Università degli Studi di Milano Bicocca

Scuola di Scienze

Dipartimento di Informatica, Sistemistica e Comunicazione

Corso di laurea in Informatica

Ricostruzione di schemi da collezioni MongoDB tramite l'uso di tecniche di big data analysis

Relatore: *Prof. Andrea Maurino*

Relazione della prova finale di:

Paolo Giannone

Matricola 764081

Anno Accademico 2015-2016

Sono le scelte che facciamo che
dimostrano quel che siamo veramente,
molto più delle nostre capacità.
(Albus Silente)

Ringraziamenti

«*Mia mamma diceva sempre che le cose che perdiamo trovano sempre il modo di tornare da noi... Anche se non sempre come noi ce l'aspettiamo*»

Quattro anni fa conclusi la maturità con tantissima delusione, non riuscii a dimostrare le mie capacità e tutta la mia autostima venne completamente annullata. Fui, probabilmente, tra i peggiori. Decisi di rispondere con prepotenza a questo scherzetto che la vita mi aveva organizzato: via, a mille chilometri di distanza da casa senza pensarci due volte.

Raggiungo questo importante traguardo con molta consapevolezza: adesso sono cosciente dei miei punti di forza e delle mie debolezze, della mia grandissima volontà e dell'eccessiva testardaggine.

Difficilmente dimenticherò del teorema di Rado ed i matroidi al mio ultimo esame, dei *"syntax error"* durante l'implementazione del mio primo compilatore, degli optional durante l'esame di programmazione («Oh Pè, ma cosa sono gli optional?») e delle nottate trascorse per comprendere le equazioni differenziali e gli integrali doppi.

Desidero ringraziare il professor Maurino che mi ha supportato durante lo svolgimento del mio stage con grande pazienza e disponibilità.

Benedico la mia volontà che mi ha permesso, quando le forze scarseggiavano, di tenere duro e continuare a lavorare.

Ringrazio i miei genitori per avermi sostenuto economicamente e psicologicamente e mia sorella che è sempre stata disponibile ad aiutarmi.

Voglio ringraziare Matteo per avermi sostenuto tantissimo nella preparazione degli esami e Giuseppe per avermi accolto e sopportato nonostante la mia, a volte eccessiva, esuberanza.

Per finire, ringrazio tutti coloro che, in qualche modo, hanno contribuito a farmi crescere facendomi diventare la persona che sono attualmente.

Indice

Elenco delle tabelle

Elenco delle figure

Abstract

L'obiettivo dello stage è stato quello di ideare e realizzare uno strumento per l'analisi e la profilazione dello schema implicito di un database non relazionale basato sul modello documentale. Negli ultimi decenni il traffico sulla rete è fortemente aumentato e i database non relazionali hanno registrato un progresso esponenziale nel loro sviluppo ed utilizzo grazie al crescente bisogno di scalare in orizzontale, dove i classici RDBMS presentano diverse limitazioni. I sistemi non relazionali sono schemaless, ovvero non prevedono vincoli nella definizione dello schema dei dati prima della loro istanziazione, al contrario di quanto previsto dai sistemi relazionali. I DBMS non relazionali forniscono, pertanto, flessibilità ed elasticità ma possono portare ad errori. Per questo motivo è stato sviluppato un sistema in grado di ricostruire lo schema inferito da collezioni di documenti al fine di individuare errori di modellazione o di inserimento. Per garantire l'efficienza nell'implementazione si sono utilizzate tecniche di big data anlaysis come le map-reduce e la distribuzione dei dati tramite lo sharding dei dataset.

Parole chiave: MongoDB · Sharding · Map/reduce · NoSQL databases · Schemaless databases · big data analysis · JSON

Capitolo 1

Introduzione

Il World Wide Web (WWW) nacque nel 1991 e da quel momento le aziende cominciarono ad utilizzare la rete per vendere i propri beni e servizi. Si registrò, pertanto, un forte incremento del traffico dei dati su internet e i database iniziarono ad aver bisogno di una maggiore organizzazione e flessibilità. Nella *figura 1*, si può analizzare in che modo la quantità dei dati sulla rete sia aumentata negli ultimi 25 anni[1] [1].

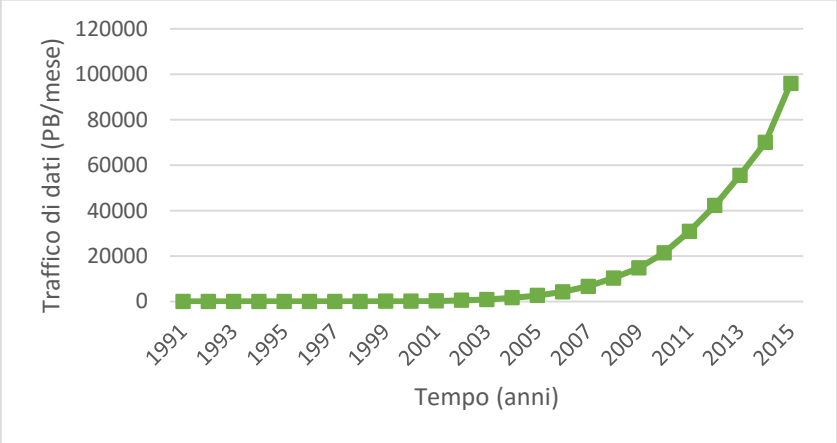

Figura 1. Andamento del traffico sulla rete negli ultimi 25 anni

[1] https://en.wikipedia.org/wiki/Internet_traffic

L'immagine precedente mette in evidenza come, con l'avvento del Web 2.0 nel 2005, ci sia stato un notevole aumento dei dati scambiati sulla rete. Per questo motivo i colossi di Internet come Amazon[2], Google[3], Facebook[4] e tutti coloro che avevano la necessità di gestire una grande quantità di dati in modo tale che fossero sempre disponibili all'utente, migrarono dai classici RDBMS ai moderni database non relazionali. Il termine NoSQL, acronimo di ***"Not only SQL"***, indica una nuova generazione di DBMS che prediligono l'alta disponibilità e la tolleranza alle partizioni piuttosto che la consistenza dei dati. Per esempio, si pensi alla classifica degli album musicali più venduti su un sito di e-commerce come Amazon.com: dal punto di vista dell'utente è preferibile accedere immediatamente alla pagina della classifica piuttosto che dover aspettare un tempo imprecisato per avere delle informazioni consistenti o aggiornate al secondo; l'interesse di una libreria online è garantire la disponibilità dei dati anche a costo di ottenere delle informazioni approssimative e non del tutto attendibili [2]. Le caratteristiche fondamentali dei database non relazionali sono **l'alta disponibilità**, **la flessibilità dello schema, la possibilità di scalare orizzontalmente** su più nodi in completa trasparenza e **l'elevata efficienza nell'estrazione delle informazioni** dalla base di dati. L'aspetto più accattivante dei database non relazionali è sicuramente, oltre che la scalabilità orizzontale, l'assenza di uno schema prefissato esplicito. Se in un database relazionale, prima di popolare la base di dati è necessario aver costruito lo schema, in uno non relazionale i dati sono memorizzati senza la necessità di avere uno schema prefissato. Si deve però fare una considerazione: questo non significa che

[2] https://www.amazon.it/
[3] https://plus.google.com/collections/featured
[4] https://www.facebook.com/

non sia necessario elaborare una struttura ben precisa da seguire nella popolazione del database ma piuttosto che, in determinate circostanze, si può decidere di salvare i dati in modo diverso rispetto a quanto fatto precedentemente, inserendo nuovi attributi o eliminandone altri che sono relativi a delle informazioni inesistenti oppure non conosciute. Se per esempio supponiamo di voler costruire un sistema non relazionale "document oriented" che tenga traccia delle informazioni relative agli attori che hanno recitato nel film *Harry Potter*, potrebbe avere senso inserire un attributo *"date of death"* che descriva la data di morte di un attore. Chiaramente questo campo sarà presente solo ed esclusivamente per quegli attori defunti, mentre per quelli ancora in vita questa informazione non sarà aggiunta. Nella *figura 2* sono stati modellati due documenti direttamente dalla corrispondente pagina di Wikipedia[5].

[5] https://en.wikipedia.org/wiki/Harry_Potter_(film_series)

```
{
    "_id" : OjectId("573f972f4fff1c8da589e9ee"),
    "name" : "Alan",
    "middle name" : "Sidney Patrick",
    "surname" : "Rickman",
    "date of birth" : ISODate("1946-02-21T00:00:00.000+0000"),
    "date of death" : ISODate("2016-01-14T00:00:00.000+0000"),
    "age" : NumberInt(69),
    "role" : "Severus Snape"
}
{
    "_id" : ObjectId("573f972f4fff1c8da589e9ed"),
    "name" : "Emma",
    "middle name" : "Charlotte Duerre",
    "surname" : "Watson",
    "date of birth" : ISODate("1990-04-15T00:00:00.000+0000"),
    "age" : NumberInt(26),
    "role" : "Hermione Jean Granger"
}
```

Figura 2. Documenti che descrivono due attori che hanno recitato nella serie cinematografica di Harry Potter

Nel documento che descrive Alan Rickman[6], recentemente deceduto, sarà definito l'attributo che riporta la data di morte dell'attore; per l'attrice Emma Watson[7] , l'attributo *"date of death"* viene omesso. Si può notare, pertanto, come esista a tutti gli effetti uno schema da seguire, ma in assenza di determinate informazioni che potrebbe non avere senso inserire, non si ha alcun tipo di limitazione come accadrebbe in un database relazionale in cui si dovrebbe fare attenzione a gestire correttamente i valori Null per quei campi il cui valore non si conosce o è inesistente. Spesso infatti, prima di inserire un record all'interno di un database non relazionale, è buona prassi utilizzare dei software appositi che consentano di validare lo schema, per garantire la correttezza dei dati; per fare un paragone è lo stesso lavoro che si svolge quando si utilizza il DTD o l'XML-schema per validare un documento XML, verificando se quest'ultimo segue o meno la sintassi che si è definita a priori.

[6] https://en.wikipedia.org/wiki/Alan_Rickman
[7] https://en.wikipedia.org/wiki/Emma_Watson

Analizzare un sistema noSQL per individuare eventuali errori di modellazione o di inserimento risulta, a causa dell'elevata elasticità dello schema, un lavoro particolarmente complesso. Con questa relazione si descriverà l'implementazione di un applicativo scritto in **java** che si serve di tecniche per l'analisi di big-data per profilare lo schema di un database document oriented, nel particolare MongoDB.

Nella prima parte di questo lavoro si analizzeranno brevemente i database non relazionali e ci si focalizzerà in modo particolare sulla descrizione di MongoDB. Successivamente si esamineranno le scelte implementative adottate per la realizzazione di un algoritmo per la profilazione del database soffermandosi sulle map-reduce e le politiche di sharding. Si passerà, in seguito, ad esaminare in che modo è stata realizzato l'applicativo, mettendo in risalto l'implementazione front end e back end. Per finire, si dedicherà un capitolo al testing ed al debugging dell'applicazione per valutare le performance e descrivere i possibili miglioramenti che possono essere realizzati.

Capitolo 2

La struttura di MongoDB

In questo capitolo si analizzeranno i componenti che costituiscono un mongo database. Dopo aver esaminato le strutture di alto livello, si prenderà in considerazione l'elemento cardine di MongoDB, cioè il documento. Si vaglieranno tutti i possibili tipi, sia primitivi sia complessi, che un attributo della base di dati può assumere. Dopo aver studiato le singole parti che compongono il database, si illustrerà un modello generale che stabilirà delle regole sintattiche complessive da rispettare per il corretto riconoscimento dello schema di un mongo database.

2.1. Gli elementi di MongoDB

MongoDB è costituito dai seguenti elementi:

- **Database:** rappresenta l'elemento di livello più elevato; all'interno di un'istanza di mongoDB possono sussistere zero o più database, univocamente identificati dal nome [3]. Se un database relazionale è un contenitore di tabelle

e viste, un mongo database è un contenitore fisico di strutture chiamate collezioni.

- **Collezione:** è un contenitore di documenti. Una collezione può essere considerata, seppur con molte differenze, come una tabella di un database relazionale. Nella stessa collezione dovrebbero essere collocati i documenti che sono semanticamente e logicamente correlati. Ciascuna collezione è univocamente identificata in un database dal nome che le viene assegnato durante la sua creazione. Spesso, riferendosi a MongoDB, si mette in evidenza l'impossibilità di effettuare dei join tra collezioni; con la versione 3.2 il DBMS ha introdotto l'operatore aggregativo *$lookup*[8] che offre la possibilità di realizzare dei left join su collezioni non shardate presenti sullo stesso database [4].
- **Documento:** è l'unità di base di un database mongoDB. I documenti sono un'entità astratta organizzata in coppie chiave-valore. All'interno di una collezione sono presenti, di norma, più documenti identificati univocamente da un campo **_id** che MongoDB genera automaticamente. A differenza dei records di un database relazionale, i documenti hanno uno schema dinamico e, nonostante appartengano alla stessa collezione, non è detto che debbano necessariamente definire lo stesso insieme di attributi.

Per maggior chiarezza, si mostra al lettore il diagramma in *figura 3* in cui sono presenti tutti gli elementi di alto livello di MongoDB [5]. Su un'istanza di MongoDB possono sussistere più database identificati univocamente da un nome; ogni database può contenere zero o più collezioni; all'interno di ogni collezione trovano posto i documenti che comprendono un certo numero di coppie chiave-valore.

[8] https://docs.mongodb.com/manual/reference/operator/aggregation/lookup/

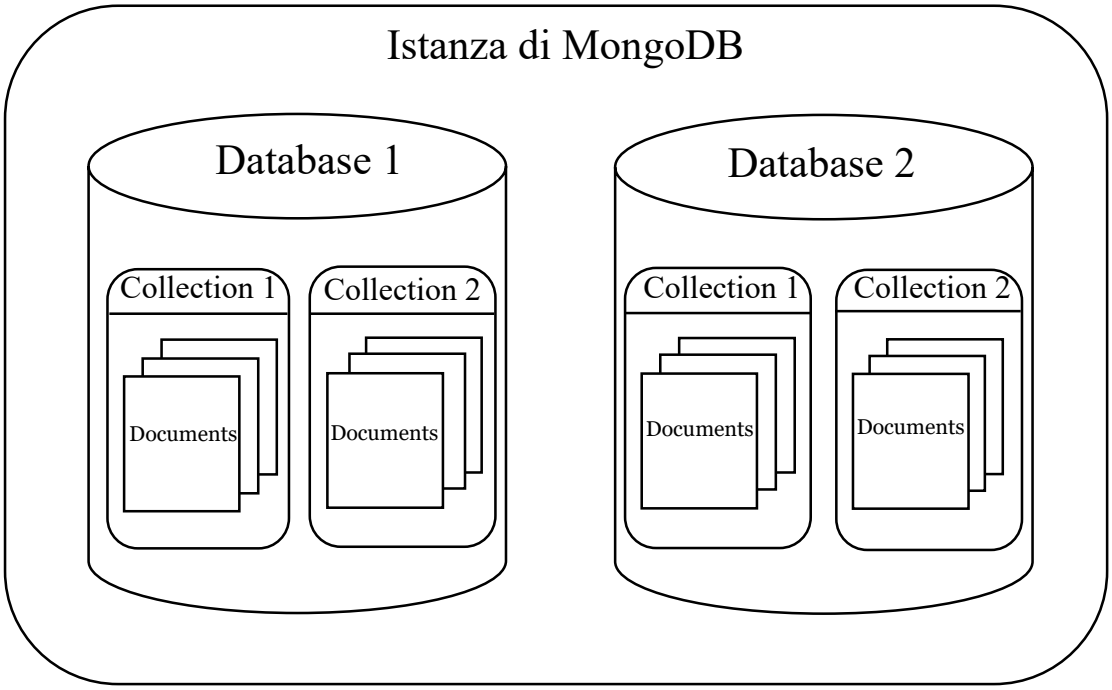

Figura 3. Struttura degli elementi di alto livello di MongoDB

2.2. Il cuore di MongoDB: il documento

Il cuore di MongoDB è il documento, un'entità astratta costituita da coppie chiave-valore. Nella *figura 4* si mostra come è costituito un documento che descrive un personaggio dei fumetti DC Comics' modellato utilizzando le informazioni rinvenute sulla corrispondente pagina di Wikipedia[9]:

[9] https://en.wikipedia.org/wiki/Green_Arrow

```
{
    "_id" : ObjectId("573f972f4fff1c8da589e9bf"),
    "name" : "Oliver",
    "surname" : "Queen",
    "alter ego" : ["The hood", "Arrow",
                   "Al Sahim", "Green Arrow"],
    "created by" : [
        {
        "name" : "Mort",
        "surname" : "Weisinger",
        "nationality" : "American"
        },
        {
        "name" : "George",
        "surname" : "Papp",
        "nationality" : "American"
        }
    ]
    "portrayed" : {
        "name" : "Stephen",
        "surname" : "Amell",
        "age" : 35,
        "nationality" : "American"
    }
}
```

Figura 4. Documento che descrive un supereroe dei fumetti

Dall'analisi della struttura del documento sopra riportato, si evince che MongoDB utilizza il **JSON** (JavaScript Object Notation) per memorizzare i records nella base di dati. Internamente i documenti vengono serializzati in una codifica binaria del JSON, nota come **BSON** (binary JSON); tale codifica permette di innestare oggetti ed array all'interno di altri oggetti esattamente come fa il JSON. Inoltre, utilizzando una codifica binaria del JSON, l'estrazione delle informazioni dalla base di dati risulta più efficiente rispetto all'uso di un formato non serializzato [6].

Come detto precedentemente, un documento contiene coppie chiave-valore. La **chiave è sempre una stringa** che permette di identificare univocamente un campo all'interno di un documento; **in altri termini la key è la stringa utilizzata per definire il nome di un attributo all'interno di un documento.** Il valore associato alla chiave può essere di **tipo primitivo** (ad esempio una stringa, un numero intero, un booleano, un time-

stamp e così via) oppure di **tipo complesso**, cioè un array oppure un object. La *figura 5* riassume la conformazione di un generico attributo.

