui si attende la conferma di due nodi, e così via, oltre ad altre possibilità quali `ANY` per l'attesa di tutti i nodi della rete o `QUORUM`, effettuando quindi un majority voting tra i nodi dell'anello.

2.2.2 MongoDB

MongoDB[39] è un database sviluppato da MongoDB Inc., divenuto il principale database NoSQL utilizzato grazie alla sua estrema facilità di installazione e manutenzione. Questa semplicità non preclude però di ottenere ottime prestazioni da questo database, che grazie ad una campagna di vendita aggressiva e un supporto continuo, oltre alla fornitura di API in molti linguaggi di programmazione, continua la sua scalata verso le prime posizioni tra i database più usati, anche più di alcuni database relazionali.

Tipologia

MongoDB è un database document-oriented, in cui quindi l'unità fondamentale sono i documenti[41], equivalenti alle singole tabelle di un database relazionale ma molto più espressivi. Questi sono identificati tutti da una chiave speciale ed univoca, `_id`, che li identifica. A questi si aggiunge il concetto di *collection*, insieme di documenti, paragonabili a schemi. La possibilità di innestare documenti o di avere delle referenze tra essi rende molto potente, flessibile e dinamico questo modello.

Per quanto riguarda il CAP theorem, MongoDB utilizza le proprietà di partitioning e consistency. I dati non sono infatti disponibili nel momento in cui il nodo principale, detto primario e spiegato nel paragrafo successivo, non sia più disponibile, dovendo aspettare un suo recupero prima di poter accedere ai dati in esso contenuti.

Replica set

La replicazione in MongoDB avviene creando un *replica set*[11]. Questo è un gruppo di nodi in cui, tramite una votazione interna al set, uno di questi viene eletto come *primario*, mentre gli altri vengono catalogati come *secondari*. Questi ultimi memorizzano solo copie esatte degli stessi dati contenuti nella macchina primaria, permettendo, nel caso questa non dovesse essere più funzionante, di essere sostituita con una delle secondarie, scelta tramite una votazione effettuata dall'intero gruppo. Importante è sottolineare che, mentre

la macchina primaria può eseguire sia operazioni di scrittura che di lettura, le secondarie possono servire solo operazioni di lettura, portando MongoDB ad essere particolarmente efficace in applicazioni che fanno uso intensivo di read. Di default MongoDB permette comunque la sola lettura sul nodo primario, attribuendo ai nodi secondari una funzione finalizzata solo al recupero dati in caso di guasto.

Un replica set è vincolato ad avere una sola macchina primaria e al massimo fino a sei macchine secondarie, a cui possono essere aggiunte fino a cinque macchine che entrano in funzione solamente durante la fase di votazione per eleggere un nuovo nodo primario. Le macchine dedicate a questa funzione sono denominate *arbiter*, e al contrario delle macchine secondarie che possono diventare primarie, non cambiano mai il proprio stato.

Un esempio di replica set è mostrato in Fig.2.10, in cui sono presenti tutti ruoli.

Si noti che, al contrario di Cassandra, questo sistema permette quindi fino ad un massimo di sette replicazioni, limitazione dovuta al numero di macchine secondarie implementabili in un replica set.

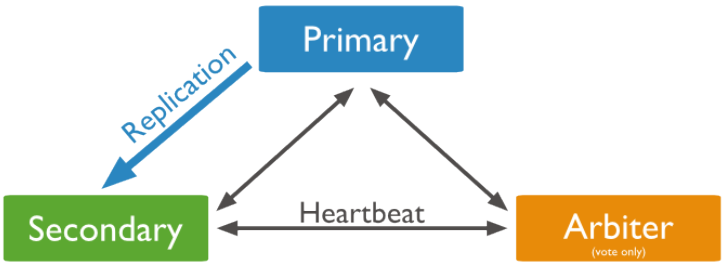

Figura 2.10: Replica set in MongoDB

Wire protocol e Heartbeat message

Tutti i nodi che compongono una implementazione di MongoDB comunicano attraverso un protocollo denominato *wire protocol*, che è sostanzialmente un protocollo TCP/IP semplificato. Questo consiste nel creare un contenitore con un header, al cui interno sono descritti il tipo di operazione da eseguire, la lunghezza del messaggio, il nome della collezione sulla quale eseguire la richiesta e un documento, generalmente in JSON, rappresentante l'informazione.

Un altro messaggio scambiato è il *heartbeat request*, richiesta inviata ogni due secondi tra ogni macchina per verificare lo stato del cluster. Altra interessante funzione di questa richiesta è verificare che il nodo primario possa raggiungere la maggioranza degli altri nodi e, in caso contrario, questo si auto-squalifica come primario della rete, procedendo a far partire una votazione tra i membri del cluster per una nuova elezione.

Sharding e processi

Per distribuire i dati ed eseguire il partizionamento di questi sui diversi nodi della rete, MongoDB utilizza il processo detto *sharding*. Ogni shard, il cui numero non è limitato, diventa così proprietaria di un certo numero di dati, i quali possono essere replicati rendendo la shard un replica set, come mostrato dall'esempio in Fig.2.11. Questo abilita l'interessante caratteristica di poter avere un numero di repliche diverso per ogni shard.

Per consentire il corretto funzionamento di questa struttura, devono essere eseguiti diversi processi, che possono essere eseguiti su una o più macchine.

- `mongod`[38]: mongod è il processo base per MongoDB. Esso può essere eseguito per avere un cluster di MongoDB composto da un solo nodo, oppure decidere di avviarne molti per creare un replica set. Questi processi sono in grado di comunicare tra loro ed eseguire votazioni per dichiarare il nodo primario, oltre alla possibilità di essere eseguiti con un particolare attributo per definirli arbiter.
  Questi processi sono quindi incaricati di eseguire le interrogazioni richieste dai client, compiendo le relative operazioni di lettura e scrittura in base al ruolo della macchina (primaria o secondaria).

- `config server`[37]: Nel momento in cui si decide di avere una rete multi shard, bisogna inserire due nuovi processi all'interno dell'architettura di MongoDB. Il primo tra questi è il *config server*, processo chiave dell'intero cluster. Questo contiene tutti i metadata riguardo a quale nodo contiene quale dato, diventando così un nodo estremamente importante per la vita del database. Esso è una particolare istanza del processo mongod ed è possibile, nell'intero cluster, istanziarne solamente uno o tre, per permettere un risultato sicuro ed immediato nel momento in cui debba svolgersi un majority voting, necessario nel caso di inconsistenza dati.

- `mongos`[40]: è il secondo processo necessario qualora si decida di creare più shard. Questo dispone di un elenco contenente tutti i nodi che compongono il database e il relativo replica set di appartenenza, dati recuperati e continuamente aggiornati attraverso la continua comunicazione con i config server. E' inoltre l'unico processo ad essere abilitato a comunicare con i client, reindirizzando la loro richiesta ai nodi delle varie shard e restituendo i risultati al richiedente del servizio. Non ci sono limiti sul numero di istanze possibili di questo processo, definendo quindi il numero di entry point per MongoDB uguale al numero di processi mongos istanziati.

Journal

Per ogni istanza del processo mongod viene creato, sulla stessa macchina, il suo rispettivo *journal*, sul quale vengono memorizzati l'esatta locazione sul disco, ed i relativi byte

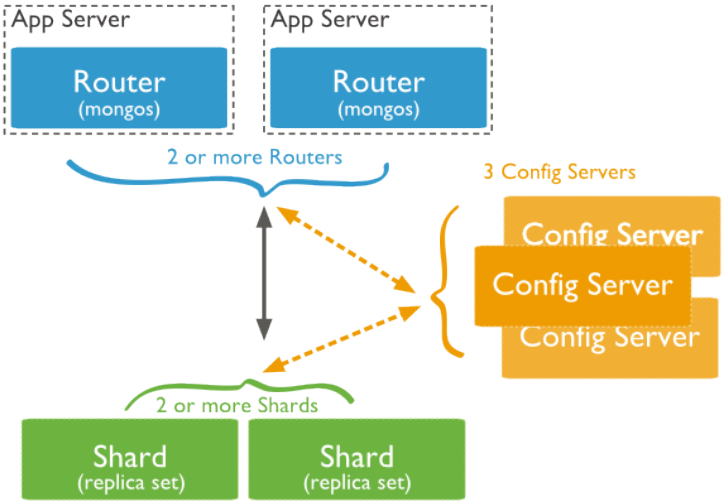

Figura 2.11: Sharding in MongoDB

cambiati, riguardo ai dati inseriti in seguito ad una operazione di scrittura. Questo permette, in seguito ad un crash di alcuni nodi della rete, di poter recuperare e rieseguire i cambiamenti svolti sul database dalle write che non sono state fisicamente scritte su disco. Le operazioni vengono infatti memorizzate su disco ogni 60 secondi (di default), perciò il journal necessita di memorizzare solo gli ultimi 60 secondi circa di quanto accaduto al database riguardo le scritture.

Write concern

Questa voce rappresenta quanta garanzia MongoDB fornisce all'applicazione client riguardo al successo di una operazione di tipo write. Ci sono diversi livelli di garanzia, sviluppati per favorire la scelta di quella che può essere considerata sufficiente per la propria specifica applicazione.

I livelli di write concern sono:

- `error ignored:` con questa configurazione, al client non viene mai dato errore, indipendentemente dalla tipologia di questo.
- `unacknowledged:` con questa modalità, il client viene reso consapevole di errori dovuti solamente a problemi di rete.
- `normal:` MongoDB fornisce al client una risposta sull'avvenuta ricezione dell'operazione di scrittura.

- `journaled:` con un journaled write concern il client viene avvisato della corretta esecuzione della scrittura solo una volta che il database abbia non solo eseguito l'operazione, ma anche aver eseguito una commit di essa sul journal.

- `replica acknowledged:` l'ultima possibilità offerta prevede un invio di avvenuta esecuzione della scrittura solo quando questa è stata eseguita sul nodo primario oltre ad essere stato replicato sull'intero replica set.

2.2.3 HBase

HBase è un database nato come clone di BigTable, un database fortemente distribuito e costruito sopra un file system apposito, anch'esso distribuito. Questa caratteristica necessita di applicazioni e tecnologie che forniscano sincronizzazione e gestione di un ambiente particolarmente dinamico. Il file system su cui HBase si appoggia è *HDFS*, uno dei principali componenti di Hadoop, un progetto sviluppato da Apache Software Foundation molto utilizzato per gestire grandi quantità di dati, cui si concede un paragrafo poichè punto essenziale per un funzionamento ottimale di HBase.

2.2.3.1 Hadoop

Hadoop[19] è un framework che permette di controllare e gestire insiemi di dati di dimensioni particolarmente elevati, fortemente orientato alla distribuzione di questi su cluster che presentino un alto numero di nodi. Il componente principale di questo framework è il file system distribuito HDFS *Hadoop Distributed File System*, capace di organizzare e distribuire i dati fornitogli dalle applicazioni che sono costruite su di esso, fornendo prestazioni di I/O particolarmente elevate.

Blocks

Così come qualunque file system, anche HDFS presenta questa unità fondamentale, in cui però questi sono particolarmente grandi rispetto ad un normale file system: la dimensione di default è di 64 MB o 128 MB[25], contro i 512 KB di un comune file system. La ragione per cui questi file sono estremamente grandi è data dal fatto che, essendo esso distribuito e finalizzato alla gestione di enormi quantità di dati, si vuole ridurre al massimo il tempo di seek time, cioè il tempo speso dalla testina del disco fisso per raggiungere l'inizio del blocco dati. Come vedremo, avere blocchi così grandi sarà di fondamentale importanza per le performance di HBase.

Architettura

Un cluster HDFS ha due fondamentali unità che devono essere correttamente configurate per il funzionamento del sistema: un *Namenode* e uno o più *Datanode*, rispettivamente il master e i workers del file system distribuito.

Il Namenode contiene tutti i metadata e l'albero del file system, memorizzandoli persistentemente sul disco fisso, oltre a conoscere tutti i datanode su cui sono memorizzati tutti i blocchi di un dato file. Quest'ultima informazione non è persistente, ma ricostruita comunicando direttamente coi datanode nel momento in cui il sistema viene avviato.

I Datanode sono coloro che si occupano di recuperare e memorizzare i blocchi, comunicando periodicamente il loro stato e la mappa dei loro dati direttamente al Namenode.

Il Namenode risulta essere quindi un punto critico per HDFS, poichè un suo malfunzionamento causerebbe l'interruzione dell'intero servizio. La soluzione più utilizzata per ovviare a questo problema è data dall'istanziare una terza unità, generalmente affidata ad una macchina diversa da quelle su cui girano Namenode e Datanode, denominata *Secondary Namenode*. Il nome non deve però trarre in inganno, perchè questa non si occupa di rimpiazzare il Namenode in caso questo abbia guasti o dare supporto ad esso. Il suo compito è invece quello, non meno importante, di tenere periodicamente traccia delle operazioni svolte dal master tenendo una copia del log operativo e di memorizzare il file system namespace. In caso di guasto, questa unità servirà per poter fornire al Namenode, una volta ripristinato, una lista di operazioni da rielaborare o da controllarne l'integrità.

Da notare che è possibile eseguire il Namenode, più Datanode e anche il Secondary Namenode, tutte su una stessa macchina, oppure distribuirle a piacimento su più nodi, soluzione sicuramente più affidabile ed usata. Un esempio di quest'ultima scelta è rappresentata in Fig.2.12.

Replicazione

HDFS è in grado di replicare i propri blocchi di dati su tutti i nodi da cui è composto. Di default il sistema è configurato per creare due copie del dato, distribuendone una sullo stesso nodo dove viene memorizzato il dato originale. Nel caso le singole macchine eseguano più istanze del processo datanode, la seconda copia va nello stesso nodo della prima, su un'istanza differente di datanode, altrimenti viene scelto un secondo nodo dove memorizzare la copia[46].

I vari nodi sono in grado di comunicare tra loro lo stato e la quantità dei propri dati, permettendo una distribuzione di questi lungo tutto il cluster. Alle applicazioni soprastanti queste operazioni appaiono completamente trasparenti.

L'analisi di Hadoop è stata svolta per permettere una più facile comprensione di alcune proprietà e di alcuni meccanismi di HBase, come verranno spiegati nella sezione seguente.

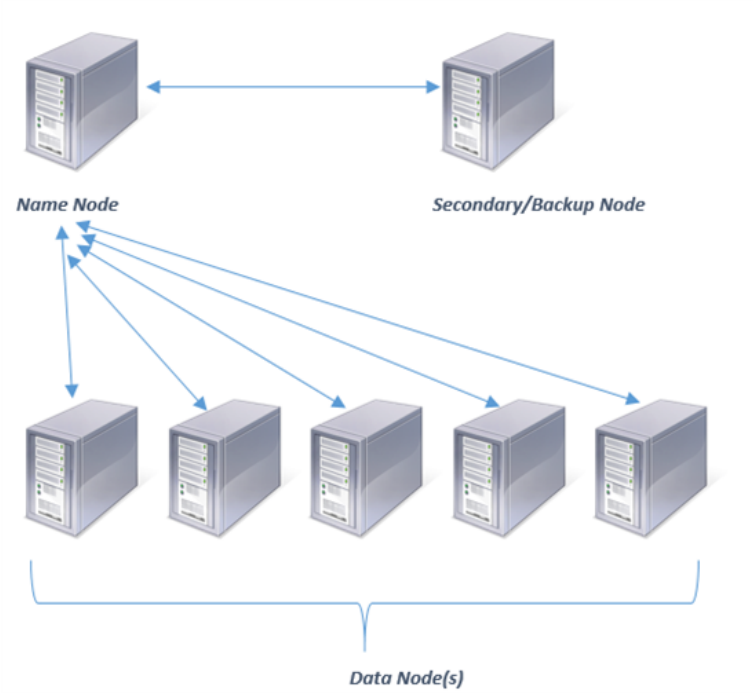

Figura 2.12: Esempio di un cluster HDFS

2.2.3.2 HBase

HBase[24] è un progetto della Apache Software Foundation nato dall'idea di ricreare il database di Google *BigTable* e che sta riscuotendo particolare successo negli ultimi tempi. Essendo infatti in grado di gestire cluster di notevoli dimensioni, si sposa adeguatamente alle tecniche di analisi necessarie vista la crescita esponenziale dei dati di questi anni.

Tipologia

HBase fa parte della famiglia column-oriented database. Al contrario di Cassandra però, qui non si hanno Super Column Family, concetto esclusivo di Cassandra. Particolarità interessante è però data dal fatto che HBase supporta esclusivamente *Ordered Partitioning*[26], implicando che le righe di una column family siano sempre memorizzate in ordine di row-key. Questo permette di interrogare il database con operazioni quali `scan` definendo un range di row-key, estraendone tutti i valori così trovati con delle performance particolarmente elevate. Si noti che i file di una column family possono essere sparsi su vari nodi, ma visto l'utilizzo di Hadoop, il quale presenta dei blocchi di dimensioni elevati, una operazione di scan su un range elevato di row-key permette di far uso di questi grossi blocchi, diminuendo la latenza data dal seek time.

Facendo riferimento al CAP theorem, come MongoDB, anche HBase presenta le proprietà di partitioning e consistency. Quando un nodo non è più riconosciuto dal cluster, i suoi dati non sono più accessibili. Si avvia quindi una fase di recupero, nel caso in cui dati siano stati precedentemente replicati, in cui HDFS e HBase cooperano per ripristinare i dati persi.

Regions

L'unità base in HBase è detta *region*, sostanzialmente un insieme di righe contigue memorizzate insieme riguardanti una tabella. La particolarità di queste è che il nodo che le contiene è in grado, una volta raggiunta una certa dimensione particolarmente grande di default, di eseguire uno *split* di esse, cioè un partizionamento in una o più regions di dimensioni minori, creando diverse regions per un'unica tabella. In questo modo si vengono a creare diverse regioni cui l'intero sistema si prenderà cura, bilanciando il carico sulla rete distribuendole lungo l'intero cluster. Questo metodo viene visto come una sorta di *auto-sharding* proprio per il suo automatico partizionamento dei dati. Si noti che la region iniziale risiede su un singolo nodo e che il partizionamento avviene gradualmente, generando carichi di lavoro non uniformemente distribuiti se non per carichi di dati estremamente elevati. Questo sarà un fattore molto importante ai fini dei test da noi eseguiti.

Due tabelle sono fondamentali per l'intero database ed il corretto funzionamento riguardante le regions: le tabelle `-ROOT-` e `.META.`. La tabella `-ROOT-` tiene traccia di dove sia memorizzata la tabella `.META.`, con informazioni relative ai nodi su quali risiede e sulle regions nei quali è partizionata. La tabella `.META.` è incaricata di contenere una lista sempre aggiornata sull'elenco di tutte le regions presenti su quali nodi. Queste due tabelle sono create ed assegnate automaticamente dal master di rete durante l'avvio dell'intero database.

Architettura

HBase si compone di due processi: HMaster, il master della rete, e uno o più HRegionServer, coloro che si occupano della gestione dei dati. Il primo si occupa di gestire l'assegnazione, per ogni HRegionServer, delle varie regioni nel momento in cui queste vengano create o partizionate, prendendosi cura anche della corretta gestione delle regions speciali `-ROOT-` e `.META.`. I secondi contengono invece le regions, gestendo la loro crescita e dimensione, oltre ovviamente a fornire le operazioni base di recupero e memorizzazione dei dati. Questi secondi processi possono essere eseguiti tutti su una macchina, oppure distribuiti su più nodi.

Zookeeper

A causa della possibilità di avere più HMaster, i quali sono il punto critico dell'intera infrastruttura di HBase e che possono essere anche spostati da una macchina all'altra, HBase necessita di un coordinatore esterno. Questa funzione viene fornita da Zookeeper[22], un servizio centralizzato di configurazione e sincronizzazione per grandi sistemi distribuiti. Questo si occupa quindi di tenere traccia dello stato dei master di HBase, prendersi carico di messaggi *heartbeat*, cioè di controllo stato, tra HMaster e HRegionServer e tenere traccia delle tabelle di `-ROOT` e `.META.`, fornendo dati per far comunicare direttamente il client con i nodi della rete. Si noti che, vista questa comunicazione diretta con gli HRegionServer, il numero di entry point della rete non è definibile: il client accederà direttamente al nodo su cui risiedono i dati a cui è interessato.

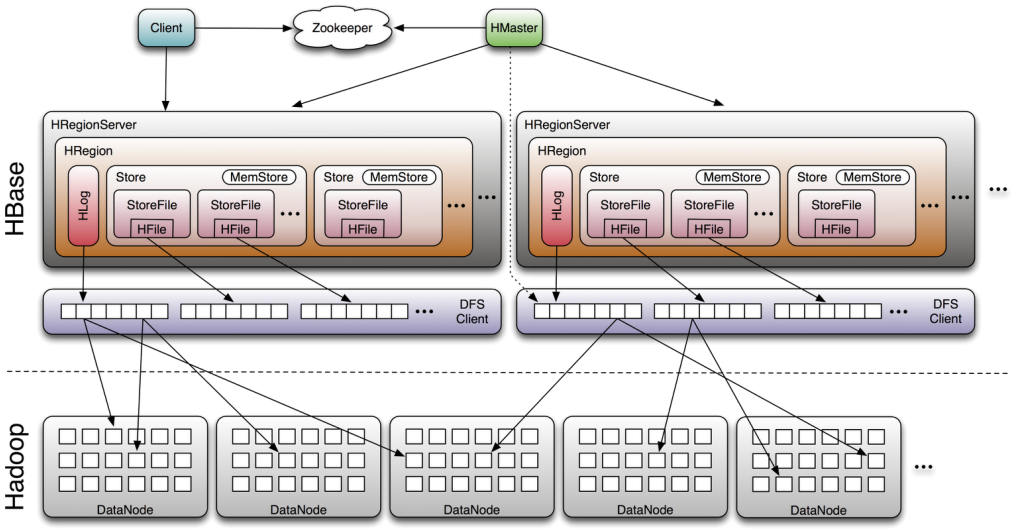

Figura 2.13: Cluster di HBase completo di tutte le sue componenti

Zookeeper è generalmente eseguito su un cluster esterno rispetto a quello dove avvengono i processi di HBase, per ragioni di sicurezza e di disponibilità. Il numero di istanze dei processi di Zookeeper deve inoltre essere dispari perchè, in caso di recupero dati o conflitti, un majority voting deve essere eseguito.

Un esempio di cluster completo di HBase che presenti tutti i componenti è mostrato in Fig.2.13.

Client-side write buffer

Una caratteristica estremamente importante per l'analisi di questo database è data dalla possibilità di abilitare sul client la funzione *client-side write buffer*. Questa permette di immagazzinare in un buffer tutte le richieste di tipo `put`, per essere poi inviate al

server tramite una sola richiesta RPC (*Remote Procedure Call*) solo quando il buffer ha raggiunto una certa dimensione. Come se non bastasse, il client diventa inoltre in grado di ordinare le richieste per HRegionServer, mandando una singola richiesta RPC per ogni singolo HRegionServer. Questo strumento sarà molto evidente sui risultati dei nostri test, portando anche a dover costruire i modelli secondo una specifica configurazione per rappresentare questa caratteristica. Il funzionamento di questo strumento è ben raffigurato in Fig.2.14

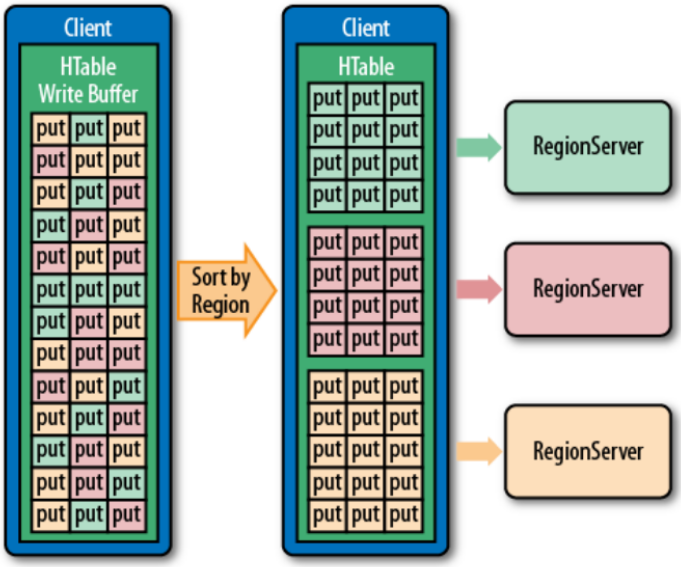

Figura 2.14: Funzionamento del client-side write buffer

Replicazione

La replicazione fornita da questo database permette di replicare dati esclusivamente tra diversi cluster di HBase: questa replicazione, asincrona, viene effettuata solitamente per avere una copia in caso di malfunzionamenti sul cluster principale.

Per avere una replicazione sul singolo cluster ci si appoggia invece ad Hadoop e al relativo HDFS, come visto nella rispettiva descrizione del file system. Per HBase questa replicazione è assolutamente ininfluente da un punto di vista prestazionale, poichè essa viene eseguita in modo del tutto trasparente al database. Il vantaggio di cui HBase è consapevole è dato dal fatto che, in caso di guasto ad una macchina, il database è conscio della presenza sul file system di copie dei dati andati perduti, permettendo quindi un processo di recupero dati.

Da notare infine che, al contrario di Cassandra, il numero di replicazione impostato tramite Hadoop non può essere configurato come superiore al numero di HRegionServer presenti nella rete.

2.3 Amazon Elastic Cloud Computing

Amazon Elastic Computing (EC2) è un sottoinsieme di tecnologie all'interno dell'ecosistema *Amazon Web Service* (AWS), la piattaforma cloud offerta da Amazon. Questa offre capacità computazionale dinamica su cloud, offrendo un'interfaccia grafica completa per la gestione delle proprie macchine virtuali.

E' allora possibile creare ed avere accesso, tramite rete, ad un numero molto elevato di macchine configurabili a piacimento, scegliendo l'hardware tra quelli messi a disposizione, il sistema operativo e facoltativamente anche la pre-installazione di software applicativi, quali database o programmi di business intelligence. Particolare da tenere in considerazione è la possibilità, una volta modificato lo stato della propria macchina con programmi personali, di salvarne lo stato e poterlo utilizzare per lanciarlo su una nuova istanza, con la possibilità di utilizzare un hardware diverso. Questo permette di avere molto velocemente, una volta configurata una singola istanza, di averne un numero elevato, uguali e con gli stessi identici dati. Si pensi per esempio al nostro caso specifico, in cui una installazione di un database NoSQL poteva essere lanciata sfruttando un unico stato salvato in cui fosse preconfigurato il database, avendo in poco tempo più nodi perfettamente funzionanti.

