Big Data Analytics

Attività 1 – 02/11/2020

Traccia 2
Indexing e query optimization su MongoDB

Pietro Orlandi – matricola 123125

Indici

Introduzione agli indici

Gli indici sono speciali strutture dati che permettono di migliorare enormemente l'efficienza e le prestazioni di un database, minimizzando la quantità di dati da processare dal sistema per risolvere una determinata query.

Gli indici infatti possono prevenire lo scanning dell'intera collection. Delle buone ottimizzazioni tramite indici avvengono in query selettive (ad esempio tramite l'operatore `$eq` o l'operatore `$gt`), ovvero query in cui verranno ritornati solamente una piccola percentuale di documenti dell'intera collection.

Indici in MongoDB

MongoDB solitamente rappresenta internamente un indice come un B-Tree ovvero una struttura dati che permette la rapida localizzazione delle chiavi, riducendo il carico di lavoro in lettura. Inoltre questa struttura dati garantisce il bilanciamento di altezza dell'albero.

I B-Tree facilitano molte tipologie di query, come il match esatto (`$eq`), range queries (`$gt` o `$lt`), l'ordinamento e il prefix matching.

Un indice su MongoDB è costruito in due step:

1) I valori del campo da indicizzare sono ordinati. Un data-set ordinato infatti è molto più efficiente.
2) I valori ordinati sono inseriti dentro all'indice

Anche se gli indici sono essenziali per ottenere delle buone performance in molte query, ogni indice comporta un costo di mantenimento. Infatti, ogni volta che si aggiunge o si rimuove un documento da una collection, verrà modificata la struttura dell'indice.

Solitamente in applicazioni con molte letture, il costo di mantenimento dell'indice è sempre giustificato a fronte delle migliori performance delle query.

Dati utilizzati

I dati utilizzati provengono dal database Amazon fornito dal Prof. Martoglia.

Sono presenti quindi due collection:

- Meta

- Reviews

Collection Meta

Per importare la collection *meta* è stato eseguito da terminale il comando:

mongoimport --db amazon --collection meta meta_Video_Games.json.

La collection contiene i vari metadati di un prodotto Amazon.

In particolare è stata usata la collection inerente alla categoria *Video Games* fornita dal prof nella traccia di questa attività e presente a questo link:
http://www.isgroup.unimore.it/files/meta_Video_Games.json.gz

Questa collection è composta da 50953 documenti.

Collection Review

È stata utilizzata anche la collection *Review*. In particolare non è stata usata tutta la collection ma un subset di 500.000 documenti.

Per riuscire a acquisire tale collection si è utilizzato il software Robo3T per collegarsi al server *pascal3.hpc.unimo.it*, poi si è acceduto alla collection *reviews* e si è cliccato il tasto *"View results in text mode"* (come mostra il cerchio rosso nella figura 2.1) e si è selezionato il range di documenti da 0 a 500000 (sempre come è mostrato in figura 2.1).

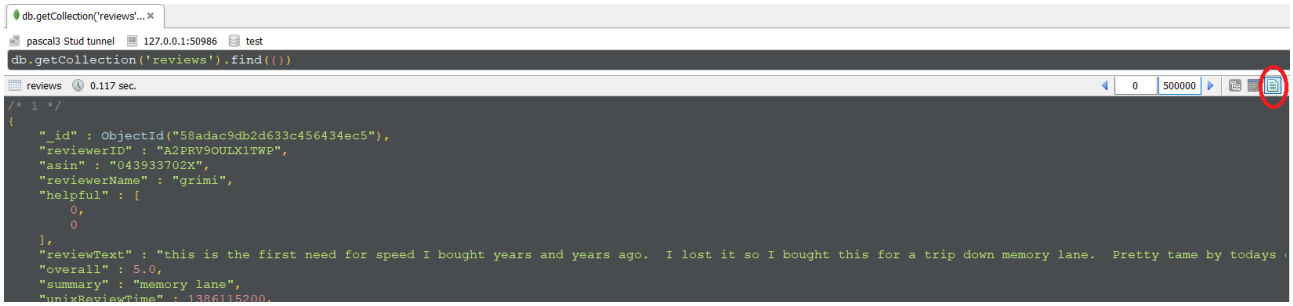

Figura 2.1

Successivamente si è copiato manualmente il contenuto testuale e si è creato un nuovo file *"reviews.json"*.

