



Università degli studi di Milano Bicocca
Dipartimento di Informatica, Sistemistica e Comunicazione
Elaborato per il corso di complementi di basi di dati

Analisi di MongoDB

Gianlorenzo Occhipinti
g.occhipinti3@campus.unimib.it
829524

Milano, Giugno 2020

Indice

Introduzione	1
1 Architettura di MongoDB	2
1.1 Gestore delle interrogazioni	2
1.1.1 La fase di Analysis	3
1.1.2 L'ottimizzazione di una query.	3
1.1.3 Query Plans	4
1.1.4 Query Shape	5
1.1.5 Gli indici	5
1.2 Gestore dei metodi d'accesso	7
1.2.1 Ruoli	7
1.3 Gestione della memoria secondaria con Wired Tiger	8
1.3.1 Wired Tiger Journaling System	8
1.4 Replica Set	9
1.4.1 Oplog	10
1.4.2 Sincronizzazione	10
1.4.3 Elezioni	11
1.4.4 Sicurezza	11
1.5 Sharding	12
1.5.1 Primary Shard	12
1.5.2 Shard Keys	13
1.5.3 Chunks	13
1.5.4 Hashed Sharding	14
1.6 Tipologie di letture e scritture	15
1.6.1 Tipologie di scrittura	15
1.6.2 Tipologie di lettura	17
1.6.3 Failover	18
2 Le transazioni in MongoDB	19
2.1 Le transazioni	19
2.2 Gestore della concorrenza	19
2.2.1 I tipi di lock	20
2.2.2 Isolamento	21
2.3 Gestore delle affidabilità	22

2.3.1	Gestione dei guasti	22
2.4	Un confronto con PostgreSQL	23
3	Setup e test di MongoDB	26
3.1	Config Server	26
3.2	Mongos	28
3.3	Shard	29
3.4	Import dei dati	29
3.5	Sharding di una collezione	30
3.6	Chunk	31
3.7	Inserimento	32
3.8	Distribuzione dei chunk	34
3.9	Transazioni	35
4	Conclusioni	38

Introduzione

Questo elaborato è stato scritto come prova sostitutiva all'appello dell'esame di complementi di basi di dati di giugno 2020. Mi scuso col Lettore se dovessero esserci delle imprecisioni; il documento è stato redatto durante la mia fase di apprendimento, perciò, potrebbe essermi sfuggito qualche dettaglio significativo. È stato comunque svolto un grande lavoro di revisione sul testo, sugli esempi proposti e sulle figure presentate.

In questa relazione vedremo quali sono le scelte architetturali adottate nella versione 4.2 di MongoDB, come sono stati implementati i vari gestori e infine mostreremo come creare un setup completo utilizzando la versione Server Enterprise; non saranno oggetto della relazione argomenti troppo tecnici e lunghi come l'aggregation framework o la sintassi del linguaggio MQL.

Auguro al Lettore una buona lettura.

1. Architettura di MongoDB

MongoDB è un DBMS non relazionale, orientato ai documenti. Classificato come un database di tipo NoSQL, MongoDB si allontana dalla struttura tradizionale basata su tabelle dei database relazionali in favore di documenti in stile JSON con schema dinamico (MongoDB chiama il formato BSON), rendendo l'integrazione di dati di alcuni tipi di applicazioni più facile e veloce. Basato su una architettura multilivello (rappresentata in figura 1.1), dalla versione 3.2 utilizza di default uno storage engine chiamato **Wired Tiger** (in figura WT), in grado di garantire un controllo di concorrenza a livello di documento per le operazioni di scrittura.