Questo modello di business offre la possibilità di effettuare un pagamento pay-per-use, cioè di pagare solamente l'utilizzo attivo di una macchina: nel momento in cui questa venga rimossa dal proprio account, viene considerato finito l'utilizzo del servizio.

Di seguito vengono illustrate le principali caratteristiche di questa piattaforma.

Availability Zone

Amazon EC2 è distribuito in tutto il mondo, dividendo le proprie regioni di operatività in 8 parti: `N. Virginia`, `Oregon`, `N. California`, `Ireland`, `Singapore`, `Tokyo`, `Sidney` e `Sao Paolo`. Ognuna di queste è indipendente e completamente separata dalle altre, permettendo quindi di impostare livelli di sicurezza, creare macchine e immagini di istanze personalizzate per ogni regione. Questa separazione viene effettuata a garanzia contro possibili imprevisti, poichè sviluppando la propria architettura su diverse regioni, si garantisce la possibilità di avere una maggior sicurezza contro i guasti: le probabilità che tutte le regioni di questa infrastruttura abbiano contemporaneamente un blocco sono davvero molto basse.

AMI

Una *Amazon Machine Image* fornisce le informazioni necessarie per l'avvio di una nuova istanza. Un AMI include le informazioni riguardo al sistema operativo corrente e le applicazioni salvate, oltre ai permessi applicati su di essa.

Amazon fornisce delle AMI di default, basati su diversi sistemi Linux o Microsoft, con la possibilità di avere alcuni software applicativi già installati. Queste immagini possono poi essere modificate, salvate e utilizzate successivamente durante la creazione di altre macchine, avendo a disposizione delle personali AMI da usare a piacimento. Amazon offre anche la possibilità, una volta personalizzate le proprie immagini, di renderle disponibili a pagamento ad altri utenti.

La Fig.2.15 mostra il ciclo di vita di una AMI: una volta registrata, cioè creata, questa può essere de-registrata, utilizzata per lanciare una nuova istanza o essere copiata.

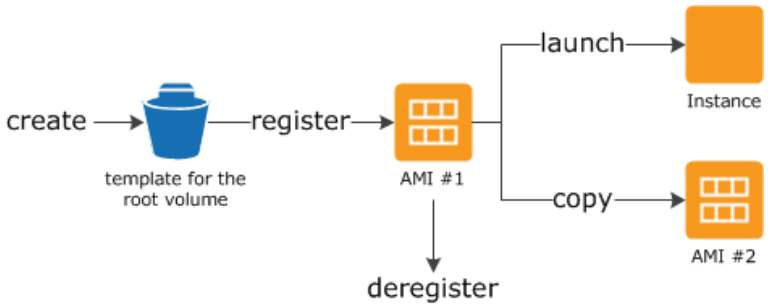

Figura 2.15: Ciclo di vita di una AMI

Instance type

Una volta scelta la AMI da utilizzare, si deve selezionare l'hardware su cui la si vuole eseguire. Ci sono quindi diverse *instance* tra cui scegliere[3], che variano per potenza di calcolo, quantità di memoria disponibile, dimensione del disco fisso o prestazioni di rete. Queste sono raggruppate per categorie come segue:

- `Micro instances:` comprende soluzioni economiche o parzialmente gratuite, scelte da chi vuole eseguire piccole applicazioni spendendo cifre contenute.

- `General purpose:` in questo gruppo sono raggruppate instances che presentano un buon compromesso tra potenza computazionale, memoria e risorse di rete, essendo consigliate per un gran numero di applicazioni.

- `Memory optimized:` la caratteristica principale di queste instances è la grande quantità a loro disposizione di memoria, diventando indispensabili per applicazioni che

necessitano di un elevato numero di GigaByte per questa risorsa.

- `Storage optimized:` il loro punto di forza è la grande capacità di memorizzazione fisica, con instances predisposte ad essere connesse a diverse soluzioni di dischi fissi o SSD, oltre a presentare una grande quantità di memoria.

- `Compute optimized:` l'ultimo gruppo di istanze disponibili sono ottimizzate per la computazione, offrendo un gran numero di virtual CPU e una elevata capacità di calcolo.

Il ciclo di vita di una istanza è definito dalla Fig.2.16. Questa può essere essere attiva, spenta, riavviata o terminata, cioè eliminata completamente dopo un'ora dall'immissione del comando. Gli stati evidenziati in giallo sono fasi di transizione tra uno stato e l'altro.

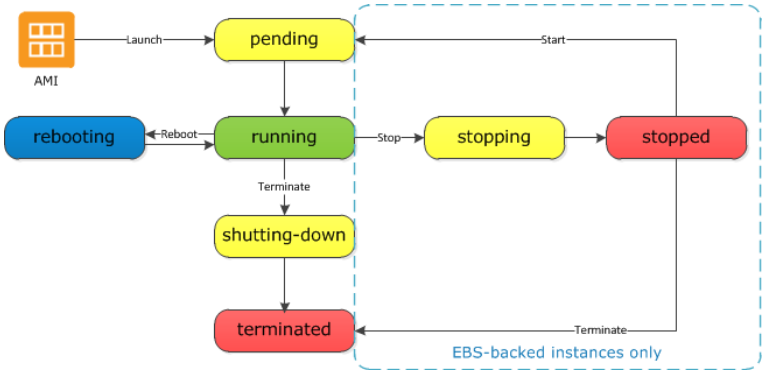

Figura 2.16: Ciclo di vita di una istanza

Si tenga conto di non confondere il termine `instance` con `istanza`: il primo termine verrà usato come particolare configurazione hardware offerta dalla piattaforma cloud, mentre il secondo sarà utilizzato per definire una singola entità, più generale, di un processo in esecuzione o di una macchina creata.

Security group

Un *security group* si comporta come un firewall presente sulla propria macchina, utile a consentire o meno il traffico in entrata e in uscita su di essa. E' possibile associare alla propria istanza più security group, in modo da avere già configurati i criteri di sicurezza all'avvio. Si possono quindi decidere quali protocolli sono abilitati (TCP, UPD, custom, etc.), la porta sulla quale è consentita una connessione e definire un range di indirizzi IP validi su di essa.

Come le AMI, anche i security group sono personalizzabili per poter essere riutilizzati.

Security keypairs

Per evitare l'accesso sulle proprie macchine da parte di sconosciuti, è obbligatorio creare un *keypair*, una coppia di chiavi crittografate da utilizzare per l'autenticazione durante la connessione, con protocollo SSH, alle proprie macchine. All'avvio di ogni nuova istanza bisogna definire a quale keypair questa sia associata, dal momento in cui è possibile disporre di diverse keypair, per permettere ad esempio a diversi utenti che condividano lo stesso account l'accesso ad un determinato numero di macchine.

Elastic IP address

Nel momento in cui una nuova istanza viene lanciata, ad essa viene attribuito un indirizzo IP privato ed uno pubblico. Questi indirizzi verranno quindi persi nel momento in cui la macchina verrà spenta, assegnando nuovamente degli indirizzi IP quando questa verrà riavviata. L'indirizzo privato viene utilizzato per la connessione tra macchine all'interno della stessa availability zone, mentre l'IP pubblico è usato per la connessione a quella istanza attraverso la rete internet.

La funzione di IP elastico permette di avere un IP statico associato ad un utente piuttosto che ad una istanza, permanente fino al suo rilascio esplicito. In questo modo è possibile lanciare macchine con assegnati esplicitamente degli indirizzi IP.

Load balancing

Questa funzione, se abilitata, permette di distribuire automaticamente il carico di lavoro tra le istanze attive e connesse tra loro, oltre ad individuare macchine guaste redirigendo il traffico verso istanze funzionanti. La configurazione permette di decidere rispetto a che soglia attivare o meno questo auto bilanciamento.

Autoscaling

L'*autoscaling* permette di modificare dinamicamente la capacità computazionale di una rete di macchine attiva. Si può quindi permettere al servizio di aumentare o diminuire la potenza di una macchina, arrivando ad aggiungere o togliere automaticamente dalla rete alcune macchine virtuali. Anche qui, come per il load balancing, è possibile configurare le soglie di attivazione. Notare che configurando correttamente questa impostazione è possibile avere sistemi capaci di sopportare carichi di lavoro poco al di sopra della richiesta attuale, per poi aumentare la capacità nel solo periodo richiesto dai client. Combinando questo al servizio di pagamento pay-per-use si possono avere, per alcuni tipi di applicazioni, dei consistenti risparmi economici.

2.4 Lavori precedenti

L'idea iniziale di questa tesi voleva concentrarsi sulla costruzione di modelli che potessero permettere una efficace modellazione dei database scelti. La scarsa disponibilità di risultati abbastanza dettagliati da poterci permettere la costruzione di modelli affidabili, ci ha visto costretti a svolgere dei test personalmente, finalizzati a fornire risultati precisi per il nostro scopo. Tra i lavori tenuti in considerazione, ai fini di avere una visione di come sono state svolte precedenti misurazione di performance, viene segnalato primo tra tutti quello pubblicato da *Yahoo*[13], in cui viene anche presentato il software di benchmarking utilizzato per i nostri test. Un secondo lavoro redatto da *Datastax*[14], società che tra i suoi prodotti annovera una versione personalizzata di Cassandra, oltre ad uno da parte di *Cubrid*[30] ed un terzo da Altoros[7], hanno permesso di analizzare punti critici dei test, configurazioni maggiorente usate e risultati con cui potersi confrontare.

Nonostante utili per vedere che strada seguire riguardo all'utilizzo di macchine e tecnologie, i dati offerti sono risultati incompleti sotto alcuni profili su cui questa tesi voleva concentrarsi. Parametri come latenze per singole operazioni, configurazioni dettagliate per ogni singola implementazione e variazioni di questi valori, sono in parte o del tutto mancanti in questi lavori, informazioni di cui questa tesi vuole invece fornire e tenere in considerazione, necessari inoltre per la costruzione di modelli che riescano a catturare in modo affidabile i trend prestazionali dei database.

2.5 Conclusioni

In questo capitolo sono state fornite le principali descrizioni dei database NoSQL presenti oggi sul mercato, andando a spiegare in maniera dettagliata le caratteristiche fondamentali dei database utilizzati per i test, oltre ad illustrare le possibilità offerte dalla piattaforma cloud. Queste sono state illustrate per permettere al lettore di comprendere al meglio quali strumenti sono stati studiati ed analizzati in fase preliminare di questo lavoro, mostrando anche l'esistenza di altri lavori che possono risultare di interessante lettura per avere una visione ancora più completa dell'ambiente dei database NoSQL.

Nel prossimo capitolo verranno introdotte le configurazioni specifiche per ogni tecnologia utilizzata, permettendo di comprendere meglio in quale ambiente di lavoro sono stati svolti i test.

Capitolo 3

Metodologie e impostazioni

Il lavoro che si vuole realizzare con questa tesi è quello di poter studiare i comportamenti prestazionali dei tre database scelti in maniera approfondita, fornendo una dettagliata descrizione delle impostazioni e delle scelte riguardo alle tecnologie utilizzate, eseguendo successivamente dei test per poter caratterizzare le prestazioni dei database in termini di throughput e latenze.

Una volta in grado di descrivere l'andamento delle performance in base al cambiamento delle configurazioni, si vuole verificare se sia possibile costruire dei modelli in grado di simulare in maniera abbastanza fedele questi comportamenti, rispettando con un certo grado di confidenza i risultati ottenuti al passo precedente. L'idea di base è comunque quella di costruire dei modelli semplici, permettendo un loro studio senza la necessita di avere calcolatori particolarmente potenti e fornire semplici strumenti su cui poter lavorare, senza l'obbligo di avere competenze tecniche specifiche riguardo.

I test che si andranno ad eseguire una volta configurati i database, si dividono principalmente per quattro proprietà. Una prima serie di test sarà eseguita sui database per valutare quanto la potenza hardware della macchina su cui essi vengono eseguiti impatta sulle loro prestazioni. Ulteriori test verranno eseguiti per osservare il comportamento dei database in base al numero di entry point del sistema, mentre altri saranno finalizzati all'osservazione di quanto il numero di nodi che compongono il sistema possa migliorarne le prestazioni. Infine verrà descritto l'andamento delle performance al variare del numero di replicazioni dei dati all'interno del database.

La precisa descrizione delle configurazioni viene ritenuta di particolare importanza, poichè offrire risultati senza le esatte impostazioni dei sistemi su cui i test sono stati effettuati, non permetterebbe un'analisi ed uno studio così preciso da rendere i dati ottenuti utilizzabili o rilevanti.

Vengono allora qui descritte le architetture e i parametri di configurazione utilizzati per ogni singolo database, illustrando le ragioni e i significati di tali scelte, oltre che

considerare prime supposizioni di come, al variare di suddetti valori, ci si aspetti che il database risponda.

Il cloud utilizzato necessita anch'esso di una spiegazione riguardo alle istanze scelte e delle variabili più influenti ai fini dell'esecuzione dei test.

Un'ulteriore sezione sarà dedicata al software utilizzato per eseguire le operazioni di benchmarking, spiegando esattamente come esso opera e, in base a questo, come sono stati operati i test e la raccolta dati.

Infine verrà introdotto il framework che ha permesso la progettazione e la creazione dei modelli, illustrando i principali strumenti di cui dispone e i componenti utilizzati per il nostro scopo.

3.1 Configurazione dei database

Mentre nel capitolo precedente è stata fornita una panoramica sulle principali caratteristiche dei database utilizzati, si vuole qui fornire una dettagliata descrizione degli attributi configurabili, per ogni database, rilevanti per l'esecuzione dei test. Si tenga presente che le installazioni e i comandi mostrati in questa sezione fanno tutti riferimenti ad un sistema operativo Linux e che gli attributi qui indicati sono una parte, seppur la principale, di tutte le impostazioni configurabili per ogni database.

3.1.1 Configurazione per Cassandra

La versione di Cassandra utilizzata in questa tesi è la 2.0.2, l'ultima rilasciata al momento dell'inizio di questo lavoro.

L'installazione di Cassandra è particolarmente semplice, poichè una volta scaricato la cartella contenente tutti i file, uno solo di questi risulta rilevante: il file `cassandra.yaml`, il solo file di configurazione utile per un corretto funzionamento del database. In questo file vengono definiti tutti i valori degli attributi riguardo all'istanza eseguita sulla macchina considerata.

Di seguito vengono elencati, in ordine all'interno del file cassandra.yaml, gli attributi più rilevanti all'interno del file di configurazione, nonchè quelli da noi ritenuti di particolare interesse per questo lavoro.

- `initial_token:` questo campo contiene il valore iniziale riguardo al range di token appartenenti al nodo. Ogni nodo contiene quindi un valore, che è anche il limite massimo (meno uno) del range di token posseduti dal nodo precedente. Si ricorda infatti la struttura ad anello di Cassandra: in presenza di due nodi, il primo con $initial\_token = 1$ e il secondo con $initial\_token = 10$, si avrà che il primo nodo possiederà i token da 1 a 9, mentre il secondo da 10 al massimo consentito, permettendo

una distribuzione dei dati attraverso una funzione di hash.

- `partitioner:` questo attributo identifica il modo in cui si vuole venga gestita l'assegnazione dei token. Due sono le principali modalità: RandomPartitioner e Murmur3Partitioner. Nel primo caso, gli initial_token vengono assegnati manualmente ad ogni nodo, secondo la regola indicata dalla seguente formula:

$$(2^{127}/number\_of\_nodes) * i) \ for\ i\ in\ range(number\_of\_nodes)$$

  Questa partiziona in modo bilanciato il range dei token assegnato ad ogni nodo. Si noti che questa formula è valida solamente dalla versione 1.2 in avanti, poichè precedentemente un token poteva assumere anche un valore negativo. Questa scelta viene preferita nel caso non si sappia esattamente il modello dati che verrà utilizzato sul database.
  Nel secondo caso i dati sono distribuiti secondo un algoritmo di hash automatico eliminando la necessità di assegnare manualmente i token. Nonostante questa modalità sia considerata più efficiente, al momento dell'inizio dei test non era ancora pienamente supportata, facendoci preferire la prima soluzione. All'attributo in questione è stato quindi assegnato il valore `org.apache.cassandra.dht.RandomPartitioner`.

- `seeds:` i seed sono gli indirizzi IP dei nodi che si occuperanno di comunicare lo stato attuale della rete ai nodi che si aggiungono ad essa, oltre ad essere i soli ad eseguire per primi il gossip protocol. La configurazione scelta è stata quella di inserire gli indirizzi IP di tutti i nodi presenti nella rete. Poichè questa funzione entra in gioco solamente alla creazione del cluster (i nodi sono avviati in serie) o alla modifica del sistema quando questo è operativo, non ha in nessun modo avuto impatto sui risultati ottenuti, favorendo maggior controllo da parte di tutti i nodi in fase di creazione dell'anello.

- `listen_address:` indica come i nodi si identifichino l'uno con l'altro ed è usato per tutte le comunicazione intra rete. Identificare ogni nodo col proprio indirizzo IP, sicuramente univoco, è sembrata una scelta appropriata.

- `rpc_address:` questo attributo identifica quali interfacce sono in ascolto per chiamate RPC da parte dei client o da parte degli stessi nodi della rete. E' possibile configurarlo a 0.0.0.0 abilitando qualunque interfaccia, cosa che è stata fatta perchè nei nostri test sapevamo a priori chi fosse il client e su quale interfaccia andasse ad eseguire le operazioni, senza doverci preoccupare di gestire più canali di comunicazione.

Una volta che tutti i nodi hanno il proprio file configurato, è possibile lanciare, su ogni nodo, il comando "`/CASSANDRA_HOME/bin/cassandra`" per eseguire l'avvio del database. All'avvio, viste le configurazioni date, tutti i nodi si parleranno tra loro, comunicando il proprio range di token ai restanti nodi e diventando operativi nel giro di pochi secondi. Essendo che in Cassandra non ci sono gerarchie tra nodi, ognuno di essi viene considerato uguale ad un altro senza dover specificare particolare impostazioni.

A questo punto è quindi possibile connettersi ad uno qualsiasi dei nodi e verificare lo stato della rete con il comando "`/CASSANDRA_HOME/bin/nodetool status -h ip_macchina`", oltre che avere una comoda percentuale indicante, per ogni nodo, il range di token da lui posseduto.

Una volta verificata l'integrità della rete ci si può collegare all'interfaccia del database tramite il comando "`/CASSANDRA_HOME/bin/cassandra-cli -h ip_macchina`" da dove sarà possibile creare i propri keyspaces e relative column families. Per una corretta esecuzione dei test, i seguenti comandi sono stati utilizzati nell'ordine qui proposto:

```
1] create keyspace usertable
    with placement_strategy ='org.apache.cassandra.locator.SimpleStrategy'
    and strategy_options = {replication_factor:1};
2] use usertable;
3] create column family data;
```

Il primo comando crea il keyspace che si andrà ad utilizzare: si noti che è in questo momento che viene specificata la strategia di replicazione ed anche il fattore di replicazione, permettendo di creare keyspaces con valori diversi per questi attributi. Il secondo comando viene usato per accedere al keyspace appena creato, mentre l'ultimo crea la column family che sarà utilizzata come riferimento dal software di benchmark.

Si tenga presente che, per tutti i test effettuati, il software di benchmark utilizzato imposta come eventual consistency un valore di default pari a `ONE`.

In questo momento il cluster di Cassandra è perfettamente funzionante e pronto per essere utilizzato.

3.1.2 Configurazione per MongoDB

MongoDB è stato utilizzato nella sua versione 2.4.5, l'ultima considerata stabile all'inizio di questo lavoro. Nonostante la configurazione della singola istanza sia più semplice rispetto a Cassandra, questo database ha bisogno di una procedura specifica per abilitare la creazione di shard, dovuta alla necessità di creare istanze gerarchicamente diverse.

La configurazione scelta inizialmente per questo database era data da un numero variabile

di nodi sui quali venivano eseguiti i processi mongod. In aggiunta a questi erano installate altre due macchine, dedicate una al deploy del processo config server e una al processo mongos. Il numero di mongod era considerato il numero utile ai fini del test: ad esempio, se il comportamento del database voleva essere studiato per un numero di nodi uguale a quattro, si dovevano aggiungere due ulteriori nodi dedicati ai due processi sopra elencati. Questa decisione è stata presa poichè, per un gran numero di nodi, l'aggiunta di due ulteriori macchine non sarebbe stata rilevante per l'impatto economico o implementativo del database. Inoltre, si pensava inizialmente che il lavoro dei processi mongos e config server fosse trascurabile ai fini della gestione del database stesso, ritenendo che questi entrassero in funzione solo in maniera estremamente marginale e non rappresentassero un possibile collo di bottiglia. Dopo l'esecuzione dei primi test si è però visto che, mentre l'utilizzo del config server era effettivamente trascurabile da un punto di vista prestazionale, il processo mongos si rivelava essere il collo di bottiglia dell'intero sistema.

A questo punto si è pensato di aggiungere un numero maggiore di processi mongos, ognuna installata su macchine diverse, per permettere ai processi di cooperare e distribuirsi il carico di lavoro. Questo avrebbe però comportato l'aggiunta di una notevole quantità di macchine dedicate a questo processo. Ciò non avrebbe consentito una comparazione adeguata con gli altri database, poichè il numero di macchine dedicate ai processi mongos sarebbe cresciuta significativamente al crescere del numero dei nodi dedicati ai processi mongod, dovendo installare molte più macchine rispetto a quante il test ne avrebbe dovuto effettivamente validare.

La soluzione adottata, mostrata in Fig.3.1, rappresenta un compromesso tra la prima e la seconda soluzione. E' stato infatti deciso di eseguire, su ogni singola macchina, un processo mongos e un processo mongod, in parallelo, mentre il processo config server, essendo irrilevante ai fini di utilizzo della CPU, sarebbe stato eseguito su un nodo scelto in maniera casuale. Questo ha permesso di avere un numero di entry point pari al numero dei nodi, mantenendo comunque un numero di macchine uguale a quello considerato utile ai fini del test.

Le singole istanze del processo mongod sono state lanciate tramite il comando `mongod` seguito dai seguenti parametri:

- `--dbpath:` percorso ad una cartella contenente i file temporanei, e non, del database.
- `--replSet:` nome identificativo del replica set a cui l'istanza appartiene, necessario per definire diversi replica set.
- `--bind_ip:` indirizzo IP associato all'istanza.
- `--port:` porta su cui mongod è in ascolto. Se ci sono diverse istanze in esecuzione sulla stessa macchina, le porte devono essere ovviamente diverse.

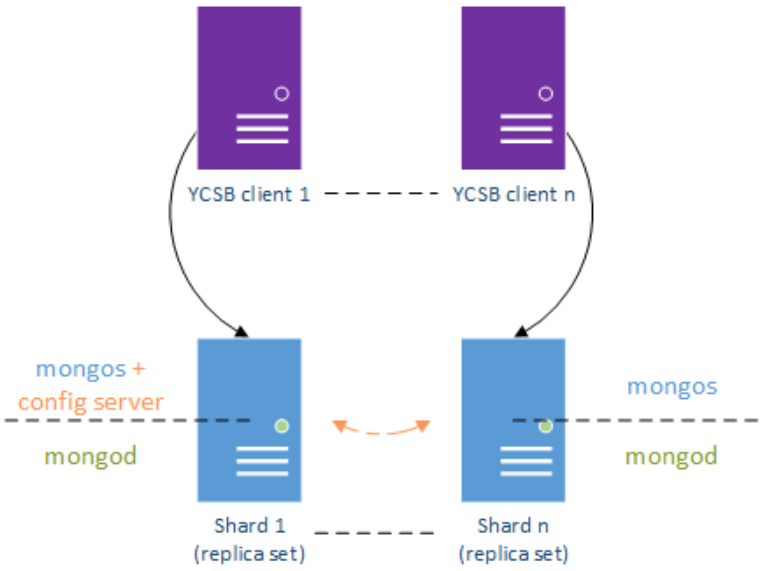

Figura 3.1: Schema della configurazione adottata per MongoDB

- `--logpath:` percorso in cui viene salvato il log dell'istanza corrente.

Una istanza del processo config server è stata invece lanciata sempre con il comando `mongod`, seguito dall'opzione `--logpath` con relativo percorso e l'attributo `--configsvr`, necessario per eseguirlo come config server.

Infine le istanze del processo mongos sono state lanciate attraverso il comando `mongos` specificando l'attributo `--configdb`, seguito in ordine uguale per tutte le istanze mongos eseguite dall'elenco degli indirizzi IP dei config server.

I test su questa configurazione sono stati svolti impostando il numero di shard al minimo possibile o al massimo consentito, in quest'ultimo caso uguale al numero di nodi. Si è così osservato l'andamento delle prestazioni avendo uno o tutti i nodi abilitati alla esecuzione di operazioni di scrittura. Si noti che la configurazione scelta non riserva alcuna macchina per l'allocazione di processi mongod che fungano da arbiter, perchè non utili ai fini dei risultati cercati. Infine, in tutti i test è stato mantenuto il valore di default del `write concern`, configurato uguale a `normal`.

Ogni cluster è stato avviato nel seguente modo: è stato eseguito un processo config server su un nodo, seguito da tutti i processi mongod su ogni rispettiva macchina per permettere ai vari componenti di poter interagire tra loro. Una volta configurati i replica set opportuni, sono stati avviati i vari mongos cui uno, scelto a caso, utilizzato per poter sincronizzare

le varie shard. Per controllare la corretta installazione, ci si è collegati all'interfaccia fornita da MongoDB contattando un qualsiasi processo mongos, interrogando la tabella di sistema e verificando la configurazione.

A questo punto il database è correttamente configurato e disponibile a gestire le richieste dei client.