Tuttavia, quando si andava a eseguire il comando *mongoimport*, questo file risultava che avesse dei problemi di struttura. Usando il seguente script Python (figura 2.2) che utilizza le espressioni regolari è stato modificato il file json nella struttura corretta.

```
import re
with open("reviews.json", 'r') as outfile:
    outfile=re.sub(r'/\*\s\d*\s\*/','',outfile.read())
```

Figura 2.2

Successivamente eseguendo il comando

mongoimport --db amazon --collection reviews reviews.json

è stata importata la collection reviews.

Query

Query 1

Descrizione:

I documenti che sono nei primi 10 della classifica di vendita (salesRank) per la categoria VideoGames.

```
db.meta.find({
    "salesRank.Video Games":{$lte:10}}
    )
```

Figura 3.1 - Query 1

Query 1 senza indice

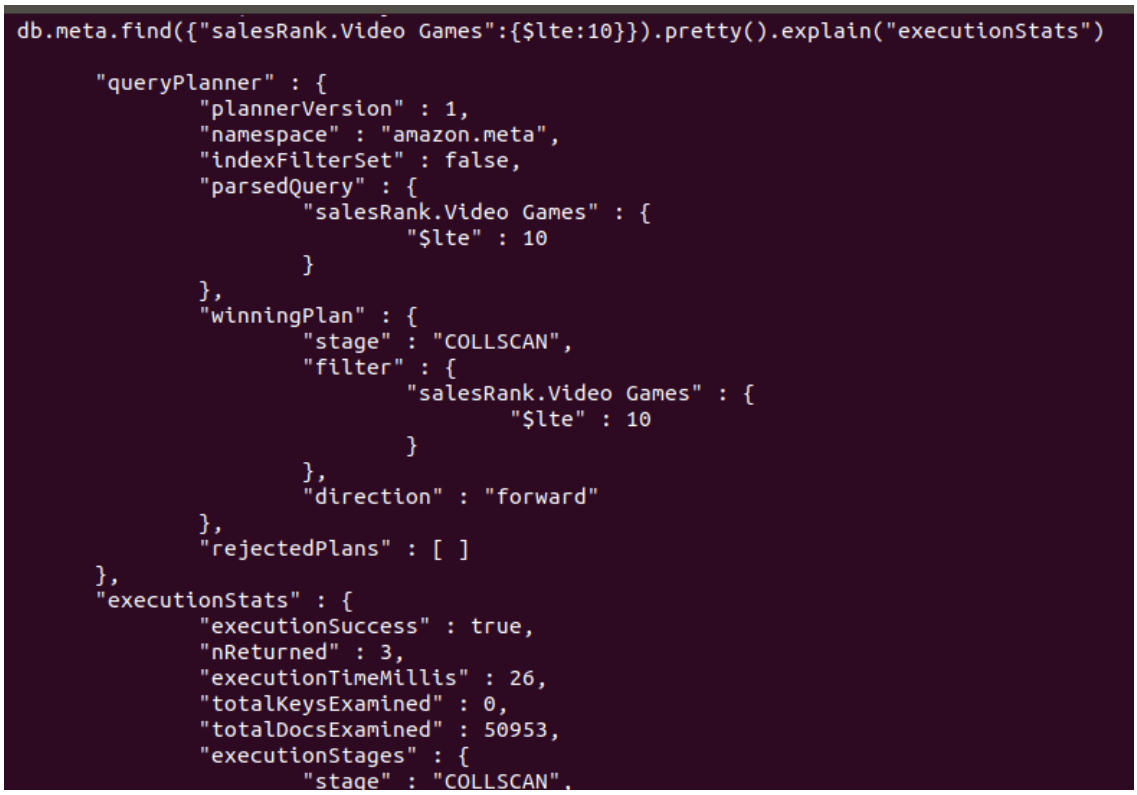

Figura 3.2

Non utilizzando un indice su *salesRank.Video Games* si andrà a vedere la condizione "`$lte:10`" in maniera sequenziale sull'intera collection. Infatti, come si vede dalla figura 3.2, sono stati esaminati 50953 documenti (ovvero tutti) a fronte di solamente 3 documenti ritornati (da notare che teoricamente dovevano tornare i documenti che avevano un valore di "*salesRank.Video Games"* minore uguale a 10; sono solamente 3 perché nella collection "*meta_Video_Games.json*" non erano presenti tutti.)

La query a eseguire, come mostrato dal campo *executionTimeMillis*, ci mette 26ms.