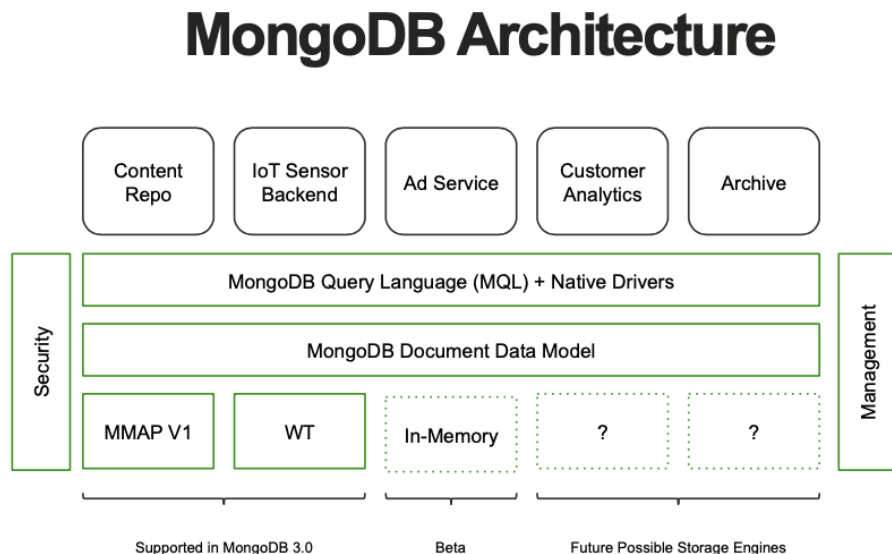


Figura 1.1: L'architettura di MongoDB

In questo capitolo analizzeremo nel dettaglio il modo in cui le classiche componenti di un DBMS sono implementate in MongoDB, soffermandoci in particolare sulle sue componenti che ne hanno caratterizzato il successo.

1.1 Gestore delle interrogazioni

Il gestore delle interrogazioni è un modulo cruciale dell'architettura di un DBMS, in quanto responsabile della corretta esecuzione di operazioni specificate a livello molto alto. Essendo basato su NoSQL, MongoDB non ha un linguaggio standardizzato per le interrogazioni (come SQL) ma, ha ideato un suo linguaggio di interrogazioni proprietario chiamato MQL (MongoDB Query Language) con una sintassi basata a quella del JSON. Per scrivere un'interrogazione bisogna effettuare la combinazione di uno o più **filtri** (che possono a loro volta essere combinati con **espressioni**) così da avere un documento chiamato **Query Filter Document** che rappresenterà l'interrogazione scritta in MQL.

Ad esempio, in questa query

```
{id: 829524, grade: {$lt: 28 }}
```

abbiamo un filtro nel campo `id`, che selezionerà solo i documenti con il valore specificato e un'espressione nel campo `grade`, che selezionerà solo i documenti che rispettano il valore imposto dall'espressione indicata.

Il lavoro svolto dal gestore delle interrogazioni può essere suddiviso in 3 tempi:

- La fase di **Analysis**.
- La fase di **Ottimizzazione**.
- La ricerca di un **Query Plan**.

1.1.1 La fase di Analysis

Questa è la fase più semplice e comune alla maggior parte dei DBMS sul mercato, dove l'unica operazione che viene effettuata è un controllo alla ricerca di eventuali errori lessicali, sintattici o semantici.

1.1.2 L'ottimizzazione di una query.

MongoDB, a differenza dei database relazionali, non è in grado di effettuare un'ottimizzazione algebrica in quanto non esiste una corrispondente rappresentazione (come l'algebra relazionale). E' in corso uno studio condotto da Botoeva e Calvanese intitolato **Formalizing MongoDB Queries** [2] che tenta di formalizzare il MongoDB Query Language, così da aprire le porte in un futuro, ad un eventuale ottimizzazione algebrica.

Una delle funzionalità più apprezzate in MongoDB è l'**aggregation framework**, capace di eseguire delle query in sequenza dove l'input di ogni interrogazione è l'output della precedente.

L'unica ottimizzazione che è in grado di fare il gestore delle interrogazioni, è quella di riorganizzare l'ordine delle query durante un'aggregazione, in modo da consumare meno risorse (RAM/CPU) e renderle più performanti.

Ad esempio, se durante una query viene prima effettuato l'ordinamento di un insieme e poi vengono selezionati solo un tot di elementi che soddisfano determinate condizioni, in questa fase vengono scambiate queste 2 query in modo da effettuare prima la selezione e poi l'ordinamento; in questo modo l'ordinamento (fatto in RAM), in genere, dovrà essere fatto con meno documenti rispetto al caso di partenza.