3.1.3 Configurazione per Hadoop & HBase

La configurazione per HBase e del file system offerto da Hadoop è molto più articolata rispetto ai due precedenti database. Per una corretta esecuzione bisogna infatti avere dapprima un cluster funzionante di HDFS, avere un cluster attivo di Zookeeper, e solo allora avviare il database, il quale deve essere configurato in modo da comunicare correttamente con i due componenti appena citati. Poichè a differenza degli altri database Hadoop comunica internamente attraverso il protocollo SSH, si è dovuto innanzitutto creare una chiave pubblica da distribuire su ogni macchina, al fine di abilitare la comunicazione attraverso questo protocollo tra le diverse macchine, funzione altrimenti non disponibile poichè considerata non sicura. Inoltre, a causa delle impostazioni di default per cui Hadoop identifica i nodi della propria rete attraverso la tupla `hostname` e indirizzo IP, è necessario modificare anche su ogni macchina il file `/etc/hosts`, inserendo l'elenco delle tuple di ogni nodo, cosa che ha creato non pochi problemi dal momento in cui questi dati sono ogni volta assegnati dinamicamente dalla piattaforma cloud. Sono stati infatti creati degli script bash appositi per l'automatica allocazione delle tuple su ogni singola macchina, necessari anche per la modifica automatica dei file di configurazione.
Anche per questo database è stato fatto un ragionamento simile a quanto fatto inizialmente per MongoDB. Sulle macchine considerate operative sono stati installati i Datanode (Hadoop) e gli HRegionServer (HBase), il cui numero era uguale a quello utile per la valutazione del test. A queste macchine ne son state aggiunte due, una per l'installazione del Namenode, del SecondaryNamenode (Hadoop) e del HMaster (HBase), e una per l'istanza di Zookeeper. La supposizione era che i due nodi aggiuntivi utilizzassero solo una irrilevante percentuale delle risorse di sistema, cosa che in questo caso si è dimostrata vera. L'idea viene supportata anche dal fatto che, come spiegato nel capitolo 2.2.3.2, HBase consente comunicazione diretta tra gli HRegionserver e i client, al contrario di MongoDB, in cui i gestori del sistema, i processi mongos, sono sempre interpellati per qualsiasi transazione. Ancora, grazie al client-side write buffer abilitato, il client comunica con l'istanza di Zookeeper poche volte, visto che le richieste sono accumulate nel buffer e inviate solo una volta tutte insieme. Si nota infine che l'istanza di Zookeeper utilizzata è solo una, poichè averne di più non sarebbe stato utile ai fini dei risultati. La configurazione finale utilizzata ai fini dei test è mostrata in Fig.3.2.

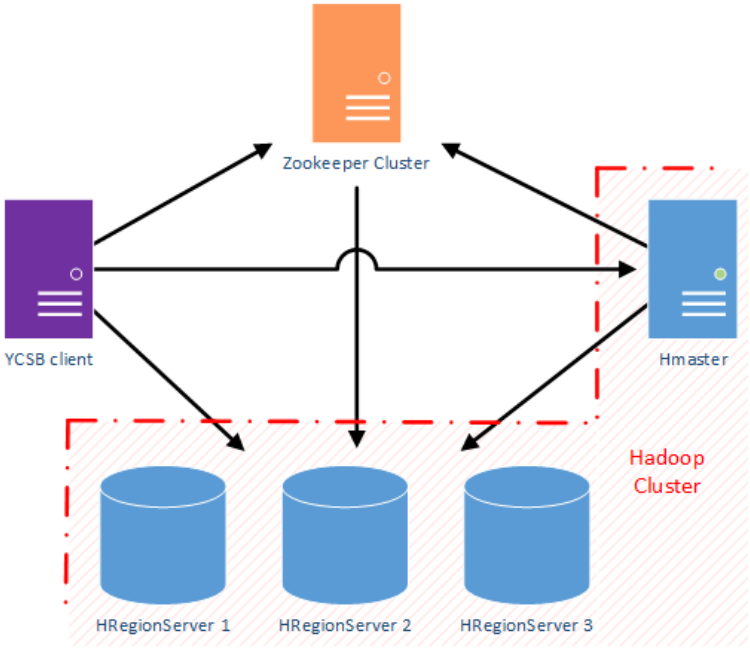

Figura 3.2: Configurazione utilizzata per HBase in fase di test.

Oltre a tutti file di configurazione impostati per un corretto funzionamento del database, un file è stato importante per poter ottenere interessanti risultati. Il file `hdfs-site.xml` consente infatti di stabilire quante repliche Hadoop dovrà svolgere riguardo i blocchi che andrà a memorizzare.

Per verificare che tutto sia corretto, sia Hadoop che HBase presentano due interfacce web che mostrano lo stato dell'intero sistema, verificando che ogni nodo sia correttamente inizializzato. HBase consente anche l'utilizzo di un interfaccia testuale per poter controllare l'allocazione di ogni singola region rispetto agli HRegionServer. Una volta che il database è attivo, è possibile accedere all'interfaccia di HBase collegandosi al master per creare la tabella e la column family utilizzate dal client, eseguendo il comando "`create 'usertable', 'family'`".

HBase è finalmente pronto per essere testato.

3.2 Configurazione della piattaforma Cloud

La piattaforma cloud, come visto nel capitolo 2.3, offre molte possibilità riguardo alle configurazioni possibili. Si vuole qui descrivere le principali funzionalità tenute in considerazione per ottenere risultati utili.

Tabella 3.1: Specifiche dell'hardware utilizzato

Instance Type	vCPU	Physical Processor	Memory(GiB)
`m1.large`	2	Intel Xeon Family	7.5
`c1.xlarge`	8	Intel Xeon Family	7
`c3.2xlarge`	8	Intel Xeon E5-2680 v2	15

Instance

Le instance utilizzate sono state scelte tra quelle appartenenti alle categorie *general purpose* e *compute optimized*. Per avere dei risultati migliori, per tutti i test tranne che per quelli inerenti l'impatto del numero di core, in cui si aveva bisogno di macchine con un alto numero di virtual CPU, sono state utilizzate macchine di tipo `m1.large`, le cui principali caratteristiche hardware sono elencate in Tab.3.1.

Questa macchina è stata scelta per la quantità di memoria disponibile, poichè per HBase è consigliata una configurazione che comprenda 7 GiB di RAM, visto che Hadoop utilizza da solo, di default, 1 GiB di questa. Si consideri che, per una più omogenea caratterizzazione dei test, tutte le macchine utilizzavano lo stesso tipo di instance indipendentemente dal ruolo che ricoprivano all'interno della rete.

Per i test in cui si necessitava di una macchina con più core è stata utilizzata l'istanza di tipo `c1.xlarge`, che permette di avere otto core virtualizzati.

Il client è invece stato continuamente osservato, poichè per alcune configurazioni diventava lui stesso il bottleneck del sistema. Questo comportamento non voleva essere considerato perchè si voleva essere in grado di stressare il database portandolo a saturazione. E' allora stato continuamente monitorato per verificare che l'utilizzo delle sue risorse non superasse mai il 50/55%, caso in cui veniva ripetuto l'intero test istanziando il client su un'istanza con maggiore potenza. In base ai casi, si è quindi utilizzata un'istanza `m1.large`, passando all'istanza `c1.xlarge` nel momento in cui l'utilizzazione delle risorse del client superavano la soglia accettata, arrivando ad utilizzare, nei casi con un numero molto alto di nodi o con molti core attivi, ad istanziare il software di benchmarking su una macchina di tipo `c3.2xlarge`.

AMI

Per avere una maggiore velocità nell'esecuzione dei test sono state create in precedenza delle AMI prevalentemente di due tipi, tutte configurate partendo da un sistema operativo *Ubuntu Server 12.04*[35]. Un'immagine è stata creata con installati tutti i database preconfigurati, permettendo così di avere subito macchine pronte su cui avviare il database desiderato. Una seconda serie di immagini sono state create per il client, ognuna per

ogni database per una questione organizzativa, con installata una versione del software di benchmarking pronta ad eseguire test per il database interessato.

Security Group

Per permettere ai vari nodi di comunicare tra loro, ogni database ha bisogno di utilizzare delle porte specifiche che devono essere abilitate esplicitamente creando un security group. Le porte abilitate sono elencate di seguito in base al servizio che le utilizza. Tutte le porte sono state attivate rispetto a qualunque indirizzo IP (0.0.0.0/0).

- `SSH:` 22
- `HTTP:` 80
- `HTTPS:` 443
- `Various:` 1024 - 1099
- `Zookeeper:` 2181, 2888, 3888
- `Cassandra:` 700, 7199, 9160
- `MongoDB:` 27017, 27018, 27019
- `Hadoop:` 50000, 50010, 50030, 50070, 51721, 54310, 54311
- `HBase:` 60000, 60010, 60020, 60030

Varie

Nonostante siano state create delle AMI con i database preconfigurati, alcuni parametri dovevano essere settati in base al test che si voleva effettuare. Questo significava andare a modificare, manualmente e su ogni macchina, i singoli file di configurazione del database. Un così lento procedimento non avrebbe permesso l'esecuzione di tutti i test cui avevamo bisogno in un tempo ragionevole. Sono stati allora creati ad hoc degli script *bash*, preparati e caricati sulle AMI del client, che hanno permesso di configurare i parametri principali di ogni database, come numero nodi o replication factor, e di avviarlo nel modo più automatico possibile.

Altri script sono stati sviluppati per effettuare test ripetuti, collezionare i risultati e inviarli alla macchina locale su cui venivano salvati per essere successivamente analizzati.

Altra nota da osservare riguarda l'instabilità architetturale della piattaforma cloud di Amazon. Nonostante durante tutta la durata di benchmarking non sono stati riscontrati particolari problemi, si è notato personalmente come i test effettuati durante i giorni lavorativi presentassero latenze più variabili rispetto a quelli svolti durante il fine settimana. Questo ha costretto ad eseguire set di benchmark durante la stessa finestra temporale per i test che necessitavano una comparazione tra loro.

3.3 Software di benchmarking: YCSB

Per svolgere i test e riuscire a collezionare dati utili per lo studio delle prestazioni dei database è stato usato un software di benchmark sviluppato da Yahoo!, chiamato *Yahoo! Cloud Serving Benchmark* (*YCSB*). Questo tool permette di eseguire operazioni di insert, update e read, su una vasta gamma di database e di ottenere delle misurazioni sulle prestazioni. La versione rilasciata al momento dell'inizio dei test è la numero 0.1.4, che ha dovuto però essere integrata con alcune librerie aggiuntive poichè non aggiornata alle più recenti versioni di Cassandra, MongoDB ed HBase.

L'utilizzo di uno strumento come questo ha permesso di testare i nostri database tutti con lo stesso metodo e con lo stesso tipo di dato inserito, in modo da poterci concentrare sull'architettura del database.

Il tool è scritto in Java sfruttando le API supportate in questo linguaggio dai tre database. Una volta impostato il tipo di test che si vuole eseguire e il database che si vuole utilizzare, configurando tutti i parametri del caso, il programma invia le operazioni da eseguire al database selezionato, riportando alcune statistiche utili come numero di operazioni al secondo e latenze medie per ogni singola operazione, con la possibilità di salvarle direttamente su file. Ogni singola istanza del client è stata avviata su una macchina diversa rispetto a quelle utilizzate per l'infrastruttura del database, per evitare che l'utilizzo delle risorse destinate al client non andasse ad impattare le misure rilevate dal benchmark. L'architettura del tool utilizzato è mostrata in Fig.3.3

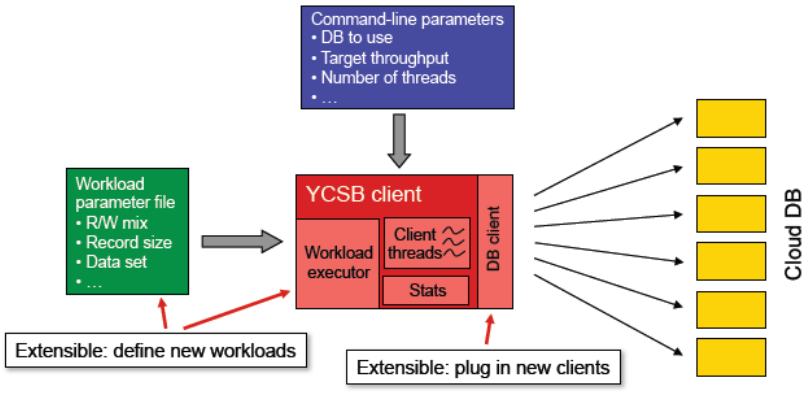

Figura 3.3: Architettura del tool YCSB

Il workload selezionato per effettuare i test, mantenuto sempre uguale per avere omogeneità nei risultati, è composto da un numero configurabile di operazioni, qui impostato come 50% in task di tipo update e 50% di task di tipo read. Il data set utilizzato è quello di default, configurato come, per ogni singolo dato, una chiave `user` seguita da un numero casuale e dieci attributi, questi ultimi identificati attraverso un nome numerico e un valore alfanumerico.

Le opzioni generiche da assegnare all'avvio del tool comprendono infine il numero di thread che si vuole adoperare lato client, il numero totale di operazioni da eseguire e gli indirizzi delle macchine che fungono da entry point del sistema da testare.

Riguardo il numero di operazioni da eseguire è stata posta particolare attenzione per accertarsi che il tool fosse eseguito per abbastanza tempo. Un test troppo corto non avrebbe consentito di osservare il comportamento a regime del database, mentre un test troppo lungo non avrebbe dato informazioni aggiuntive, sprecando le risorse a nostra disposizione.

Il numero di entry point, oltre ad essere soggetto di una tipologia dei test svolti, ha portato a dover creare più macchine client nel caso di MongoDB. Mentre per Cassandra è possibile definire più indirizzi IP per una singola istanza del tool, e per HBase, a causa della comunicazione diretta instaurata tra client e HRegionServer, questo parametro risulta irrilevante, per MongoDB è possibile definire un solo indirizzo IP di un processo mongos per tool, di fatto imponendo al database un solo entry point. Per ovviare a questo problema sono state istanziate, per MongoDB, diverse macchine client, una per ogni istanza di mongos. Lo sviluppo su più macchine è stato fatto per evitare conflitti di interfacce delle singole istanze e per non rischiare di saturare il client. Essendo il nostro obbiettivo quello di saturare il database, avere uno o più client non inficia la qualità e la consistenza dei risultati.

Ogni singolo test si divide in due fasi.

La prima serve ad inserire un numero rilevante di dati per essere in presenza di un database non vuoto, cosa che non rappresenterebbe una situazione generica. Una considerazione da fare per questa fase è riguardo la stabilità dei database. Mentre Cassandra e MongoDB, salvo alcuni rari casi, non hanno mai avuto inconvenienti durante il caricamento dei dati, HBase ha riscontrato parecchi problemi nel momento in cui si caricavano grandi quantità di dati attraverso un numero elevato di thread. Questo deriva dall'utilizzo del client-side write buffer e dall'invio della richiesta RPC nello stesso momento in cui una region sta venendo divisa in due. Questo causa un conflitto poichè, mentre il write buffer ha ordinato le operazioni in base al HRegionserver su cui eseguirle, una region può essere stata spostata su un altro HRegionserver, a seguito di uno split di questa. In questo caso il database necessita, una volta finito lo split di una regione, di attivare una comunicazione aggiuntiva col client ricomunicandogli le posizione aggiornate delle region, operazione che può richiedere parecchi secondi.

La seconda fase consiste nell'esecuzione vera e propria del test. Una volta lanciato il comando, il software esegue una sequenza casuale di operazioni di update e di read attendendo per ognuna di esse la risposta da parte del database, fino al raggiungimento del numero di operazioni voluto. A questo punto il test fornisce una serie di risultati in forma di numero operazioni al secondo medie, latenze medie per ogni tipologia di

operazione e un sommario di questi valori per ogni dieci secondi in cui il test è stato eseguito, permettendo così di controllare che non ci siano stati errori o comportamenti anomali durante l'esecuzione del test.

Un esempio di output fornito dal tool è quello mostrato in Fig.3.4, relativo alla esecuzione di un test su cassandra.

```
YCSB Client 0.1
Command line: -db com.yahoo.ycsb.db.CassandraClient10 -p hosts=10.249.5.33,10.225.152.70 -p operationcount=500000 -threads 16 -s -P YCSB_complete/
  workloads/workloada -t
Loading workload...
Starting test.
 0 sec: 0 operations;
 10 sec: 41571 operations; 4143.84 current ops/sec; [UPDATE AverageLatency(us)=3223.36] [READ AverageLatency(us)=4366.67]
 20 sec: 84268 operations; 4268.42 current ops/sec; [UPDATE AverageLatency(us)=3165.68] [READ AverageLatency(us)=4313.59]
 30 sec: 127495 operations; 4322.27 current ops/sec; [UPDATE AverageLatency(us)=3109.82] [READ AverageLatency(us)=4281.91]
 40 sec: 169947 operations; 4244.78 current ops/sec; [UPDATE AverageLatency(us)=3176.01] [READ AverageLatency(us)=4344.12]
 50 sec: 213646 operations; 4369.9 current ops/sec; [UPDATE AverageLatency(us)=3113.64] [READ AverageLatency(us)=4195.84]
 60 sec: 256734 operations; 4308.37 current ops/sec; [UPDATE AverageLatency(us)=3087.39] [READ AverageLatency(us)=4318.14]
 70 sec: 300166 operations; 4343.2 current ops/sec; [UPDATE AverageLatency(us)=3081.45] [READ AverageLatency(us)=4271.83]
 80 sec: 341692 operations; 4152.18 current ops/sec; [UPDATE AverageLatency(us)=3314.01] [READ AverageLatency(us)=4373.96]
 90 sec: 384834 operations; 4314.2 current ops/sec; [UPDATE AverageLatency(us)=3154.92] [READ AverageLatency(us)=4254.8]
 100 sec: 427428 operations; 4258.97 current ops/sec; [UPDATE AverageLatency(us)=3171.47] [READ AverageLatency(us)=4336.39]
 110 sec: 467233 operations; 3980.5 current ops/sec; [UPDATE AverageLatency(us)=2931.77] [READ AverageLatency(us)=4020.92] [CLEANUP AverageLatency
  (us)=70.83]
 120 sec: 498298 operations; 3106.19 current ops/sec; [UPDATE AverageLatency(us)=2670.04] [READ AverageLatency(us)=3586.47] [CLEANUP
 AverageLatency(us)=60.29]
 124 sec: 500000 operations; 414.41 current ops/sec; [UPDATE AverageLatency(us)=4186.05] [READ AverageLatency(us)=4361.92] [CLEANUP AverageLatency
  (us)=83.67]
[OVERALL], RunTime(ms), 124147.0
[OVERALL], Throughput(ops/sec), 4027.4835477297074
[UPDATE], Operations, 250171
[UPDATE], AverageLatency(us), 3113.8042698794025
[UPDATE], MinLatency(us), 887
```

Figura 3.4: Esempio di output restituito dal tool YCSB

Questa seconda fase, per ogni singolo test, è stata eseguita venti volte. Questo è stato fatto per evitare di considerare le prime esecuzioni del test, anche dette *esecuzioni a freddo*, che non avrebbero rappresentato correttamente un database utilizzato a regime e costantemente. Si è infatti visto che per la maggior parte dei test, le prime tre o quattro esecuzioni mostravano delle performance altalenanti, salvo poi stabilizzarsi a valori stabili per tutte le successive esecuzioni. I valori iniziali ed i valori particolarmente fuori norma, dovuti ad un calo di banda all'interno della rete o processi improvvisi del sistema operativo, sono stati eliminati per avere una media solo tra i dati considerati rappresentativi.

Tramite alcuni script bash, la macchina client era in grado di eseguire in serie il numero voluto di test, collezionare i dati e inviarli al computer da noi usato per memorizzare e in seguito trattare i dati. Inoltre, sempre tramite alcuni script, il client è stato in grado di collezionare dati riguardo l'utilizzazione della propria CPU tramite il comando "`top`" preinstallato sul sistema operativo, necessario per controllare che il client non saturasse, diventando il collo di bottiglia del sistema invalidando così i risultati.

Impostazioni particolari del tool per ogni database, riguardo alla fase di run, saranno mostrate nel capitolo successivo, dove verranno analizzati in dettaglio i test e i loro risultati.

3.4 Progettazione dei modelli: Java Modelling Tools

La creazione dei modelli è stata affidata ad uno strumento progettato e sviluppato dal Politecnico di Milano, il cui sviluppo è iniziato nel 2002 e da allora continuamente migliorato. Il *Java Modelling Tools* (JMT)[5] è un framework contenente diversi tool, sviluppati in Java, utile per l'analisi prestazionale di sistemi informatici utilizzando modelli basati sulla teoria delle code[34].

La teoria delle code si occupa di studiare i sistemi in cui siano presenti dei processi attivi in grado di elaborare, con determinate tempistiche, delle richieste, le quali viaggiano all'interno del sistema avendo la possibilità di finire in linee di attesa, o code, in attesa di ricevere il servizio. Questa è molto utilizzata nell'informatica poichè permette caratterizzazioni e simulazioni molto accurate di sistemi multi-tier, sistemi distribuiti o sistemi di reti.

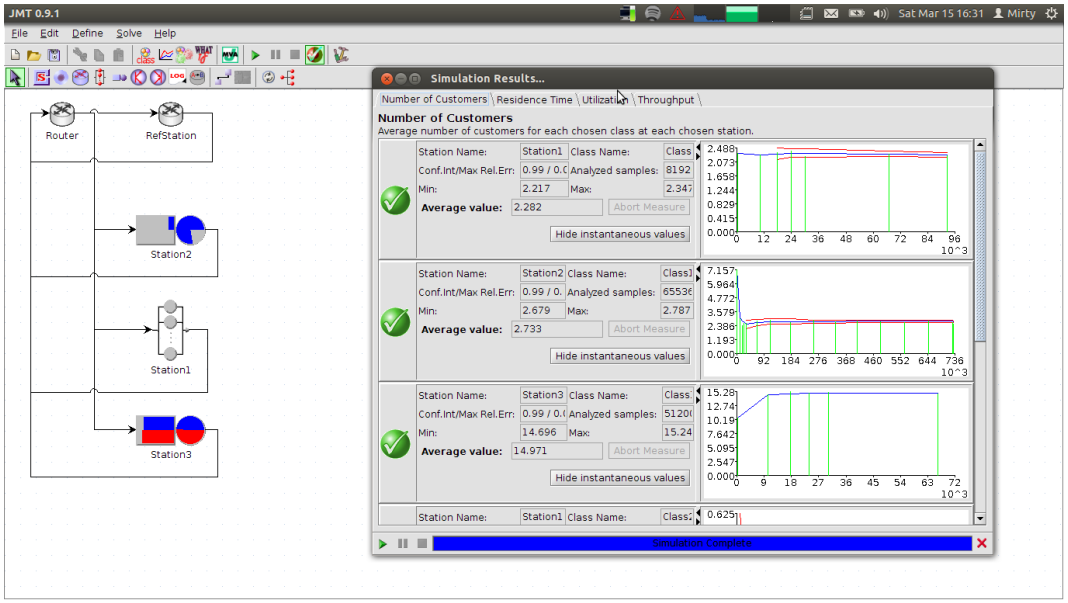

Figura 3.5: Un modello costruito tramite JSIMgraph

Lo specifico tool scelto all'interno del framework è JSIMgraph, che permette la creazione di modelli tramite interfaccia grafica, mostrata in Fig.3.5. Questo ha permesso anche di poter visualizzare al meglio il modello creato, permettendo un rapido ed efficace confronto con il sistema realmente utilizzato per i test su cloud.

In particolare vengono definiti i seguenti componenti, illustrati in Fig.3.6:

- `service station`(Fig.3.6-a): rappresenta un componente capace di elaborare una richiesta. E' possibile caratterizzare il suo comportamento impostando i tempi di risposta che esso ha rispetto ad ogni tipologia di richiesta, detta *classe*, oltre che al

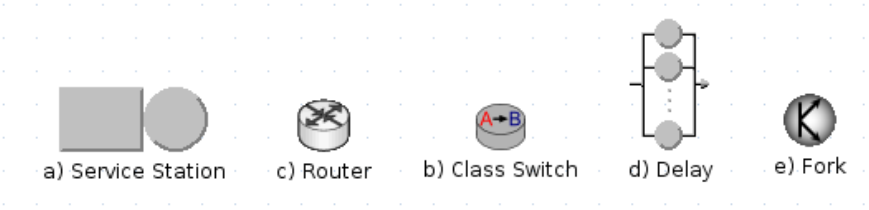

Figura 3.6: Componenti di JMT utilizzati

numero di server che esso rappresenta o la sua strategia di servizio (*First Count First Service*, *Process Sharing*). La legge fondamentale che regola questo componente è detta *Little's law*, definita come: $N = X * R$, o anche *numero medie di richieste* equivalente a *throughput* moltiplicato per *residence time medio*. Con questo oggetto vengono prevalentemente astratti i componenti delle virtual machine installati sulla piattaforma cloud.

- `router`(Fig.3.6-b): serve per reindirizzare una certa richiesta ricevuta come input ad un numero variabile di componenti, con una certa probabilità per ogni classe.

- `class switch`(Fig.3.6-c): se si necessita di diversificare i tempi di risposta in base all'operazione da effettuare, questo componente si occupa di prendere un'unica classe di richiesta come input e creare, con una certa distribuzione, un numero desiderato di classi pre assegnate.

- `delay`(Fig.3.6-d): utilizzato per modellare dei ritardi dovuti a diverse cause, ad esempio una latenza di rete. E' possibile quindi configurare un ritardo da assegnare ad ogni richiesta che attraverserà questo componente.

- `fork`(Fig.3.6-e): utile per creare più copie identiche della stessa richiesta ricevuta come input, simulando per esempio delle replicazioni. Esiste anche una sua contro-parte, chiamata `join`, che si preoccupa di riportare le richieste al numero originale.

I modelli costruiti e i relativi trend prestazionali, confrontati con quelli dei database reali, saranno presentati nel capitolo 5, in cui sarà data una specifica ragione ad ogni componente utilizzato.

3.5 Conclusioni

In questo capitolo sono state descritte le configurazioni di tutti gli strumenti utilizzati per un corretto ed efficiente svolgimento di questa tesi. Le impostazioni sono state fornite,

oltre che per una lettura più chiara, anche per rendere consapevole il lettore che si appresta a lavorare con questi strumenti, sotto quali pre-concetti sono stati preparati i test che si sono andati ad effettuare e come questi possano essere replicati o modificati in base alle proprie personali esigenze.

Nel successivo capitolo saranno mostrati i risultati dei test svolti, considerando le impostazioni qui presentate.

Capitolo 4

Studio e realizzazione dei Test

In questo capitolo si vogliono presentare i risultati ottenuti durante lo svolgimento delle diverse categorie di test, sviluppate in base allo studio dei database e alle relative considerazioni che ne sono seguite. Analizzando di volta in volta l'output dei test, espresso in termini di throughput, latenze di operazioni di tipo update e di tipo read, l'obbiettivo perseguito è stato sempre quello di ottenere dei risultati che ci permettessero di comprendere il trend prestazionale di ogni configurazione, confrontandolo quindi con le supposizioni fatte a priori. Altro obbiettivo è stato quello, quando possibile, di cercare di sfruttare al massimo le risorse della configurazione adottata. Si è quindi cercato di portare alla saturazione ogni elemento del database, potendo confrontare così i valori massimi di numero di operazioni che questo riesce ad eseguire a pieno regime. Proprio per questo i test sono stati svolti aumentando di volta in volta il numero di thread operativi sul client, ottenendo così i punti di flesso indicanti la saturazione del sistema.