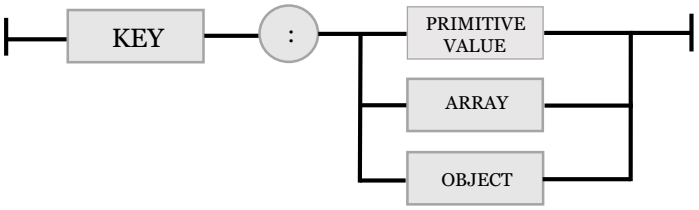

Figura 5. Struttura di un attributo

2.2.1. I tipi primitivi

In questa sezione si analizzeranno i tipi primitivi che MongoDB mette a disposizione. Nel documento in *figura 4* compaiono attributi di tipo *String*, *Double* e *ObjectID*. MongoDB offre una ricca quantità di data types primitivi per consentire all'utente di utilizzare il formato migliore in base alle circostanze [7]:

- **Double**: è utilizzato per salvare numeri in virgola mobile. Si tenga presente che MongoDB utilizza di default il tipo *Double* per salvare dati numerici. Per esempio l'attributo *"age" : 22,* sarebbe salvato come un *Double* nonostante non sia un numero in virgola mobile.
- **32-bit integer**: permette di memorizzare i numeri interi minori o uguali di 2.147.483.647. Per salvare un numero come un *Int32* è necessario utilizzare la clausola *NumberInt(x)*, per esempio *"age" : NumberInt(22)* sarebbe salvato come un intero a 32 bit.
- **64-bit integer**: consente di registare i numeri interi maggiori di 2.147.483.647. Per salvare un numero come un *Int64* è necessario utilizzare la clausola *NumberLong(x)*, per esempio l'attriburo *"age" : NumberLong(22)* è un Int64.

- **String**: è probabilmente il tipo più utilizzato e serve per memorizzare dati che rappresentano delle stringhe di testo. Per salvare un attributo come *String* è sufficiente racchiudere il valore tra apici.
- **Binary data**: consente di registrare nella base di dati delle stringhe arbitrarie di byte. JSON e BSON sono in grado di codificare e decodificare soltanto stringhe valide secondo la codifica *UTF-8*. I dati che non rispettano tale codifica posso essere memorizzati come binary data, sotto forma di stringhe di byte [8].
- **Null**: mongoDB offre la possibilità di memorizzare degli attributi di tipo *Null*, per esempio *"nullField" : null.*
- **ObjectId**: il campo *_id,* che identifica univocamente un documento in una collezione, viene automaticamente generato da MongoDB e memorizzato come un *ObjectId*. Tale attributo consiste di una stringa di 12 byte calcolata considerando il nome della macchina e il timestamp.
- **Boolean**: permette di salvare attributi il cui valore può essere *true* o *false*. Si presti attenzione a non racchiudere tra apici i valori di verità, pena la memorizzazione dell'attributo come una stringa.
- **Date**: è il formato che MongoDB mette a disposizione per il salvataggio delle date. Per registrare correttamente una data, bisogna utilizzare la clausola *ISODate("date")*, per esempio il campo *"date": ISODate("2016-07-28T00:00:00.000+0000")* corrisponde alla data del 28-07-2016.
- **Regular Expression**: è il data type che MongoDB offre per salvare le espressioni regolari. Per registrare un campo come *Regular Expression*, è sufficiente racchiudere l'espressione regolare tra slash, per esempio *"regex": /[a-zA-Z]+/.*

- **Timestamp**: MongoDB permette di memorizzare nella base di dati le marche temporali utilizzando la clausola *Timestamp()*, per esempio "*tstamp": Timestamp*() è un timestamp che rappresenta la data e l'ora di inserimento dell'attributo nel database.

2.2.2. Gli array

Gli array sono dei tipi complessi che possono contenere valori di tipo primitivo oppure, a loro volta, campi di tipo complesso (array o oggetti). L'*immagine 6* illustra la struttura di un array.

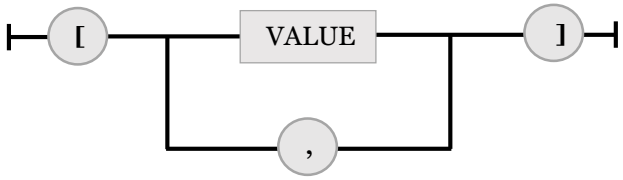

Figura 6. Struttura di un array

Gli elementi contenuti all'interno di un array, sono racchiusi in delle parentesi quadrate e divisi da virgole.
Un array è un contenitore di valori: per esempio il campo *"alter ego"* del documento rappresentato in *figura 4*, è un array che contiene valori di tipo primitivo, nella fattispecie stringhe. L'attributo *"created by"* della stessa immagine è un array i cui valori sono di tipo complesso, cioè due oggetti che consentono di descrivere i creatori del fumetto.

2.2.3. Gli oggetti

Una caratteristica fondamentale di MongoDB è la possibilità di innestare documenti all'interno di documenti ed array. Nel gergo del BSON i documenti vengono definiti semplicemente oggetti. La *figura 7* mostra la conformazione degli oggetti.

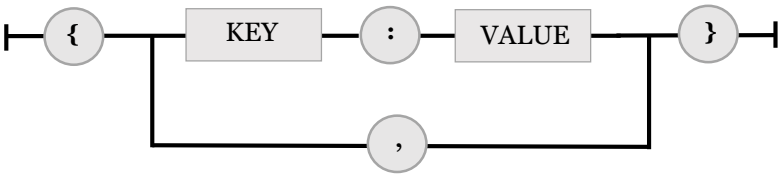

Figura 7. Struttura di un oggetto

All'interno di un oggetto possono essere inserite nuove coppie <*key, value*>; il valore associato alla chiave può essere, a sua volta, sia di tipo primitivo sia di tipo complesso.

L'attributo *"portrayed"* del documento raffigurato in *figura 4* è un oggetto (cioè un documento) che consente di specializzare con nome, cognome, età e nazionalità l'attore che interpreta il supereroe nelle rappresentazioni cinematografiche. In MongoDB, quando si innestano documenti all'interno di altri documenti si parla di **modello embedded o denormalizzato**.

2.3. La struttura di un dataset

Nei paragrafi precedenti si sono analizzati singolarmente tutti gli elementi che costituiscono un documento. Un dataset è l'insieme dei documenti in cui è organizzata una collezione. In questa sezione si delineerà un modello generale che l'algoritmo per la profilazione, descritto prossimamente in questa trattazione, deve rispettare per riuscire a riconoscere correttamente un qualsiasi dataset.

La *figura 8* illustra graficamente la conformazione di un dataset. Uno o più oggetti fanno parte di un dataset. La relazione che intercorre tra il dataset e i documenti che lo costituiscono è una **composizione** perché se il dataset venisse cancellato anche tutto il suo contenuto sarebbe perso. Ogni documento comprende zero o più chiavi a ciascuna delle quali è associato un unico *value*. Se il documento viene eliminato dalla collezione anche tutte le sue coppie <*key, value*> vengono cancellate, quindi oggetti e chiavi sono anch'essi legati da una **composi-**

zione. Il *value* associato ad una certa chiave è una **generalizzazione** di tre possibili forme che un valore può assumere: object, array e valore di tipo primitivo. Un array può contenere zero o più object e/o zero o più valori di tipo primitivo.
Si tenga presente che, grazie alla generalizzazione, il *value* di una certa chiave può essere anche di tipo complesso.

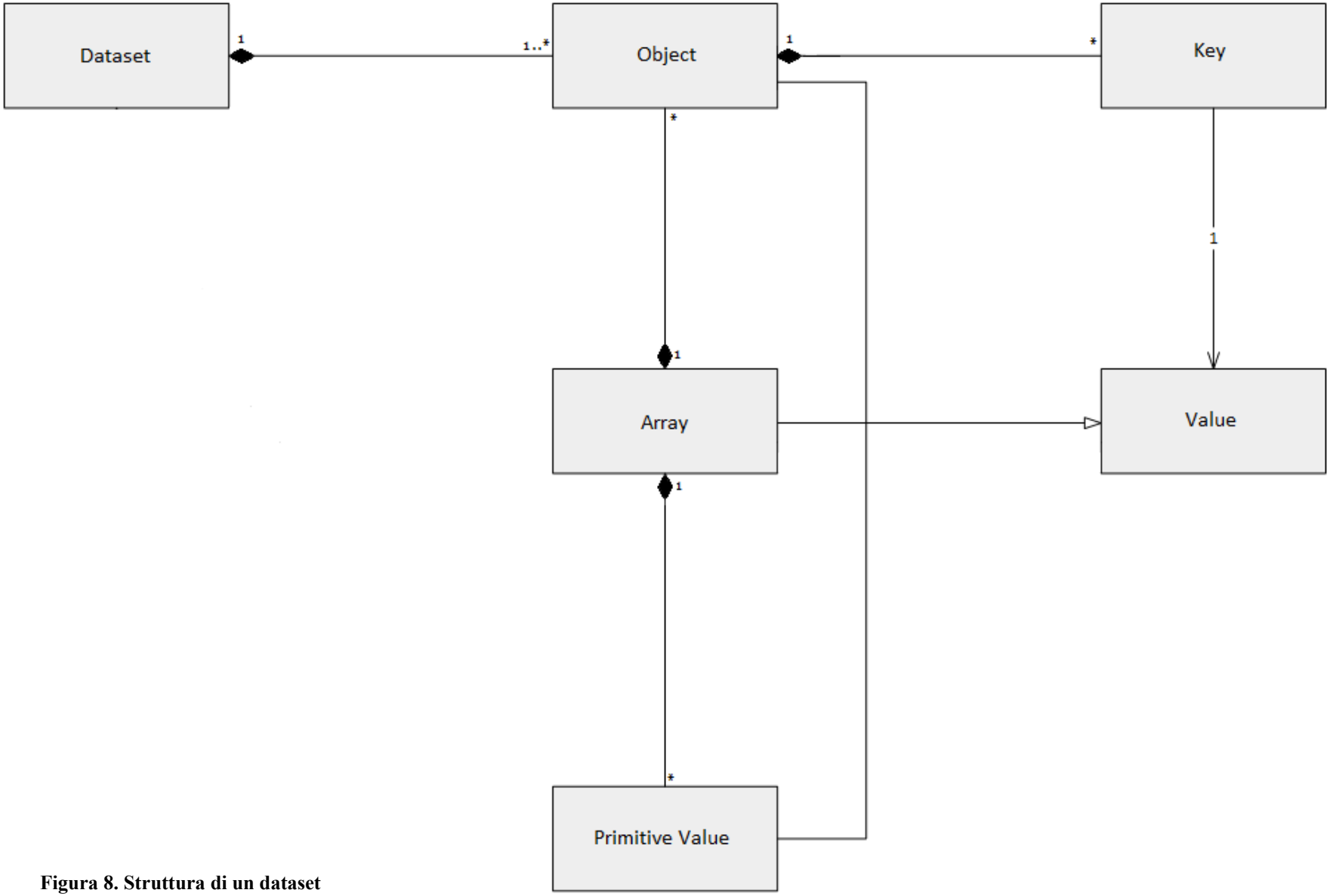

Figura 8. Struttura di un dataset

2.4. Caratteristiche dell'applicativo per la profilazione

Si vuole concludere questo capitolo offrendo al lettore una visione generale delle caratteristiche e delle funzionalità che l'applicativo per la profilazione di un dataset implicito deve necessariamente garantire come conseguenza delle peculiarità dello schema che sono state descritte nei paragrafi precedenti:

- MongoDB è un database schemaless, pertanto l'algoritmo per la profilazione del database deve **scandire tutti i documenti** presenti in una collezione; infatti nonostante si stiano analizzando documenti semanticamente correlati, non è detto che essi definiscano lo stesso insieme di attributi;
- per ciascun documento è necessario **analizzare tutti gli attributi** al fine di identificare il tipo (o i tipi), il numero di occorrenze e la frequenza relativa di ciascun campo all'interno del dataset; l'applicativo deve riconoscere tutti i tipi descritti nei paragrafi 2.2.1, 2.2.2 e 2.2.3.
- l'algoritmo per la profilazione deve funzionare a **qualsiasi livello di profondità** e, per questo motivo, è necessario definire un **meccanismo ricorsivo** se il valore associato ad una chiave è un oggetto oppure un array di oggetti, come si evince dal diagramma di flusso in *figura 9*. Per ogni documento presente in una collezione si dovrà eseguire la scansione degli attributi. Se l'attributo è di tipo primitivo, allora si salvano le informazioni necessarie e si prosegue con l'analisi del prossimo campo; se un attributo è di tipo complesso sarà necessario analizzare il contenuto di tale campo prima di procedere con l'analisi degli altri attributi.

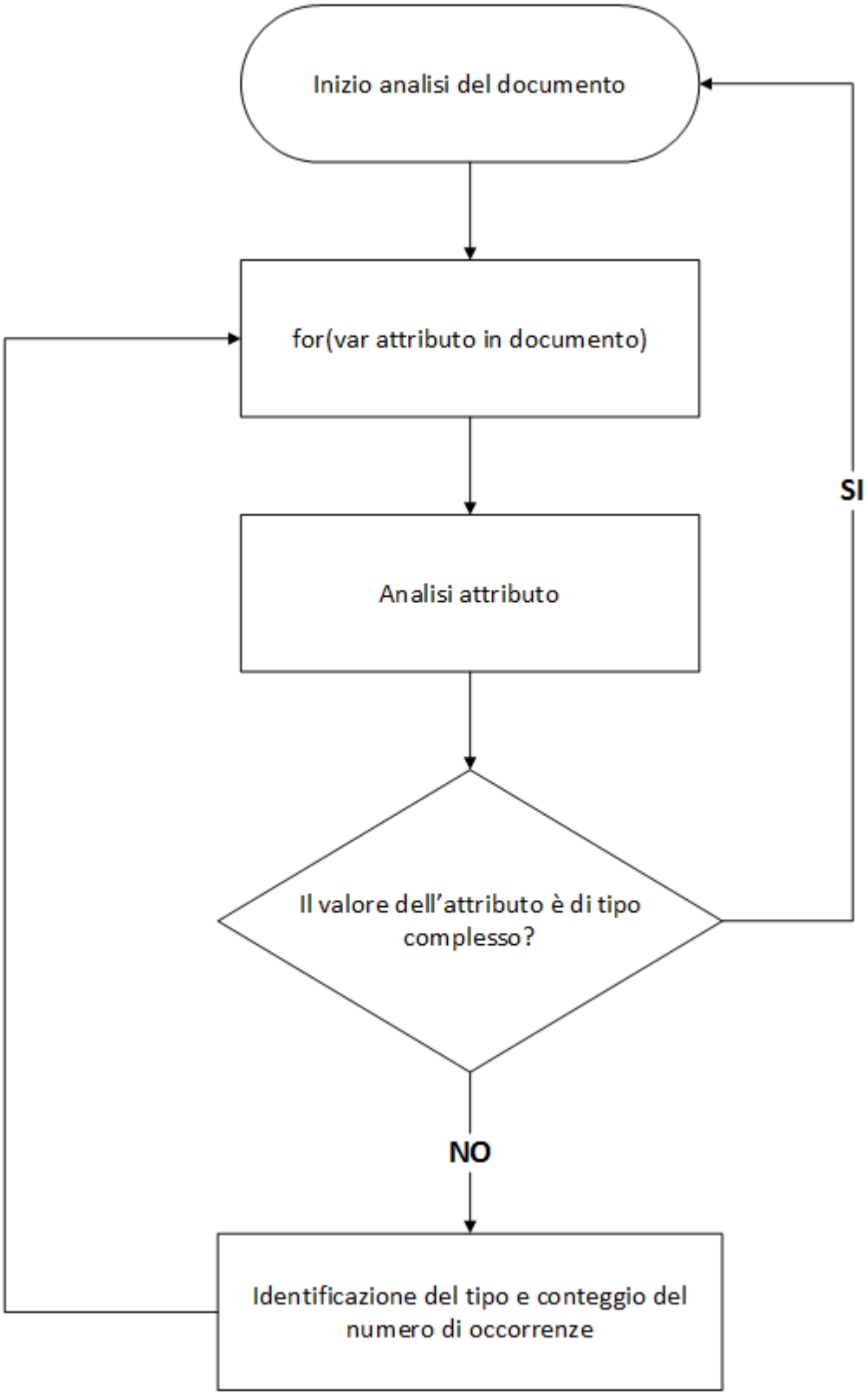

Figura 9. Diagramma di flusso per spiegare la necessità della ricorsione nell'algoritmo per la profilazione di un dataset implicito

- l'applicativo deve **estrarre dalla base di dati tutti i valori** associati ad un determinato attributo disegnando una tabella come quella che segue, che descrive il campo *"cuisine"* di un dataset di esempio:

Valori del campo *"cuisine"* | Tipo dei valori del campo *"cuisine"* | # di occorr. dei valori del campo *"cuisine"*

cuisine	type	Number of occurre...
American	String	1981
Chinese	String	1174
Other	String	972
Café/Coffee/Tea	String	511
Spanish	String	410
Pizza	String	371
Japanese	String	325
Italian	String	282

Figura 10. Tabella ottenuta estraendo i valori associati all'attributo cuisine

Considerando la tabella in *figura 10*, l'attributo *"cuisine"* assume 1981 volte il valore "American", 1174 volte il valore "Chinese", 371 volte il valore "Pizza" e così via. Ad ogni valore è associato anche il tipo: in questo caso sono presenti solamente stringhe.

- sulla base dei records presenti nella tabella in *figura 10*, l'applicazione deve disegnare un istogramma per rappresentare graficamente la distribuzione dei valori associati ad un determinato attributo. La *figura numero 11* mostra l'istogramma che sarà implementato.