I casi in cui vengono eseguite queste ottimizzazioni non sono molti e sono preimpostati [3].

1.1.3 Query Plans

In questa fase finale, il gestore delle interrogazioni, per decidere quale sia la strategia più efficace per risolvere il goal della query, cerca quale è l'insieme più efficaci di **work unit**.

Una *work unit* non è altro che un'operazione effettuata dal DBMS per raggiungere il goal della query; ad esempio, la ricerca di tutti gli indici in un range (chiamato **IXSCAN**) è uno stage.

L'insieme di *work unit* eseguiti consecutivamente uno dopo l'altro è detto Query Execution Plan che, fa parte di un possibile Query Plan.



Figura 1.2: Un esempio di un Query Plan con 3 work unit

L'ottimizzazione di una query in MongoDB avviene attraverso la scelta e il salvataggio in cache del Query Plan più efficiente; il criterio di scelta è basato sul numero di "work units" previsti dal query execution plan e, ovviamente, dal loro carico di lavoro previsto.

Molto rilevante nell'individuazione di un Query Plan vincente (quindi più veloce di altri) è il supporto degli indici. Indicizzando un attributo (in figura 1.2 in grassetto) si evita un work su tutta la collezione ("COLLSCAN") e si ottiene invece un work solo su determinati indici ("IXSCAN").

Gli indici sono rappresentati con una struttura ad albero binaria, dove il valore di ogni nodo è un puntatore al documento rappresentato; la ricerca per indici, quindi ha un costo pari a $T(n) = \mathcal{O}(\log n)$ contro $T(n) = \mathcal{O}(n)$ nel caso in cui non si abbia il supporto degli indici.

Proviamo ad eseguire una query con vincoli su un attributo non coperto da indice

Nella figura 1.3a si nota come una query non supportata da nessun indice, abbia un tempo di esecuzione non indifferente; infatti, costringe il DBMS a controllare se il filtro può essere applicato su ogni singolo documento del sistema (si noti il campo "docsExamined"). I tempi di esecuzione fanno la differenza nel caso di collezioni di grandi dimensioni, una collezione con 20.000 documenti viene considerata piccola se pensiamo ai database industriali con milioni e milioni di record. Nell'esempio in figura 1.3a la query ha esaminato 23.5000 documenti e ha impiegato 18m; pur ipotizzando un andamento lineare dei tempi di esecuzione, nel caso di milioni di documenti i tempi salgono tremendamente.

Notiamo immediatamente le differenze in figura 1.3b rispetto alla figura 1.3a; in una IXSCAN non vengono analizzati altri documenti oltre a quelli cercati e il tempo di esecuzione è istantaneo (0ms!).

```

MongoDB Enterprise atlas-00k7z-shard-0.PRIMARY> db.movies.explain('executionState').find({type: 'movie'})
{
  "queryPlanner" : {
    "planCacheVersion" : 1,
    "namespace" : "sample_movie.movies",
    "indexFilterSet" : false,
    "parsedQuery" : {
      "type" : {
        "$eq" : "movie"
      }
    },
    "winningPlan" : {
      "stage" : "COLLSCAN",
      "filter" : {
        "type" : {
          "$eq" : "movie"
        }
      },
      "direction" : "forward"
    },
    "rejectedPlans" : [ ]
  },
  "executionState" : {
    "executionSuccess" : true,
    "returned" : 23285,
    "executionTimeMillis" : 18,
    "totalKeysExamined" : 8,
    "totalDocsExamined" : 23039,
    "executionStages" : {
      "inputStage" : {
        "stage" : "COLLSCAN",
        "filter" : {
          "type" : {
            "$eq" : "movie"
          }
        },
        "returned" : 23285,
        "executionTimeMillisEstimate" : 8,
        "works" : 23641,
        "advanced" : 23285,
        "needTime" : 258,
        "needYield" : 0,
        "saveState" : 181,
        "restoreState" : 183,
        "isEOF" : 1,
        "direction" : "forward",
        "docsExamined" : 23039
      }
    },
    "serverInfo" : {
      "host" : "sandbox-shard-00-02.rnab.mongodb.net",
      "port" : 27017,
      "version" : "4.2.0",
      "gitVersion" : "2836484808f1af16917e4c23c1b5f5ef68b352f8"
    },
    "ok" : 1,
    "clusterTime" : {
      "clusterTime" : Timestamp(1591827889, 1),
      "signature" : {
        "hash" : BinData(0,"a0c9862f8hu5a0zWVb1p(PndwA=)",
        "keyID" : NumberLong("688886825823997954")
      }
    },
    "operationTime" : Timestamp(1591827889, 1)
  }
}