I test sviluppati sono stati divisi in quattro categorie:

- **numero di core:** si è cercato innanzitutto di capire fino a che punto la potenza della CPU della singola macchina potesse influire sulle prestazioni di un database, valutando così quanto il sistema fosse in grado di sfruttare la parallelizzazione fornitagli. E' con questo primo test che, in base ai risultati qui ottenuti, sono state scelte le configurazioni hardware dei test seguenti.

- **numero di entry point:** avendo considerato la possibilità che un singolo entry point potesse diventare rapidamente il collo di bottiglia dell'intero sistema, si è voluto studiare quanto un singolo nodo potesse reggere il lavoro relativo alla comunicazione con il client per conto dell'intero sistema. Il confronto è stato poi effettuato rispetto ad una configurazione che permettesse il maggior numero di nodi possibili abilitati alla comunicazione l'esterno.

- **numero di nodi:** essendo, come stato detto più volte, la scalabilità orizzontale una delle caratteristiche principali dei database NoSQL, questo test si è occupato di valutare quanto il numero dei nodi impattasse sulle prestazioni del sistema che si è di volta in volta creato. Volendo evidenziare il più possibile questa caratteristica, è stata scelta per ogni database la configurazione più performante rispetto al test precedentemente svolto.

- **replicazione:** di particolare interesse questa ultima tipologia di test, poichè la replicazione, come visto nel capitolo 2, viene gestita in maniera molto differente da ognuno dei tre database analizzati. Variando il numero di repliche effettuate dal sistema, si cerca di comprendere come i vari database tendano a gestire le copie create, aspettandosi dei risultati in linea con quanto studiato.

Per ogni tipo di test sarà svolta un'analisi delle ragioni che ci hanno portato ad eseguirlo, i parametri specifici dei database e la presentazione dei risultati opportunamente commentati, con un particolare risalto ad un confronto tra i database e le differenze con i risultati aspettati.

I risultati più rilevanti mostrati in questo capitolo saranno poi utilizzati per costruire dei modelli che riescano a catturare le caratteristiche dei sistemi reali, ricercando lo stesso trend prestazionale ottenuto nei test eseguiti.

4.1 Test A: Core

La prima tipologia di test è stata pensata per osservare come i database si comportassero al variare della potenza della macchina su cui venivano installati. Per fare questo, è stata scelta una macchina che disponesse di un massimo di otto core, l'instance `c1.xlarge`, su cui di volta in volta sono stati tenuti operativi solamente il numero di core voluti.

Essendo il sistema operativo Ubuntu, eseguire una operazione del genere è resa possibile tramite il comando "`echo 0 | sudo tee /sys/devices/system/cpu/cpu`X`/online`", con X indicante il numero del core da disattivare.

Questo modo di operare è stato scelto perché lavorare sempre con un tipo di macchina variandone i core attivi permette di essere certi di operare sempre con la stessa tipologia di CPU. Lavorare con macchine aventi direttamente un numero diverso di core avrebbe portato il rischio di utilizzare tipologie di CPU differenti, invalidando i risultati.

Per focalizzare i risultati di questo test solo ed esclusivamente sulla potenza della macchina, è stato deciso di utilizzare, per ogni database, lo stesso numero di nodi e lo stesso numero di thread per client, impostati rispettivamente a un nodo e 50 thread.

Per Cassandra e MongoDB sono stati rispettivamente installati sul singolo nodo il processo di Cassandra e tutti e tre i processi di MongoDB, ossia mongos, mongod e config server. Per HBase, essendo problematica e mai consigliata la sistemazione dei processi di HMaster, Zookeeper e HRegionserver su un singolo nodo, sono state utilizzate tre macchine differenti. Due, di tipo `m1.large`, per l'esecuzione di Zookeeper e del HMaster, mentre una terza per l'avvio del HRegionserver e di tipo `c1.xlarge`, potente quanto le macchine usate per Cassandra e MongoDB e l'unica tra le tre a scalare di potenza.

Questa scelta è stata fatta anche in merito all'architettura di HBase che, lo si ricorda, permette la comunicazione diretta tra HRegionserver e client, facendo in modo che le latenze tra HMaster e HRegionserver siano irrilevanti.

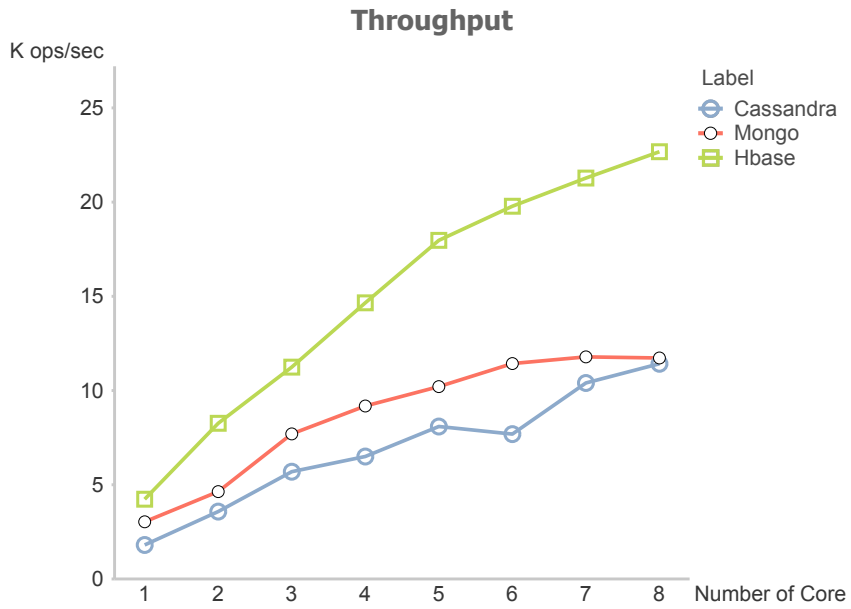

Figura 4.1: Throughput totale in funzione del numero di core

In Fig.4.1 è possibile vedere i risultati ottenuti dal test. Come era facile prevedere, tutti i database riescono a migliorare il proprio throughput al salire del numero di core utilizzati, con Cassandra e MongoDB che presentano un andamento praticamente lineare. Più interessante il risultato di HBase, che riesce a sfruttare decisamente meglio la potenza datagli, aumentando sensibilmente le proprie prestazioni.

Mentre quest'ultimo database presenta un tasso di crescita particolarmente elevato, si nota che Cassandra, nonostante parta da un risultato minore con un solo core, riesce a raggiungere MongoDB mano a mano che vengono aggiunti core alla macchina, arrivando ad avere un risultato molto simile. Proprio MongoDB mette in evidenza come, raggiunto un numero di core pari a 6, l'aggiunta di core non sembra più essere di particolare impatto

per il sistema, al contrario degli altri due database che mostrano un trend monotono crescente.

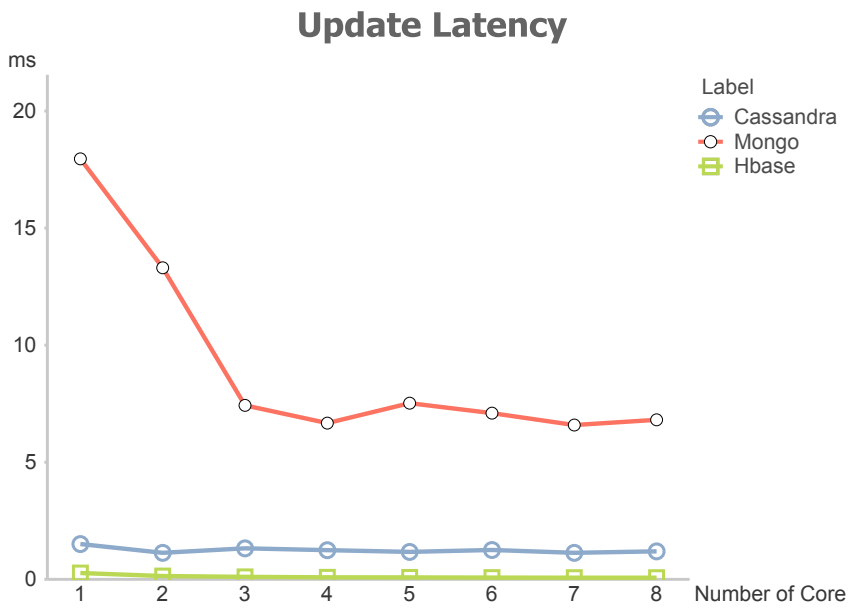

Figura 4.2: Update latency rispetto al numero di core

La Fig.4.2 mostra le latenze che i database hanno mostrato per quanto riguarda le operazioni di tipo update. Si nota innanzitutto che MongoDB ha delle latenze estremamente alte per un numero di core inferiore a tre, dopo il quale raggiunge una certa stabilità per un qualunque numero di core. L'installazione del processo mongos e mongod su una stessa macchina può essere ricondotta come principale causa di questa, visto che i due processi necessitano in continuazione di essere operativi in parallelo. La latenza di questo database è particolarmente alta confrontata con gli altri due database che, mentre per Cassandra è molto stabile con una latenza poco al di sotto dei due millisecondi, vede HBase come un database particolarmente indicato per le operazioni di update. HBase infatti ha presentato sempre, come vedremo anche dai test successivi, ottimi risultati nell'eseguire questo tipo di task, andando sempre ad ottenere latenze nell'ordine delle centinaia di microsecondi, contro i diversi millisecondi impiegati dagli altri database. Questa velocità di scrittura è da ricercarsi nell'utilizzo del già citato *client-side write buffer*, che rende le latenze di update quasi irrilevanti per il client che continua a riempire il buffer, operazione che rimane interna al client stesso annullando qualsiasi ritardo fino al riempimento del buffer stesso.

Le operazioni di read, come mostrato dalla Fig.4.3, rivelano un andamento iperbolico, segno che questo task riesce a sfruttare meglio la parallelizzazione delle CPU. In questo caso è MongoDB ad ottenere le latenze più basse a qualsiasi numero di core, mentre Cassandra sembra far maggiore fatica ad eseguire read in presenza di un numero molto basso

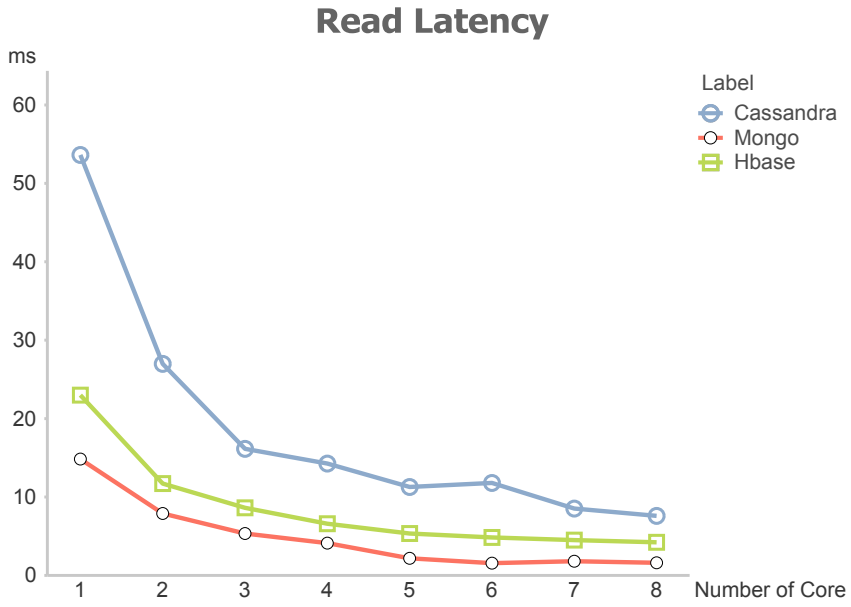

Figura 4.3: Read latency rispetto al numero di core

di core, mostrando quindi una maggiore predisposizione ad operazioni di scrittura. Di particolare interesse la netta differenza presentata da questo database tra un single-core e un dual-core, punto da cui comincia a comportarsi con lo stesso trend degli altri database.

In base ai risultati ottenuti e qui mostrati, tutti i test successivi sono stati eseguiti su macchine con processore dual-core. Questa decisione è stata presa in merito a diverse considerazioni. Si è voluto innanzitutto puntare sull'utilizzo di macchine che presentassero un hardware non particolarmente costoso, per poter considerare la reale costruzione di un database NoSQL con un alto numero di nodi, con un costo non troppo elevato, puntando sulla scalabilità piuttosto che sulla potenza della singola macchina.

Inoltre, i grafici precedenti mostrano come la configurazione con due core rappresenti per tutti i database il punto in cui i database comincino un trend senza particolare instabilità. Questo vale anche per MongoDB, dato che la configurazione mongod, mongos e config server verrà eseguita solamente su un solo nodo, mentre sulle altre macchine saranno principalmente due i processi operativi. L'utilizzo di macchine non troppo potenti ha anche permesso di non mettere il client nella situazione di diventare il collo di bottiglia del sistema, evidenziando invece il punto di saturazione dei database anche per un numero non troppo elevato di thread.

4.2 Test B: Entry point

Una delle differenze più evidenti tra i database è quella riscontrata nel modo in cui i nodi contenenti i dati vengono contattati dal client. In Cassandra si ha che tutti i nodi possono essere considerati come punti d'accesso, senza dover creare particolari gerarchie e nemmeno specificandolo nelle impostazioni del database. Per MongoDB gli unici punti d'accesso sono i processi mongos, mentre per HBase il client instaura una connessione diretta con i HRegionserver attraverso una rapida comunicazione con l'istanza di Zookeeper. Proprio per quest'ultimo motivo, HBase non è stato preso in considerazione per questo test, poichè indipendentemente dall'indirizzo IP che si andava a specificare nelle opzioni del tool di benchmark, il nodo veniva sempre messo in comunicazione con l'istanza di Zookeeper e conseguentemente reindirizzato al HRegionserver utile.

Questo test ha dimostrato inoltre come l'architettura iniziale pensata per MongoDB, in cui ogni processo aveva una macchina dedicata, non fosse la scelta giusta per una corretta comparazione tra i vari database.

Mentre in Cassandra gli entry point sono dichiarati direttamente immettendo come opzione l'elenco degli indirizzi IP dei singoli nodi che si vuole abilitati a questa funzione, per MongoDB si è dovuto creare un numero di client pari ai processi mongos eseguiti. Questo, come già anticipato nella sezione 3.3, è dovuto al fatto che nelle opzioni di YCSB, riguardo a MongoDB, è possibile indicare un solo indirizzo per volta, e per non rischiare di saturare il client, ne sono stati avviati tanti quante le istanze dei processi mongos. In questo caso il numero dei thread indicato come dimensione nei grafici indica il numero totale di thread cui il sistema viene sottoposto, e non il numero di thread per client. I dati sono stati poi raccolti su un singolo nodo e incrociati, sommando i throughput e calcolando una media delle latenze rispetto al numero di client, il tutto per simulare l'esecuzione del test da parte di un singolo client.

Di seguito vengono presentati i risultati suddivisi per database, permettendo una maggior comprensione delle configurazioni utilizzate.

Cassandra

Cassandra è stato testato prendendo in considerazione la possibilità di avere il numero di entry point pari a uno oppure pari al numero di nodi appartenenti all'anello costruito, mantenendo in ogni caso il numero di replicazione uguale ad uno.

Le Fig.4.4 e 4.5 evidenziano il massimo carico di lavoro gestibile da un singolo nodo abilitato come entry point. In particolare, si può notare come fino ad un numero di nodi pari a quattro le differenze siano irrilevanti, mostrando come un nodo solo sia in grado di gestire le richieste del client senza saturare per la troppa gestione di operazioni. Bisogna

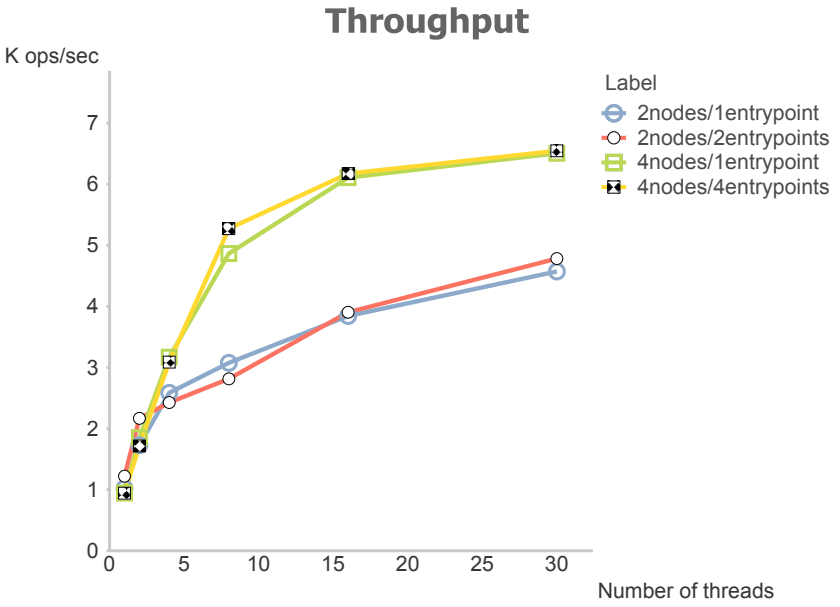

Figura 4.4: Throughput medio per Cassandra in presenza di due e quattro nodi, al variare dei thread e degli entry point

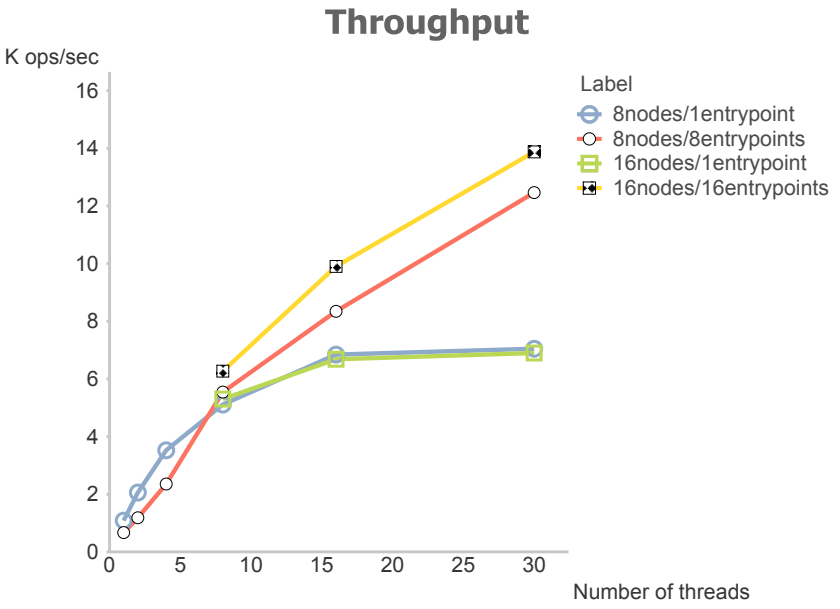

Figura 4.5: Throughput per ring di Cassandra composte da otto e sedici nodi, in funzione dei thread e degli entry point

infatti considerare che il nodo in questione si prende carico, oltre che delle operazioni assegnate ad esso, della comunicazione con gli altri nodi reindirizzando le opportune richieste ai nodi corretti, terminando il suo compito con l'invio al client di tutti i risultati. La laboriosità del lavoro a cui è sottoposto è evidente in presenza di un numero maggiore di nodi. Per otto o sedici macchine, la presenza di più entry point appare determinante per consentire al sistema di essere sfruttato appieno. In questo caso il singolo nodo non riesce a gestire tutto il lavoro assegnatogli, saturando e rendendo irrilevante il numero di nodi che compongono il database.

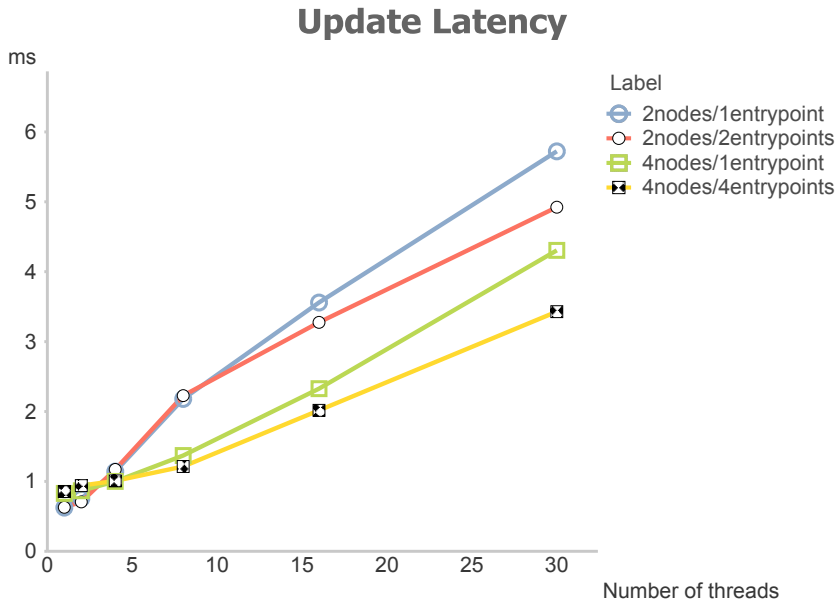

Figura 4.6: Update latency per Cassandra, con differente numero di entry point e thread, per due e quattro nodi

Quanto visto per il throughput è riscontrabile anche sulle latenze. In Fig.4.6 le latenze per le operazioni di update presentano lo stesso trend, anche se si osserva come il singolo nodo fatichi a gestire il carico di lavoro. Dalla Fig.4.7 è evidente come l'avere più entry point riduce drasticamente la latenza di queste operazioni. Il singolo nodo, superata la soglia degli otto thread attivi, tende a impiegare troppo tempo a gestire la comunicazione con gli altri nodi. Si noti sempre come le latenze per un numero maggiore di quattro nodi diventino uguali indipendentemente dal numero dei nodi, conferma che si è raggiunta la saturazione dell'entry point.

Dalle Fig.4.8 e 4.9 è interessante notare come il trend delle latenze riguardo alle operazioni di read sia praticamente identico a quello seguito per le update. E' anche qui evidente come il singolo nodo riesca a gestire le richieste fino ad un determinato numero di nodi, superato il quale le latenze crescono ad un ritmo molto elevato.

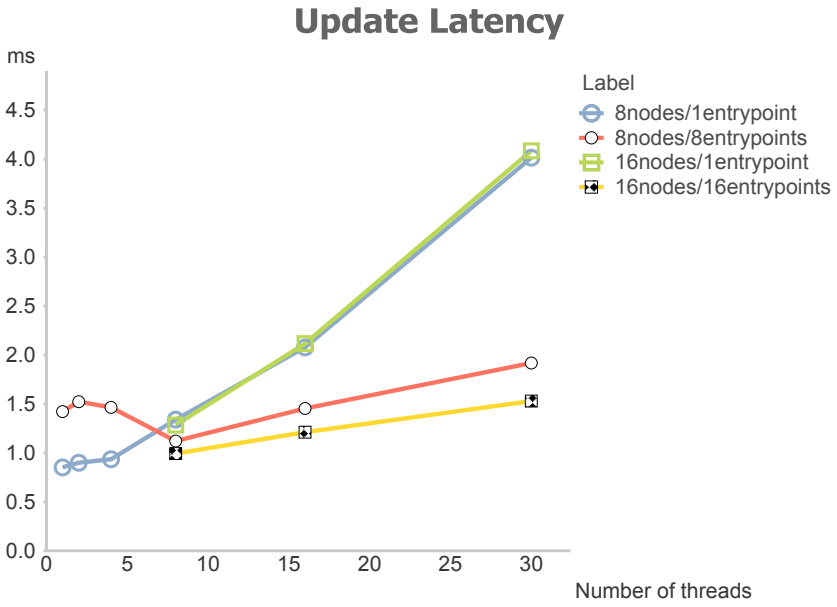

Figura 4.7: Update latency per Cassandra al variare di entry point e thread, per otto e sedici nodi

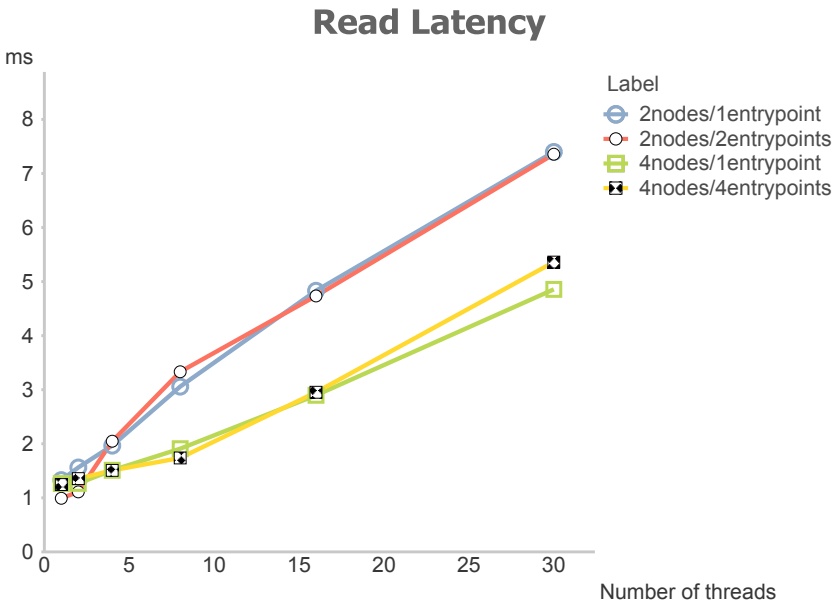

Figura 4.8: Read latency per Cassandra con due o quattro nodi, variando numero di thread ed entry point.