Query 1 con indice su salesRank.Video Games

Prima di eseguire la query viene creato un indice sul field *salesRank.Video Games* utilizzando il comando:

```
db.meta.createIndex({"salesRank.Video Games":1})
```

Successivamente se eseguiamo la query abbiamo i risultati mostrati in figura 3.3:

Figura 3.3

Come si vede dalla figura 3.3, una volta che viene creato l'indice *salesRank.Video Games_1*, viene scelto questo indice dal query optimizer per risolvere la query.
Usando questo indice ci consente di risolvere la query in 0 ms e di esaminare solamente 3 documenti invece dei 50953 documenti confrontati senza usare l'indice. Si può dire quindi che la query è stata effettivamente ottimizzata creando l'indice su *salesRank.VideoGames.*

Query 2

Descrizione:

Prodotti che sono stati comprati dopo aver visto (campo *related.buy_after_viewing*) il prodotto con ASIN 0439573947.

```
db.meta.find({
    "related.buy_after_viewing":"0439573947"
})
```

Figura 3.4 – Query 2

Query 2 senza indice

Figura 3.5

Come si può vedere dalla figura 3.5, non mettendo nessun indice si dovrà fare uno scan lineare di tutta la collection per risolvere la condizione posta da `$eq:"0439573947”.`

Si vede infatti che a fronte dei 2 documenti ritornati dalla query, sono stati esaminati 50953 docs, con un tempo di esecuzione di 34ms.

Query 2 con indice su related.buy_after_viewing

Per prima cosa creiamo l’indice su *related.buy_after_viewing* con il comando

```
db.meta.createIndex({“related.buy_after_viewing”:1})
```

Dopo di che eseguiamo la query, come mostrato in figura 3.6.

Come si nota dalla figura 3.6, innanzitutto si vede che l’indice è di tipo *multikey*: il field related.buy_after_viewing infatti è composto da un array di valori. Come si può intuire dal nome, un indice di tipo multikey permette di associare a varie key uno stesso documento nell’indice.

Si può vedere come utilizzando questo indice la query venga ottimizzata: i documenti esaminati sono pari ai documenti ritornati dalla query ovvero 2. Inoltre il tempo di esecuzione della query è di 1ms, ottenendo uno speed-up di 34x rispetto alla query senza indice.

Figura 3.6

Query 3

Descrizione:

Numero di prodotti di ciascun brand con prezzo maggiore di 250, ordinati in ordine decrescente per numero di prodotti.

```
db.meta.aggregate(
    [
        {$match:{price:{$gte:250}}},
        {$group:{_id : "$brand", num_products : {$sum : 1}}},
        {$sort:{num_products:-1}}
    ]
)
```

Figura 3.7 – Query 3

Query 3 senza indice

Figura 3.8

Come si nota dalla figura 3.8, la query 3 senza indice effettua uno scan lineare di tutta la collection per verificare la prima fase ($match) della pipeline, mettendoci 23ms per eseguire.

Query 3 con indice su price

Viene creato un indice su *price* con questo comando:

```
db.meta.createIndex({price:1})
```

e si esegue la query usando il metodo explain(“executionStats”) per analizzare i risultati (mostrati in figura 3.9)

Figura 3.9

Utilizzando un indice su *price*, si riesce a ottimizzare la fase $match. Infatti grazie all'indice si riuscirà a procurarsi velocemente i documenti che matchano il filtro. Il tempo di esecuzione cala di fino ad arrivare a 4ms, mentre i documenti esaminati passano da 50953 a 939.

Query 4

Descrizione:

Media delle recensioni per il prodotto con ASIN 0078764343

```
db.reviews.aggregate(
    [
        {$match:{asin:"0078764343"}},
        {$group:{_id:"$asin",mean_overall_reviews:{$avg:"$overall"}}}
        ]
)
```

Figura 3.10 – Query 4

Gli indici infatti possono essere sfruttati non solo nella `find` e nella `sort`, ma anche in alcune fasi dell'aggregate, come il `$match` e la `$sort`.

Query 4 senza indice

Figura 3.11

Come si vede dalla figura 3.11, si nota che la query per trovare tutti i documenti che hanno *asin* uguale a 0078764343 debba scannerizzare l'intera collection (ovvero in questo caso 500.000 docs) a fronte di solamente 3 documenti ritornati. Il tempo di esecuzione della query è di 155ms.