```

(a) Senza il supporto degli indici.

```

"inputStage" : {
  "stage" : "IXSCAN",
  "nReturned" : 23285,
  "executionTimeMillisEstimate" : 0,
  "works" : 23286,
  "advanced" : 23285,
  "needTime" : 0,
  "needYield" : 0,
  "saveState" : 181,
  "restoreState" : 181,
  "isEOF" : 1,
  "keyPattern" : {
    "type" : 1
  },
  "indexName" : "type_1",
  "isMultiKey" : false,
  "multiKeyPaths" : {
    "type" : [ ]
  },
  "isUnique" : false,
  "isSparse" : false,
  "isPartial" : false,
  "indexVersion" : 2,
  "direction" : "forward",
  "indexBounds" : {
    "type" : [
      "[\"movie\", \"movie\"]"
    ]
  },
  "keysExamined" : 23285,
  "seeks" : 1,
  "dupsTested" : 0,
  "dupsDropped" : 0
}

```

(b) Con supporto degli indici

Figura 1.3: Esecuzione di una query sia con che senza il supporto degli indici.

1.1.4 Query Shape

Per determinare quali Query Plan possono essere usati per risolvere una query, il gestore delle interrogazioni controlla se il Query Shape della query è presente in cache, così da ottenere subito il Query Plan vincente.

Una Query Shape è una combinazione dei vari predicati di una query, dei sort e delle proiezioni. Nei predicati della query, solo la struttura del predicato (inclusi i nomi degli attributi) è significativa; ad esempio la query { type: 'phone' } ha la stessa query shape di { type: 'tablet' }.

Per ciascuna Query Shape viene calcolato un **queryHash** con una funzione non iniettiva; in questo modo due query diverse, con Query Shape uguale, hanno lo stesso queryHash.

Il Query Shape introduce il concetto di "somiglianza" di query, grazie alla quale è possibile identificare delle strategie di risoluzione comune valide per più query simili tra di loro.

MongoDB non appena identifica il Query Plan vincente per una query con un dato Query Shape, salva in cache la strategia che ha utilizzato così da non dover ripetere nuovamente la ricerca per una futura query simile a quella appena elaborata.

1.1.5 Gli indici

Abbiamo appena visto come un utilizzo corretto degli indici possa avere una grande influenza nei tempi di esecuzione di una query. Gli indici in MongoDB sono memorizzati in un albero binario di ricerca su un file apposito (lo vedremo più avanti in figura 1.5).

Notiamo come l'implementazione degli indici in MongoDB non si discosta troppo dal funzionamento rispetto ad altri DBMS; un vantaggio senza dubbio rimane la possibilità di avere massimo 64 indici per ciascuna collezione.

Analizziamo alcuni dei più importanti indici presenti su MongoDB:

Single Field Indexes

Indicizzano un attributo in una collezione di documenti, in modo da offrire supporto a tutte le query che filtrano o riordinano documenti su quell'attributo. Di default tutte le collezioni hanno un indice sull'attributo *_id* ma ovviamente è possibile specificarne ulteriori per dare supporto alle query che si vogliono ottimizzare.

Compound Indexes

Gli indici composti sono, come suggerisce il nome, composti da più attributi di un documento in una collezione.

Il supporto dato da questi indici alle query è limitato all'ordine in cui gli attributi sono indicizzati; quindi ad esempio se dovessi creare un indice su questi attributi { "item": 1, "stock": 1 }, questo indice coprirà tutte le query sui campi **item** e **stock** create con determinati criteri.