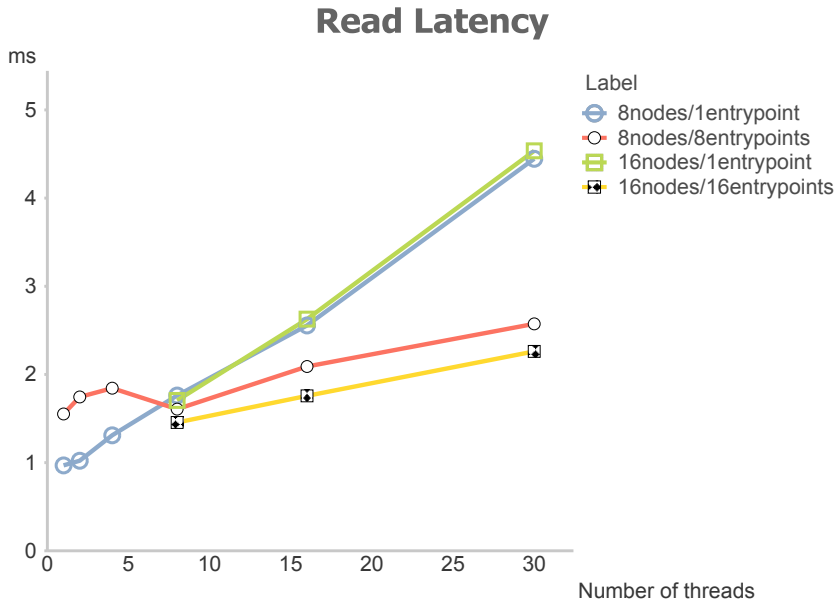

Figura 4.9: Read latency per Cassandra in base a differenti entry point e numero thread, per otto e sedici nodi

MongoDB

Così come quanto fatto per Cassandra, anche per MongoDB sono stati testati vari sistemi configurati con un numero di entry point pari ad uno o al numero di nodi. Nel caso di singolo entry point si è optato quindi di utilizzare la prima soluzione pensata per il deploy di questo database, ossia di avere due macchine dedicate rispettivamente all'esecuzione del processo mongos e una all'esecuzione del processo config server, mentre il numero di macchine utilizzate per i processi mongod è stato considerato il numero utile ai fini del test. La configurazione con più entry point presenta invece, per ogni macchina, l'avvio di un processo mongos e uno mongod, mentre il config server è stato istanziato su di una sola macchina scelta in modo casuale. Il numero di shard è stato invece impostato al massimo possibile, rendendo il numero di replicazioni uguale ad uno.

Dalla Fig.4.10 è subito evidente il limite della prima configurazione. Nel momento in cui il sistema viene sottoposto ad uno sforzo ragionevole, intorno ai dieci thread, l'avere più entry point si rivela indispensabile fin da un sistema con quattro nodi. E' invece interessante notare come avendo due nodi, non si abbia variazione del throughput in presenza di un maggior numero di entry point, indice che i nodi funzionali, dedicati cioè ai processi mongod, saturino ancora prima di sfruttare tutte le risorse degli entry point presenti nel sistema.

Le latenze sono, ovviamente, in linea con il trend del throughput. Sia dalla Fig.4.11 che

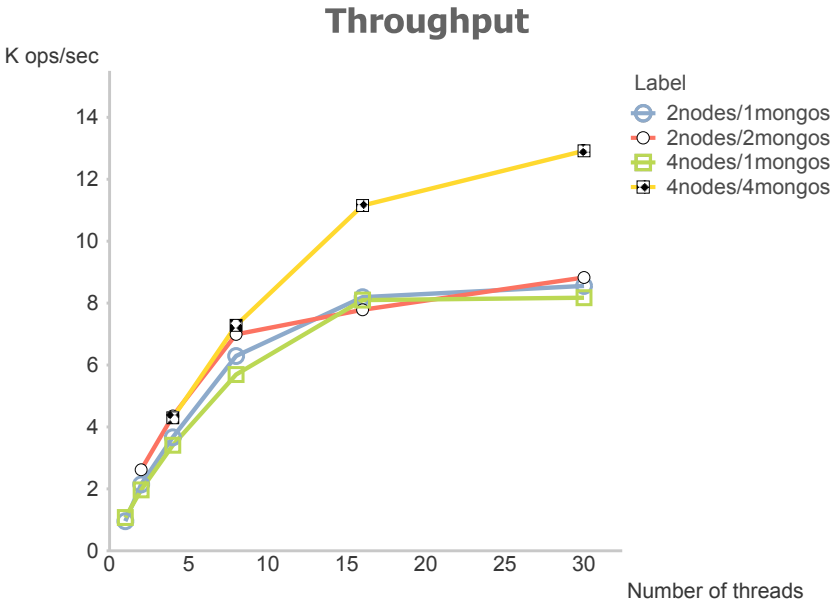

Figura 4.10: Valori del throughput per MongoDB al variare dei thread e degli entry point

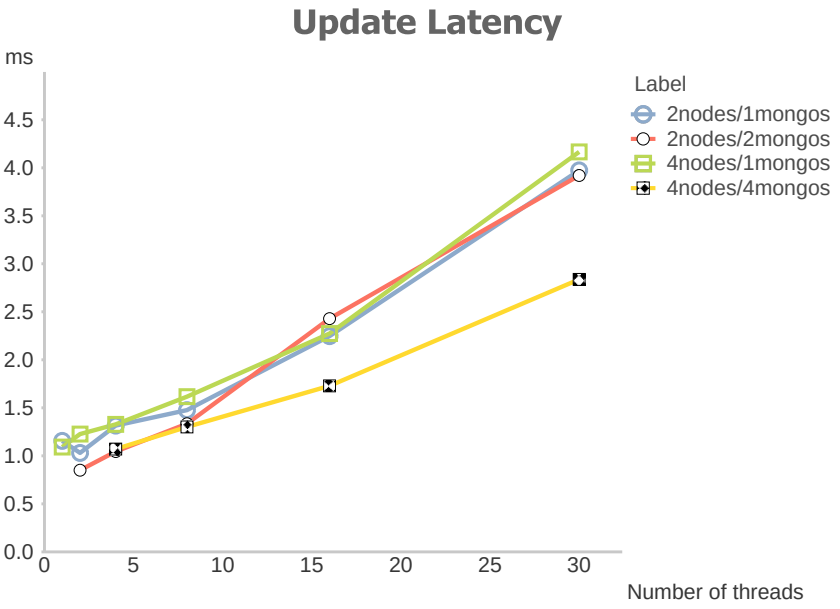

Figura 4.11: Update latency per MongoDB in funzione dei thread e degli entry point

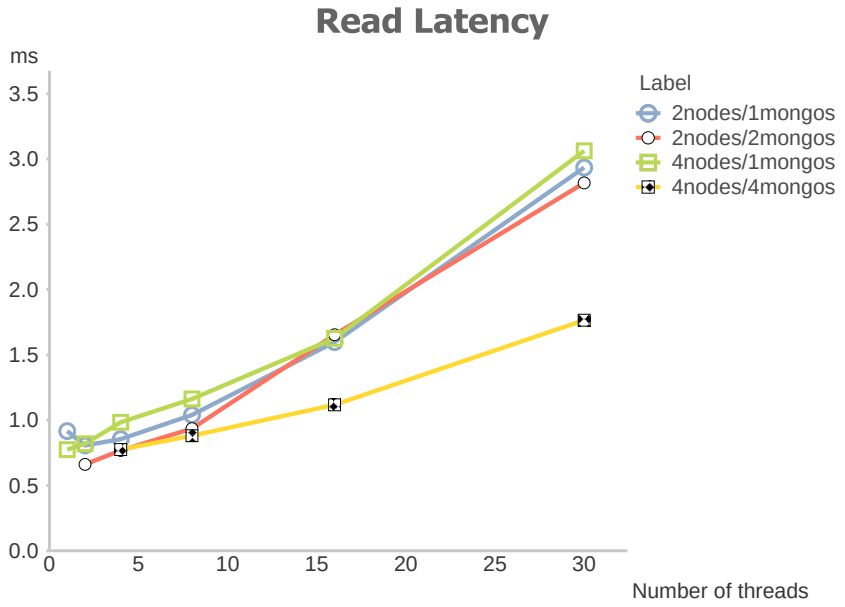

Figura 4.12: Read latency per MongoDB al variare dei thread e degli entry point

dalla Fig.4.12 è evidente come l'unica configurazione in grado di gestire carichi di lavoro maggiori sia quella che, al crescere dei nodi, presenta un maggior numero di entry point. In entrambi i grafici è infatti possibile notare come, in presenza di quattro macchine ed un solo entry point, questo diventi immediatamente il collo di bottiglia, rendendo inutile l'installazione di un numero maggiore di nodi.

I risultati appena mostrati risaltano in modo evidente l'importanza di definire un numero elevato di entry point al crescere della dimensione del sistema. Come era facile da dedurre, riuscire a distribuire il carico di lavoro, inerente la gestione del database stesso, su di un numero elevato di nodi, permette al sistema di ottenere prestazioni migliori. Mentre per Cassandra era facile aspettarselo, essendo l'anello composto da nodi uguali tra loro, per MongoDB non era scontato come comportamento. Questo ha dimostrato come il processo mongos venga fortemente utilizzato, preferendo l'allocazione di più processi su di una singola macchina piuttosto che averne una dedicata per ognuno di essi.

A fronte di questi risultati, i test successivi sono stati eseguiti su configurazioni che permettessero di concentrarsi sull'utilizzazione del sistema nella maniera più completa possibile, utilizzando sempre il numero maggiore di entry point disponibili.

4.3 Test C: Numero di nodi

La scalabilità rispetto al numero dei nodi viene analizzata in questa serie di test. L'obbiettivo è stato quello di essere in grado di trovare una relazione tra il numero di nodi e il throughput del sistema. Oltre ad aumentare il numero di componenti del sistema, per analizzare meglio l'andamento delle prestazioni rispetto ad ogni configurazione si è incrementato il numero di thread fino ad un numero tale da poter essere considerato soddisfacente per l'analisi in questione.

Di seguito sono riportati i risultati conseguiti per ogni database.

Cassandra

Cassandra è stato testato imponendo il massimo numero di entry point possibile e variando le configurazioni rispettivamente ad uno, due, quattro, otto e sedici nodi. Il numero di thread è stato poi incrementato fino ad un massimo di sessanta thread concorrenti, valore per cui i trend prestazionali del database diventano evidenti.

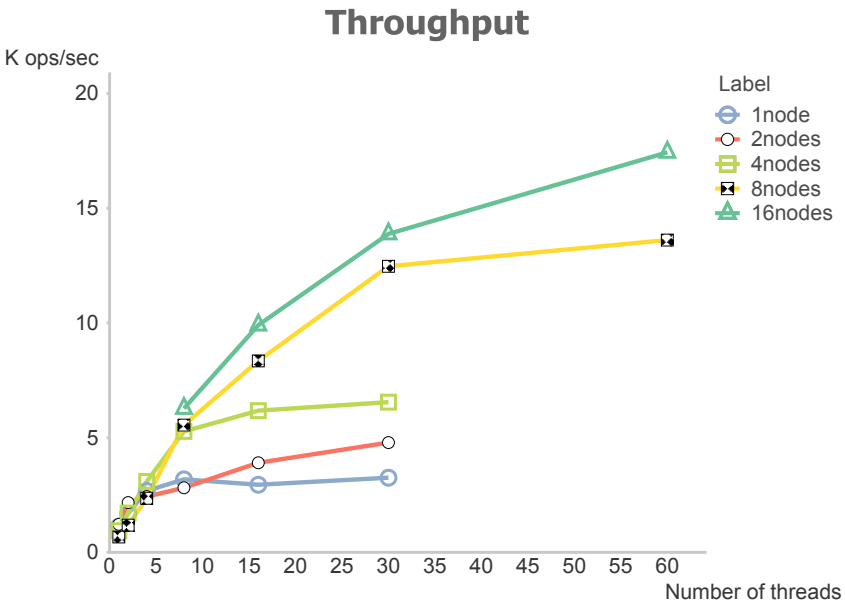

Figura 4.13: Throughput di Cassandra al crescere del numero di nodi e di thread

Il throughput del sistema, come evidente dalla Fig.4.13, diventa sempre maggiore all'aumentare del numero di nodi, in maniera costante. E' di particolare interesse il punto di stacco tra le diverse configurazioni: mentre la differenza tra uno, due e quattro nodi

si nota attorno ai dieci thread, è per un numero maggiore di trenta che le configurazioni dotate di un numero maggiore di nodi cominciano ad assumere andamenti differenti. La configurazione ad otto nodi riesce infatti a seguire quella con sedici macchine fino ai trenta thread, dopo i quali la prima implementazione comincia a dare segni di saturazione. Da tenere in considerazione come, per le configurazioni che mostravano evidenti segni di saturazione già per un numero di thread pari a trenta, non sono stati effettuati ulteriori test poichè non considerati di nessuna utilità.

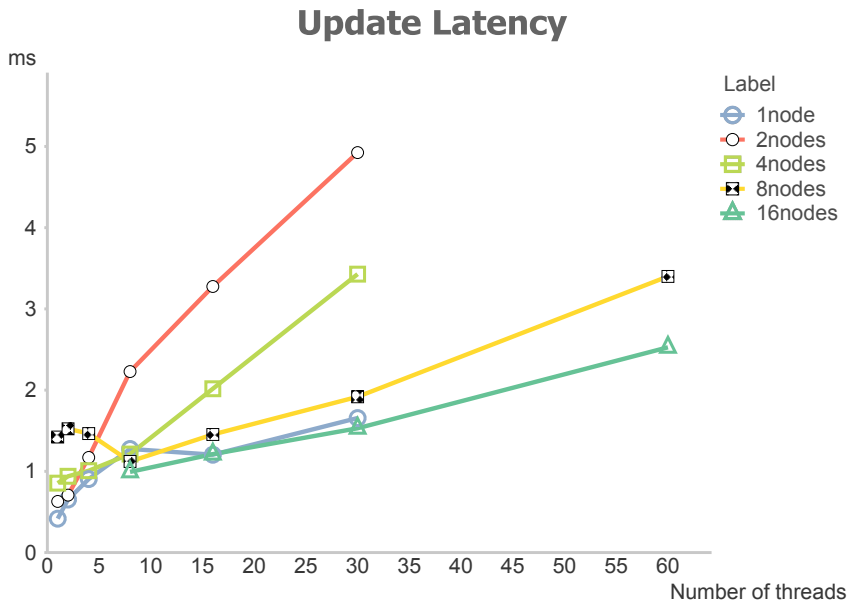

Figura 4.14: Update latency di Cassandra al variare del numero di nodi e thread

L'analisi delle latenze mostra invece diverse particolarità. Mentre all'aumentare dei nodi le latenze tendono ovviamente a diminuire, segno che il sistema riesce a sopportare maggiori carichi di lavoro, la configurazione dotata di un solo nodo presenta delle latenze di update particolarmente basse, comparabili con quella avente il maggior numero di nodi. Questa particolarità, evidente in Fig.4.14, indica come l'avere un singolo nodo eviti al sistema di dover gestire la sincronizzazione e la comunicazione tra tutti i nodi della ring, facendo anche capire la portata del lavoro di cui Cassandra necessita per eseguire tali operazioni, almeno per quanto riguarda le operazioni di scrittura.

Dalla Fig.4.15 è invece possibile notare come, sebbene la configurazione con un nodo abbia delle latenze di update evidentemente basse, le operazioni di lettura risentano particolarmente della presenza di più componenti. Il tasso di crescita di tale configurazione è infatti decisamente più elevato, in proporzione, rispetto a quella con un nodo in più. Per le

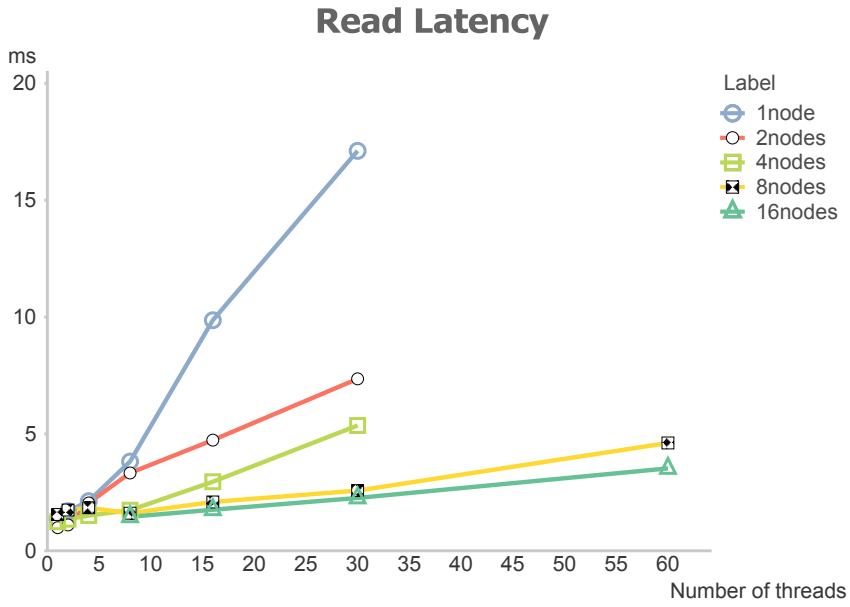

Figura 4.15: Read latency di Cassandra in funzione di numero di nodi e thread

configurazioni che hanno invece un alto numero di nodi, al crescere di questi la differenza si fa sempre più sottile, come le configurazioni con otto e sedici nodi fanno notare.

MongoDB

Per questo database la scelta delle configurazioni, a fronte dei test illustrati precedenti, è ricaduta sull'avvio dei processi mongos e mongod su ogni macchina appartenente al sistema, impostando il numero di entry point uguale al numero di nodi, con la singola istanza del config server avviato su una macchina scelta in modo casuale. Oltre a questa decisione, è stato scelto di creare un numero di shard che fosse il massimo possibile, poichè per ottenere prestazioni massime, la presenza di un numero di nodi che potesse eseguire operazioni sia di tipo update che di tipo read è stata considerata necessaria. Si noti quindi che, come per lo stesso tipo di test su Cassandra, anche per questo database la replicazione è impostata ad uno, cioè la memorizzazione nel database della sola copia originale del dato.

Infine è da tenere presente che, essendo i test eseguiti su un numero di client pari al numero di macchine, i risultati per ogni configurazione partono da un numero di thread pari al numero dei nodi, permettendo quindi ad ogni client di avere almeno un thread attivo. Essendo interessati al trend delle prestazioni, questa accortezza non ha inficiato in alcun modo sulla bontà dei risultati.

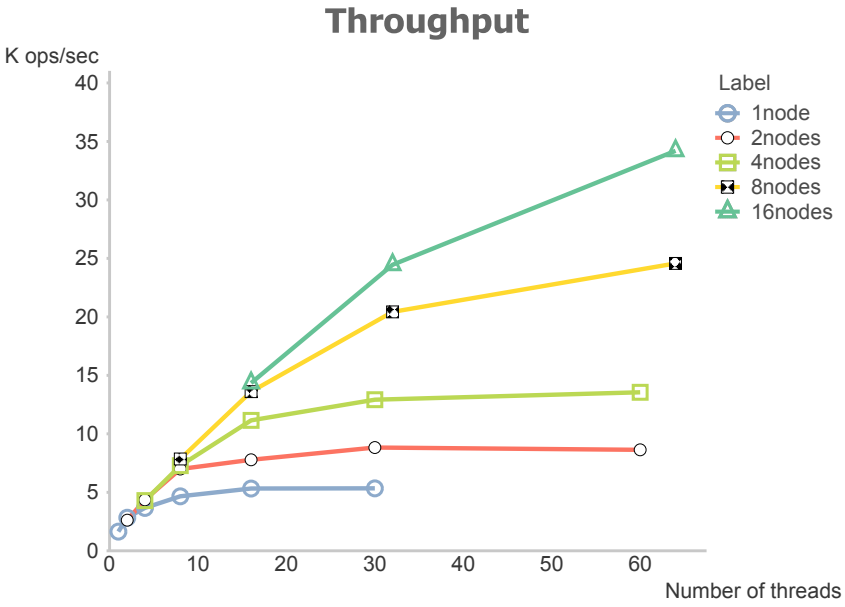

Figura 4.16: Throughput di MongoDB al variare del numero di nodi e thread

Il throughput dell'intero sistema è, per come si presenta in Fig.4.16, molto intuitivo. Al crescere del numero dei nodi, le prestazioni e il punto in cui si ha un cambiamento del trend vengono spostati verso un valore più alto. E' molto importante notare la differenza con Cassandra riguardo la capacità di sfruttare l'aumentare dei nodi. MongoDB dimostra infatti di riuscire a moltiplicare, per un fattore di circa sette, il throughput tra la configurazione con un nodo e quella con sedici, mentre Cassandra non riusce ad andare oltre ad un fattore poco maggiore di tre.

Altro confronto con Cassandra, come mostrato dalle Fig.4.17 e 4.18, si può fare sulle latenze, dove in MongoDB abbiamo una distribuzione particolarmente uniforme. In questo database è infatti interessante vedere come l'andamento tra le latenze delle diverse operazioni sia molto simile, mostrando come MongoDB non sia fortemente indicato per un tipo particolare di operazioni.

HBase

L'ultimo database utilizzato per questa tipologia di test è stato HBase, preso nella configurazione in cui sono presenti, oltre ai nodi su cui sono installati gli HRegionserver, due ulteriori nodi per eseguire Zookeeper e HMaster. Questa scelta è stata presa per le stesse motivazioni descritte riguardo alla prima tipologia di test. Va invece notato come per questo database ci siano stati problemi da un punto di vista del client. HBase, come sarà evidenziato dai risultati, tende a sovraccaricare molto il client, tendendo ad avere satura-

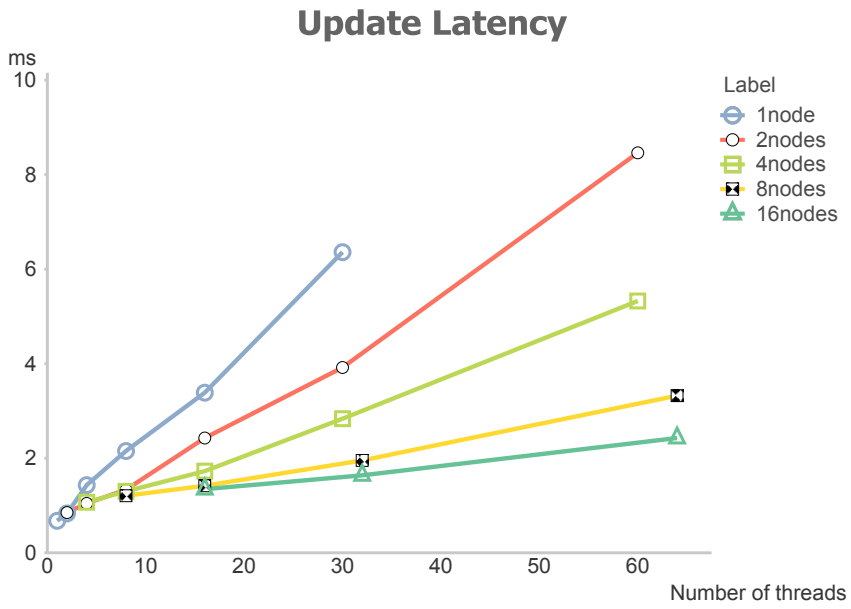

Figura 4.17: Update latency di MongoDB in funzione del numero di nodi e thread

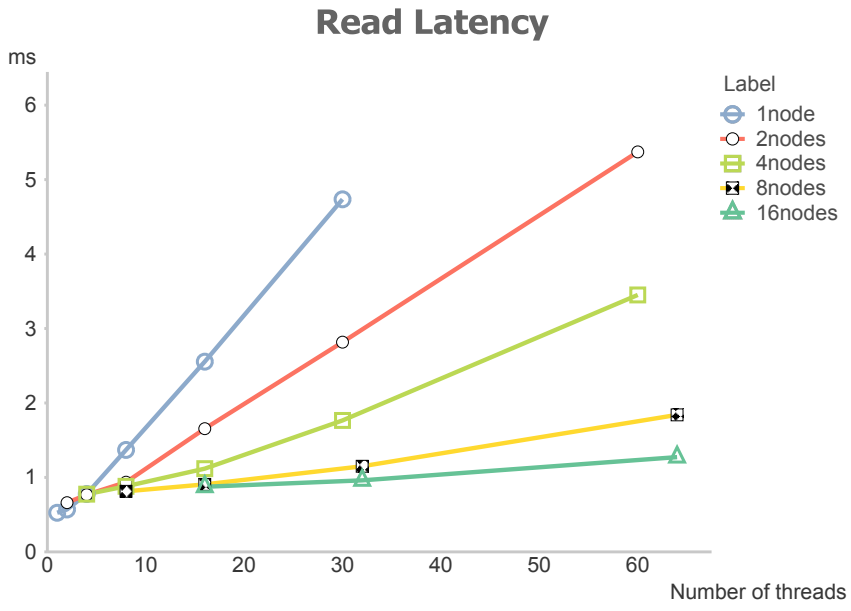

Figura 4.18: Read latency di MongoDB per un diverso numero di nodi e thread

zione di quest'ultimo. Per questo motivo, la macchina su cui è stato eseguito il software di benchmark è stato installato su una macchina che presentasse una potenza maggiore rispetto a quanto fatto per gli altri database, rendendo però utili i risultati per una architettura composta fino ad un massimo di otto nodi. Oltre quella cifra infatti, oltre ad avere un carico di lavoro particolarmente elevato per il client, il caricamento dei dati riguardante la fase di load ha presentato spesso una scarsa stabilità, dovendo riavviare completamente il database e ricominciare l'operazione di caricamento dati. Questo è dovuta principalmente alla gestione delle region, come spiegato nel capitolo 3.3. La replicazione, come per gli altri database, è stata impostata uguale a uno.