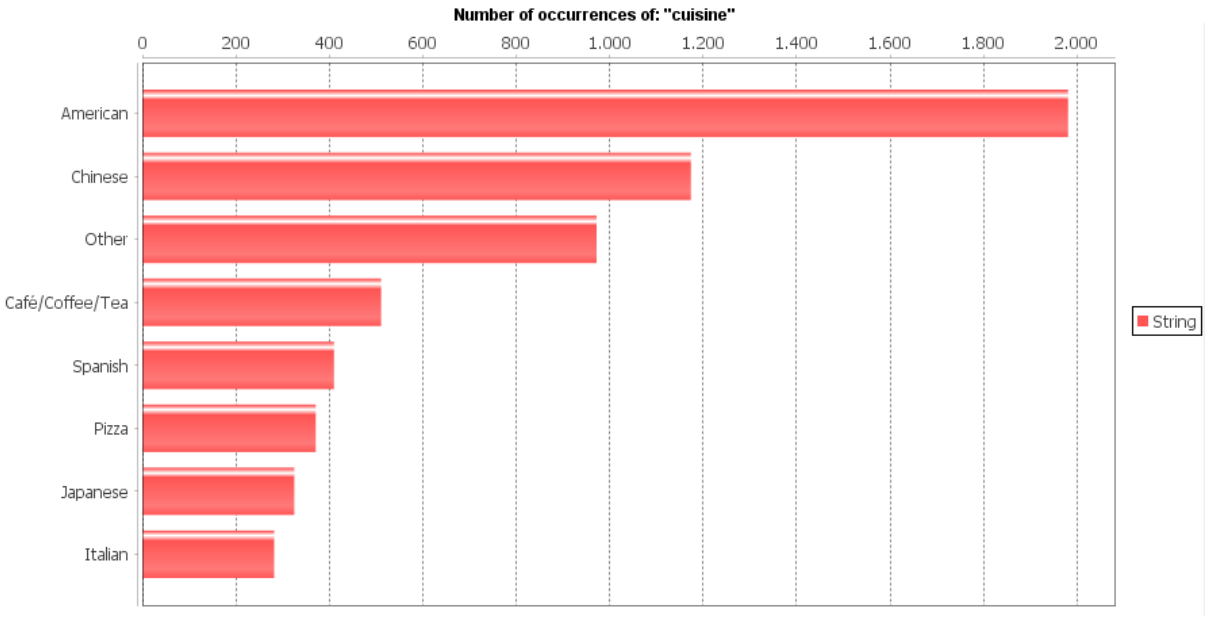

Figura 11. Istogramma per la visualizzazione della distribuzione dei valori di un attributo

L'istogramma e la tabella, rispettivamente in *figura 10 e 11* saranno tra di loro correlati: selezionado alcune tuple dalla tabella, verranno mostrate le corrispondenti barre sull'istogramma e viceversa.

Capitolo 3

Le scelte implementative: map-reduce e sharding

MongoDB (da *"humongous",* enorme*)* è stato concepito con l'intento di gestire in modo capillare una grande quantità di dati. Pertanto, per garantire delle buone prestazioni da parte dell'applicativo è stato necessario utilizzare delle tecniche di **big-data analytics: le map-reduce in un sistema distribuito su più nodi**. Le map-reduce sono state utilizzate per estrarre dalla base di dati tutte le informazioni necessarie per la ricostruzione di uno schema la cui struttura non è a priori conosciuta. Per migliorare le performance delle map-reduce, si è utilizzato lo sharding, un meccanismo che permette la costruzione di un cluster di nodi. In questo capitolo si descriverà come sono state utilizzate le map-reduce per

la profilazione dello schema e lo sharding per ottenere delle prestazioni di alto livello.

3.1. Le aggregazioni in MongoDB

MongoDB è dotato di un motore di aggregazione dei dati, denominato **aggregation pipeline framework,** che fornisce funzionalità simili all'istruzione GROUP BY del linguaggio SQL. Tale framework utilizza operatori nativi di MongoDB e viene impiegato per le operazioni di aggregazione più semplici. L'aggregation framework consente di suddividere in stages le operazioni da svolgere su un insieme di documenti. Ciascuno step produce un risultato finale che può essere dato in input ad un nuovo stage. Si consideri, per esempio, una collezione *students* che contiene dei documenti che descrivono gli studenti dell'università di Milano-Bicocca. Nella *figura 12* si mostrano due documenti di questa collezione.

```
{
    "_id" : ObjectId("574afff00bec1c4563cb3229"),
    "serial number" : NumberInt(764081),
    "name" : "Paolo",
    "surname" : "Giannone",
    "gender" : "male",
    "date of birth" : ISODate("1993-01-16T00:00:00.000Z"),
    "faculty" : "Mathematical, physical and natural sciences",
    "course of study" : "Computer Science"
}
{
    "_id" : ObjectId("574b011f0bec1c4563cb322a"),
    "serial number" : NumberInt(762209),
    "name" : "Giuseppa",
    "surname" : "Giubaldo",
    "gender" : "female",
    "date of birth" : ISODate("1992-06-26T00:00:00.000Z"),
    "faculty" : "Mathematical, physical and natural sciences",
    "course of study" : "Computer Science"
}
```

Figura 12. Documenti che descrivono due studenti dell'università degli studi di Milano-Bicocca

Utilizziamo l'aggregation pipeline per individuare il numero di ragazze e di ragazzi iscritti al corso di informatica:

```
db.sudents.aggregate([
                {$match:{"course of study" : "Computer Science" } },
                {$group:{"_id" : "$gender" }, "total" : { $sum : 1 } }
                      ])
```

Figura 13. **Uso dell'aggregation pipeline per estrarre dalla collezione "student" il numero di ragazzi e di ragazze iscritti al corso di informatica**

La precedente aggregazione è suddivisa in due step:

1. **$match**: tra tutti i documenti presenti nella collezione *students*, vengono selezionati solo quelli aventi la stringa *Computer Science* come valore per il campo *course of study*;
2. **$group**: i documenti ritornati dallo step precedente vengono raggruppati in base al campo *gender*; poi si definisce una variabile *total* che effettua la somma dei documenti che appartengono a ciascun gruppo, calcolando il numero di ragazzi e di ragazze.

Nella versione più recente di MongoDB, attualmente la 3.2.7, **non è presente** un operatore nativo **per analizzare la conformazione di uno schema** senza conoscerne a priori la struttura. Comunque, nella pagina web che mongoDB mette a disposizione degli utenti per lasciare feedback e proposte di aggiornamento[10], sono stati individuati molti ticket ancora aperti in cui si richiede la possibilità di utilizzare l'aggregation pipeline framework per esaminare la forma dello schema di una collezione[11][12][13][14]; MongoDB ha accolto positivamente tale richiesta e sta lavorando per produrre un operatore nativo in grado di assolvere a tale compito[15]. È probabile che in futuro si possa utilizzare direttamente il framework nativo per l'analisi della struttura dello schema, ma al momento è

[10] https://jira.mongodb.org/secure/Dashboard.jspa
[11] https://jira.mongodb.org/browse/SERVER-18210
[12] https://jira.mongodb.org/browse/SERVER-23310
[13] https://jira.mongodb.org/browse/SERVER-18794
[14] https://jira.mongodb.org/browse/SERVER-5947
[15] https://groups.google.com/forum/#!topic/mongodb-user/xy9ikK7lK1Q

necessario utilizzare le map-reduce, descritte nel prossimo paragrafo.

3.2. Le map-reduce

Le map-reduce sono un paradigma di processo dei dati che utilizza il **javascript** per realizzare aggregazioni complesse che non è possibile effettuare con l'aggregation pipeline descritto nel paragrafo precedente [9]. Eseguire la scansione dello schema di una collezione, senza sapere a priori l'insieme di attributi utilizzato nel dataset, rende necessario l'uso delle map-reduce. Così come il motore di aggregazione nativo, anche le map-reduce prevedono più stages: il primo prende il nome di **map**, il secondo di **reduce** ed il terzo, che è opzionale, è chiamato *finalize*.

La funzione **map** viene eseguita su ogni documento di una collezione ed ha la struttura presentata in *figura 14*.

```
function() {
    . . .
    . . .
    emit(key, value);
}
```

Figura 14. Struttura della funzione map

Attraverso l'invocazione del metodo **emit(key, value)**, la map definisce su quale chiave si stanno raggruppando i documenti e condensa tutti i valori associati ad una certa chiave in un array *value* [10]. La funzione map produce delle multimappe, dato che ad una chiave è associata una lista di valori.

La funzione **reduce** prende in input il risultato ritornato dalla map, cioè una chiave *key* ed una lista di valori *values*, come mostrato in *figura 15*.

```
function(key, values) {
    . . .
    . . .
    return result;
}
```

Figura 15. Struttura della funzione reduce

La reduce viene automaticamente eseguita per ogni multimappa prodotta dalla funzione map che ha la lunghezza dell'array *values*

maggiore di uno. Il compito della reduce è quello di svolgere delle operazioni sull'array di valori *"values"*, al fine di costruire delle applicazioni biunivoche <*key, value*>, dove ad una chiave univocamente identificata sia associato un unico valore.

Bisogna sottolineare che la funzione reduce non viene eseguita atomicamente in un unico step, ma viene richiamata più volte dal DBMS per renderla più performante e per garantirne il funzionamento anche quando viene lanciata su un sistema distribuito su più nodi. Pertanto, dato che la reduce viene invocata più volte, è fondamentale che sia una **funzione idempotente**: il risultato che si ottiene suddividendo il lavoro svolto dalla reduce in più step deve essere lo stesso che si otterrebbe se fosse eseguita atomicamente in un unico stage [11].

Adesso, si mostrerà al lettore il funzionamento di una map-reduce esaminando lo stesso esempio fatto nel paragrafo 3.1 con l'aggregation pipeline; si cercherà, pertanto, di estrarre con una map-reduce il numero di studenti maschi e femmine frequentanti il corso di informatica dalla collezione *students:*

```
> db.sudents.mapReduce(

    MAP ⟶ function()            { emit(this.gender,1); },
 REDUCE ⟶ function(key,values) { return Array.sum(values) },
  QUERY ⟶ { query: { "course of study" : "Computer Science" } })
```

Figura 16. Uso di una map-reduce per estrarre dalla collezione "student" il numero di ragazzi e di ragazze iscritti al corso di informatica

Per prima cosa la map-reduce esegue la query per individuare tutti gli studenti iscritti al corso di informatica; successivamente la funzione *map* costruisce un vettore di 1 separati da virgole ed associato al genere di ciascuno studente; la funzione reduce somma gli 1 presenti nella lista *values* per ciascuna chiave, producendo come risultato finale il numero di ragazze e di ragazzi.

La *figura 17* dà al lettore una visualizzazione grafica del lavoro svolto dalla map-reduce che è stata fornita come esempio.

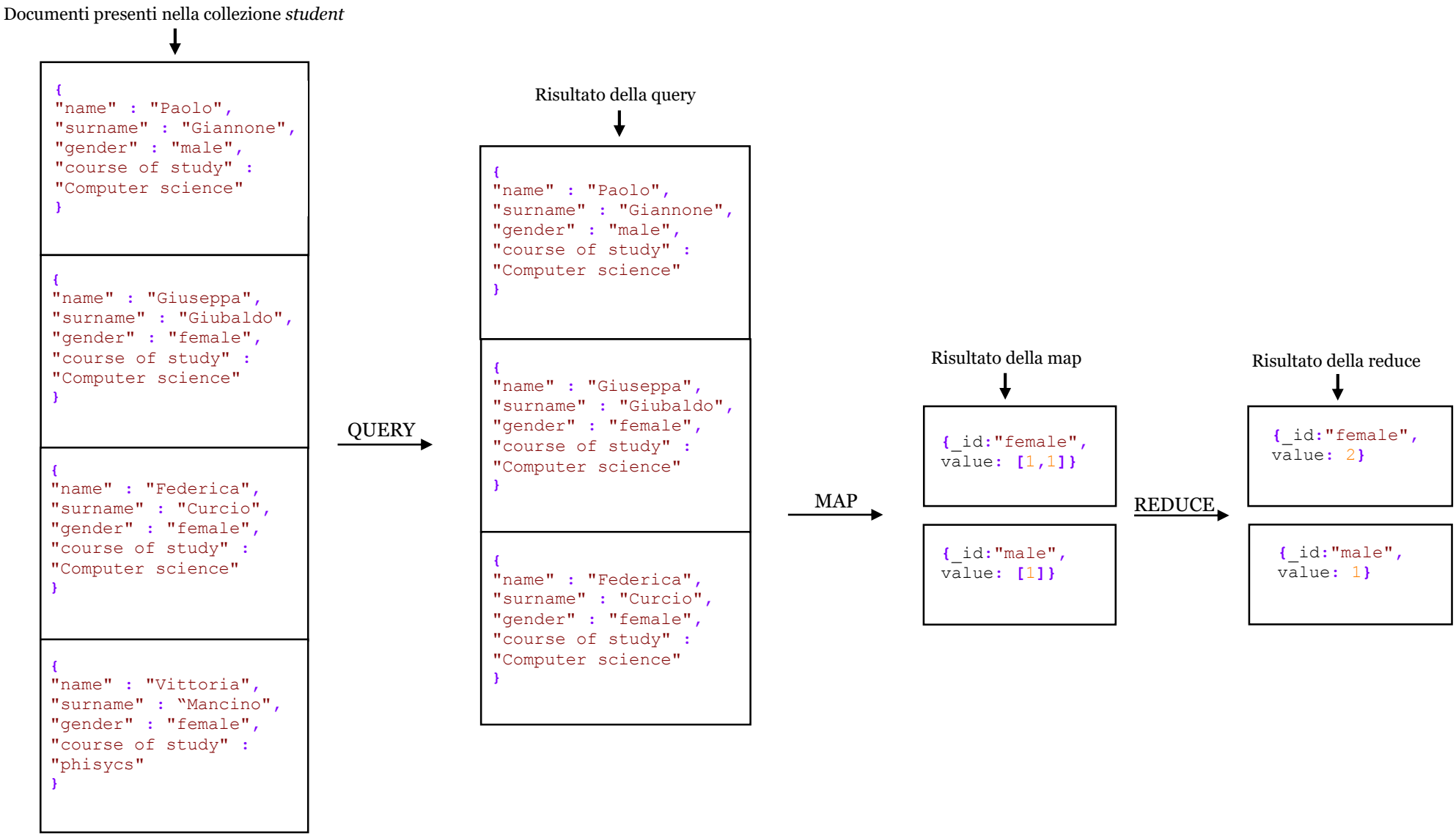

Figura 17. Visualizzazione grafica della map-reduce per l'estrazione dalla collezione *student* il numero di ragazzi e di ragazze iscritti al corso di informatica

3.3. Implementazione della map-reduce per il riconoscimento dello schema

Questo paragrafo fornirà una spiegazione del modo in cui sono state utilizzate le map-reduce per la profilazione dello schema di una collezione.

La map-reduce implementata permette di analizzare, **a qualsiasi livello di profondità**, gli attributi di ogni documento. La funzione **map** serve per mappare il nome di ciascun campo con un array di oggetti che contengono il tipo dell'attributo ed un contatore, per esempio:

```
key: "age", [ { value: {  type : "Int32", count : 1 } },
              { value: {  type : "Int32", count : 1 } },
              { value: {  type : "Int32", count : 1 } }
            ]
```

Nell'immagine precedente *key* identifica univocamente l'attributo *age* nella collezione; ogni volta che la funzione map trova tale campo associa al suo nome il corrispondente tipo ed un contatore, che servirà alla reduce per il calcolo del numero di occorrenze.

La reduce conterà il numero di volte che un attributo è presente nella collezione e condenserà l'array in un unico oggetto che conterrà il tipo (o i tipi) e le occorrenze del campo:

```
key: "age", value: { type : "Int32", count : 3 }
```

3.3.1. La funzione Map

L'implementazione della funzione map ha richiesto l'uso di tre funzioni di supporto:

- **isArray(x)**, che prende in input un attributo e ritorna true se è un array e false altrimenti [12]:

```
isArray = function (x)    {
            return x && typeof x === 'object'
                     && typeof x.length === 'number'
                     && !(x.propertyIsEnumerable('length'))};
}
```

Figura 18. Codice della funzione isArray(x)

- **bsonType(x)**, che prende in input un attributo e ritorna il suo data type:

```
bsonType = function (x)  {

  if(Object.prototype.toString.call(x) === '[object BSON]')
      return "Object";

  else if(Object.prototype.toString.call(x) === '[object Number]')
      return "Number";

  else return Object.prototype.toString.call(x).slice(8, -1)}
```

Figura 19. Codice della funzione bsonType(x)

- **map_rec(base,value)**, è una funzione ricorsiva utilizzata quando un attributo è di tipo complesso (array o object) e permette l'analisi a qualsiasi livello di profondità:

```
map_rec = function(base, value) {
            for(var key in value) {
               emit(base + "." + key,
                       {"type":bsontype(this[key]), "count":1});
               if(isArray(value[key])
                            || typeof value[key] == 'object' )
                     map_rec(base + "." + key, value[key])}
            }
}
```

Figura 20. Codice della funzione map_rec(base, value)

Il ciclo for della funzione *map_rec* permette di iterare ogni attributo presente in un documento e di associare il nome di ciascun campo ad un oggetto contenente il suo tipo ed il contatore *count*, attraverso la chiamata del metodo *emit*; se l'attributo è un **array** oppure un **object**, la funzione viene richiamata ricorsivamente

prendendo in input il nome del campo complesso (formattato nella forma x.y) ed il contenuto dell'object o dell'array.

Dopo aver definito le precedenti funzioni di supporto, si riporta in *figura 21* l'implementazione della map [12].

```
map = function(base, value) {

        for(var key in this) {

            emit(key,
                    {"type":bsontype(this[key]), "count":1});

            if(isArray(this[key])
               || typeof this[key] == 'object' )

                    map_rec(key, this[key])}

             }
}
```

Figura 21. Codice della funzione map

La funzione *map* viene eseguita su ogni documento della collezione; il ciclo for permette di iterare tutti gli attributi presenti in un singolo documento e con la chiamata del metodo *emit*, si costruiscono delle coppie *<field name, {type,count}>*. Se un attributo è di tipo complesso, viene chiamata funzione *map_rec* che si occupa di gestire la ricorsione.