2D Indexes

Un indice 2d viene usato per indicizzare i documenti che come attributo hanno un punto su un piano 2d. L'utilizzo di questo tipo di indice è limitato a pochi casi d'uso e inoltre un indice di questo tipo non può essere usato come shard key in uno sharded cluster.

Per la rappresentazione di una coordinata MongoDB dà la possibilità di usare l'index 2d appena introdotto oppure, se si sceglie di rappresentare il punto secondo lo standard GeoJSON (RFC 7946), un indice geospaziale.

Geospatial Indexes

Un indice geospaziale permette l'esecuzione efficiente di query "spaziali" (quindi su dei punti in un piano).

La grande potenzialità rispetto ad un indice 2d è quella di usare un filtro chiamato **\$geoWithin** che permette di selezionare solo i documenti con un punto definito all'interno di una determinata area.

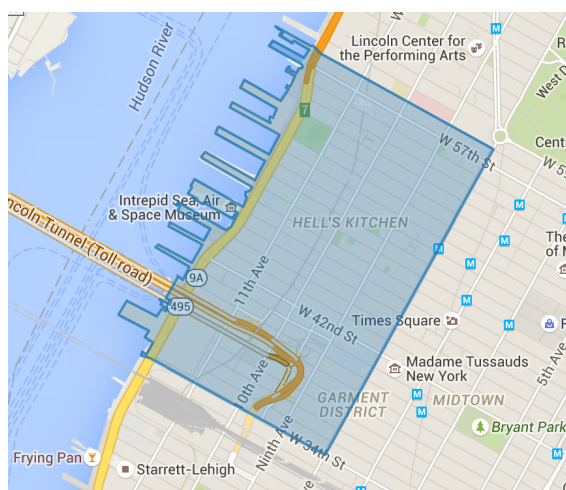
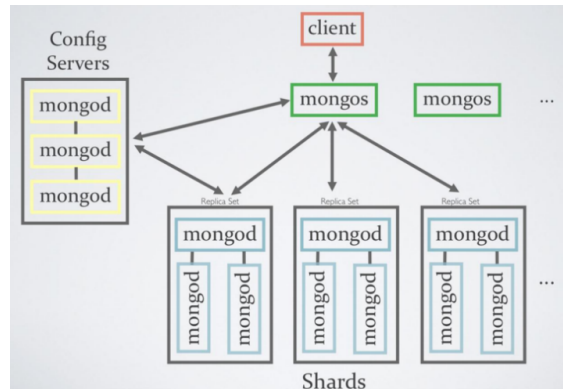


Figura 1.4: Un esempio di come il filtro **\$geoWithin** possa creare un area per filtrare dei documenti indicizzati con un geospatial index

MongoDB, inoltre, ha anche il supporto per un ulteriore tipo di indice per i punti: il **geoHaystack index**; ottimizzato per le coordinate distribuite su un'area ristretta di un piano.

1.2 Gestore dei metodi d'accesso

Il compito fondamentale di un gestore dei metodi d'accesso è quello di individuare il blocco in cui è presente la tupla di interesse; in MongoDB, il cui concetto cardine è proprio la scalabilità orizzontale, per essere sempre in grado di individuare la posizione di un dato, tutte le richieste vengono gestite da un processo chiamato **mongos**.



In base alla strategia di sharding adottata mongos grazie ai metadati salvati nei config servers saprà sempre dove andare a prendere le tuple richieste.

Nella sezione 1.5.1 viene spiegato meglio quali sono le strategie di sharding sopracitate.

1.2.1 Ruoli

Il gestore dei metodi d'accesso consente ad un utente di effettuare operazioni di lettura e scrittura se, precedentemente autorizzato da un amministratore.

Questo controllo di sicurezza è implementato con un controllo d'accesso alle risorse basato sui ruoli (Role Based Access Control). Ad un utente possono essere assegnati uno o più ruoli, che ne determinano l'accesso ai documenti nei database e le operazioni che può eseguire.

Un ruolo è formato da una lista di privilegi, garantiti all'utente che ne riceverà il titolo, salvati in una collezione nel database **admin**.