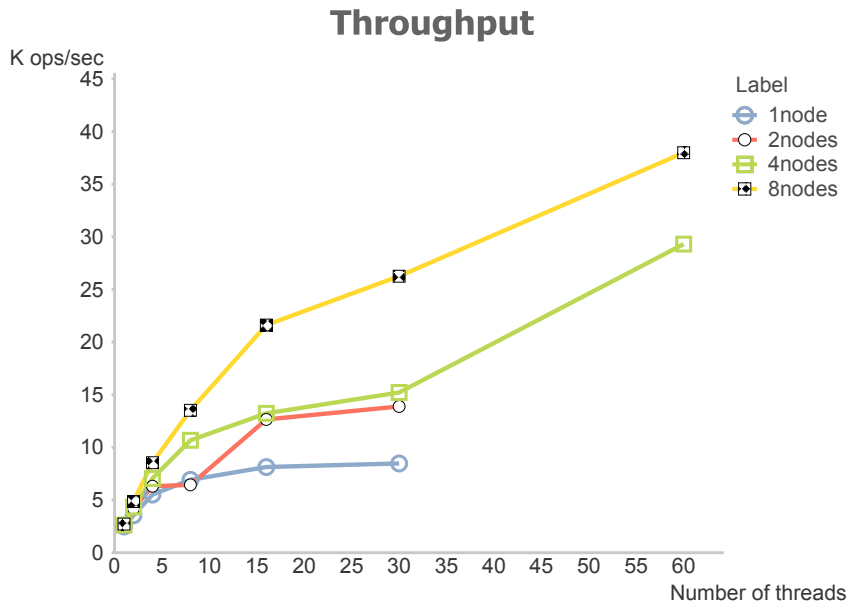

Figura 4.19: Throughput di HBase in funzione del numero di nodi e thread

Dalla Fig.4.19 è innanzitutto possibile notare quanto, a parità di nodi rispetto agli altri database, HBase offra prestazioni più elevate. In presenza di otto nodi siamo infatti già al di sopra del throughput espresso da MongoDB con il doppio dei nodi, ben distanti da qualunque configurazione di Cassandra. Si noti inoltre che il trend per le configurazioni con nodi pari a quattro e otto non tende a saturare, ma anzi è in continua crescita, cosa che negli altri database non si è riusciti ad ottenere. La causa di queste alte prestazioni è da ricercarsi principalmente nelle latenze.

La Fig.4.20 mostra in maniera efficace il perchè delle ottime prestazioni di HBase. Si fa infatti notare che, mentre per tutti i database le latenze riguardo le operazioni di update sono espresse in millisecondi, per HBase questa misura viene misurata in microsecondi. Si è quindi in presenza di latenze da uno a due ordini di grandezza inferiori rispetto a MongoDB e Cassandra, il che inoltre porta anche ad avere una instabilità nella misurazione di tali

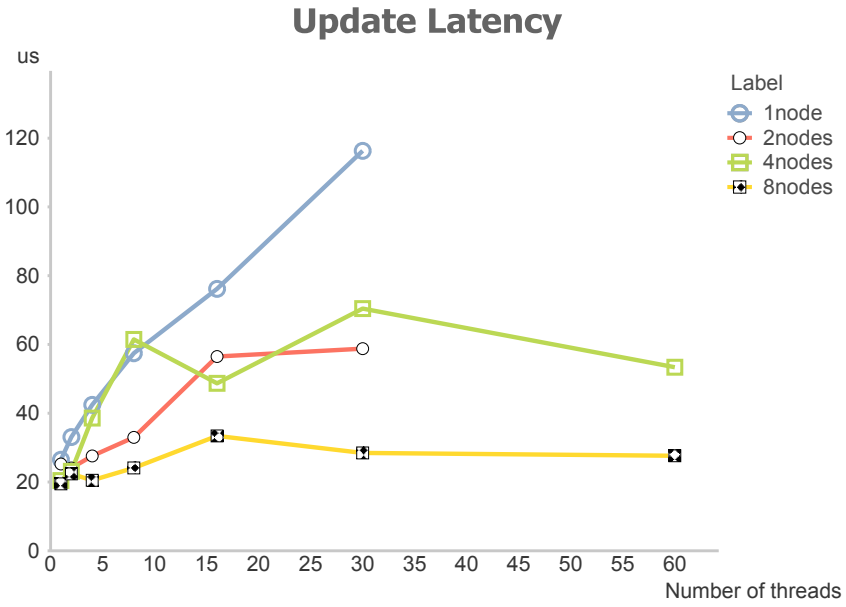

Figura 4.20: Update latency di HBase in funzione del numero di nodi e thread

valori poichè, essendo misure molto piccole, una lieve instabilità della rete causa delle ripercussioni marcate su questi valori. Questo risultato mostra in modo evidente quanto il client-side write buffer lavori in maniera efficiente, concentrando molto lavoro sul client e ottimizzando al meglio le chiamate al server. Ovviamente, come per gli altri database, un numero maggiore di nodi riduce ancora le latenze, per cui però siamo in presenza di valori medi estremamente bassi.

Nonostante nelle latenze per operazioni di update HBase sia molto efficiente, la Fig.4.21 mostra come il database torni a livelli comuni per quanto riguarda le latenze per operazioni di tipo read. Come ci si poteva aspettare, le latenze si riducono mano a mano che vengono aggiunti nodi, mostrando come un nodo solo sia particolarmente lento nell'effettuare questo tipo di operazioni.

Questa serie di test è stata fondamentale per capire quanto sono in grado di migliorare le prestazioni dei database rispetto al numero dei nodi che compongono il sistema e al numero di thread eseguiti sul client, permettendo l'identificazione dei punti di possibile saturazione dell'intero sistema. Sono inoltre stati utili al fine di analizzare diverse proprietà dei database, come le latenze di operazioni di tipo update per Cassandra o ancora meglio quelle per quanto riguarda HBase, mostrando in maniera chiara quanto l'utilizzo di un meccanismo quale il client-side write buffer possa essere il fattore determinante per un certo tipo di applicazioni.

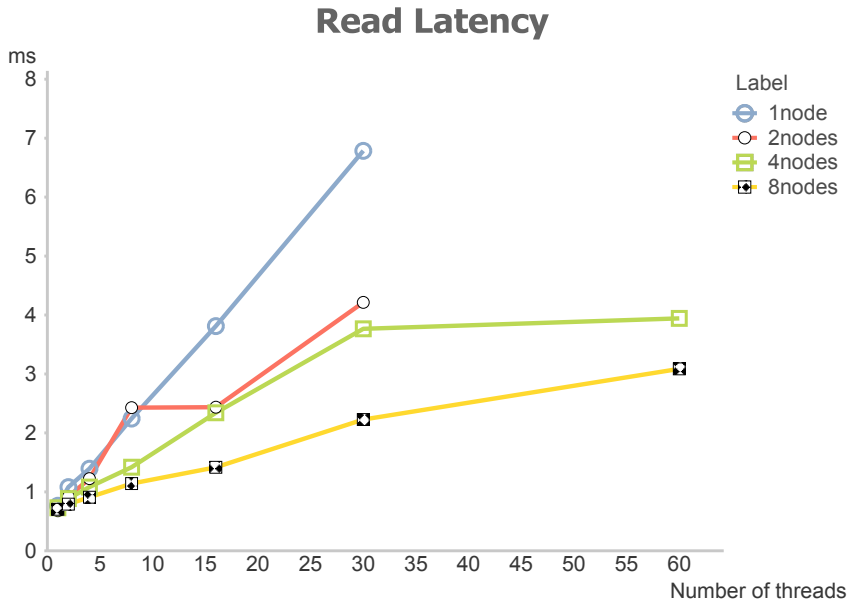

Figura 4.21: Read latency di HBase al variare del numero di nodi e thread

4.4 Test D: Replicazione

L'ultima tipologia di test riguarda la gestione delle replicazioni all'interno del database. Come per i test riguardo il numero di core, anche qui ci si è voluti focalizzare sulla proprietà principale da analizzare, decidendo quindi di fissare il numero dei thread uguale a 100. I database sono stati poi configurati in modo da ottenere risultati per un numero di nodi variabile da due a nove, con un fattore di replicazione crescente da uno a tre.

Per Cassandra si è quindi installato il sistema con il numero di nodi interessato e, al momento della creazione del keyspace, impostando la strategia di replicazione come *SimpleStrategy* e il numero di replicazioni voluto. Il numero di entry point è stato impostato uguale al numero dei nodi.

Per quanto riguarda MongoDB invece, a causa della presenza di replica set, per controllare le replicazioni si è andati a modificare il numero di nodi all'interno di ogni singola shard, creando quindi replica set di diverse dimensioni. Per la replicazione pari a uno è stato sufficiente creare un numero di shard pari al numero di nodi, avendo quindi replica set composte da un singolo nodo. Per gestire due repliche si è dovuto creare shard a cui appartenevano due nodi ognuna, portando quindi alla creazione di replica set composti da due nodi e di fatto dimezzando le capacità di update, poichè i nodi primari, differentemente da prima, non erano più tutti i nodi. Stessa modalità per replicazione pari a tre, dove il numero di macchine abilitati ad operazioni di tipo update è sceso ulteriormente. Per ottenere dei risultati chiari e in linea con gli altri database è stato scelto, quando in

presenza di replication factor maggiore di uno, di creare replica set tutti composti dallo stesso numero di nodi. In questo modo si ha mantenuto per tutti i dati, così come operato da Cassandra e da HBase, lo stesso numero di replicazioni. Si ricorda infatti che MongoDB associa ad ogni shard un set di dati, replicati solamente all'interno di quella stessa shard, offrendo quindi la possibilità di impostare, implicitamente attraverso il numero dei nodi di un replica set, un diverso numero di replicazioni per ogni shard. Si ricorda che il numero di entry point, anche in questo caso, è uguale al numero di nodi del sistema.

Infine, per HBase, prima di avviare il cluster di Hadoop, si è dovuto modificare il file di configurazione di HDFS in cui è possibile impostare il numero desiderato di repliche operate dal file system. Ci aspettiamo da questo database risultati interessanti, in quanto per HBase queste replicazioni appaiono completamente trasparenti.

Si fa infine notare la semplicità, per tutti i database, nell'impostare il numero di replicazioni, anche se MongoDB è quello che necessita di maggiori attenzioni durante la creazione di shard e replica set.

Di seguito sono riportati i risultati per ogni database.

Cassandra

Cassandra presenta il sistema di replicazione più standard e intuitivo: con la strategia adottata, il nodo incaricato di memorizzare il dato crea una copia passandola al nodo successivo, seguendo l'anello rispetto al numero crescente di token. Nel caso la replicazione sia maggiore di due, il secondo nodo creerà una copia da passare al terzo, e così via. Si ricorda come sia possibile, per questo database, avere un numero di replicazioni maggiore del numero di nodi, cosa impedita da parte degli altri due database.

Il throughput mostrato in Fig.4.22 fa capire quanto su Cassandra impatti la gestione delle replicazioni. E' particolare notare come si ha una perdita di throughput praticamente fisso, indipendentemente dal numero di nodi, così come la gestione di un numero crescente di replicazioni porti ad una sempre maggiore perdita di prestazioni.

Più interessanti sono invece le latenze. Come mostrato in Fig.4.23, le latenze per le operazioni di tipo update sono molto basse in presenza di un numero maggiore di replicazioni, anche se è da notare che tendono tutte quante alla stessa misura. E' inoltre particolare l'andatura perfettamente iperbolica nella configurazione in cui è presente il solo dato originale, mentre quelle con replicazioni maggiori presentano una solida stabilità una volta raggiunto un numero di nodi pari al numero di replicazione impostato.

Diverse invece le latenze per operazioni di tipo read. In Fig.4.24 è possibile notare un andamento iperbolico per tutte le configurazioni, notando che anche qui, come per le operazioni di update, le curve tendono tutte ad una misura comune. Questa volta è invece la configurazione con una sola replica ad avere prestazioni maggiori, avendo però un distacco sempre minore all'aumentare del numero di nodi. Una maggior latenza in presenza

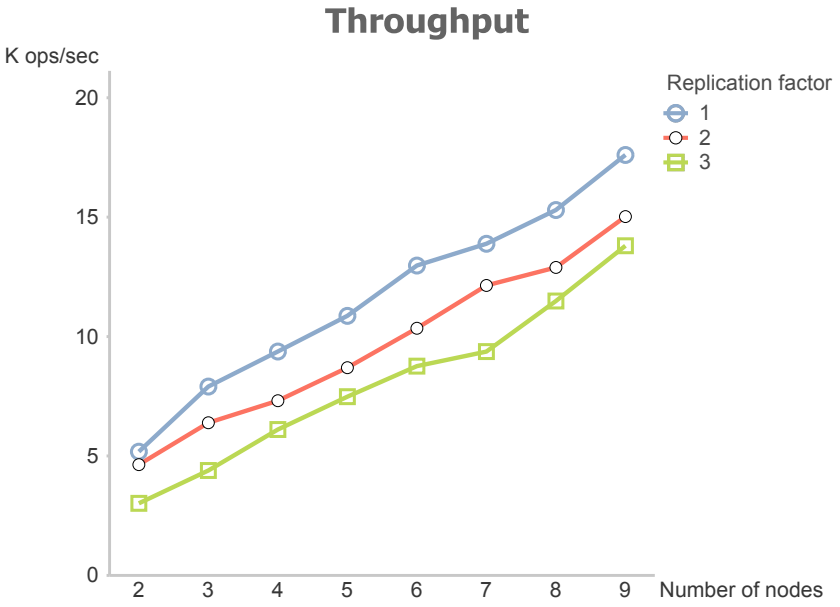

Figura 4.22: Throughput per Cassandra al variare del numero di nodi e del numero di replicazioni

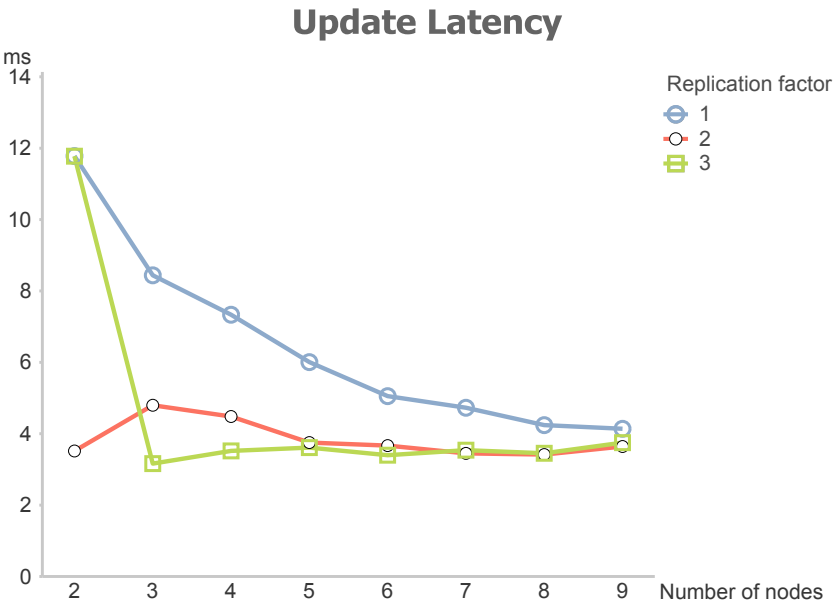

Figura 4.23: Update latency per Cassandra in funzione del numero di nodi e del numero di replicazioni

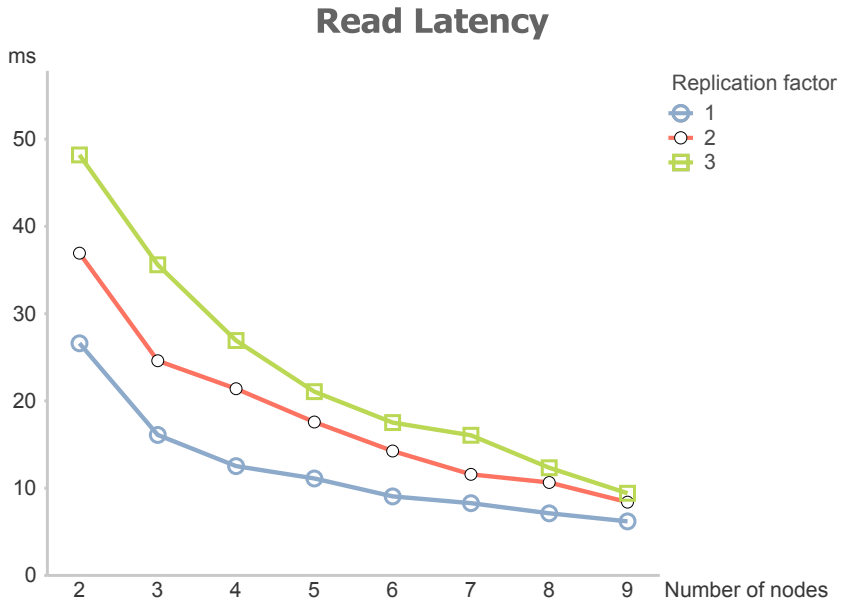

Figura 4.24: Read latency per Cassandra in funzione del numero di nodi e del numero di replicazioni

di un maggior numero di copie può essere dovuto ad un maggior tempo di gestione di queste, per cui periodicamente il database controlla consistenza tra tutte le copie presenti sul sistema.

MongoDB

La particolare gestione delle repliche offerta da MongoDB presuppone un forte impatto sulle prestazioni, a causa del numero di nodi primari che viene a diminuire. Da notare come, per replicazione pari a tre, non sia possibile costruire una architettura in cui sia mantenuto questo grado di replicazione in presenza di due soli nodi. Si ricorda inoltre che, di default, MongoDB esegue le letture sempre contattando il nodo primario, utilizzando quelli secondari solo in caso di guasto.

Dalla Fig.4.25 si nota come effettivamente avere replicazione pari ad uno, quindi solamente nodi primari, porti ad un beneficio nelle prestazioni. E' però particolare il comportamento, molto simile, nel momento in cui si vengono a creare dei replica set cui appartiene più di un nodo, per cui il throughput presenta lo stesso andamento e valori molto simili tra loro indipendentemente dal numero di nodi appartenente ad ogni replica set. Da notare anche come il tasso di crescita per replicazione uguale ad uno sia decisamente più elevato rispetto alle altre due possibilità.

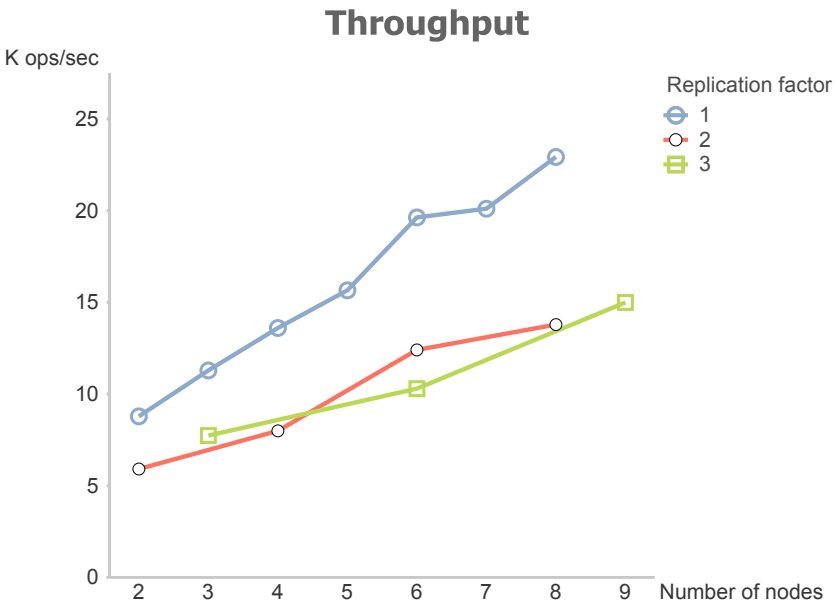

Figura 4.25: Throughput per MongoDB al variare del numero di nodi e del numero di replicazioni

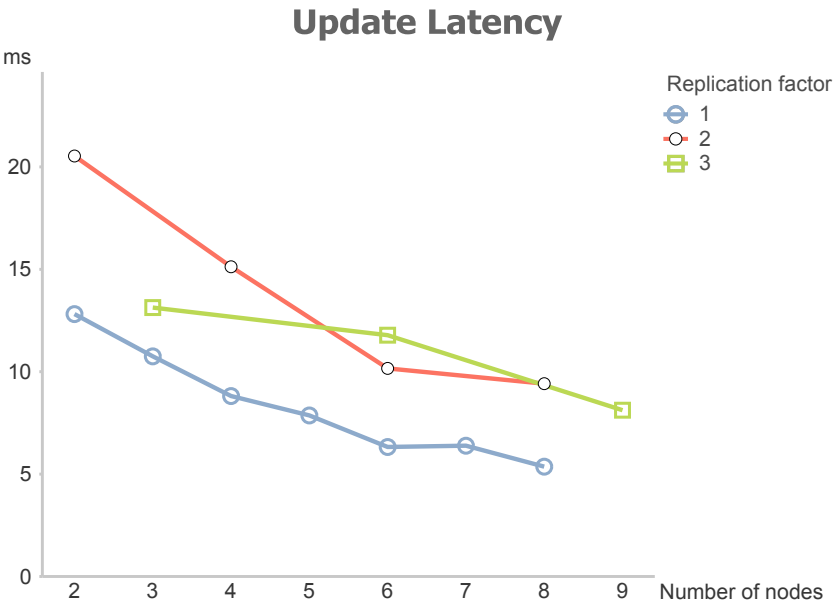

Figura 4.26: Update latency per MongoDB in funzione del numero di nodi e del numero di replicazioni

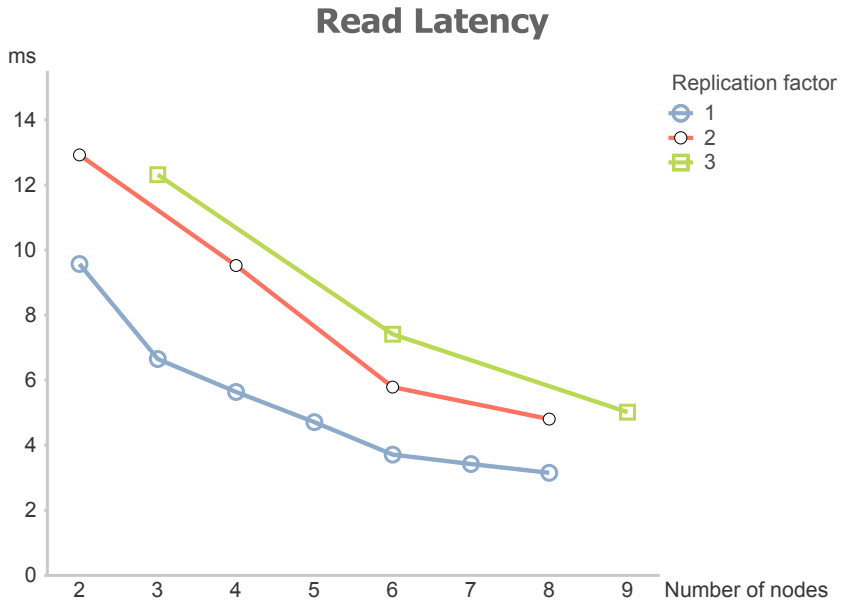

Figura 4.27: Read latency per MongoDB al variare del numero di nodi e del numero di replicazioni

Dalle latenze di tipo update, come mostrato dalla Fig.4.26, si ha la stessa caratteristica per cui avere un numero maggiore di repliche porti sì ad un deterioramento delle prestazioni, ma l'incremento di questo fattore non porta ad un conseguente aumento delle latenze.
Le latenze di tipo read tendono invece ad accentuare la differenza tra un diverso numero di replicazioni, come si nota in Fig.4.27. In questo caso, l'aumento del fattore di replicazione da due a tre porta ad un leggero aumento delle latenze, anche se in questo caso il trend rimane indifferente per tutte e tre le configurazioni.

HBase

Da HBase ci aspettavamo i risultati più interessanti riguardo la replicazione, a causa della gestione di questa affidata a HDFS.

Dalla Fig.4.28 possiamo già ritenerci soddisfatti, osservando come non ci sia una netta distinzione tra le diverse configurazioni, segno che per questo database la replicazione non comporti nessun impatto. Molto più particolare invece notare come, analizzando ogni singola configurazione, l'aggiunta di nodi possa portare ad una perdita di prestazioni, come ad esempio passando da cinque a sei nodi per una sola replica. Questo comportamento è dovuto alla gestione delle region da parte degli HRegionserver. Una distribuzione non uniforme e non predeterminata dei dati causa, in maniera del tutto non controllata, presta-

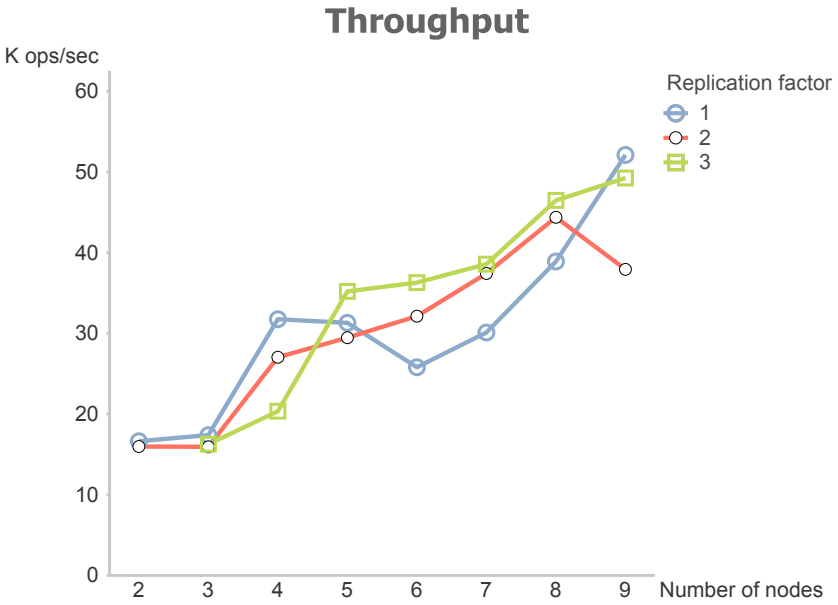

Figura 4.28: Throughput per HBase al variare del numero di nodi e del numero di replicazioni

zioni altalenanti, anche se si può comunque osservare una tendenza comune nel migliorare le prestazioni all'aggiunta di nodi. Questa cosa non è stata notata durante i test riguardo al numero dei nodi a causa del numero di questi considerato per ogni singolo test, cui spaziavano da quattro ad otto nodi, presentando un intervallo ampio. Si noti infatti che qui si è analizzata ogni configurazione con ogni numero di nodi, per cui viene accentuata questa particolarità.

Le latenze mostrano lo stesso comportamento, distinguendo a fatica le diverse configurazioni. Dalla Fig.4.29 si nota come si è sempre in presenza di latenze di update estremamente basse, con un andamento particolarmente uniforme rispetto alle latenze per le operazioni di read, mostrate in Fig.4.30. In questo caso è molto più evidente osservare l'instabilità causata dalla gestione dei dati, anche se globalmente si osserva sempre un trend portato alla diminuzione delle latenze al salire del numero di nodi.