Di seguito si offre al lettore una visualizzazione grafica del funzionamento della map sul documento riportato in *figura 22*.

```
{
    "_id" : ObjectId("574c6b546d2a7d265114fd76"),
    "name" : "Paolo",
    "surname" : "Giannone",
    "address" : {
        "street" : "Giovanni Spadolini",
        "number" : 19,
        "city" : "Lecce"
    }
}
```

Figura 22. Documento che descrive le generalità di una persona

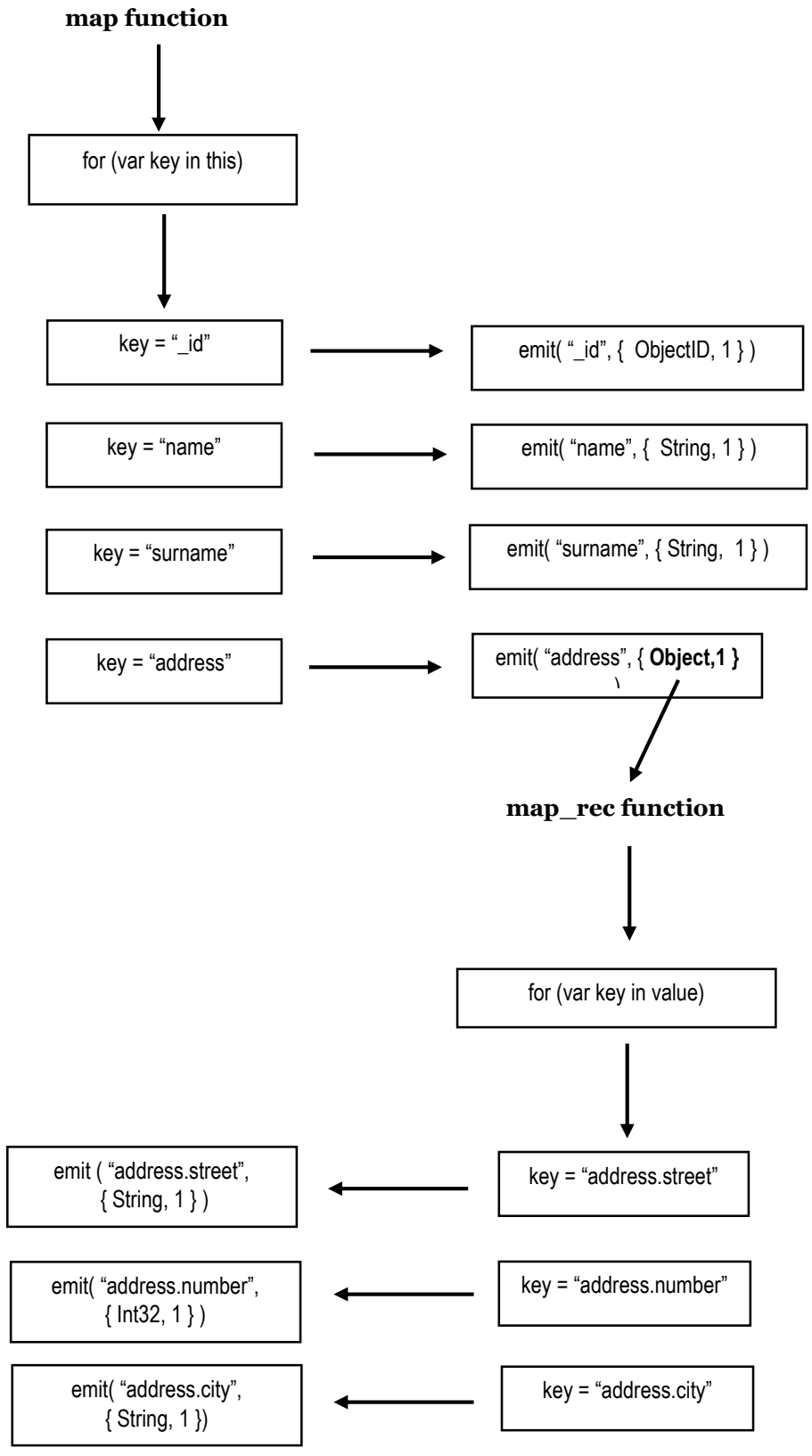

Figura 23. Funzionamento della funzione map sul documento riportato in figura 18

Dall'immagine in *figura 23*, si evince la natura ricorsiva dell'algoritmo realizzato: la funzione *map* itera tutti gli attributi di un documento e, se il tipo di tale campo è un object o un array, chiama la funzione *map_rec* che implementa la ricorsione.

3.3.2. La funzione reduce

La funzione reduce prende in input la chiave e la lista degli oggetti ritornata dalla funzione map del paragrafo precedente; la reduce condensa il vettore in un unico oggetto che descrive il tipo di un attributo ed il suo numero di occorrenze nella collezione selezionata:

```
reduce = function(key, stuff)  {
        reduceVal = { type: "", count: 0 };
        var array = [];
        var sum = 0;
        for (var idx =0; idx < stuff.length; idx++) {
            sum += stuff[idx].count;
            if(array.indexOf(stuff[idx].type) == -1)
                array.push(stuff[idx].type);}
        }
        reduceVal.count=sum;
        var x = array.toString();
        x = Array.from(new Set(x.split(','))).toString();
        reduceVal.type = x;
        return reduceVal;
}
```

Figura 24. Codice della funzione reduce

La funzione reduce viene chiamata per ogni coppia *<key, [{type, count}]>* prodotta dalla map che abbia l'array associato alla key di lunghezza maggiore di 1. Essa si occuperà di effettuare la somma dei contatori definiti nella funzione map, calcolando, pertanto, le occorrenze di un certo attributo nella collezione; inoltre salverà in una stringa il tipo dell'attributo facendo attenzione ad eliminare eventuali duplicati.

3.3.3. I limiti delle map-reduce

Come più volte indicato nel corso della trattazione, le map-reduce conferiscono maggiore flessibilità rispetto all'aggregation pipeline nativo di MongoDB; d'altro canto l'estrazione dei dati risulta essere più lenta. Infatti, l'uso delle map-reduce richiede che ogni documento analizzato venga convertito dal BSON al JSON prima che possa essere utilizzato dal motore javascript. Per incrementare la potenza computazionale delle map-reduce è possibile costruire un'applicazione distribuita su più macchine; in questo modo ciascun server esegue in parallelo la map e la reduce solamente su una porzione di dati ed una volta che tutti i nodi completano la computazione, le informazioni vengono ricompattate e rese visibili all'utente finale in completa trasparenza.

A tale scopo, nel prossimo paragrafo, si descriverà lo sharding, processo che consente a MongoDB di distribuire il proprio workload su più macchine con l'implementazione di un cluster di nodi.

3.4. Lo sharding

Memorizzare grandi quantità di informazioni su un singolo server potrebbe rendere l'estrazione dei dati poco efficiente, sovraccaricare la CPU della macchina rendendo quest'ultima temporaneamente non disponibile ed eccedere la capacità di archiviazione del disco [13]. Per far fronte a questi problemi MongoDB utilizza un meccanismo volto alla creazione di un database distribuito su più nodi, in cui ogni server contiene una porzione del dataset iniziale. Tale procedimento prende il nome di **sharding**. La *figura 27* mostra in che modo viene realizzato lo sharding su una collezione da 1TB [14].

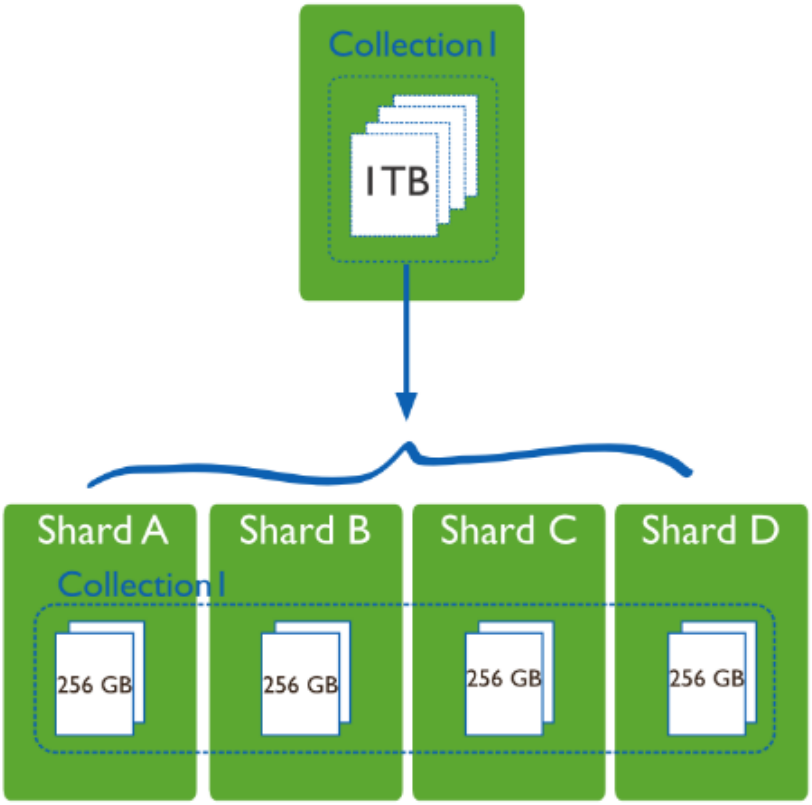

Figura 25. Sharding su 4 macchine di una collezione di 1TB

Al crescere delle dimensioni del database, le politiche di sharding consentono la suddivisione del dataset su più server in modo completamente automatico e trasparente mantenendo equilibrata la distribuzione dei dati tra i vari shards ed evitando che ci siano nodi più sovraccarichi di altri. In questo modo ciascuno shard dovrà processare una minore quantità di dati e sarà in grado di restituire il risultato di un'interrogazione in tempi più brevi rispetto ad un unico server.

Uno sharded custer è costituito da seguenti elementi:

- **Config Servers**: contengono i metadati necessari al corretto funzionamento del cluster. Sono processi mongod che devono essere configurati ed avviati prima di ogni altro elemento del cluster. I config servers sono di vitale importanza per il funzionamento del sistema distribuito, pertanto si deve garantire che al loro avvio il journaling sia abilitato, in modo tale che in caso di malfunzionamenti i metadati siano ripristinati in

uno stato consistente [15]. Prima della versione 3.2 di MongoDB, si era obbligati ad utilizzare esattamente 3 config servers per la configurazione di un cluster[16]. Con le ultime release, è possibile servirsi dei config servers come un *replica set*, con la possibilità di avviare fino a 50 config servers per un unico cluster e preservare l'integrità dei dati grazie alle politiche offerte dalla *replication*. Se il *replica set* dei config servers dovesse perdere il suo nodo primario e non riuscisse ad eleggere un nuovo nodo principale, i metadati risulterebbe accessibili solo in lettura e, pertanto, tutte le operazioni che richiedono una scrittura sui config servers, come la migrazione o la suddivisone dei chunks, non sarebbero eseguibili fino all'elezione di un nuovo nodo primario.

Se tutti i config servers diventano irraggiungibili, il cluster va fuori uso. I config servers non necessitano né di un elevato spazio di archiviazione né di particolare potenza computazionale, quindi possono essere lanciati su nodi su cui sono in esecuzione altri processi [16].

- **Query router (mongos instances)**: sono responsabili dell'instradamento delle operazioni di lettura e di scrittura verso gli shards su cui risiede la porzione di dati interessata. Le applicazioni, infatti, non comunicano mai direttamente con gli shards: i query router sono dei middleware che ricevono delle istruzioni da un mittente, le indirizzano ad uno o più shards che le processa e, dopo aver ricevuto una risposta, la inviano indietro al mittente. In questo modo, se si interrogasse un mongo database distribuito su più nodi, la query sarebbe rivolta solo ed esclusivamente agli shards che contengono le informazioni ricercate limitando il tempo di estrazione delle informazioni. [17]

[16] https://docs.mongodb.com/manual/core/sharded-cluster-config-servers/

- Shard: è un processo standalone *mongod* oppure un *replica set* e contiene una certa porzione del dataset. In un ambiente di produzione ogni shard dovrebbe essere un *replica set* in modo tale da preservare l'integrità dei dati da eventuali malfunzionamenti [15]. Ciascuno shard è suddiviso in segmenti adiacenti chiamati chunks, aventi una capacità di default pari a 64MB, ma modificabile dall'utente tra 1 e 1024 MB. I documenti di una collezione shardata vengono disposti nei vari chunks sulla base del valore assunto da una shard key, cioè un attributo o un insieme di attributi presente in ogni documento della collezione.

  La *figura 26* mostra l'architettura di uno sharded cluster. [17]

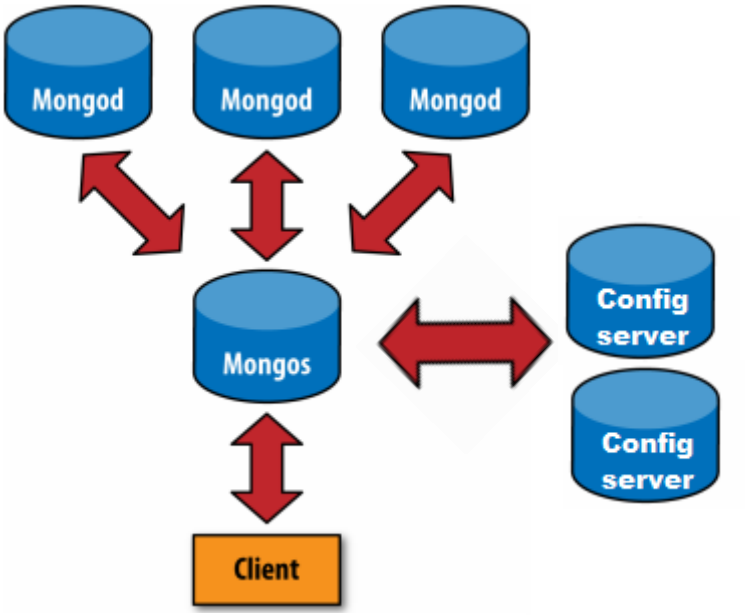

Figura 26. Architettura di uno sharded cluster

La scelta della shard key risulta essere particolarmente importante dato che le operazioni di scrittura e di lettura possono essere rese più o meno performanti. Per esempio, scegliendo con oculatezza la shard key, si potrebbe riuscire ad utilizzare un unico shard per le proprie operazioni di lettura invece che coinvolgerli tutti, limitando la complessità delle interrogazioni e riducendo le probabilità di indisponibilità di un dato [15]. Per distribuire i documenti tra i vari chunks, si possono utilizzare due tecniche differenti: *range based partioning* e *hashed based partitioning*.

La prima politica prevede di suddividere ogni shard in chunks individuati da intervalli **non sovrapposti** del tipo *[K-i, K+i]*, dove *K* è il valore assunto dalla shard key: tutti i documenti aventi come attributo della shard key un valore compreso all'interno di questo range verranno inseriti nel corrispondente chunk. Nel caso della *hashed based partitioning*, mongoDB applica una funzione di hash sul valore della shard key, determinando qual è il chunk in cui deve essere inserito un certo documento.

Se un chunk supera la sua massima capacità, MongoDB attua un meccanismo noto come *splitting*, che serve a dividere in due porzioni il singolo chunk. Nella *figura 27* [18], per esempio, un chunk ha superato la dimensione massima di 64 MB e viene splittato in due chunks di 32.1 MB ciascuno.

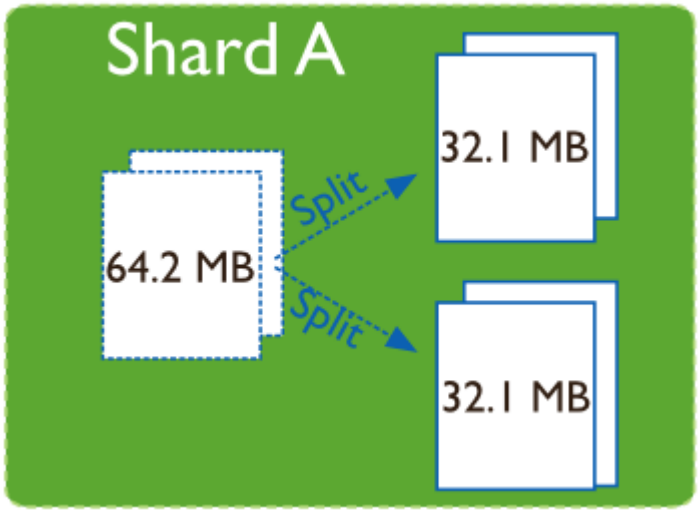

Figura 27. Splitting di un chunk che ha superato la massima capacità in due chunks più piccoli

Lo splitting può produrre una ripartizione non equa dei chunks tra i vari shards; se questo si verifica alcuni chunks possono essere fatti migrare da uno shard all'altro da un processo chiamato *balancer*, che ha il compito di tenere traccia del numero di chunks presenti su ciascuno shard e di provvedere alla migrazione nel caso in cui i chunks siano distribuiti in modo sproporzionato sui vari shards. [19]

3.5. Uso dello sharding nell'applicativo

Per rendere le map-reduce più performanti, si è implementato un piccolo cluster costituito da due nodi. Le specifiche tecniche delle macchine sono elencate di seguito:

- Computer 1:
  - Nome della macchina: DESKTOP-DNO2SN3
  - Produttore: ASUSTeK COMPUTER INC.
  - Modello: X550CC
  - Processore: Intel(R) Core(TM) i7-3537U CPU @ 2.00GHz (4 CPUs), ~2.5GHz
  - Memoria (RAM): 8192MB
  - Sistema Operativo: Windows 10 Home 64-bit
  - Memoria di massa: Kingston SSD V300 120GB (lettura fino a 450MB/s)

- Computer 2:
  - Nome della macchina: pc-Giuseppe
  - Produttore: ASUSTeK COMPUTER INC.
  - Modello: X550LD
  - Processore: Intel(R) Core(TM) i7-4510U CPU @ 2.00GHz (4 CPUs), ~2.6GHz
  - Memoria (RAM): 8192 MB
  - Sistema Operativo: Windows 10 Home 64-bit
  - Memoria di massa: Seagate ST1000LM024

Si è configurato un replica-set costituito da due config servers, uno per ciascuna macchina. Su ogni server sono stati lanciati cinque processi mongod come si può vedere nella *figura 28*.

```
{ "_id" : "shard0000", "host" : "DESKTOP-DNO2SN3:27059" }
{ "_id" : "shard0001", "host" : "DESKTOP-DNO2SN3:27060" }
{ "_id" : "shard0002", "host" : "DESKTOP-DNO2SN3:27061" }
{ "_id" : "shard0003", "host" : "DESKTOP-DNO2SN3:27062" }
{ "_id" : "shard0004", "host" : "DESKTOP-DNO2SN3:27063" }
{ "_id" : "shard0005", "host" : "pc-Giuseppe:27021" }
{ "_id" : "shard0006", "host" : "pc-Giuseppe:27022" }
{ "_id" : "shard0007", "host" : "pc-Giuseppe:27023" }
{ "_id" : "shard0008", "host" : "pc-Giuseppe:27024" }
{ "_id" : "shard0009", "host" : "pc-Giuseppe:27025" }
```

Figura 28. Distribuzione degli shards sulle due macchine

Dato che il dataset da analizzare non è conosciuto a priori, l'unico attributo sicuramente presente in tutti i documenti è il campo *_id*. Pertanto si è deciso di utilizzare l'*_id* come *hashed shard key* per distribuire i documenti nei vari chunks.