Query 4 con indice su asin (basato su BTree)

Creiamo innanzitutto l'indice sull'*asin* con il comando:

```
db.reviews.createIndex({asin:1})
```

In figura 3.12 è mostrato l'output della query utilizzando la funzione *explain.*

Utilizzando un indice su asin si può vedere come le perfomance di questa query migliorino di molto: in figura 3.12 si vede che i documenti esaminati passano da 500.000 (query senza indice) a 3, riuscendo a sfruttare l'indice nell'operazione `asin:{$eq: 0078764343}`. Il tempo di esecuzione passa da 155 ms a 2 ms, ottenendo uno speedup di circa 77x.

Figura 3.12

Query 4 con indice su asin (hashed-index)

Oltre al classico indice basato su Btree in MongoDB esiste anche un indice *hashed*, ovvero un indice che sfrutta una funzione hash.

Per prima cosa eliminiamo l'indice creato in precedenza con il comando

```
db.reviews.dropIndex("asin_1")
```

e poi eseguiamo il comando

```
db.reviews.createIndex({asin:"hashed"})
```

Ora eseguendo di nuovo la query 4 otteniamo i seguenti risultati mostrati in figura 3.13:

Figura 3.13

Le prestazioni sono addirittura maggiori rispetto all'indice normale (infatti qui si raggiungono i 0 ms come tempo di esecuzione rispetto ai 2 ms di prima). Gli indici hashed però hanno meno funzionalità rispetto agli indici basati su BTree, ad esempio non supportano le condizioni sul range (verrà mostrato nella query 5).

Query 5

Descrizione:

Tutte le recensioni del 2015 ordinate in modo decrescente per unixReviewTime.

```
db.reviews.find(
    {unixReviewTime:{$gte:1388534401,$lte:1451606399}}).sort({unixReviewTime:-1}
)
```

Figura 3.14 – Query 5

Da notare che per risolvere la condizione temporale sull'anno è stato utilizzato l*'unixReviewTime* e non il *reviewTime*. È stato fatto questo perché l'*unixReviewTime* è rappresentato come numero e quindi più semplice rispetto all'ordinamento del *reviewTime* che è espresso nel formato "*mm dd, YYYY*". Per convertire una determinata data in unixReviewTime è stato utilizzato il seguente sito: *https://www.unixtimestamp.com/index.php*

Query 5 senza indice

```
db.reviews.find({unixReviewTime:{$gte:1388534401,$lte:1451606399}}).sort({unixReviewTime:-1}).explain("executionStats")

      "queryPlanner" : {
              "plannerVersion" : 1,
              "namespace" : "amazon.reviews",
              "indexFilterSet" : false,
              "parsedQuery" : {
                      "$and" : [
                              {
                                      "unixReviewTime" : {
                                              "$lte" : 1451606399
                                      }
                              },
                              {
                                      "unixReviewTime" : {
                                              "$gte" : 1388534401
                                      }
                              }
                      ]
              },
              "winningPlan" : {
                      "stage" : "SORT",
                      "sortPattern" : {
                              "unixReviewTime" : -1
                      },
                      "inputStage" : {
                              "stage" : "SORT_KEY_GENERATOR",
                              "inputStage" : {
                                      "stage" : "COLLSCAN",
                                      "filter" : {
                                              "$and" : [
                                                      {
                                                              "unixReviewTime" : {
                                                                      "$lte" : 1451606399
                                                              }
                                                      },
                                                      {
                                                              "unixReviewTime" : {
                                                                      "$gte" : 1388534401
                                                              }
                                                      }
                                              ]
                                      },
                                      "direction" : "forward"
                              }
                      }
              },
              "rejectedPlans" : [ ]
      },
```

```
"executionStats" : {
        "executionSuccess" : true,
        "nReturned" : 28948,
        "executionTimeMillis" : 356,
        "totalKeysExamined" : 0,
        "totalDocsExamined" : 500000,
```

Figura 3.15

Siccome nella query è presente un filtro sul range dell'*unixReviewTime* e successivamente un sort, il query optimzer, come mostrato nella sezione winningPlan della figura 3.15, sceglie di fare prima un sort di tutti i documenti (500.000) rispetto al field unixReviewTime, e poi applicare sullo stesso campo il filtro per trovare i documenti che soddisfano la condizione `unixReviewTime:{$gte:1388534401,$lte:1451606399}`.