Un privilegio è una coppia <resource, actions> costituita da:

- **una risorsa**: quindi una collezione o un database.
- **un insieme di azioni**, che saranno operazioni consentite sulla risorsa selezionata.

Ecco un esempio di privilegi concessi in un ruolo:

```
privileges: [
  { resource: { db: "products", collection: "inventory" }, actions: [ "find",
    "update", "insert" ] },
  { resource: { db: "products", collection: "orders" }, actions: [ "find" ] }
]
```

Di default il sistema fornisce dei ruoli built-in pronti all'uso ma è sempre possibile creare dei ruoli in base alle proprie esigenze.

Lo scope di un ruolo è sempre a livello indicato, l'unica eccezione si applica nel caso del database **admin** dove, i privilegi assegnati, si estendono a tutti i database del sistema.

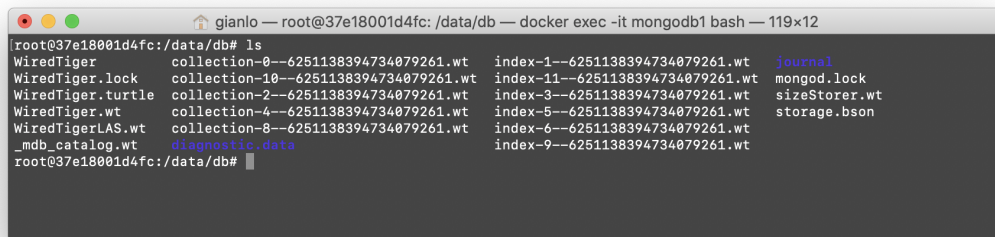
Criptamento

WiredTiger nella versione Enterprise del server di MongoDB offre la possibilità di criptare i dati sul disco utilizzando il protocollo **AES256-CBC**. La scelta di criptare i dati sul disco unita alla possibilità di criptare le connessioni tra i nodi [ref. 1.4.3] rende il sistema sicuro da possibili attacchi atti a rubare i dati.

1.3 Gestione della memoria secondaria con Wired Tiger

Dalla versione 3.2 MongoDB utilizza di default un engine storage chiamato **WiredTiger** in grado di effettuare un controllo di concorrenza sui documenti durante le operazioni di scrittura. WiredTiger è stato acquisito da MongoDB nel 2014 ed il suo sviluppo continua in modo opensource sotto una licenza GPL.

Come sono organizzati i files su disco



```
root@37e18001d4fc:/data/db# ls
WiredTiger          collection-0--6251138394734079261.wt  index-1--6251138394734079261.wt  journal
WiredTiger.lock     collection-10--6251138394734079261.wt  index-11--6251138394734079261.wt  mongod.lock
WiredTiger.turtle   collection-2--6251138394734079261.wt  index-3--6251138394734079261.wt  sizeStorer.wt
WiredTiger.wt       collection-4--6251138394734079261.wt  index-5--6251138394734079261.wt  storage.bson
WiredTigerLAS.wt    collection-8--6251138394734079261.wt  index-6--6251138394734079261.wt
_mdb_catalog.wt     diagnostic.data                        index-9--6251138394734079261.wt
```

Figura 1.5: L'organizzazione dei files su disco usata da WiredTiger

Quando avviamo il server di Mongo di default tutti i dati del database vengono salvati nella directory `"/data/db"` secondo lo schema di figura 1.5. Per ogni collezione, o indice, WiredTiger crea un file individuale che, se non diversamente specificato, viene compresso in formato **zlib** così da minimizzare lo spazio usato a discapito di un costo computazionale maggiore pagato dalla CPU.

WiredTiger genera diversi tipi di files:

- **WiredTiger.lock/mongod.lock**: Usato per evitare che due istanze di MongoDB operino sullo stesso dataset.
- **collection-*.wt**: I files della collezione.
- **index-*.wt**: Gli indici delle collezioni.
- **diagnostic.data**: Questa cartella contiene i dati raccolti dall' FTDC module (Full Time Data Capture) per scopi diagnostici.
- **journal**: Questa directory contiene tutti i file usati dal WiredTiger Journaling System.