Inoltre si è modificata la dimensione massima dei chunks portandola da 64MB a 16MB; questa scelta si è resa necessaria perché si sono analizzati dataset di dimensione compresa tra i 20 MB e i 6 GB, pertanto utilizzare dei chunks troppo grandi non avrebbe portato molti benefici.

All'inserimento dei documenti in una collezione abilitata allo sharding, il *balancer* si è occupato autonomamente ed in massima trasparenza di inserire i documenti nei vari chunks in modo da rendere equa la loro distribuzione tra gli shards rappresentati in *figura 28*.

Nel capitolo 5, in cui viene descritto il testing, vengono analizzate le performance dell'applicativo su dataset di dimensioni variabili quando l'applicazione viene lanciata su un unico nodo non shardato, su un singolo nodo localmente shardato e sul cluster distribuito su due nodi.

Capitolo 4

Implementazione dell'applicativo

In questo capitolo si analizzerà in che modo è stato implementato il software per la profilazione. Nel particolare si mostrerà al lettore l'interfaccia utente e le varie funzionalità di cui è dotato l'applicativo facendo riferimento a tutte le API che sono state utilizzate per la realizzazione del tool.

4.1. Collegamento dell'applicativo ad un server MongoDB in esecuzione

L'applicativo stabilisce una connessione con un'istanza di MongoDB in esecuzione, inserendo l'indirizzo IP e la porta a cui collegarsi. Per realizzare il collegamento tra l'applicazione e MongoDB si è utilizzata l'API [**MongoDB Java Driver 2.14.1**][17], che fornisce la classe *MongoClient* dotata di appositi metodi per simulare il collegamento ad un client MongoDB.

Per favorire l'usabilità del tool è stato implementato un pannello *"Connection Manager"* che consente all'utente di gestire le proprie connessioni:

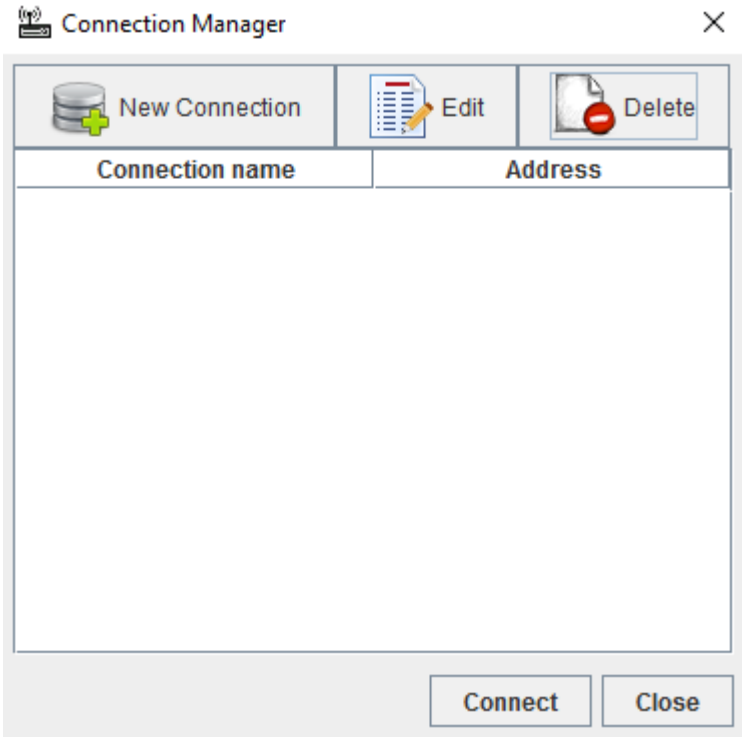

Figura 29. Pannello Connection Manager per la gestione delle connessioni

A partire dalla finestra precedente, possono essere create nuove connessioni cliccando sul bottone *"New Connection"*.

[17] http://api.mongodb.com/java/2.14/

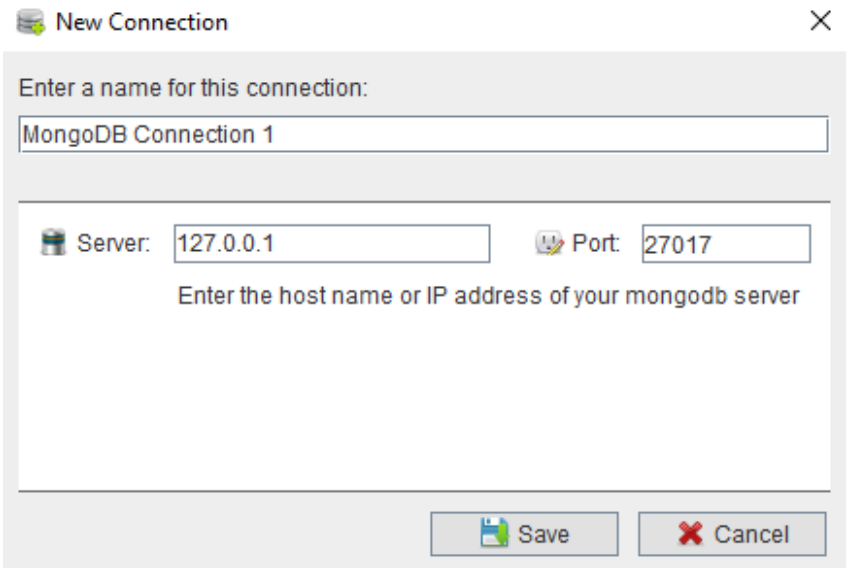

Figura 30. Finestra per la creazione di nuove connessioni a MongoDB

Una volta che l'utente inserisce correttamente i dati, essi vengono memorizzati in un file di configurazione *congif.ini* che ha la seguente struttura:

```
[Settings]
Connections_count = 1

[Connection1]
Connection_name = MongoDB Connection 1
Server_name = 127.0.0.1
Port_number = 27017
LimitDocuments = 0
MaxDepth = 0
```

Le informazioni di configurazione vengono estratte dal file *config.ini* per popolare la tabella rappresentata in *figura 31*.

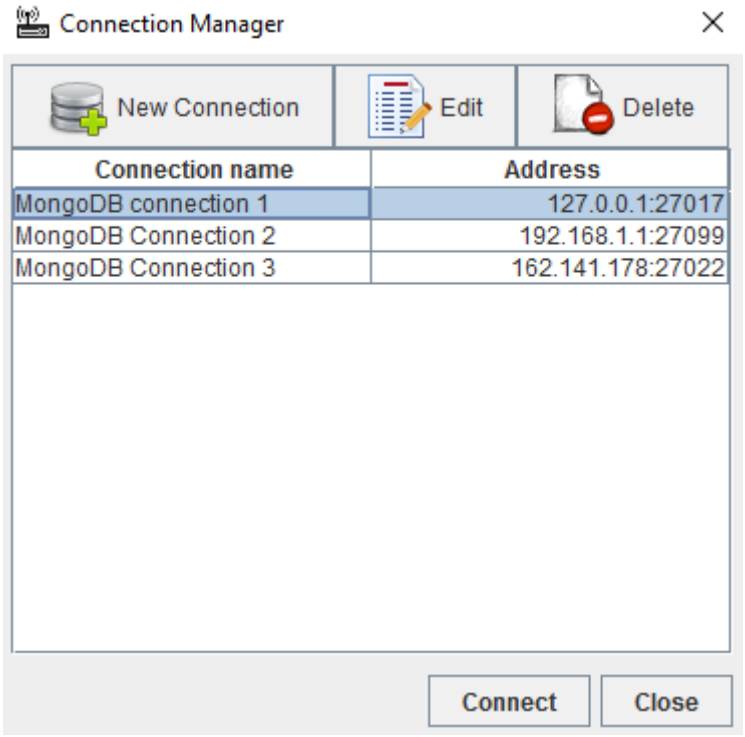

Figura 31. Connection Manager con le connessioni inserite dall'utente

Selezionando una connessione tra quelle esistenti nella tabella di configurazione in *figura 31* e premendo il tasto *"Connect"*, l'applicativo cerca di collegarsi al server MongoDB, individuato dalla stringa *Address.* L'utente può anche editare o cancellare una connessione della tabella di configurazione utilizzando rispettivamente i bottoni *"Edit"* e *"Delete"*. Per l'implementazione del file *config.ini*, è stata utilizzata l'API **[ini4j][18]**, una libreria che consente la gestione di file con estensione *.ini* in modo semplice ed elementare.

4.2. Renderizzazione dei database e delle rispettive collezioni

Per consentire all'utente di avere una visione generale dei database e delle collezioni presenti sul server, si è implementato un **JTree[19],** i cui nodi sono i database e le foglie le collezioni. In questo paragrafo si spiegherà in che modo è stato implementato il

[18] http://ini4j.sourceforge.net/
[19] https://docs.oracle.com/javase/8/docs/api/javax/swing/JTree.html

JTree utilizzando le API **[Swing]**[20] e **[AWT]**[21] di java. L'API [**MongoDB Java Driver 2.14.1]** fornisce il metodo *getDatabasesNames* per mezzo del quale si è estratto dal server MongoDB il nome di tutti i database presenti e, per ciascuno di essi, si è istanziato un oggetto *DB* passandogli il nome come parametro. Per ogni database si è utilizzato il metodo *getCollectionNames*, attraverso il quale sono state indentificate e poi istanziate le collezioni appartenenti a ciascun database. Dopo aver individuato per ogni database le rispettive collezioni, è stata creata una multimappa avente per chiave un database e per valore la lista delle sue collezioni. Utilizzando la multimappa si è implementato il JTree della *figura 32*. Nell'immagine sono presenti quattro nodi, a ciascuno dei quali corrisponde un database; questi nodi possono essere dilatati mostrando le collezioni in essi contenute.

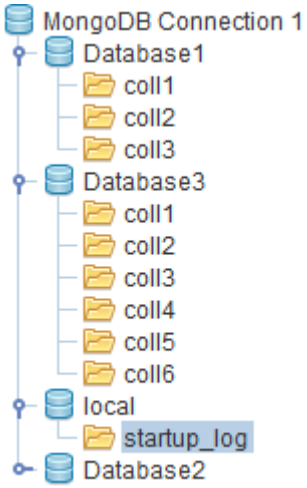

Figura 32. JTree dei database e delle rispettive collezioni

4.3. Renderizzazione dei risultati prodotti dalla map-reduce

Nella sezione 3.3 di questa trattazione si è esaminata la map-reduce per l'estrazione del nome, del tipo, della frequenza relativa

[20] https://docs.oracle.com/javase/8/docs/api/javax/swing/package-summary.html
[21] https://docs.oracle.com/javase/8/docs/api/java/awt/package-summary.html

ed assoluta degli attributi presenti in una collezione. In questo paragrafo si intende spiegare al lettore in che modo sono stati renderizzati i dati ottenuti dalla map-reduce.

In modo particolare, si descriverà l'implementazione di un **JXTreeTable**[22], costruito sfruttando l'API **[SwingX]**[23].

Quando si clicca con il tasto destro su una foglia del JTree in *figura 32*, l'applicativo lancia l'esecuzione della map-reduce per la profilazione dello schema sulla collezione che si è selezionata. Al termine dell'esecuzione della map-reduce, le informazioni estratte sono salvate all'interno di un cursore che viene iterato con un ciclo for. Per ogni elemento del cursore si è creato un array di stringhe con le informazioni estratte dalla map-reduce.

```
String record = new String [] {
                                <field name>,
                                <field type>,
                                <occurrences numbers>,
                                <occurrences rate>
                               }
```

Poi, si è costruita una lista di records, che è stata utilizzata per l'implementazione del **JXTreeTable** in *figura 33*.

Nell'immagine è stata selezionata la collezione *restaurants*; l'applicativo, dunque, lancia la map-reduce su tale collezione e visualizza per ciascun attributo della collezione, il nome, il tipo, la frequenza assoluta e relativa. Il *JTreeTable* permette all'utente di analizzare dettagliatamente tutti i campi presenti nella collezione esaminata; nel caso di attributi di tipo complesso si dà la possibilità di esplorare il contenuto in profondità: per esempio, il nodo "*address*" è un Object e pertanto può essere dilatato, rendendo visibile la struttura interna dell'oggetto; gli attributi di tipo primitivo vengono mostrati come foglie all'interno del *JTreeTable*.

[22] http://javadoc.geotoolkit.org/external/swingx/org/jdesktop/swingx/JXTreeTable.html
[23] https://swingx.java.net/

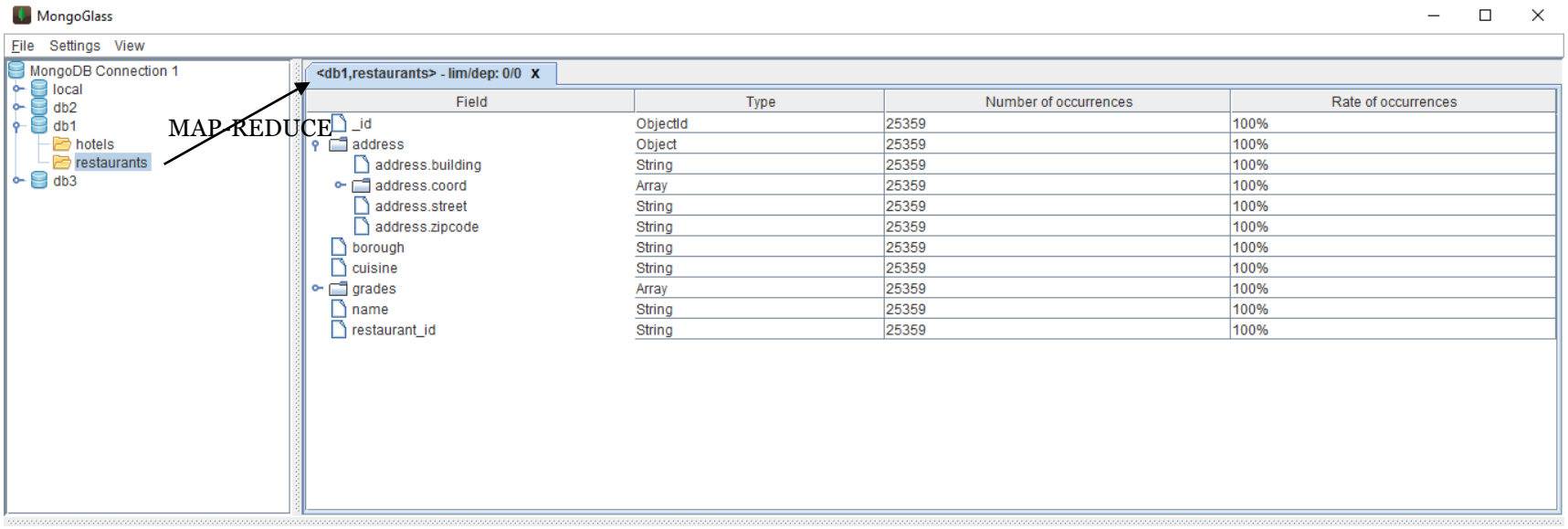

Figura 33. Renderizzazione dei risultati della map-reduce per mezzo di un JXTreeTable

4.4. Istogramma per l'analisi della distribuzione dei valori di un attributo

In questo paragrafo si esaminerà l'implementazione di un istogramma che dia all'utente una visualizzazione grafica della distribuzione dei valori associati ad un attributo. Per l'implementazione dell'istogramma è stata utilizzata l'API **[JFreeChart]**[24] che offre una grande quantità di strumenti per la realizzazione e la personalizzazione di grafici.

Nella sezione precedente si è mostrata l'implementazione del *JTreeTable*; Quando l'utente seleziona un record del *JXTreeTable* rappresentato in *figura 33*, l'applicativo lancia una query sul database che consente l'estrazione di tutti i documenti in cui è presente l'attributo a cui il record è associato.

È bene precisare che, in base al tipo del campo selezionato, è stato necessario implementare l'interrogazione in modo differente, pertanto si suddividerà la trattazione in modo tale da esaminare nel dettaglio tutte le possibilità.

4.4.1. Attributo di tipo primitivo con un unico data-type

Quando si seleziona dal *treetable* in *figura 33* un attributo che compare nella base di dati con un unico data type primitivo, l'applicativo lancia sul database la seguente interrogazione:

```
db.collection.find(
        {<field name> : {"$type" : <field type> }},
        {_id: 0, <field name>: 1  })
```

Figura 34. Query per estrarre i valori di attributi aventi un unico data-type primitivo

La query estrae tutti i documenti in cui l'attributo <*field name*> è di tipo <*field type*>, eseguendo la proiezione sul campo stesso. Il risultato dell'interrogazione è salvato in un cursore che viene iterato con un ciclo while; attraverso un contatore, si calcolano le

[24] http://www.jfree.org/jfreechart/

volte che un attributo assume lo stesso valore e si costruisce l'istogramma della *figura 35*.