4.5 Conclusioni

In questo capitolo sono stati analizzati i risultati ottenuti attraverso una serie di test, finalizzati alla visualizzazione e alla comprensione dei comportamenti dei database da un punto di vista prestazionale rispetto alle principali proprietà offerte da un database di

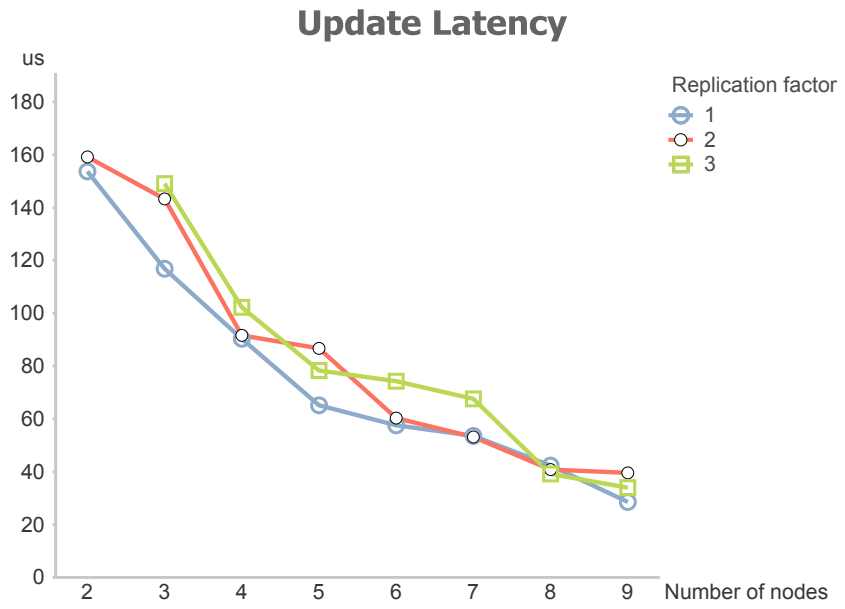

Figura 4.29: Update latency per HBase in funzione del numero di nodi e del numero di replicazioni

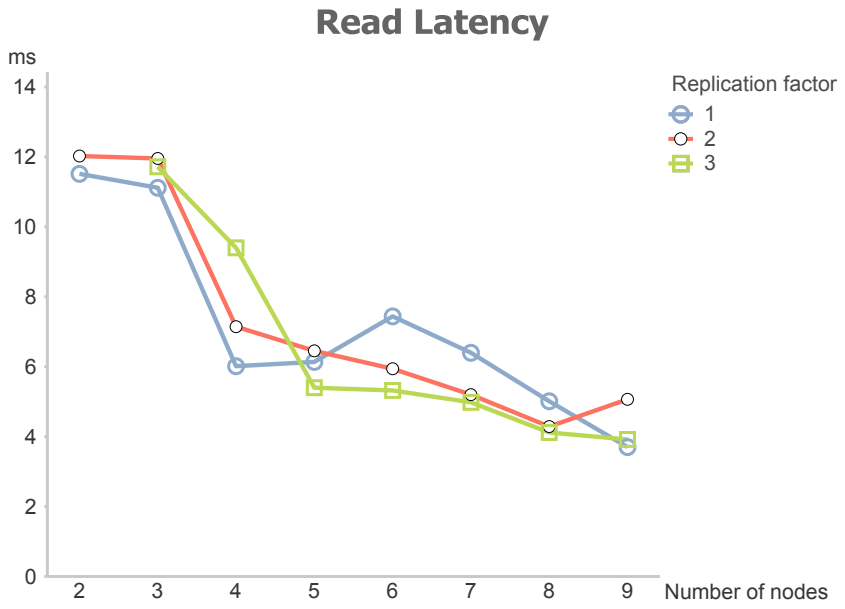

Figura 4.30: Read latency per HBase al variare del numero di nodi e del numero di replicazioni

tipo NoSQL. Molte proprietà studiate nel capitolo 2 sono state effettivamente riscontrate, come le basse latenze per operazioni di update da parte di HBase o la gestione delle sue replicazioni, ottenendo importanti informazioni anche sulla reale capacità di calcolo offerta da ognuno di questi database. Il cloud si è rivelato strumento essenziale per l'esecuzione in serie di tutti i test, grazie alla gestione delle AMI e la creazione rapida di un nodo. Si deve infatti considerare che per ogni database e ogni sua relativa configurazione, è stata ogni volta avviata una nuova istanza partendo da una macchina nuova, evitando che dati precedenti compromettessero la validità del test che si andava a svolgere.

I risultati più importanti analizzati qui saranno quindi oggetto di studio nel prossimo capitolo, finalizzato a creare uno o più modelli che riescano a catturare gli stessi comportamenti osservati in questo capitolo per ognuno dei database presi in esame.

Capitolo 5

Modelli e simulazione

Un modello è l'astrazione di un sistema che consente di studiarne gli aspetti essenziali, permettendo di non considerare il grande numero di dettagli, irrilevanti ai fini della suddetta analisi, di cui questo è composto. L'approccio modellistico permette quindi di poter risparmiare risorse che verrebbero altrimenti spese nella costruzione fisica di tali sistemi.

La costruzione di modelli che presentino un certo grado di affidabilità permette di osservare, al cambiamento di alcuni parametri come il *response time* o il numero di client, le variazioni comportamentali dei sistemi studiati, ottenendo utili informazioni a priori dell'implementazione reale dell'architettura. Le simulazioni riguardano quindi l'esecuzione di operazioni che verrebbero eseguite dal sistema così costruito, permettendo di verificare lo stato della rete, l'utilizzazione delle risorse modellate e potendo identificare eventuali problemi costruttivi derivanti dalla configurazione scelta, come colli di bottiglia o utilizzo di un servizio troppo lento e inadeguato.

Nonostante si sia consapevoli che i database sono caratterizzati da complessi meccanismi di sincronizzazione e intercomunicazione, i quali richiederebbero modelli molto dettagliati per riprodurre con particolare dettaglio il loro comportamento[44, 45], il tipo di modelli usati ai fini di questo lavoro sono le così dette *reti di code*[33], tipologia fortemente usata nell'informatica poichè permette molto facilmente di catturare le peculiarità di tali sistemi. La rappresentazione avviene attraverso dei *centri di servizio*, rappresentanti le risorse di sistema, e dei *clienti*, che identificano gli utenti e le operazioni da eseguire. Questi ultimi generano delle richieste che vengono inviate ai vari centri di servizio, i quali vengono caratterizzati da un determinato tempo medio d'esecuzione (detto *service demand*) per ogni tipologia di operazione richiesta. Questi modelli sono molto semplici, consentendo una soluzione analitica senza dover ricorrere a tool troppo complicati che richiederebbero risorse di calcolo troppo dispendiose.

I risultati ottenuti nel capitolo precedente sono qui utilizzati per permettere la costru-

zione di modelli affidabili da un punto di vista comportamentale. Si è quindi interessati a trovare modelli che riescano a ricreare il trend prestazionale dei parametri misurati nel capitolo 4, finalizzando questo lavoro all'esecuzione di simulazioni su tali modelli, permettendo uno studio più approfondito del sistema in seguito a variazioni sui parametri delle risorse. Le reti di code sono state scelte, essendo modelli non particolarmente complicati, proprio per permettere facili ed immediate modifiche al sistema così simulato, eliminando la necessità di operare uno studio, a priori, particolarmente approfondito riguardo al modello utilizzato.

L'obbiettivo preposto durante la fase di progettazione dei modelli è stato quello di creare un modello unico che riuscisse a caratterizzare tutti e tre i database, puntando a variare solamente i tempi di risposta delle singole *service stations*.

Sono stati quindi creati due modelli, il primo utilizzato per modellare i database in presenza di un singolo nodo, il secondo per configurazioni con un numero di nodi pari a quattro. Il secondo modello, grazie ai componenti di cui è composto, può essere utilizzato anche per svolgere simulazioni riguardo alla gestione di un numero di repliche maggiore di uno.

5.1 Caratterizzazione dei modelli

In Fig.5.1 è rappresentato il modello riguardante l'architettura per un singolo nodo. Come spiegato nel Capitolo 3.3, il software di benchmark è composto da un numero fissato di thread che eseguono richieste di update e di read casualmente al 50%, non satura mai (l'obbiettivo è stressare il database, non il client), ed esiste la possibilità, per HBase, di utilizzare un buffer per accumulare richieste di scrittura.

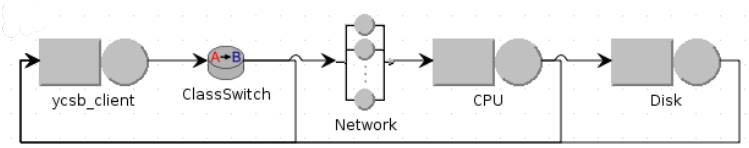

Figura 5.1: Modello di rete di code rappresentante un'architettura single node

Abbiamo modellato quindi l'intero sistema con una rete di code chiusa, dove ogni operazione lanciata dalla macchina client rappresenta la richiesta di un singolo thread del YCSB tool. Il tempo di servizio per il client, la cui istanza è rappresentata dalla service station *ycsb_client*, e il suo rispettivo numero di core sono stati scelti in modo da non rappresentare mai un collo di bottiglia per il sistema stesso. Essendo le operazioni richieste dal client appartenenti a due diverse tipologie, è stato aggiunto un *class switch*,

grazie al quale ogni operazione richiesta dal client ha il 50% di probabilità di diventare una operazione di read o, per la restante percentuale, una operazione di update.

I singoli database sono stati modellati utilizzando due service station distinte. La prima rappresenta la CPU, cioè le risorse computazionali disponibili per quel nodo, mentre la seconda rappresenta il disco rigido. Siccome nel precedente capitolo la maggior parte dei test sono stati operati con macchine fornite di CPU dual core, il numero di servizi rappresentati da ogni stazione CPU è stato impostati uguale a due, abilitando la stazione alla parallelizzazione delle operazioni. Il disco è invece caratterizzato da un solo servizio, utile per rappresentare l'accesso seriale caratteristico di questi componenti.

A questo punto sono state trovate le misure fisse, che non variano cioè in base al database o alla configurazione analizzata. La prima è la latenza di rete, introdotta come una delay station, dovuta alla connessione attraverso la rete tra tutti gli attori partecipanti, che ha mostrato attraverso il comando "`ping`" pre installato sul sistema operativo una stima di $0.45375ms$, con una varianza dello 0.032%. Le altre misure fisse riguardano la velocità di scrittura e lettura da parte del disco fisso, stimate e impostate rispettivamente a $20ms$ (update) e $17ms$ (read).

L'ultima considerazione da fare riguarda il modo in cui vengono processate le due diverse classi di richieste di cui disponiamo. Mentre le read, per tutti i database, non sono coinvolte in metodi di caching, queste vengono processate da tutte le stazioni fino ad arrivare al disco fisso. Le richieste di tipo update, invece, hanno per tutti i database la caratteristica di poter essere poste in cache, senza dover accedere al disco per ogni singola richiesta. Questo viene rappresentato da una bassa probabilità di essere inviate dalla CPU al disco, facendo tornare la richiesta come compiuta direttamente al client. Per quanto riguarda il solo HBase invece, la particolarità data dal client-side write buffer viene modellata attraverso un loop creato direttamente sul client. Questo ha infatti la possibilità di utilizzare un buffer per memorizzare tutte le richieste di scrittura, senza che queste vengano mandate al server e quindi considerate opportunamente completate. La bassa percentuale con cui una richiesta di update possa essere inviata al database rappresenta la probabilità che l'operazione appena generata sia quella che permette di raggiungere la massima dimensione disponibile del buffer, avviando la comunicazione con il database.

In Fig.5.2 è invece rappresentato il modello riguardo la configurazione *multi nodes*. Mentre l'architettura rimane sostanzialmente invariata, con ovviamente la presenza di una CPU e un disco per ogni nodo del sistema, vengono qui introdotti tre nuovi componenti, utili per rappresentare con un unico modello sia le variazioni dovute ad un maggior numero di nodi sia quelle dovute ad una replicazione dei dati.

Il primo componente è il *fork*, che consente di prendere una richiesta e replicarla un numero di volte desiderato, modellando così il carico di lavoro dovuto alla gestione di diverse copie dello stesso dato. La sua controparte, il *join*, consente invece di prendere le repliche precedentemente create e compattarle insieme prima di essere inviate al client.

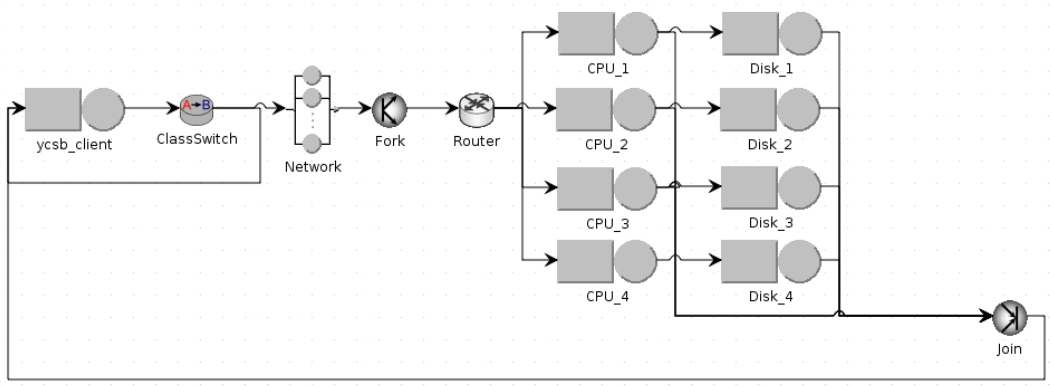

Figura 5.2: Modello di rete di code rappresentante un'architettura multi nodes

Tabella 5.1: Parametri del modello per Cassandra.

node(s)	**replication factor**	**cpu demand in *ms* (update)**	**cpu demand in *ms* (read)**
1	-	0.075	1.15
4	1	0.8	1.25
4	4	0.15	0.55

Il valore attribuito al componente fork è stato impostato uguale ad uno o a quattro in base alla simulazione desiderata. Si noti che, essendo risultato dal capitolo precedente come HBase non fosse in particolar modo suscettibile rispetto alla replicazione, questa simulazione è stata svolta per Cassandra e MongoDB.

Il terzo componente è il *router*, una stazione che permette di indirizzare le richieste ai vari nodi che lo succedono, utile per modellare il partizionamento dei dati attraverso i vari nodi del sistema e l'esecuzione abilitata di una certa operazione da parte di un solo nodo. Siccome si è qui voluto testare i database considerati con il massimo valore disponibile di entry point, tutte le stazioni CPU presentano la stessa probabilità di essere raggiunti da una operazione passante per la stazione di router.

Le Tab.5.1, 5.2 e 5.3 mostrano, in base alla configurazione adottata, i valori dati ai tempi di servizio ad ogni stazione CPU. I diversi valori assegnati a queste rappresentano come il sistema debba modificare la gestione delle proprie operazioni in seguito ad una maggior presenza di overhead, dovuto a comunicazione tra un maggior numero di nodi, oltre che a controlli di consistenza nel caso di replicazioni di dati.

Nella tabella riguardante HBase viene inoltre introdotta la colonna *P(flush)*, indicante la probabilità che la richiesta appena generata faccia raggiungere al buffer del client la

Tabella 5.2: Parametri del modello per MongoDB.

node(s)	**replication factor**	**cpu demand in *ms* (update)**	**cpu demand in *ms* (read)**
1	-	0.42	0.28
4	1	0.7	0.3
4	4	0.42	0.9

Tabella 5.3: Parametri del modello per HBase.

node(s)	**cpu demand in *ms* (update)**	**cpu demand in *ms* (read)**	**P(flush)**
1	0.25	0.43	0.03
4	0.2	0.87	0.01

sua dimensione massima, avviando una richiesta direttamente al server. Con probabilità $1 - P(flush)$, la richiesta non è spedita al server, venendo completata direttamente dal client stesso. Si noti che, per quanto riguarda l'utilizzo di caching sulle operazioni di scrittura da parte delle CPU dei vari nodi, la probabilità che una tale operazione acceda al disco è stata impostata al 10%. Al 90%, una richiesta di scrittura verrà gestita dalla CPU attraverso la cache.

5.2 Simulazioni e risultati

Una volta completati i modelli è stato possibile eseguire delle simulazioni attraverso JMT, utilizzando il componente JSIMgraph. Così come per i test svolti nel precedente capitolo, anche qui la simulazione è stata effettuata aumentando di volta in volta il numero di thread, cercando di trovare gli stessi punti di saturazione riscontrati nei database reali.

I risultati ottenuti sono mostrati di seguito, divisi in base alla configurazione modellata.

Modello single node

La Fig.5.3 mostra il throughput relativo alle simulazioni effettuate utilizzando il primo modello costruito, rappresentante l'architettura con un singolo nodo.

Tutte le simulazioni riescono a seguire il trend dei sistemi reali, per quanto si noti che il modello di HBase faccia più fatica in presenza di un numero di thread minore di quindici. Da HBase ci si aspettava infatti i risultati più discordanti, dovuti alla complessità del database, essendo costruito tramite la cooperazione di più applicazioni. Gli altri due

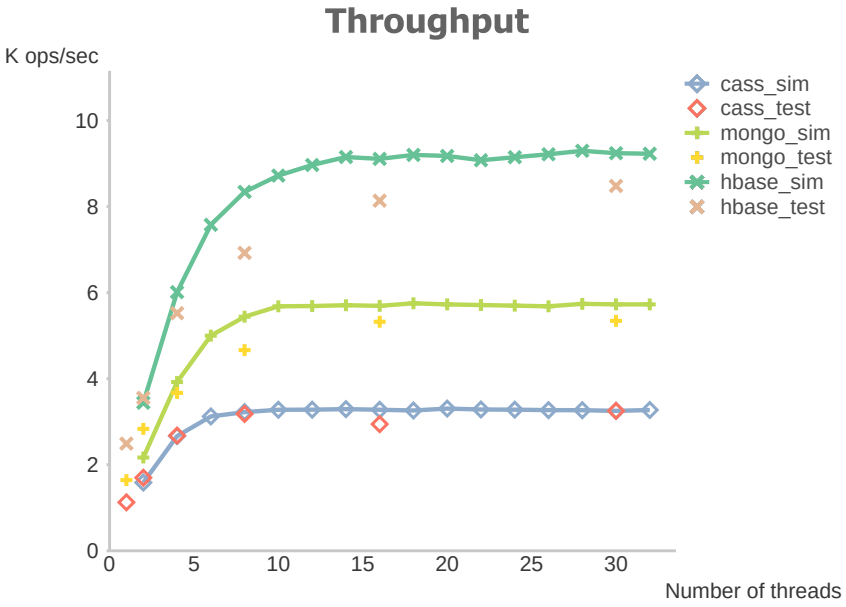

Figura 5.3: Comparazione del throughput tra modello e sistema reale per configurazione single node

modelli rispecchiano invece molto bene il trend dei sistemi reali, ottenendo valori molto vicini a quelli reali.

Le Fig.5.4 e 5.5 mostrano invece i risultati riguardo le latenze simulate. Grazie alla costruzione del modello, in particolare al loop creato intorno al client, è stato possibile ottenere degli ottimi risultati, soprattutto per quanto riguarda le operazioni di scrittura riguardo ad HBase. Si osserva infatti che le latenze sono estremamente basse, proprio come nel test originale, riuscendo a caratterizzare appieno la capacità del client di sfruttare il client-side write buffer. Gli altri due database, come per il throughput, riescono a simulare molto bene il trend reale, anche se MongoDB vede, per le update, delle latenze più alte del normale per un basso numero di thread.

Modello multi nodes

Il secondo modello analizzato riguarda la configurazione con quattro nodi presenti nel sistema. Si tenga presente che per MongoDB il sistema modellato è quello con quattro shard, impostando implicitamente una replicazione pari ad uno, valore così configurato anche per gli altri database.

In Fig.5.6 viene mostrato il throughput riscontrato attraverso la simulazione di questo modello. Anche in questo caso possiamo ritenerci molto soddisfatti, i trend di tutti i database vengono rispettati pienamente, con andamenti che tendono a saturare negli stessi

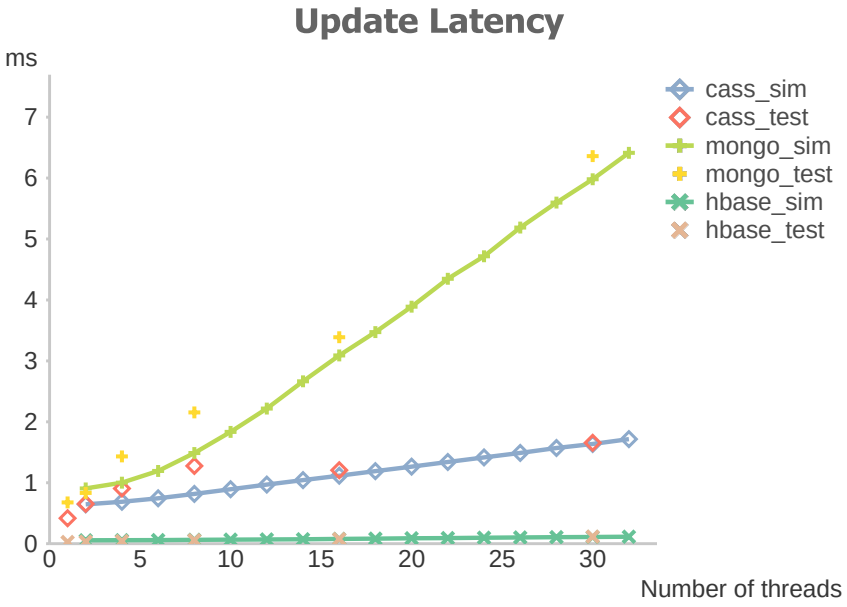

Figura 5.4: Update latency simulate e reali a confronto per configurazione single node

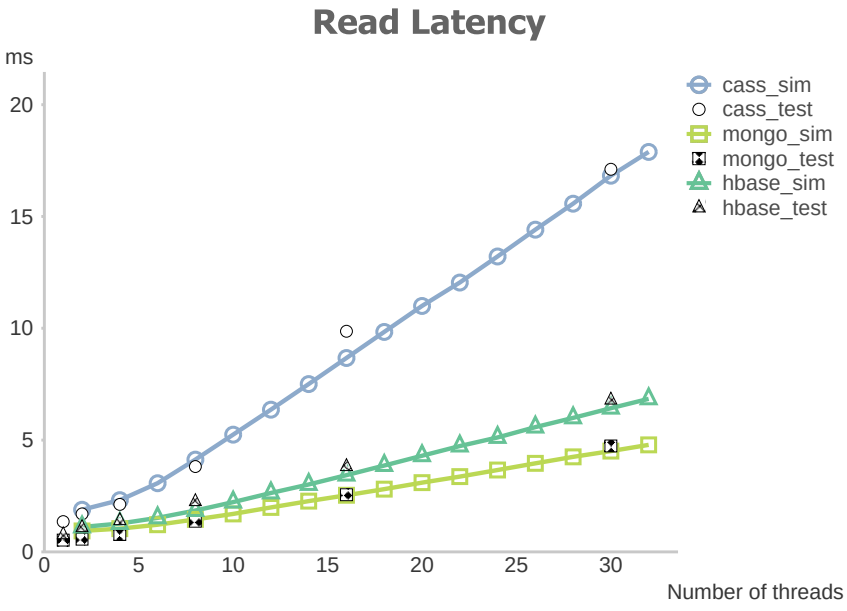

Figura 5.5: Comparazione di read latency tra modelli e sistemi reali con configurazione single node

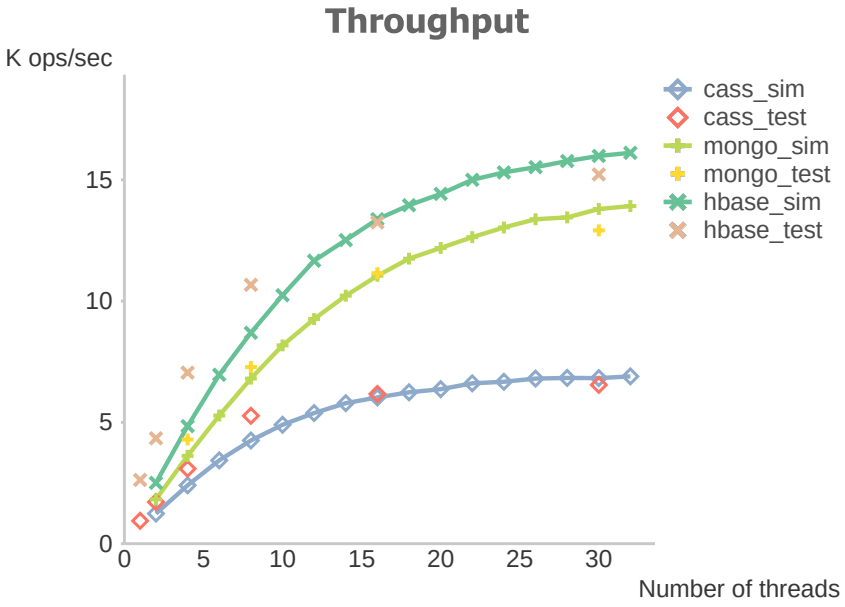

Figura 5.6: Throughput simulati e reali a confronto per configurazione multi nodes

punti dei sistemi reali. In questo caso, HBase e Cassandra sono i database che tendono ad avere problemi con un numero basso di thread. Si consideri che la simulazione, per questo modello, deve tenere conto di tutti gli overhead che vengono aggiunti da una configurazione multi-nodo, calcolo che non riesce ad essere particolarmente preciso per un numero di thread inferiore a otto.

Simulate le latenze, HBase continua sempre a dare ottimi risultati comportamentali rispetto alle operazioni di scrittura, come mostrato in Fig.5.7. In questo caso è invece particolarmente evidente come sia Cassandra che MongoDB, al contrario della configurazione con un singolo nodo, facciano molta fatica a simulare i risultati ottenuti con un numero di thread minore di dieci. Anche le letture presentano lo stesso problema: le latenze appaiono via via sempre meno precise al diminuire del numero di thread, fatto evidenziato dalla Fig.5.8. Interessante infatti notare come tutti i database, per entrambe le operazioni, mostrino per un numero basso di thread delle latenze minori rispetto alle simulazioni.

Ulteriori simulazioni su questo modello sono state svolte in funzione del numero di repliche, impostate ad un valore pari a quattro. In questo caso, per Cassandra, la cui replicazione viene espressamente esplicitata alla creazione del keyspace, il numero di repliche è stato simulato creando quattro copie, attraverso la fork, della stessa operazione. Queste venivano poi gestite dal database e, una volta completate, ricompattate in una singola operazione restituita al client.