Viene quindi esaminata l'intera collection per poi ritornare 28948 documenti che risolvono la query. Il tempo di esecuzione per la query 5 senza indice è di 356ms.

Query 5 con indice su unixReviewTime

Creo l'indice usando il comando:

```
db.reviews.createIndex({unixReviewTime:1})
```

Successivamente elaboro la query ottenendo i risultati mostrati in figura 3.16

Figura 3.16

Usando l'indice {*unixReviewTime:1*} si riescono a recuperare velocemente e selettivamente i documenti che soddisfano la condizione {`$gte:1388534401,$lte:1451606399}`.

Inoltre il sort che verrà chiesto successivamente sarà molto veloce siccome i documenti nell'indice *unixReviewTime* vengono mantenuti ordinati.

Si può notare che vengono esaminati solamente i 28948 documenti che verranno poi ritornati dalla query. Il tempo di esecuzione della query con indice è di 40ms, ottenendo un speedup rispetto alla query senza indice di circa 9x (prima infatti era di 356 ms).

Query 5 con indice hashed su unixReviewTime

Si prova a creare un indice di tipo hashed sul field unixReviewTime. Per farlo eseguiamo questo comando per droppare l'indice precedente:

```
db.reviews.dropIndex("unixReviewTime_1")
```

Successivamente creiamo il nuovo indice con il comando

```
db.reviews.createIndex({unixReviewTime:"hashed"})
```

Andando ad eseguire la query i risultati mostrati in figura 3.17.

Figura 3.17

Si può notare infatti che anche se abbiamo creato correttamente l'indice, il query optimizer non sceglie di risolvere la query sfruttando esso. Questo perché sono presenti due filtri del tipo `$lte` e `$gte`: l'indice hash non supporta range queries e per questo non riusciamo ad ottenere benefici per questa query. Per questo, infatti, viene effettuato un scan dell'intera collection come nel caso senza indice.

Query 6

Descrizione:

Tutte le reviews del 2011 con overall >= 4.0 dell'user con reviewerID "A3V6Z4RCDGRC44"

```
db.reviews.find(
    {
        reviewerID:"A3V6Z4RCDGRC44",
        overall:{$gte:4.0},
        unixReviewTime:{$gte:1293840000,$lte:1325375999}

    }
)
```

Figura 3.18 – Query 6

Si può notare come la query filtri i documenti in base a tre campi: il *reviewerID*, *l'overall* e *l'unixReviewTime*. Si andranno a analizzare le prestazioni di questa query utilizzando vari indici tra cui un compound-index formato dalle tre chiavi.
Come prima la condizione sull'anno è stata fatta sul campo *unixReviewTime*.

Query 6 senza indice

Figura 3.19

Come si può vedere dalla figura, viene fatto un scan sequenziale dell'intera collection per risolvere le 3 condizioni (a fronte dei 4 docs ritornati). Si può notare dalla figura che le condizioni sono risolte in `$and`.

Come è scritto nella documentazione di MongoDB (https://docs.mongodb.com/manual/reference/operator/aggregation/and/), l'operatore `$and` usa la valutazione a corto circuito, ovvero l'operazione si stoppa quando incontra la prima espressione falsa. Quindi in pratica vengono scannerizzati tutti i documenti, però appena una delle tre condizioni valutate è falsa si passa al documento successivo.

Il tempo di esecuzione non utilizzando nessun indice è di 153ms.

Query 6 con indice su overall

Viene creato un indice su *overall* utilizzando il comando:

```
db.reviews.createIndex({overall:1})
```

Figura 3.20

Si può notare che creando un indice su *overall* non si ha uno speedup ma si ottiene l'effetto contrario. Infatti si ha un tempo di esecuzione di 322ms rispetto ai 153ms ottenuti senza usare un indice. Si vede infatti vengono esaminati ancora 363083 documenti per risolvere le altre due condizioni (ovvero quella sull'*unixReviewTime* e sul *reviewerID*). Proviamo quindi a creare un altro indice.

Query 6 con indice su unixReviewTime

Viene creato un indice su *unixReviewTime* utilizzando il comando:

```
db.reviews.createIndex({unixReviewTime:1})
```

Figura 3.21

Come si nota dal tempo di esecuzione, utilizzando solamente un indice su *unixReviewTime*, le prestazioni migliorano. Infatti, questa condizione è molto più selettiva rispetto alla condizione rispetto all'*overall*. Questo lo si può notare confrontando i documenti esaminati: utilizzando l'indice su *overall* si esaminano 363083 docs, mentre utilizzando l'indice su *unixReviewTime* se ne esaminano 28046, ovvero solamente l'8% rispetto al precedente indice. Inoltre si può vedere come si ottenga uno speedup rispetto alle versioni precedenti siccome il tempo di esecuzione è di 70ms.