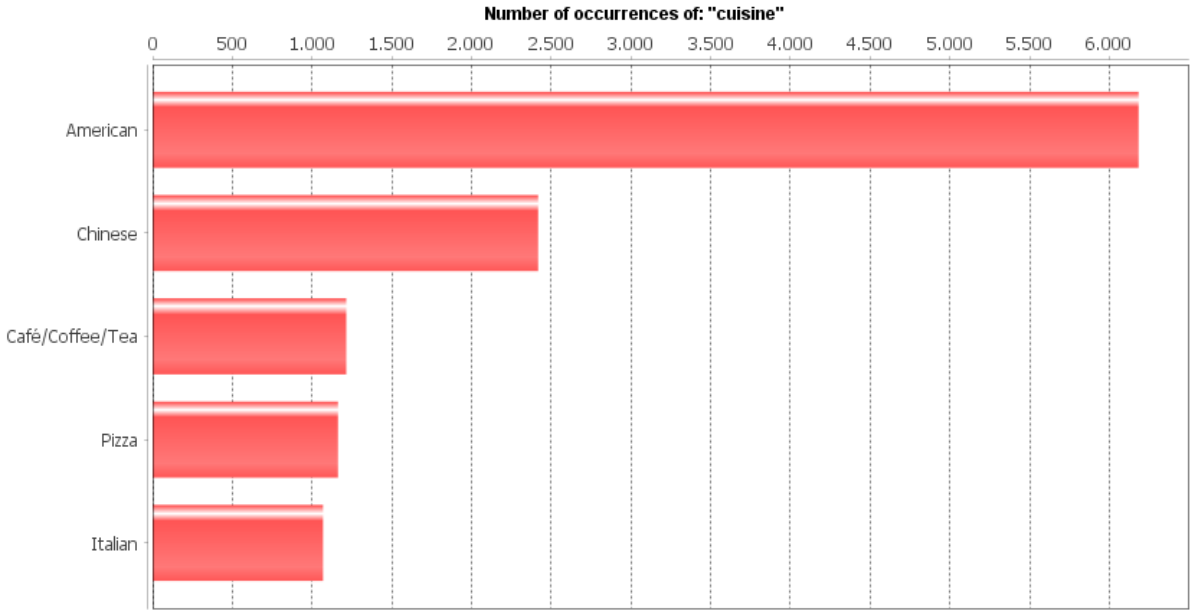

Figura 35. Istogramma per la rappresentazione della distribuzione dei valori di un attributo che compare nella base di dati con un unico data type primitivo

Considerando il campo *"cuisine"* di un dataset di esempio, sull'asse delle ordinate si mostreranno tutti i valori assunti dall'attributo e sulle ascisse il numero di occorrenze.

4.4.2. Attributo con data-types multipli di tipo primitivo

Nel caso in cui l'attributo selezionato dal treetable in *figura 33* sia di diversi tipi primitivi, si seguirà fondamentalmente lo stesso procedimento esaminato nella sezione precedente. L'unica differenza è che sarà necessario eseguire più interrogazioni, una per ogni data-type possibile:

```
db.collection.find(
        {<field name> : {"$type" : <field type 1> }},
        {_id:0, <field name> : 1 })

db.collection.find(
        {<field name> : {"$type" : <field type 2> }},
        {_id:0, <field name> : 1 })
```

Figura 36. Query da eseguire per estrarre i valori di attributi aventi più data-type primitivi

Ogni query ritornerà i documenti in cui il campo *<field name>* sarà del tipo *<field type>* preso in considerazione. Anche in

questo caso si utilizza un contatore per determinare il numero di volte che un attributo assume un certo valore; inoltre, l'applicativo procederà a colorare le colonne dell'istogramma in modo diverso in base al tipo del valore assunto dall'attributo, come si può vedere nell'immagine in *figura 37*.

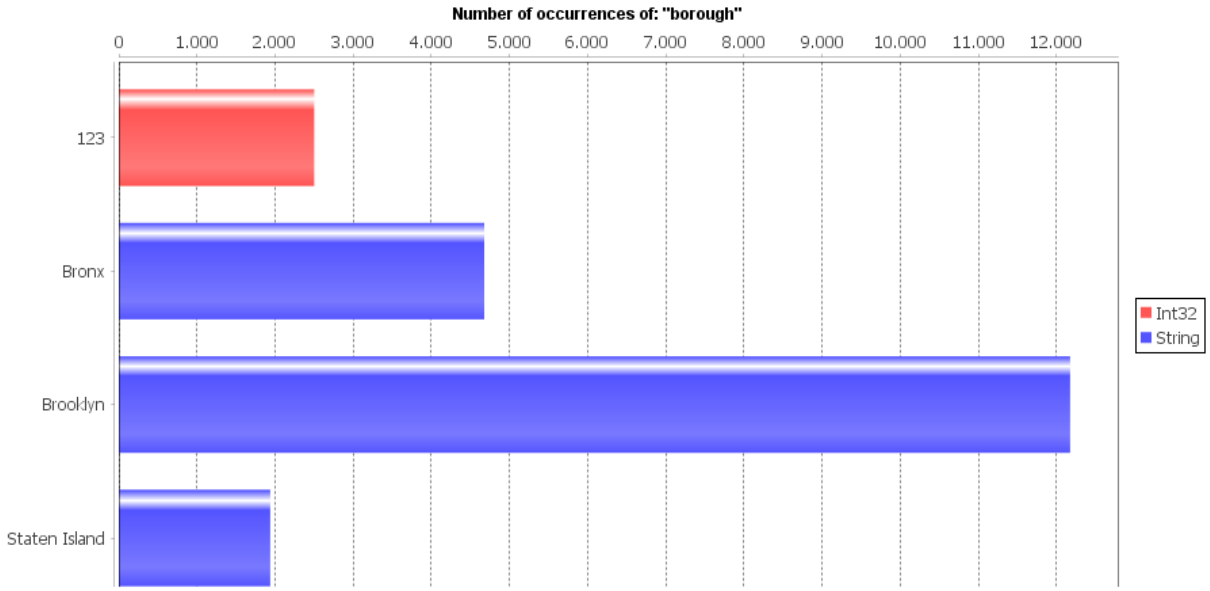

Figura 37. Istogramma per rappresentare la distribuzione dei valori di attributi di più data type primivi

Il campo *"borough"* compare in una base di dati di esempio sia come stringa sia come intero. Pertanto l'istogramma distinguerà i tipi colorando le barre in modo diverso ed inserendo una legenda sulla destra.

4.4.3. Attributo contenuto in un array

MongoDB gestisce gli array in modo molto particolare. Se un attributo è un array, le queries utilizzate nei paragrafi precedenti non funzionano correttamente ed è necessario implementare l'interrogazione in modo differente. Nel dettaglio, se si vogliono individuare i valori di un attributo che è contenuto in un array è necessario utilizzare le aggregazioni, suddividendo in più stages la query.

Il primo stage è una *$match*, attraverso la quale si prendono in considerazione solamente i documenti che contengono l'attri-

buto che è stato selezionato nella treetable della *figura 33*. Successivamente, viene eseguita una serie di *$project* che trasformano l'array in una semplice coppia <*chiave, valore*>. Anche l'aggregazione ritorna un cursore attraverso il quale si calcola il numero di volte che un attributo è presente nella collezione e si disegna l'istogramma come quello delle figure *36 e 37*.

4.5. Implementazione delle funzioni *limit* e *depth*

Un'altra funzionalità di cui è dotato l'applicativo è la possibilità di limitare i documenti analizzati dalla map-reduce:

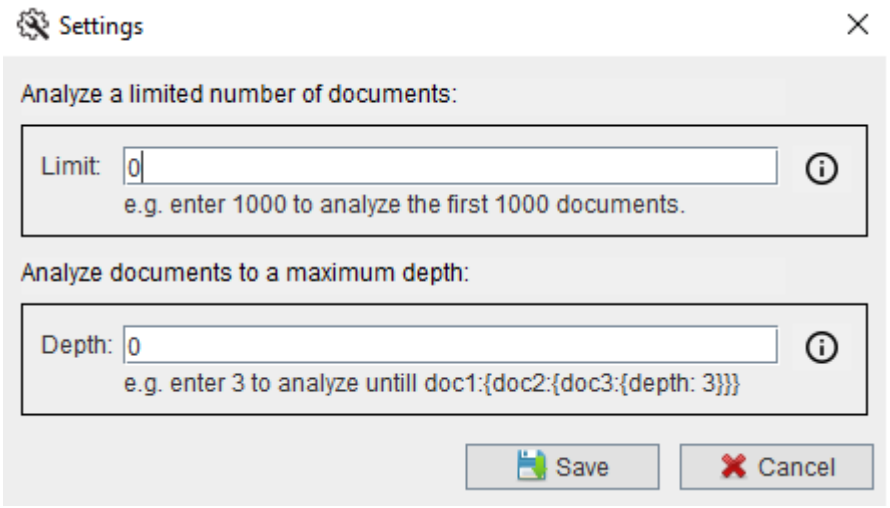

Figura 38. Pannello per settare le impostazioni

4.5.1. Limit

Quando si setta la proprietà *"limit"* pari ad un certo numero intero "N", si limita l'esecuzione della map-reduce ai primi "N" documenti della collezione considerata.

L'applicativo è in grado di riconoscere quando la proprietà è settata; in tal caso crea una copia della collezione originale contenente i primi "N" documenti e la usa come appoggio per svolgere il proprio lavoro. Si tenga presente che la collezione originale viene ordinata, in modo tale che i primi "N" documenti della collezione temporanea siano sempre gli stessi anche per più esecuzioni diverse. L'applicativo gestisce autonomamente la creazione

delle collezioni d'appoggio impedendo collisioni con altre collezioni già presenti nel database. Quando le collezioni temporanee non sono più necessarie, l'applicativo provvede ad eliminarle, gestendo anche le circostanze in cui l'applicazione non termina correttamente.

L'uso della proprietà *"limit"* può essere utile quando una collezione è particolarmente grande: limitando il numero di documenti da analizzare il tempo di esecuzione si abbassa, consentendo comunque una visione generale della struttura della collezione.

4.5.2. Depth

La proprietà depth permette di limitare l'esecuzione della map-reduce fino ad un certo livello di profondità. Secondo la documentazione, MongoDB accetta fino a 100 livelli di innestamento[25], pertanto nei casi in cui la proprietà non viene esplicitamente settata dall'utente, si supporrà che la depth sia pari a 100. L'implementazione della depth è stata eseguita direttamente sulla map-reduce presentata nel paragrafo 3.3: quando un attributo è di tipo complesso, si setta una variabile "count" che funge da contatore e che viene incrementata man mano che si scende in profondità. Se la variabile count supera il limite espresso dalla proprietà depth, la map-reduce viene interrotta. Nel caso di collezioni caratterizzate da un forte livello di innestamento, potrebbe essere utile usare la depth per interrompere la map-reduce ad una certa profondità, abbassandone i tempi di esecuzione.

[25] https://docs.mongodb.com/manual/reference/limits/

Capitolo 5

Testing dell'applicativo

In questo capitolo si proporranno una serie di test per mezzo dei quali si analizzeranno le performance dell'applicativo. Nello specifico, utilizzando dei dataset di esempio, si esamineranno le prestazioni del tool per la profilazione della base di dati quando viene lanciato su:

- singolo nodo non shardato;
- singolo nodo localmente shardato;
- sharded cluster distribuito su due nodi.

5.1. Generazione di dataset di esempio

Per ottenere dei dataset di esempio su cui eseguire i propri test, si è implementato un semplice programma in java, che prende in input un numero intero *"N"* e produce *"N"* documenti in formato JSON. Il programma è stato implementato in modo tale che non tutti i documenti definiscano lo stesso insieme di attributi: in base al risultato ottenuto simulando il lancio di un dado, alcuni campi vengono randomicamente inseriti oppure omessi.

I documenti generati dal tool cercano di descrivere gli utenti di un social-network; nella *figura 39* si mostra un documento di esempio prodotto dal tool:


```
{
    "name": "Giuseppa",
    "surname": "Giubaldo",
    "age": 23,
    "gender": "female",
    "email": "giubaldog@giuseppa.com",
    "phone": "+039 123-456-789",
    "address": {
      "street": "Via Figino",
      "number": 11,
      "city": "Marsala",
      "country": "Italy",
      "zipcode": 57407
    },
    "registered": "2015-11-15T09:08:00 -01:00",
    "tags": [
      "football",
      "computer science",
      "tv-series"
    ],
    "friends": [
      {
        "id": "575041a9847a6fcbe474cfe6",
        "name": "Mirko",
        "surname": "Mancino"
      },
      {
        "id": "575041a9c066bf25e6a18cdb",
        "name": "Vittoria",
        "surname": "Curcio"
      }
    ]
}
```


Figura 39. Schema generale dei documenti utilizzati per il testing dell'applicativo

Il tool è stato utilizzato per generare dei file JSON contenenti una quantità crescente di documenti in modo da valutare le prestazioni dell'applicativo con collezioni sempre più grandi. I file JSON creati, sono stati successivamente importati in un database MongoDB utilizzando il comando *--mongoimport.*

5.2. Analisi delle performance su singolo server

In questa sezione si esamineranno le prestazioni dell'applicativo nella profilazione della base di dati quando viene lanciato su una

singola macchina, senza utilizzare alcuna politica di sharding. Di seguito si riportano le specifiche della macchina utilizzata per svolgere l'esperimento:

- Produttore: ASUSTeK COMPUTER INC.
- Modello: X550CC
- Processore: Intel(R) Core(TM) i7-3537U CPU @ 2.00GHz (4 CPUs), ~2.5GHz
- Memoria (RAM): 8192MB
- Sistema Operativo: Windows 10 Home 64-bit
- Memoria di massa: Kingston SSD V300 120GB (lettura fino a 450MB/s)

La *tabella 1* mostra il tempo necessario all'estrazione delle informazioni da parte della map-reduce in relazione al numero di documenti da analizzare:

NUMERO DI DOCUMENTI NELLA COLLEZIONE	TEMPO DI ESTRAZIONE
1.000	0.819 secondi
2.000	1.568 secondi
5.000	3.707 secondi
10.000	7.294 secondi
20.000	14.956 secondi
50.000	38.181 secondi
100.000	1 minuto 12 secondi
150.000	1 minuto 42 secondi
300.000	3 minuti 37 secondi
500.000	6 minuti 5 secondi
750.000	8 minuti 49 secondi
1.000.000	11 minuti 39 secondi
1.500.000	17 minuti 20 secondi
2.000.000	21 minuti 45 secondi
3.000.000	35 minuti 15 secondi
4.000.000	46 minuti 36 secondi

Tabella 1. Tempo necessario per la profilazione di una collezione eseguendo l'applicativo su un singolo nodo senza utilizzare politiche di sharding.

I dati empirici ottenuti sono stati posti nel grafico a dispersione in *figura 40* per valutare la correlazione tra la dimensione di una collezione e il tempo necessario ad analizzarla:

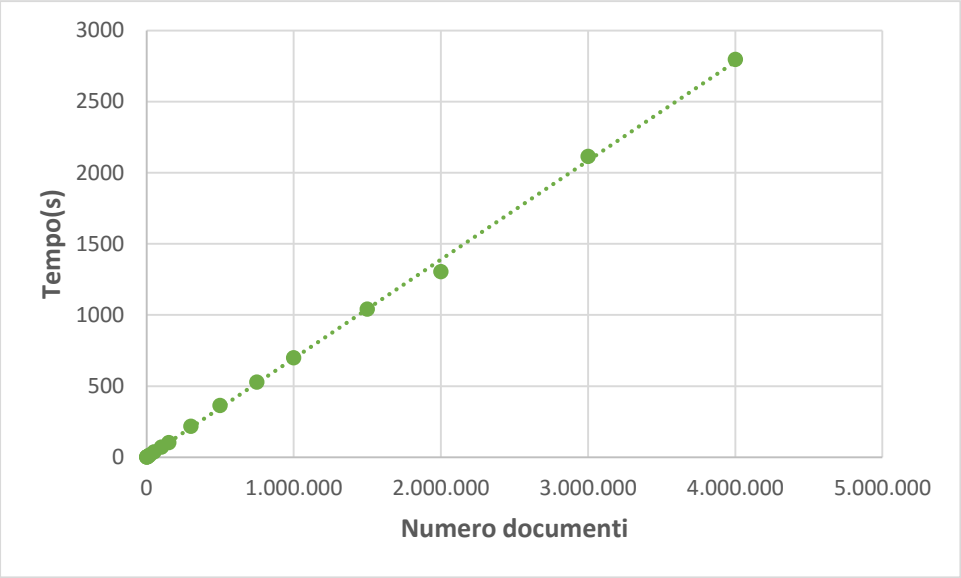

Figura 40. Il tempo necessario per la profilazione di una collezione in relazione al numero di documenti in essa contenuti eseguendo l'applicativo su un singolo nodo senza servirsi delle politiche di sharding

L'asse delle ascisse tiene traccia del numero di documenti presenti nella base di dati mentre quello delle ordinate misura il tempo per l'estrazione delle informazioni in secondi. Congiungendo i punti ottenuti con l'esperimento, si ottiene una retta passante per l'origine che evidenzia la proporzionalità diretta tra le due grandezze: quando il numero dei documenti da analizzare raddoppierà, anche il tempo necessario alla map-reduce per svolgere il proprio lavoro raddoppierà. La legge che lega il tempo $\boldsymbol{T}$ necessario per la profilazione ed il numero di documenti $\boldsymbol{\#D}$ da analizzare è data dalla seguente equazione:

$$\boldsymbol{T = \#D \cdot (6,9 \cdot 10^{-4})}$$

Pertanto, le performance offerte dall'applicativo eseguito su un singolo server, risultano essere particolarmente scarse man mano che una collezione aumenta di dimensione; si pensi ad una collezione costituita da 15 milioni di documenti:

$$T = 15.000.000 \cdot (\,6{,}9 \cdot 10^{-4}) = 6900s = 2\ ore\ 52\ minuti$$

Sarebbero necessare 2 ore e 52 minuti per la completa scansione della collezione!

5.3. Analisi delle performance applicando lo sharding su un singolo nodo

Dopo aver preso coscienza delle scarse performance dell'applicativo quando lanciato su un singolo nodo, si proverà ad utilizzare delle politiche di sharding in locale.