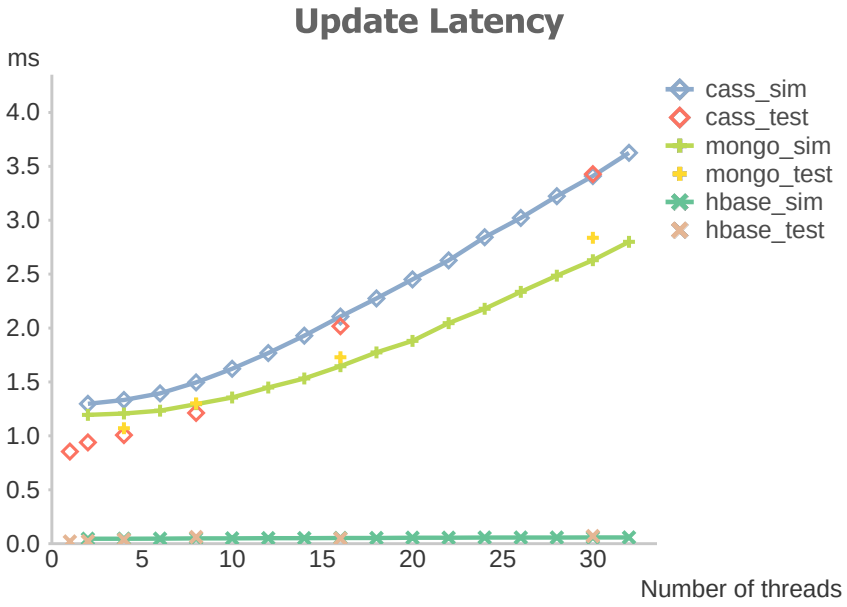

Figura 5.7: Comparazione di update latency tra modelli e sistemi reali per configurazione multi nodes

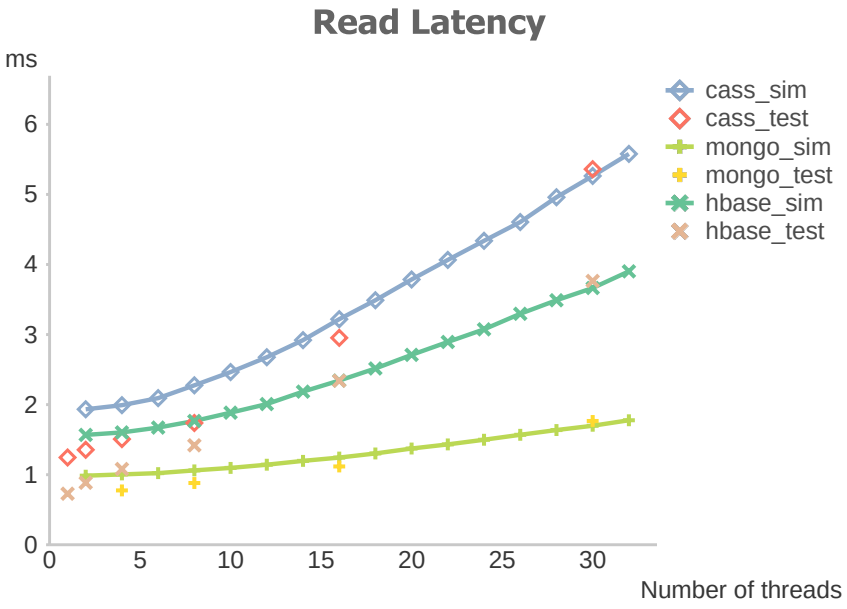

Figura 5.8: Read latency confrontate tra simulazioni e sistemi reali per configurazione multi nodes

Nel caso di MongoDB, invece, si ha bisogno di modellare il funzionamento di una singola shard composta da quattro nodi, formando così un replica set in cui si ha la presenza di un nodo primario e tre secondari. Il modello è stato allora gestito in modo da reindirizzare tutte le richieste di scrittura su una singola service station, mentre le richieste di read potevano essere eseguite da tutti i nodi. Nonostante si è consapevoli che anche le operazioni di tipo read vengono eseguite principalmente dal nodo primario, si è deciso di modellare in questo modo il sistema per avere una rappresentazione di un numero maggiore di entry point, ricordando che su ogni nodo è in esecuzione una istanza del processo mongos.

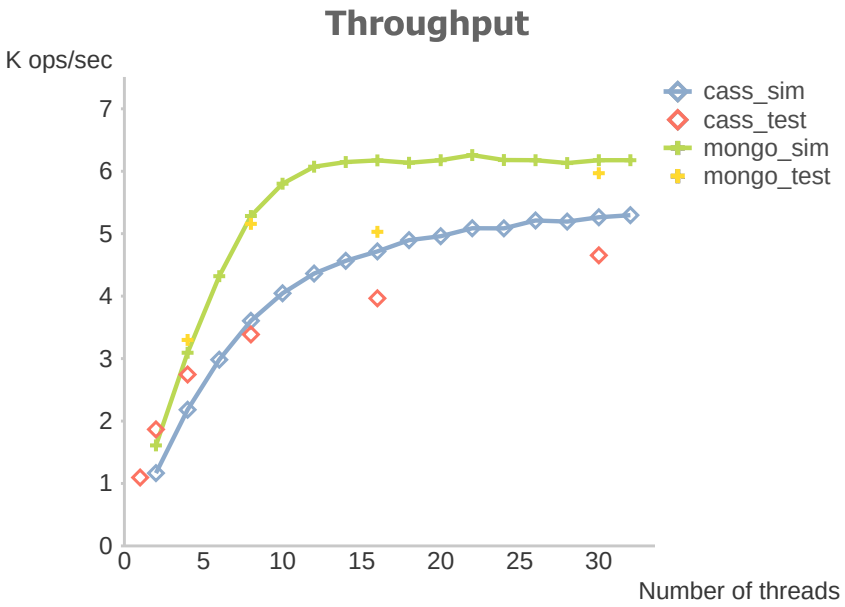

Figura 5.9: Throughput medio di modelli e sistemi reali per configurazione multi nodes con replicazione

Si osserva dunque in Fig.5.9, anche nel caso in cui il modello rappresenti un sistema particolarmente complesso, come le simulazioni riescano a rappresentare il trend delle prestazioni dei sistemi reali. Si nota una lieve perdita nell'accuratezza dei risultati, dovuta qui alla gestione da parte dei database rispetto alla distribuzione delle repliche sui vari nodi. Da considerare come MongoDB, per un numero di thread pari a sedici, mostri un valore particolarmente basso e fuori norma per quanto riguarda il test su architettura reale, segno di un calo di prestazioni dovuto all'instabilità del cloud.

La Fig.5.10 mostra le latenze delle operazioni di tipo update, mostrando come queste vengano ben simulate nonostante MongoDB si discosti leggermente, come per il throughput, per un valore di thread pari a sedici. Cassandra si comporta invece molto bene, anche se, ancora una volta, per un numero basso di thread il modello non riesce a cattu-

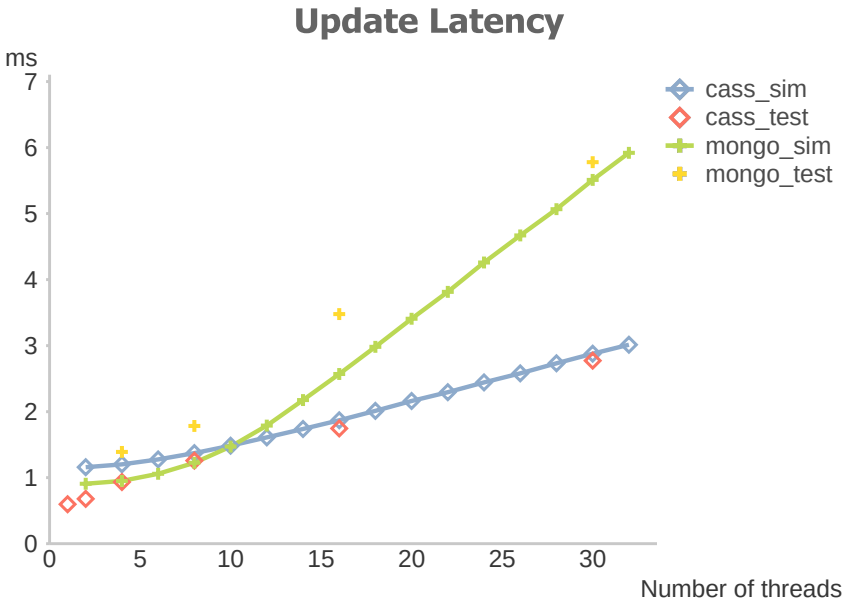

Figura 5.10: Update latency per configurazioni multi-nodes con replicazione confrontate tra modelli e sistemi reali

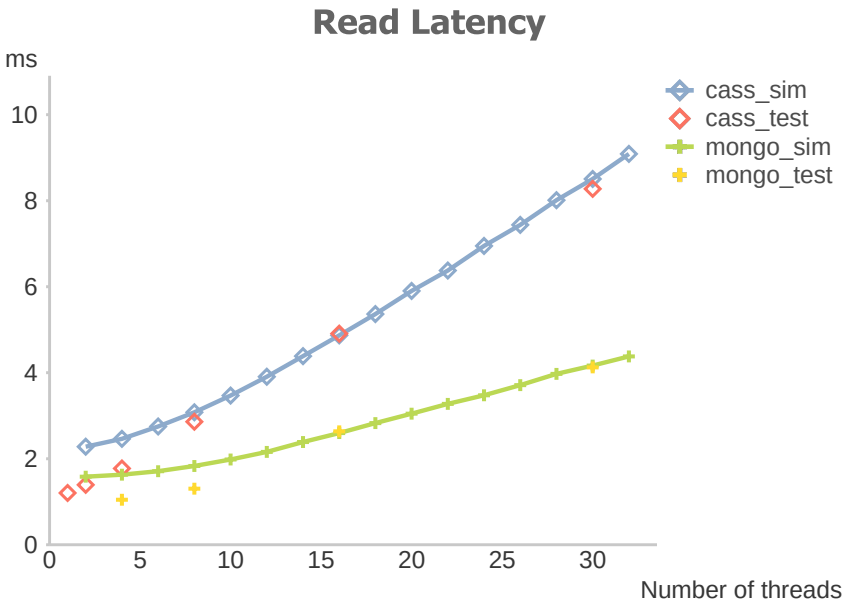

Figura 5.11: Read latency simulate e testate a confronto per configurazioni multi-nodes con replicazione

*Tabella 5.4: Errori relativi dei modelli proposti: throughput (**th**), update latency (**up**), read latency (**re**)*

node(s) repl.	**Cassandra (th/up/re)**	**MongoDB (th/up/re)**	**HBase (th/up/re)**
1 / 1	4% / 14% / 8.4%	12.2% / 16.9% / 21.9%	24.4% / 10.7% / 9%
4 / 1	15.1% / 19.7% / 23.4%	7.5% / 6.3% / 16.3%	19.6% / 33% / 30.5
4 / 4	19.3% / 23.8% / 22.4%	8.7% / 23.4% / 2.4%	N.A.

rare propriamente il comportamento reale. Infine, la Fig.5.11 presenta ottimi risultati per le latenze in lettura, mostrando come il modello riesca a simulare particolarmente bene questo tipo di operazioni.

Per concludere la presentazione dei risultati, si mostra in Tab.5.4 gli errori medi relativi per ogni database rispetto ad ogni operazione. Si noti come tutti i modelli tendano a perdere accuratezza all'aumentare della complessità del sistema simulato, mentre HBase è l'unico che presenta un errore leggermente più alto riguardo il throughput, dovuto alla sua complessità architetturale. MongoDB si dimostra invece essere il più preciso, mantenendo errori piuttosto bassi in tutte le configurazioni. Infine, Cassandra è il database che risente di più della complessità dell'architettura, perdendo via via accuratezza all'aumentare dei nodi e delle repliche gestite dal sistema.

5.3 Conclusioni

Si è visto in questo capitolo come la costruzione di modelli semplici come quelli basati sulle reti di code permetta una buona simulazione di sistemi reali. I trend di tutti i database vengono rispettati infatti in maniera precisa, riuscendo anche a trovare gli stessi punti in cui il database reale presenta una saturazione del sistema.

Le considerazioni da fare sono riguardo principalmente al numero di thread, per cui si è visto come, al diminuire di questo, la precisione del modello tende a decrescere, portando un incremento dell'errore medio relativo dell'intero modello. Si vuole poi far notare come l'errore dovuto in questo caso sia sempre positivo, portando cioè il modello ad avere valori simulati più alti rispetto al database reale. Questo può essere indice del fatto che, per un carico di lavoro molto basso, i database non riescano ad ottimizzare il loro lavoro come rappresentato invece dal modello. Si tende inoltre ad avere in generale una maggior precisione sulle latenze piuttosto che sul throughput, con quest'ultimo che tende ad essere lievemente più alto rispetto a quello riscontrato sul sistema originale. Questo indica come ci siano fattori ulteriori, rispetto alle sole latenze considerate dal modello, che affliggono

il funzionamento del database. Processi del sistema operativo o richieste di accesso in memoria da parte di altre applicazioni, possono esserne un valido esempio.

Si osserva poi come, essendo valori molto bassi, le latenze per le operazioni di update di HBase facciano salire molto l'errore in questione, valori effettivamente difficili da modellare visto il particolare modo che hanno di operare.

In ultima analisi si vuole far notare come i modelli presentino ovviamente una maggiore stabilità rispetto ai sistemi costruiti su cloud. Questo deriva dal fatto per cui il modello, volendo essere un'astrazione dei sistemi su cui si basa, non tiene conto delle fluttuazioni che ci possono essere riguardo l'utilizzo delle risorse, come può accadere, ad esempio, in seguito a momenti in cui le latenze presentino dei picchi all'interno dell'infrastruttura cloud o ad una virtualizzazione delle risorse non propriamente perfetta.

Capitolo 6

Conclusioni e sviluppi futuri

Questo lavoro di tesi è stato finalizzato allo studio delle prestazioni per tre dei maggiori database di tipo NoSQL attualmente disponibili sul mercato, con la conseguente costruzione di due modelli di reti di code in grado di simulare e catturare le caratteristiche principali osservate sui sistemi implementati.

6.1 Conclusioni

All'inizio del lavoro sono state studiate le principali proprietà dei database NoSQL, cercando un confronto diretto tra i classici database relazionali e questa nuova tipologia di database, considerando tutte le diverse famiglie in cui questi vengono suddivisi. Uno studio più approfondito su alcuni di questi, concentrandosi su implementazioni architetturali, gestione dei dati e delle replicazioni tra le tante caratteristiche considerate, ha permesso infine la scelta dei database sui quali si è poi lavorato. Un'analisi dettagliata dei database scelti è stata resa necessaria poichè si voleva comprendere, a priori, quali fossero le diverse implementazioni e proprietà che potessero impattare maggiormente sulle prestazioni di ogni database considerato. Inoltre, l'analisi di articoli inerenti a precedenti studi di prestazioni di database NoSQL ci ha portato alla decisione di effettuare dei nostri test, fornendoci dati più completi e significativi per la realizzazione dei nostri obbiettivi.

Una volta fatte queste considerazioni, si è passati alla scelta di un tool di benchmark in grado di soddisfare le nostre esigenze. Si voleva un tool in grado di stressare allo stesso modo i tre database selezionati, fornendo come dato in uscita una misura delle prestazioni riguardo alle operazioni appena eseguite. La scelta è ricaduta su di un tool sviluppato e reso disponibile da Yahoo!, strumento particolarmente diffuso e preso come riferimento da molti articoli.

Il passo successivo è stato quello di capire se la piattaforma di cloud computing, da noi inizialmente considerata per l'installazione di tutte le tecnologie di cui avevamo bisogno, fosse in grado di fornirci tutti gli strumenti di cui necessitavamo. Amazon EC2 è

stato quindi di fondamentale importanza per la rapida ed automatica installazione di un numero variabile di macchine, permettendoci di tenere in memoria le immagini dei nostri sistemi operativi per un rapido avviamento dei nodi. Sono stati quindi creati diversi script bash che ci hanno consentito di eseguire diversi test simultaneamente, oltre a collezionare i risultati ed inviarli alla macchina locale utilizzata per memorizzare tutte le misurazioni. Questo è stato necessario per l'esecuzione di un elevato numero di test, in modo da ottenere per tempo una collezione di dati abbastanza ampia da poterci considerare soddisfatti. Lo svolgimento di tutti i test è stato seguito passo passo, permettendoci di capire immediatamente quali fossero le configurazioni corrette rispetto a quello che ci si aspettava, oltre a monitorare parametri come risorse utilizzate dal client e stabilità della rete al fine di non ottenere risultati non utili.

Una volta collezionati tutti i dati e averli opportunamente studiati, si sono potuti ottenere i comportamenti di ogni singolo database in funzione delle principali caratteristiche considerate, fornendo una precisa caratterizzazione per ogni configurazione esaminata. Con questi risultati è stato poi possibile costruire dei modelli, rappresentati come reti di code, al fine di poter svolgere simulazioni e mostrare come fosse possibile un riscontro effettivo tra modello e sistema reale anche tramite l'utilizzo di modelli semplici come quelli scelti.

I principali contributi di questo lavoro sono da ricercarsi nella presentazione di risultati, relativi alle prestazioni, forniti con tutti i principali dettagli dei sistemi utilizzati per ogni singolo test. Si può quindi ora presentare un'ampia descrizione di come questi database si comportino se utilizzati a regime rispetto a precise configurazioni, essendo consapevoli di trend prestazionali e conseguenti punti di saturazione, oltre ad ovviamente valori riguardo al carico di lavoro che questi possono sopportare.

Altro contributo di questo lavoro viene dall'aver mostrato come modelli semplici quali le reti di code permettano, con una buona confidenza, di simulare sistemi complessi come i database NoSQL. E' possibile quindi poter utilizzare questi modelli per testare configurazioni basate sulle proprie necessità, cambiando la caratterizzazione del modello in pochi semplici passi.

6.2 Sviluppi futuri

Il lavoro qui svolto rappresenta un punto di partenza per lo sviluppo di nuove collezioni di dati. L'esecuzione dei test svolta per i database scelti può essere estesa su sistemi non considerati in questa tesi, offrendo ad esempio una comparazione utile tra tutte le diverse tipologie di database NoSQL, arrivando a considerare anche l'implementazione di test per database relazionali per offrire una visione completa e un confronto diretto tra questi due mondi.

Anche le caratteristiche analizzate possono essere estese, verificando ad esempio come la consistenza impatti sulle prestazioni offerte dai database, o ancora come utilizzando un modello dati differente e più specifico possa portare un aumento delle prestazioni per un determinato database.

Un capitolo a parte vedrà lo studio della *reliability*, cioè quanto un database NoSQL sia in grado di gestire un guasto e ritornare operativo nel minor tempo possibile, operando quindi test finalizzati a modificare dinamicamente la struttura del database stesso attraverso eliminazione di nodi operativi. All'aumentare del numero di nodi, infatti, le probabilità di essere in presenza di un guasto aumentano, rendendo il tempo medio di recupero un fattore determinante per un certo tipo di applicazioni.

Ulteriore attenzione potrà poi essere dedicata ai modelli. In questo lavoro sono state utilizzate le reti di code, presentando due modelli in grado di rappresentare i trend prestazionali dei database studiati. Questi potranno essere ulteriormente migliorati e arricchiti in modo da poter caratterizzare anche altre proprietà qui non considerate, portando ad un maggior dettaglio la loro espressività. Ancora, l'utilizzo di questa tipologia di modelli non vuole qui imporsi come l'unica strada possibile, volendo fornire invece uno spunto per lo sviluppo e il confronto tra il modello da noi utilizzato e l'implementazione di altre tipologie di modelli.

Bibliografia

[1] Seyed Hamidreza Afzali. Consistent range-queries in distributed key-value stores. Accessed: 2014-03-19.

[2] Inc. Amazon Web Services. Amazon ec2, http://aws.amazon.com/ec2/. Accessed: 2014-03-04.

[3] Inc. Amazon Web Services. Amazon instance types, http://aws.amazon.com/ec2/instance-types/. Accessed: 2014-03-04.

[4] Hagit Attiya and Rinat Rappoport. The level of handshake required for managing a connection. *Distrib. Comput.*, 11(1):41–57, December 1997.

[5] Marco Bertoli, Giuliano Casale, and Giuseppe Serazzi. Jmt: performance engineering tools for system modeling. *SIGMETRICS Perform. Eval. Rev.*, 36(4):10–15, 2009.

[6] Eric A. Brewer. Towards robust distributed systems (abstract). In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '00, pages 7–, New York, NY, USA, 2000. ACM.

[7] Sergey Bushik. Evaluating nosql performance: Time for benchmarking, http://gotocon.com/dl/goto-cph-2012/slides/big-data/evaluatingformance.pdf. Accessed: 2014-03-20.

[8] Rick Cattell. Scalable sql and nosql data stores. *SIGMOD Rec.*, 39(4):12–27, May 2011.

[9] Maria Chalkiadaki and Kostas Magoutis. Managing service performance in nosql distributed storage systems. In *Proceedings of the 7th Workshop on Middleware for Next Generation Internet Computing*, MW4NG '12, pages 5:1–5:6, New York, NY, USA, 2012. ACM.

[10] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, June 2008.

[11] K. Chodorow and M. Dirolf. *MongoDB: The Definitive Guide*. O'Reilly Media, 2010.

[12] LLC Citrusbyte. Redis, http://redis.io/. Accessed: 2014-03-04.

[13] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[14] Datastax Corporation. Benchmarking top nosql databases, http://www.datastax.com/wp-content/uploads/2013/02/wp-benchmarking-top-nosql-databases.pdf. Accessed: 2014-03-20.

[15] db engines. Db-engines ranking of database management systems, march 2014. Accessed: 2014-03-04.

[16] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, October 2007.

[17] Drs. R. Keur Drs. L.L. Aakster. Big data: too big to ignore. what organizations can learn from the american presidential elections.

[18] Dietrich Featherston. Cassandra: Principles and application. University of Illinois at Urbana-Champaign.

[19] Apache Software Foundation. Apache hadoop, http://hadoop.apache.org/. Accessed: 2014-03-04.

[20] Apache Software Foundation. Apache, http://www.apache.org/. Accessed: 2014-03-04.

[21] Apache Software Foundation. Apache incubator, https://incubator.apache.org/. Accessed: 2014-03-04.

[22] Apache Software Foundation. Apache zookeeper, http://zookeeper.apache.org/. Accessed: 2014-03-04.

[23] Apache Software Foundation. Cassandra, http://cassandra.apache.org/. Accessed: 2014-03-04.

[24] Apache Software Foundation. Hbase project, https://hbase.apache.org/. Accessed: 2014-03-04.

[25] Apache Software Foundation. Hdfs-default settings, http://hadoop.apache.org/docs/r2.2.0/hadoop-project-dist/hadoop-hdfs/hdfs-default.xml. Accessed: 2014-03-04.

[26] L. George. *HBase: The Definitive Guide*. O'Reilly Media, 2011.

[27] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002.

[28] E. Hewitt. *Cassandra: The Definitive Guide*. Definitive Guide Series. O'Reilly Media, 2010.

[29] IBM. Ibm cloud, http://www.ibm.com/cloud-computing/us/en/. Accessed: 2014-03-04.

[30] Lee Hye Jeong. Nosql benchmarking, http://www.cubrid.org/blog/dev-platform/nosql-benchmarking/. Accessed: 2014-03-20.

[31] Alexandros Labrinidis and H. V. Jagadish. Challenges and opportunities with big data. *Proc. VLDB Endow.*, 5(12):2032–2033, August 2012.

[32] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.

[33] Edward D. Lazowska, John Zahorjan, G. Scott Graham, and Kenneth C. Sevcik. *Quantitative System Performance: Computer System Analysis Using Queueing Network Models*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1984.

[34] Edward D Lazowska, John Zahorjan, and Kenneth C Sevcik. Computer system performance evaluation using queueing network models. *Annual review of computer science*, 1(1):107–137, 1986.

[35] Canonical Ltd. Ubuntu server 12.04, http://www.ubuntu.com/server. Accessed: 2014-03-04.

[36] Microsoft. Microsoft azure, http://www.windowsazure.com/en-us/. Accessed: 2014-03-04.

[37] Inc. MongoDB. config server instance, http://docs.mongodb.org/manual/core/sharded-cluster-config-servers/. Accessed: 2014-03-31.

[38] Inc. MongoDB. mongod instance, http://docs.mongodb.org/manual/reference/program/mongod/. Accessed: 2014-03-04.

[39] Inc. MongoDB. Mongodb, http://www.mongodb.org/. Accessed: 2014-03-04.

[40] Inc. MongoDB. mongos instance, http://docs.mongodb.org/manual/reference/program/mongos/. Accessed: 2014-03-04.

[41] Inc. Ltd. MongoDB. Mongodb data model/documents, http://docs.mongodb.org/manual/core/document/. Accessed: 2014-03-04.

[42] Inc. Neo Technology. Neo4j, http://www.neo4j.org/. Accessed: 2014-03-04.

[43] Oracle. Oracle nosql database. *white paper*, September 2011.

[44] Rasha Osman, David Coulden, and William J. Knottenbelt. Performance modelling of concurrency control schemes for relational databases. In *ASMTA*, pages 337–351, 2013.

[45] Rasha Osman and William J. Knottenbelt. Database system performance evaluation models: A survey. *Perform. Eval.*, 69(10):471–493, 2012.

[46] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 1st edition, 2009.

Ringraziamenti

Il primo ringraziamento che mi sento di esprimere va ai miei genitori, per avermi permesso di vivere questi anni al Politecnico supportandomi in ogni occasione.

Un ringraziamento speciale va a mio fratello Alessandro, per avermi sempre consigliato nella maniera più completa possibile e senza il quale non mi ritroverei ora a scrivere questa pagina.

Non posso poi non ringraziare gli amici e colleghi che mi hanno accompagnato durante tutti questi anni: Fede, Gerry, Fuma, Carlo, Fra e Mambre, sono state le persone su cui ho potuto sempre contare. Non posso poi escludere le persone all'infuori del Poli, ringraziando Lello, Chiara, Philips, Pier, Dani e Tuna.

Ringrazio infine Marco Gribaudo e Pietro Piazzolla, per la pazienza e la disponibilità che hanno saputo offrirmi durante tutto il periodo di tesi.

Milano, 1 Aprile 2014

Andrea