Query 6 con indice su reviewerID

Viene creato un indice su *reviewerID* utilizzando il comando:

```
db.reviews.createIndex({reviewerID:1})
```

Figura 3.22

Come si vede dalla figura 3.22, utilizzando l'indice su *reviewerID* otteniamo performance decisamente migliori rispetto alle versioni precedenti. Il tempo di esecuzione è di solamente 7ms (ottenendo quindi uno speedup di 10x rispetto alla versione con indice su *unixReviewTime*) e vengono confrontati solamente 745 documenti.

Vengono confrontati meno documenti perché la condizione `$eq` sull*'reviewerID* è molto più selettiva delle condizioni sul range dei campi *unixReviewTime* e *overall*.

Query 6 con compound index

Anche se le prestazioni dell'ultimo indice erano buone, si è provato a ottimizzare ulteriormente, andando a creare un compound index composto dalle tre chiavi.

Viene creato il compound-index utilizzando il comando:

```
db.reviews.createIndex({reviewerID:1,unixReviewTime:1,overall:1})
```

Figura 3.23

Utilizzando un compound index si può notare dalla figura 3.23 che si raggiungono le prestazioni migliori. In ordine, la prima chiave del compound index è *reviewerID* ovvero il campo che abbiamo visto che è il più selettivo, la seconda chiave è *unixReviewTime* e la terza chiave è *l'overall*. In questa specifica query abbiamo visto che il modo migliore per creare l'indice composto era in questo ordine di chiavi, ma potrebbero esserci diversi casi in cui, a seconda delle condizioni presenti nelle query, l'ordine migliore di chiavi per il compound index potrebbe essere diverso. In generale, è bene creare l'indice in funzione delle operazioni più usate nell'applicazione e in modo da ottenere una minimizzazione del carico di lavoro.

Ritornando all'analisi della query, si vede che grazie all'indice composto vengono esaminati solamente 4 documenti ovvero solamente i 4 documenti che vengono ritornati alla query. Il tempo di esecuzione è di 3ms, ancora migliore rispetto ai 7ms ottenuti utilizzando l'indice singolo su *reviewerID*.

Nei seguenti grafici in figura 3.24 e 3.25 sono mostrati rispettivamente i tempi di esecuzione e il numero di documenti esaminati utilizzando i diversi indici.