La macchina utilizzata è esattamente la medesima dell'esperimento del paragrafo precedente; per maggiore organizzazione si riportano nuovamente le specifiche del computer:

- Produttore: ASUSTeK COMPUTER INC.
- Modello: X550CC
- Processore: Intel(R) Core(TM) i7-3537U CPU @ 2.00GHz (4 CPUs), ~2.5GHz
- Memoria (RAM): 8192MB
- Sistema Operativo: Windows 10 Home 64-bit
- Memoria di massa: Kingston SSD V300 120GB (lettura fino a 450MB/s)

Il test è stato realizzato utilizzando 5 shards costituiti di chunks di 16 MB:

```
{  "_id" : "shard0000",  "host" : "DESKTOP-DNO2SN3:27033" }
{  "_id" : "shard0001",  "host" : "DESKTOP-DNO2SN3:27034" }
{  "_id" : "shard0002",  "host" : "DESKTOP-DNO2SN3:27035" }
{  "_id" : "shard0003",  "host" : "DESKTOP-DNO2SN3:27036" }
{  "_id" : "shard0004",  "host" : "DESKTOP-DNO2SN3:27037" }
```

Inizialmente il database risiedeva completamente sullo shard 0004 in ascolto sulla porta 27037; una volta abilitato lo sharding si è registrata la migrazione dei chunks dallo shard 0004 agli altri, al fine di equilibrare la distribuzione dei dati tra i vari shards. Completata la migrazione dei chunks, sono state valutate le performance dell'applicativo. La *tabella 2* mostra la distribuzione dei chunks nei vari shards ed il tempo necessario alla profilazione di una collezione in relazione alle sue dimensioni.

NUMERO DI DOCUMENTI	NUMERO DI CHUNKS PER SHARD					TEMPO DI ESTRAZIONE
	S0000	S0001	S0002	S0003	S0004	
50.000	1	1	1	0	1	21,065 secondi
100.000	2	1	1	1	2	44,297 secondi
150.000	2	2	2	2	2	57,130 secondi
300.000	4	4	4	4	4	2 minuti 3 secondi
500.000	7	7	6	6	7	3 minuti 17 secondi
750.000	10	10	10	9	10	5 minuti 10 secondi
1.000.000	13	13	13	13	13	7 minuti 4 secondo
1.500.000	20	20	19	19	20	10 minuti 10 secondi
2.000.000	26	26	26	26	26	13 minuti 40 secondi
3.000.000	39	39	39	39	39	19 minuti 10 secondi
4.000.000	52	52	52	52	52	28 minuti 21 secondi
5.000.000	65	65	65	65	65	35 minuti 2 secondi

Tabella 2. Tempo necessario per la profilazione di una collezione in relazione alle sue dimensioni, utilizzando localmente delle politiche di sharding

Nel grafico in *figura 41* si confrontano le prestazioni tra l'esecuzione dell'applicazione su un singolo nodo shardato e non shardato:

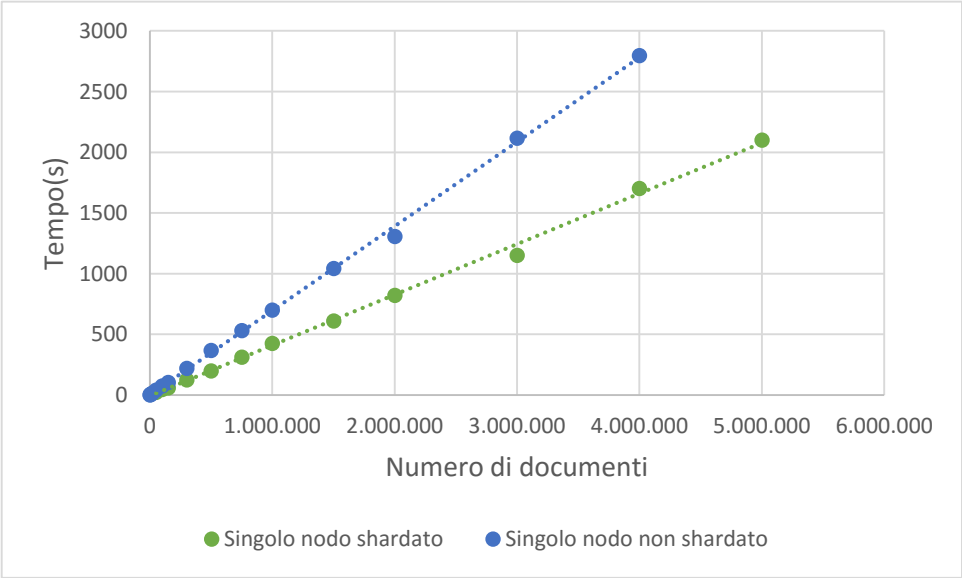

Figura 41. Il tempo necessario per la profilazione di una collezione in relazione al numero di documenti in essa contenuti eseguendo l'applicativo su un unico nodo localmente shardato o meno.

Osservando il grafico a dispersione in *figura 41*, che mette a confronto le prestazioni utilizzando o meno le politiche di sharding su un singolo nodo, si nota chiaramente un discreto miglioramento. Infatti, adoperare lo sharding su un singolo server permette di aumentare le performance della map-reduce descritta nel paragrafo 3.3; se non si utilizza lo sharding, il client mongoDB sarà dotato di un **unico motore javascript** che avrà il compito di gestire tutte le operazioni di acquisizione e rilascio dei lock, di eliminazione delle collezioni temporanee, di serializzazione dei documenti dal BSON al JSON e viceversa. Se si utilizza lo sharding, ciascuno shard sarà dotato del proprio motore javascript attraverso il quale gli shards potranno processare la map-reduce in parallelo su una porzione più piccola del dataset inziale, usufruendo, però, solamente della potenza computazionale offerta dal processore della singola macchina.
Anche in questo caso, il numero dei documenti da analizzare **#D** e il tempo necessario per la profilazione **T** sono legati da proporzionalità diretta come si può vedere dalla seguente equazione che esprime la legge fisica che intercorre tra le due grandezze:

$$\boldsymbol{T = \#D \cdot (3,78 \cdot 10^{-4})}$$

Chiaramente la costante di proporzionalità risulta essere minore rispetto all'esperimento realizzato nel paragrafo 5.2.

5.4. Analisi delle performance su uno sharded cluster di due nodi

In questa sezione si analizzeranno le prestazioni dell'applicativo quando viene lanciato su un cluster distribuito su due nodi, come già spiegato nel paragrafo 3.5 di questa trattazione.

L'immagine in *figura 42* dà una visualizzazione grafica dell'architettura del cluster costruito.

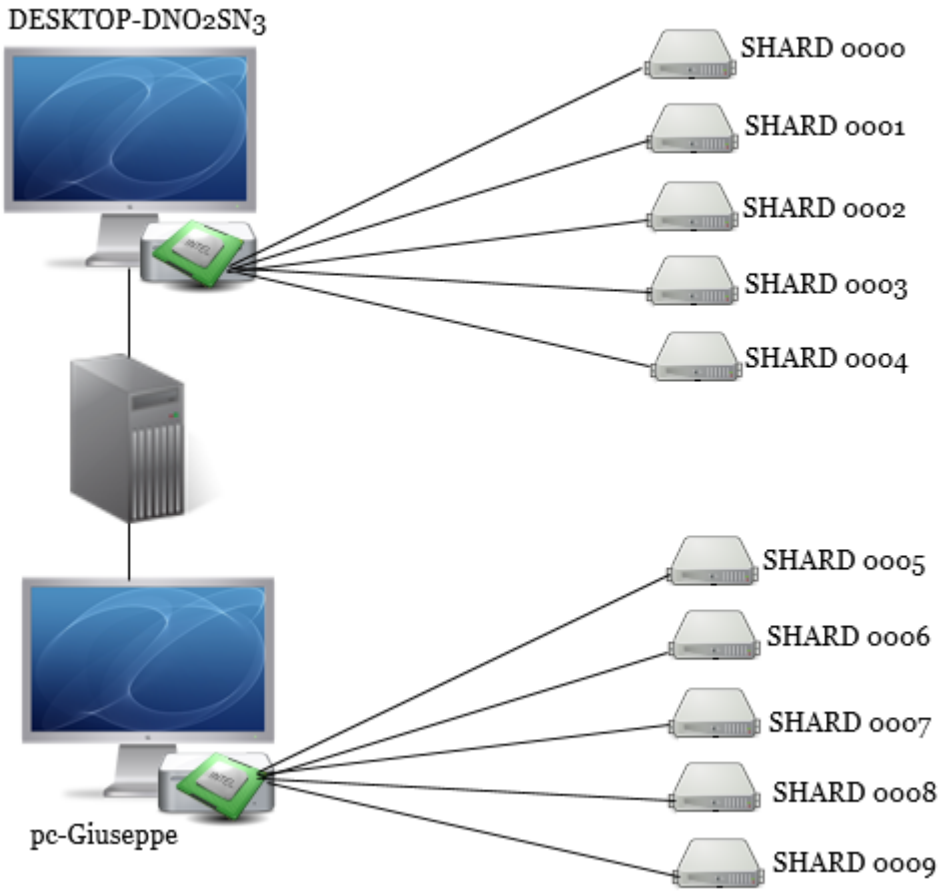

Figura 42. Architettura dello sharded cluster realizzato

Le caratteristiche tecniche delle due macchine sono elencate di seguito:

- Computer 1:
  - Nome della macchina: DESKTOP-DNO2SN3
  - Produttore: ASUSTeK COMPUTER INC.
  - Modello: X550CC
  - Processore: Intel(R) Core(TM) i7-3537U CPU @ 2.00GHz (4 CPUs), ~2.5GHz
  - Memoria (RAM): 8192MB
  - Sistema Operativo: Windows 10 Home 64-bit
  - Memoria di massa: Kingston SSD V300 120GB (lettura fino a 450MB/s)

- Computer 2:
  - Nome della macchina: pc-Giuseppe
  - Produttore: ASUSTeK COMPUTER INC.
  - Modello: X550LD
  - Processore: Intel(R) Core(TM) i7-4510U CPU @ 2.00GHz (4 CPUs), ~2.6GHz
  - Memoria (RAM): 8192 MB
  - Sistema Operativo: Windows 10 Home 64-bit
  - Memoria di massa: Seagate ST1000LM024

Come si evince dalla *figura 42*, sono stati utilizzati 5 shards per macchina; nella tabella 3 si presentano i risultati ottenuti testando l'applicativo sullo sharded cluster realizzato, evidenziando la distribuzione dei chunks tra i vari shards e il tempo necessario per la profilazione di una collezione in base al numero di documenti in essa contenuti.

NUMERO DI DOCUMENTI	NUMERO DI CHUNKS PER SHARD										TEMPO DI ESTRAZIONE
	S0000	S0001	S0002	S0003	S0004	S0005	S0006	S0007	S0008	S0009	
50.000	1	1	1	1	0	0	0	0	0	0	8 secondi
100.000	1	1	1	1	1	1	1	0	0	0	18 secondi
150.000	1	1	1	1	1	1	1	1	1	1	25 secondi
300.000	2	2	2	2	2	2	2	2	2	2	52 secondi
500.000	4	4	3	4	3	3	3	3	3	3	1 minuto 24 secondi
750.000	5	5	5	5	5	5	5	5	5	4	1 minuti 56 secondi
1.000.000	7	7	7	7	7	6	6	6	6	6	2 minuti 53 secondi
1.500.000	10	10	10	10	10	10	10	10	9	9	4 minuti 1 secondio
2.000.000	13	13	13	13	13	13	13	13	13	13	5 minuti 56 secondi
3.000.000	20	20	20	20	20	19	19	19	19	19	8 minuti 20 secondi
4.000.000	26	26	26	26	26	26	26	26	26	26	12 minuti 10 secondi
5.000.000	33	33	33	33	33	32	32	32	32	32	13 minuti 37 secondi

Tabella 3. Distribuzione dei chunks nei vari shards e tempo necessario per la profilazione di una collezione eseguendo l'applicativo su uno sharded cluster distribuito su due nodi

Nell'esperimento del paragrafo 5.3, ciascuno shard può attingere alla potenza computazionale garantita dalla CPU del singolo server su cui è in esecuzione l'applicativo. Al contrario, creando uno sharded cluster distribuito su due nodi, **l'applicativo utilizza due CPU per svolgere il proprio lavoro.** Per esempio, nella prova descritta in questa sezione, i primi cinque shards utilizzano la CPU della macchina *DESKTOP-DNO2SN3* e gli altri cinque quella di *pc-Giuseppe.* Il grafico a dispersione della *figura 43* mette in evidenza le performance dell'applicazione utilizzando o meno le politiche di sharding.

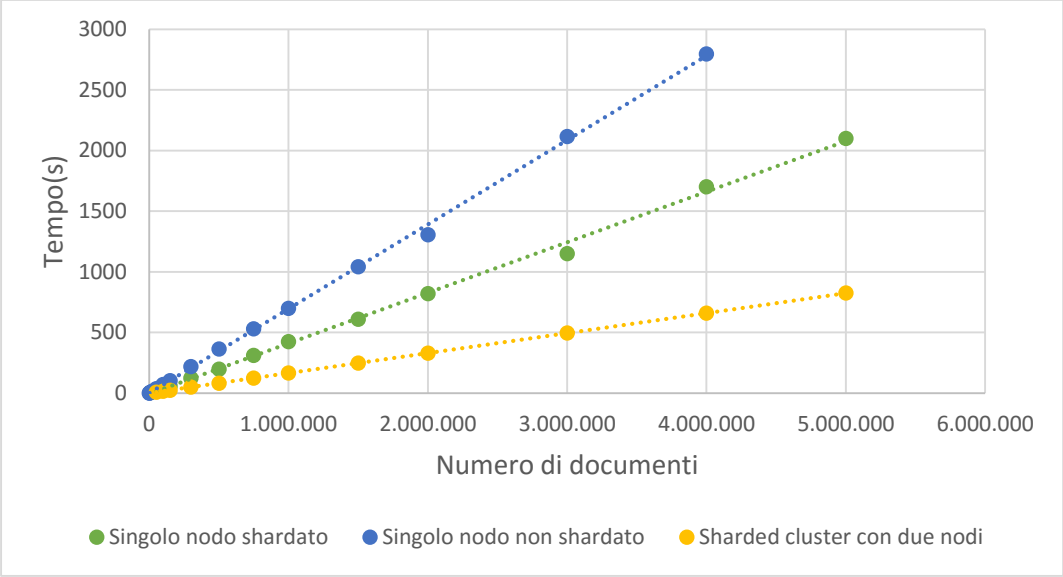

Figura 43. Tempo necessario per la profilazione di una collezione in relazione al numero di documenti in essa contenuti eseguendo l'applicativo su un unico nodo e su uno sharded cluster distribuito su due nodi.

Anche nel caso di uno sharded cluster distribuito su due nodi, il tempo **T** e il numero di documenti **#D** sono grandezze direttamente proporzionali. La costante di proporzionalità risulta, però, minore rispetto ai casi precedenti, come si evince dall'equazione presentata di seguito:

$$\boldsymbol{T = \#D \cdot (1{,}65 \cdot 10^{-4})}$$

Gli esperimenti effettuati nei paragrafi precedenti dimostrano che il tempo necessario alla profilazione di una collezione e il numero di documenti in essa contenuti sono direttamente proporzionali. Nei test realizzati sullo sharded cluster, però, la costante di proporzionalità è risultata quattro volte più piccola rispetto a quella ottenuta quando si è lanciato l'applicativo su un singolo nodo, senza utilizzare alcuna politica di sharding. L'esecuzione della map-reduce su un sistema distribuito permette di migliorare parecchio le prestazioni.

5.5. Errori individuati dall'applicazione

Per concludere il capitolo, si vogliono mostrare alcuni errori di modellazione e di inserimento che l'applicativo è in grado di riconoscere.

Si consideri per esempio l'istogramma della *figura 44* che descrive la distribuzione dei valori dell'attributo *nationality*.

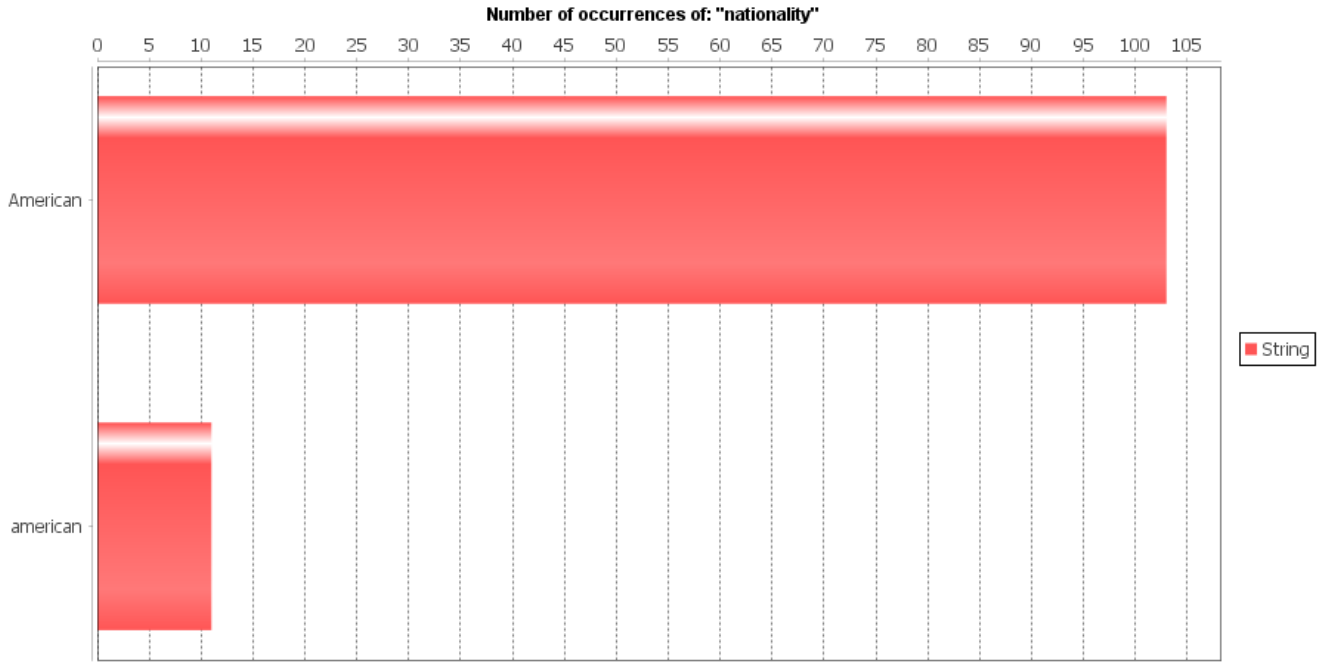

Figura 44. Ridondanza nei valori assunti dall'attributo *nationality*

L'istogramma mette in evidenza che l'attributo assume 103 volte il valore "American" e 11 il valore "american". Dato che ci si sta effettivamente riferendo alla stessa nazionalità ma che le stringhe utilizzate hanno in un caso l'iniziale maiuscola e nell'altro quella minuscola, viene prodotta una ridondanza nei dati. L'istogramma permette di riconoscere errori di inserimento in modo molto elementare.