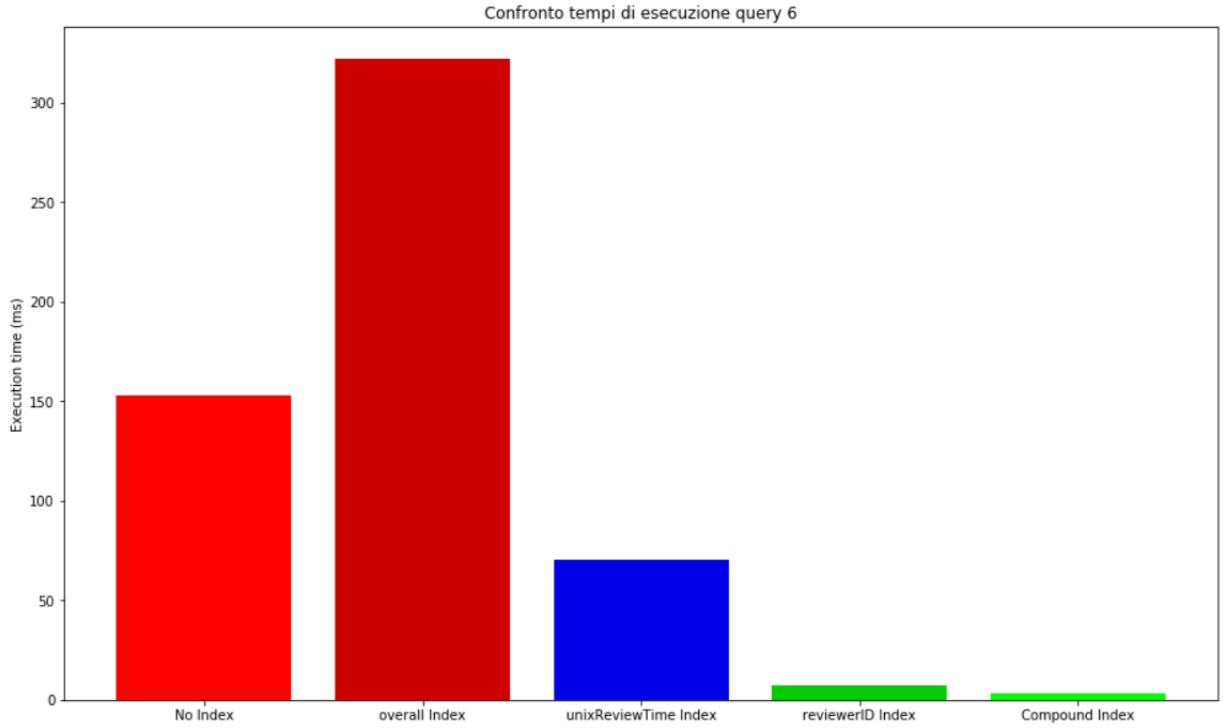

Figura 3.24 – Confronto tempi di esecuzione query 6

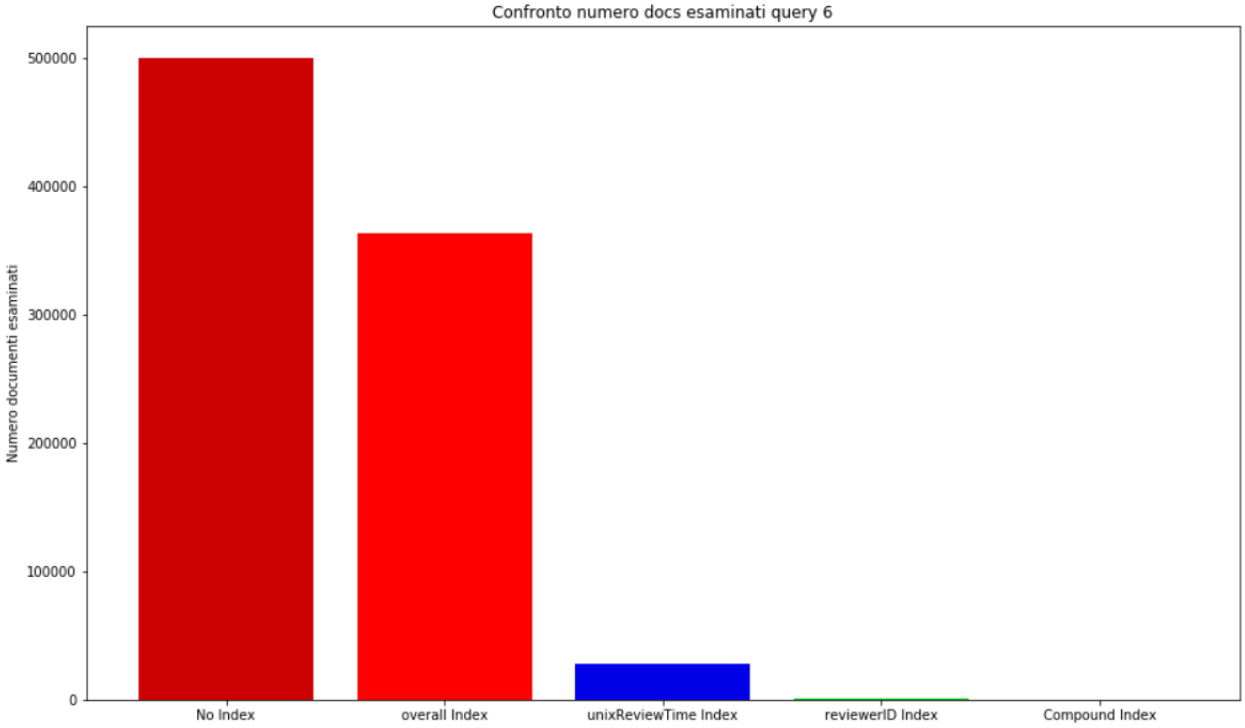

Figura 3.253 - Confronto documenti esaminati query 6

Si può quindi riassumere che per questa query le prestazioni migliori, come si vede dai grafici in figura 3.24 e 3.25, si ottengono creando un compound index. Tuttavia, si ottengono buone prestazioni anche con solamente l'indice sul *reviewerID*. Considerando che un compound index è un po' più oneroso da mantenere (in seguito ad aggiunte e rimozioni di documenti) rispetto a un indice singolo, si può decidere in base all'applicazione quale indice usare.

C’è da enfatizzare che questo compound index può essere utilizzato anche per risolvere query con filtri solamente sul *reviewerID* oppure sul *reviewerID* e il *unixReviewTime* (però ad esempio non può essere sfruttato per query che filtrano solamente sull’*unixReviewTime*).

Query 7

Descrizione:

Gli utenti che hanno effettuato più recensioni minori o uguali a 3.0 effettuate dal 1/1/2014 in poi.

```
db.reviews.aggregate(
    [
        {$match : {overall : {$lte:3.0,},unixReviewTime:{$gte:1388534400}}},
        {$group:{_id:"$reviewerID",number_reviews:{$sum:1}}},
        {$sort:{number_reviews:-1}}
    ]
)
```

Figura 3.46 – Query 7

Query 7 senza indice

Figura 3.27

Senza utilizzare un indice bisogna scannerizzare tutti i documenti mettendoci in totale 200ms.

Query 7 con indice su overall

Viene creato l'indice su *overall* utilizzando il comando:

```
db.reviews.createIndex({overall:1})
```

Figura 3.28

Come si vede in figura 3.28, utilizzando un indice sul campo *overall*, si passa da analizzare 500.000 docs esaminati senza indici a esaminarne 136917 e si passa da un tempo di esecuzione di 200ms a un tempo di 150ms. L'ottimizzazione della query avviene nella fase `$match` dell'aggregate, sfruttando l'indice per vedere i documenti che la soddisfano.

Query 7 con indice su unixReviewTime

L'indice su *unixReviewTime* era già stato creato in delle query precedenti, se però non è presente eseguire il comando:

```
db.reviews.createIndex({unixReviewTime:1})
```

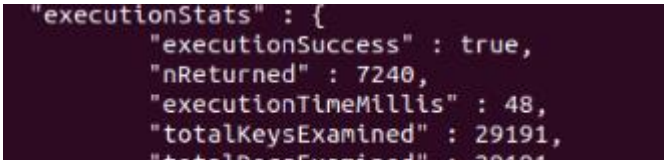

Figura 3.29

Utilizzando un indice sull'*unixReviewTime* invece che sull*'overall*, si passa da avere un tempo di esecuzione di 150ms a ottenere un tempo di 48ms. I documenti analizzati sono 29191, ovvero circa il 20% rispetto ai docs esaminati utilizzando un indice sul field *overall*. Tra le due condizioni quindi, la più selettiva è quella che filtra l*'unixReviewTime*, infatti avendo a disposizione entrambi gli indici, il query optimizer sceglierà quest'ultima.

Conclusioni

Analizzando le varie query e le loro ottimizzazioni, si è capito come gli indici siano strutture dati molto potenti che permettono di ottimizzare molto le prestazioni e diminuire il carico di lavoro del sistema.

Gli indici MongoDB basati su BTree garantiscono spesso grosse ottimizzazione e implementano molte funzionalità, come il match esatto e le range queries come dimostrato nelle sezioni precedenti, ma anche funzionalità come il prefix matching (ad esempio creando un indice con `db.meta.createIndex({brand:1})` e elaborando la query `db.meta.find({brand:{$regex:/^Log.*/}}).explain()` è possibile capire come l'indice sia sfruttato anche per il prefix matching).

Gli indici basati su funzione hash si è visto come abbiano buone performance ma non hanno molte funzionalità, per esempio non permettono di risolvere range queries.

Gli indici non possono essere utilizzati sempre, vi sono dei casi infatti in cui l'utilizzo dell'indice non ottimizza la query. Ad esempio se creassimo un indice sul campo *price* della nostra collection meta con il comando `db.meta.createIndex({price:1})` e poi elaborassimo la query `db.meta.find({price:{$ne:10}}).explain()` vedremmo come usando l'operatore `$ne`, le prestazioni migliori si ottengono scanerizzando l'intera collection senza fare uso di alcun indice.

In conclusione, possiamo dire che generalmente nella maggioranza delle applicazioni che compiono molte letture, il mantenimento di indici spesso è consigliato e su dataset molto grandi permette il risparmio di molte risorse.