La *figura 45* mostra un istogramma che descrive la distribuzione dei valori dell'attributo *isGood* in una collezione.

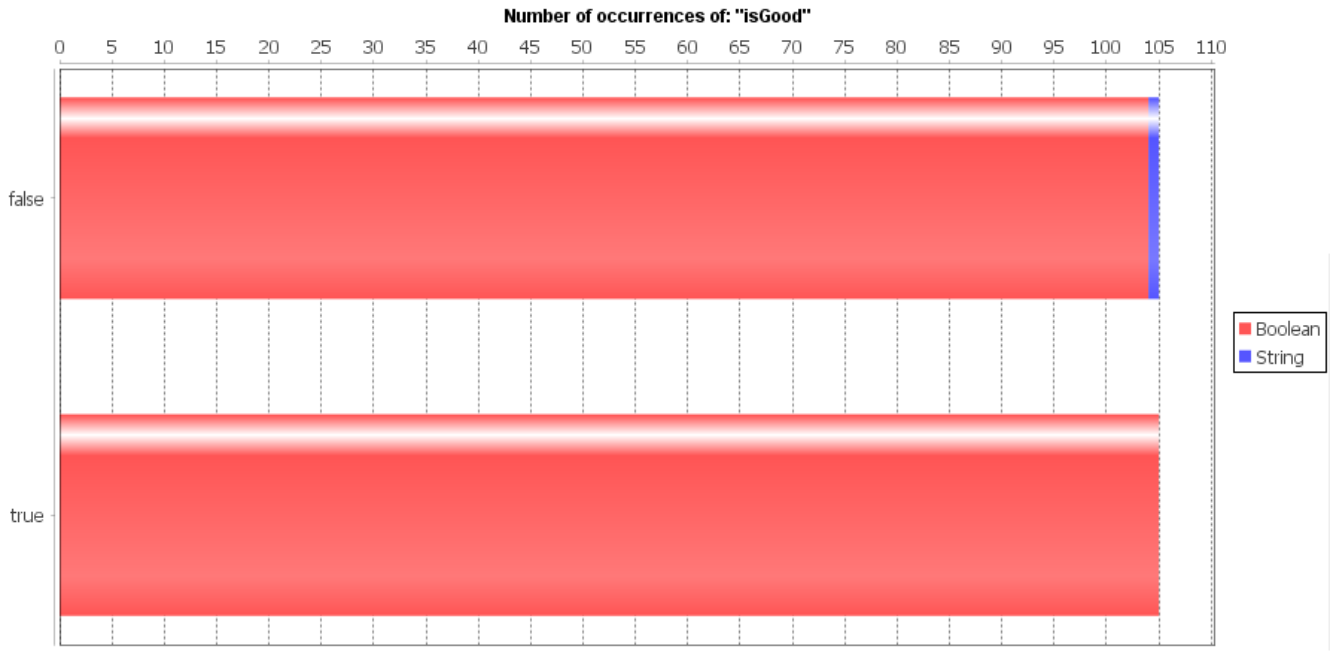

Figura 45. Inserimento di un valore booleano come una stringa

In questo caso il grafico mostra che l'attributo *isGood* assume 104 volte il valore booleano false, 105 true mentre una volta ha per valore la **stringa** "false". È molto probabile che in questa circostanza sia stato compiuto un errore di inserimento: il valore false è stato racchiuso tra apici ed è stato memorizzato dal DBMS come una stringa piuttosto che come un booleano.

Nella *figura numero 46* è illustrato in che modo si distribuiscono i valori dell'attributo *age* in una collezione. Per ben 6 volte il valore del campo *age* assume null come valore. Si ritiene necessaria un'indagine suppletiva al fine di determinare se la scelta di modellazione sia realmente inappropriata

o sei il valore null sia stato assegnato intenzionalmente, nell'eventualità che le informazioni sull'età non fossero ancora note.

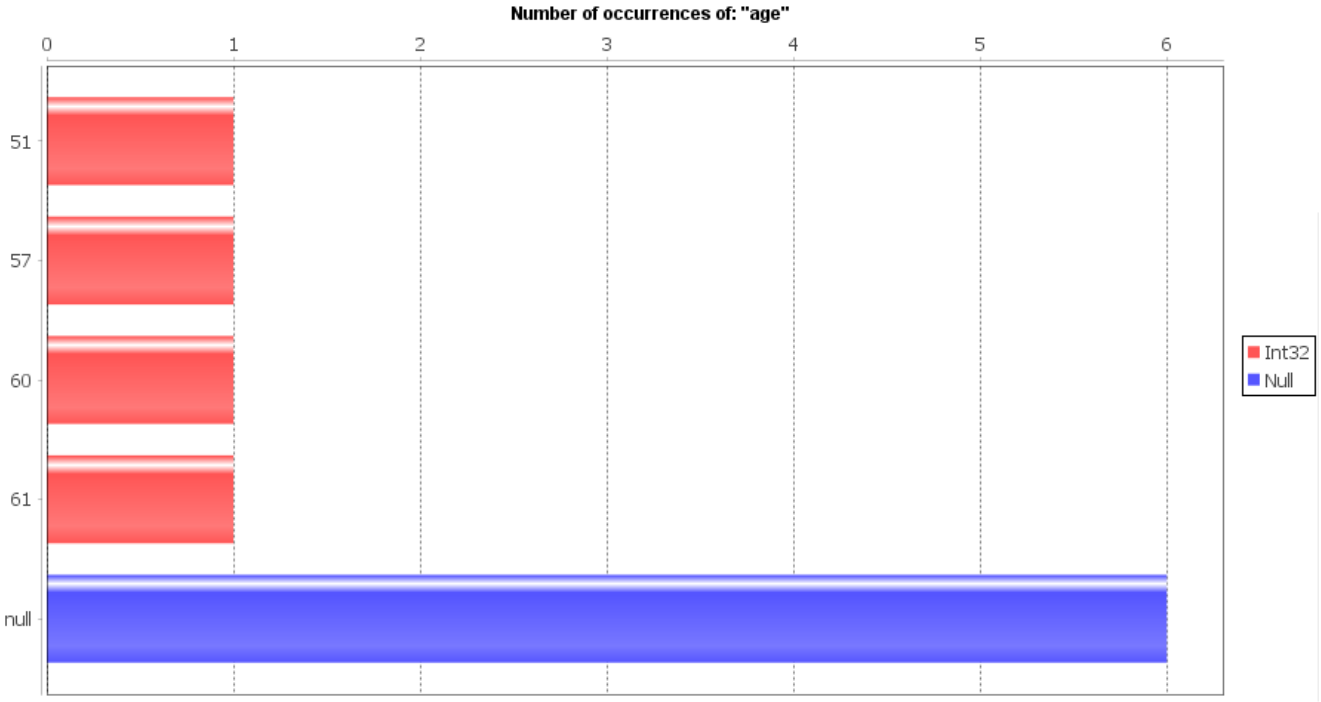

Figura 46. Il campo age assume 6 volte il valore null. Bisogna indagare per comprendere se si tratta di un errore o meno

Nella *figura 47* viene mostrata la tabella contenente il nome, il tipo, la frequenza relativa ed assoluta degli attributi che trovano posto in una collezione. Il campo *lastname* è presente soltanto 1101 volte, con una percentuale del 7% circa. Inoltre tra gli attributi della collezione c'è anche *surname.* È probabile, pertanto, che in alcuni casi il cognome sia stato memorizzato utilizzando l'attributo *lastname* ed in altri *surname*. Quando un attributo ha una frequenza relativa molto bassa, è bene verificare se siano stati commessi degli errori di modellazione.

Field	Type	Number of occurrences	Rate of occurrences
_id	ObjectId	16494	100%
address	Object	15392	93,3188%
age	Int32	15392	93,3188%
company	String	15392	93,3188%
email	String	15392	93,3188%
friends	Array	15392	93,3188%
gender	String	15392	93,3188%
lastname	String	1101	6,6752%
name	String	15392	93,3188%
phone	String	15392	93,3188%
registered	String	15392	93,3188%
surname	String	15393	93,3249%
tags	Array	15392	93,3188%

Figura 47. Il campo *lastname* compare soltanto 1101 volte. Si rende necessaria una verifica per comprendere se si tratta di un errore o meno.

Si tenga presente che quelli descritti sono soltanto alcuni errori tra quelli individuabili utilizzando l'applicativo: scandendo ed analizzando una collezione il software realizzato è in grado di fornire un grande aiuto a coloro che vogliono verificare se il proprio database è stato modellato correttamente e se sono presenti errori di inserimento.

Capitolo 6

Conclusioni

L’obiettivo della tesi è stato quello di descrivere dettagliatamente l’implementazione di un applicativo per l’analisi dello schema implicito di un database non relazionale basato sul modello documentale, nello specifico **MongoDB**. Le difficoltà incontrate nella realizzazione dell’applicazione sono state dovute prevalentemente alla flessibilità dello schema offerta da MongoDB: la completa libertà nella definizione degli attributi ha reso necessario un approccio basato sull’uso delle map-reduce, un paradigma di programmazione grazie al quale si è scansionato ogni attributo presente nella base di dati, individuandone il nome, il tipo, la frequenza relativa ed assoluta. Le map-reduce, però, non sono particolarmente performanti quando eseguite su un unico server. Per questo motivo si è costruito un piccolo cluster distribuito su due nodi servendosi delle politiche di sharding messe a disposizione dal DBMS. Nella fase di tesing, infatti, si è dimostrato come le prestazioni dell’applicativo siano quattro volte migliori shardando il dataset su due server piuttosto che eseguendolo su un'unica macchina.
Per dimostrare l'utilità della ricerca affrontata in questa trattazione, nonché l'innovazione dei metodi attraverso i quali è stata condotta, si mette in evi-

denza che ***3T Software Labs***[26], un'azienda che sviluppa software per MongoDB, ha recentemente rilasciato una nuova applicazione: ***3T Schema Explorer***[27]. L'applicativo è in grado di scansionare lo schema di una collezione rapidamente, anche se non garantisce estrema precisione e non permette di analizzare per ciascun attributo tutti i valori che può assumere. Ho scambiato diversi messaggi con la società, comunicandole dei bug e alcune possibili modifiche che avrebbero potuto effettuare; inoltre, dopo aver proposto loro di provare l'applicativo da me realizzato, ho domandato in che modo avessero reso la scansione dello schema così rapida. Chiaramente non mi hanno potuto fornire una risposta dettagliata ma mi hanno comunicato che non si sono serviti delle map-reduce per l'analisi dello schema. Per questo motivo, sarebbe molto interessante cercare delle soluzioni alternative all'uso delle map-reduce insieme allo sharding. Per esempio, una possibilità sarebbe quella di utilizzare MongoDB congiuntamente ad ***Hadoop!***[28], un framework concepito per scrivere applicazioni che elaborano una grande quantità di dati in parallelo su cluster costituiti da migliaia di nodi assicurando elevata affidabilità e disponibilità. MongoDB ha sviluppato un connettore apposito per Hadoop! nel quale vengono caricati i dati e successivamente automaticamente distribuiti su più nodi. Le interrogazioni per estrarre i dati dal database sono eseguite direttamente sfruttando la potenza computazionale del processore dei server su cui risiede la porzione del dataset interessata dalla query. Utilizzando Hadoop! non è necessario allestire un proprio cluster di nodi ma è sufficiente stabilire una connessione con il framework; nel grafico in *figura 48* sono messe a confronto le prestazioni di Hadoop! e quelle delle map-reduce utilizzate per l'implementazione dell'applicativo [20].

[26] http://3t.io/
[27] http://3t.io/schema-explorer/
[28] http://hadoop.apache.org/

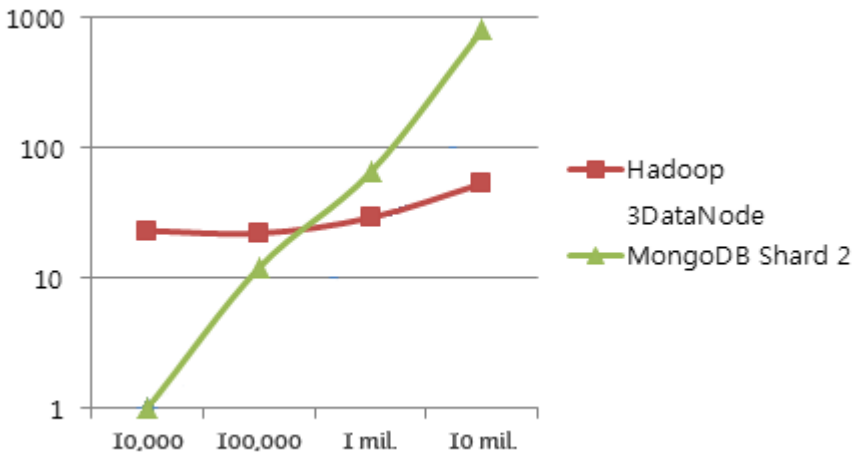

Figura 48. Prestazioni di Hadoop! e delle map-reduce di MongoDB su un cluster di due nodi a confronto

Una volta stabilita una connessione con Hadoop! i dati saranno splittati su diverse macchine in completa trasparenza e quando si interroga la base di dati, MongoDB invia la query ad Hadoop! che la prende in carico inoltrandola ai nodi che contengono le informazioni ricercate e ritornando il risultato al mittente dopo aver ricevuto una risposta.
Un'altra possibilità per la profilazione della base di dati, potrebbe essere quella di utilizzare le aggregazioni per rimodellare lo schema di qualsiasi collezione, facendogli assumere una forma prestabilita e facile da interrogare.

La trattazione, pertanto, ha cercato di descrivere le motivazioni per le quali è importante servirsi di uno strumento apposito per l'analisi di una base di dati non relazionale, fornendo informazioni dettagliate sull'implementazione dell'applicativo e sulle tecniche utilizzate.
Nel complesso, mi ritengo particolarmente soddisfatto: il tool è in grado di analizzare una qualsiasi collezione dando notevole aiuto nel riconoscimento di errori di modellazione o inserimento.

Bibliografia

[1] Carro, «NoSQL databases,» [Online]. Available: https://arxiv.org/ftp/arxiv/papers/1401/1401.2101.pdf.

[2] A. Gambella, «Analisi e sperimentazione del DBMS NoSQL MongoDB: il caso di studio della Social Business Intelligence,» 2014. [Online]. Available: http://docplayer.it/3292687-Analisi-e-sperimentazione-del-dbms-nosql-mongodb-il-caso-di-studio-della-social-business-intelligence.html.

[3] K. Seguin e P. Neal, The Little MongoDB Book, 2014.

[4] Documentazione MongoDB;, «$lookup (aggregation),» 2015. [Online]. Available: https://docs.mongodb.com/manual/reference/operator/aggregation/lookup/.

[5] F. Marchioni, MongoDB for Java Developers, Packt Publishing, 2015, p. 146.

[6] D. MongoDB, «JSON and BSON,» [Online]. Available: https://www.mongodb.com/json-and-bson.

[7] Documentazione MongoDB, «BSON Types,» [Online]. Available: https://docs.mongodb.com/manual/reference/bson-types/.

[8] F. Ragwitz, K. Chodorow e M. Friedman, «CPAN,» [Online]. Available: http://search.cpan.org/~friedo/MongoDB-0.503.2/lib/MongoDB/DataTypes.pod#AUTHORS.

[9] L. Merelli, «ANALISI DELLE PERFORMANCE DEI DATABASE NON RELAZIONALI IL CASO DI STUDIO DI

MONGODB,» [Online]. Available: http://tesi.cab.unipd.it/42073/1/Analisi_delle_performance_dei_database_non_relazionali._Il_c.pdf.

[10] M. documentation, «Map-reduce,» [Online]. Available: https://docs.mongodb.com/manual/core/map-reduce/.

[11] Documentazione MongoDB, «Requirements for the reduce Function,» [Online]. Available: https://docs.mongodb.com/manual/reference/command/mapReduce/#requirements-for-the-reduce-function.

[12] G. VP, «StackOverflow,» [Online]. Available: http://stackoverflow.com/questions/2997004/using-map-reduce-for-mapping-the-properties-in-a-collection.

[13] HTML.it, «Scalabilità: lo Sharding,» [Online]. Available: http://www.html.it/pag/54641/scalabilita/.

[14] Documentazione MongoDB, «Sharding Introduction,» [Online]. Available: https://docs.mongodb.com/manual/core/sharding-introduction/.

[15] A. Gambella, «Analisi e sperimentazione del DBMS noSQL MongoDB: il caso di studio della Social Business Intelligence,» 2015. [Online]. Available: http://amslaurea.unibo.it/8295/1/gambella_alice_tesi.pdf.

[16] Documentazione MongoDB, «Config Servers,» 2016. [Online]. Available: https://docs.mongodb.com/manual/core/sharded-cluster-config-servers/.

[17] K. Chodorow, MongoDB: The Definitive Guide, O'Reilly Media, 2013.

[18] Documentazione MongoDB, «Chunk Splits in a Sharded Cluster,» [Online]. Available: https://docs.mongodb.com/manual/core/sharding-chunk-splitting/.

[19] MongoDB documentation, «Sharded Collection Balancing,» [Online]. Available: https://docs.mongodb.com/manual/core/sharding-balancing/.

[20] J. H. Lee, «Log Analysis System Using Hadoop and MongoDB,» [Online]. Available: http://www.cubrid.org/blog/dev-platform/log-analysis-system-using-hadoop-and-mongodb/.

[21] K. Seguin e P. Neal, The Little MongoDB Book, 2014, p. 29.