1.3.1 Wired Tiger Journaling System

Tutte le operazioni di lettura sono salvate in un buffer e scritte in memoria ogni 60 secondi creando un **checkpoint** dei dati. Wired tiger per garantire atomicità e persistenza (AD) scrive i log con

la tecnica Write-Ahead Logging: le scritture sono segnate prima nei file di log (che si suppone risiedano in memoria stabile) e successivamente le modifiche vengono propagate nel database. Ciascun file di log è limitato a 100MB e per sincronizzare i dati con il disco ed evitare di avere un unico file di log troppo pesante viene usato un meccanismo che prevede la rotazione di rotazione dei log. In caso di crash durante la scrittura vengono letti i log dall'ultimo checkpoint per ricostruire lo stato del database.

Per esempio, durante l'esecuzione di scrittura WiredTiger scrive sul disco i dati ogni 60 secondi, oppure se i file di log raggiungono i 2GB. Con la scrittura dei dati sul disco si crea il checkpoint che sarà eventualmente utilizzato per recuperare i dati.

Document Level Concurrency

Il vantaggio assoluto di WiredTiger rispetto al precedente storage engine usato da MongoDB è la possibilità di effettuare dei lock sui singoli documenti (e non solo sulle intere collezioni come accadeva precedentemente).

1.4 Replica Set

Relativamente al triangolo utilizzato per visualizzare il CAP Theorem, MongoDB è situato sul lato con vertici gli angoli **CP**, quindi deve essere in grado di supportare fallimenti grazie alle varie partizioni del DB in rete (**P**artition Tolerance) mantenendo la consistenza (**C**onsistency) a discapito della disponibilità. I Replica Set sono il modo in cui MongoDB garantisce Consistenza e Tolleranza alle Partizioni.

Essi sono implementati con un meccanismo di tipo *single-master*: ogni set ha un nodo **PRIMARY** utilizzato come entry point (se non diversamente specificato) per le operazioni di scrittura. Tutti i nodi secondari replicano le operazioni (guardando il file di log) del nodo primario.

Ciascun nodo ha uno stato che lo classifica:

- **STARTUP**: un nodo che non è membro effettivo di nessun replica set.
- **PRIMARY**: l'unico nodo nel replica set che accetterà le operazioni di lettura.
- **SECONDARY**: un nodo che di effettuare la sola replicazione dei dati. (può essere promosso a PRIMARY)
- **RECOVERING**: un nodo che sta effettuando una qualche operazione di recupero dei dati. (può essere promosso a PRIMARY)
- **STARTUP2**: un nodo che è appena entrato nel set. (può essere eletto a PRIMARY)
- **ARBITER**: un nodo non atto alla replicazione dei dati con lo scopo di prendere parte alle elezioni per promuovere i nodi a PRIMARY. (può essere eletto a PRIMARY)
- **DOWN/OFFLINE**: un nodo irraggiungibile.
- **ROLLBACK**: un nodo che sta svolgendo un rollback per cui sarà inutilizzabile per le letture. (può essere promosso a PRIMARY)

La Partition Tolerance viene garantita attraverso un sistema di elezioni; nel momento in cui il nodo PRIMARY non viene più rilevato dal resto del Replica Set, i nodi secondari eleggono un nuovo nodo PRIMARY scegliendo, in genere, quello con la versione più recente dei dati. Una volta che tutti i nodi SECONDARY avranno accettato l'elezione del nuovo nodo, saranno riprese le operazioni di scrittura (precedentemente sospese durante l'elezione) così da garantire sempre la consistenza dei dati.

1.4.1 Oplog

Per mantenere i documenti consistenti nei vari nodi dei replica set MongoDB effettua un log di tutte le operazioni effettuate dal nodo PRIMARY in una speciale collezione (capped) chiamata **oplog** (operations log).

Questo tipo di replicazione è detta *Statement-based replication*, che ha la principale caratteristica di essere indipendente dalle piattaforme (e quindi dal loro sistema operativo); se avessero optato per una *Binary replication*, che scrive sui log direttamente il dato scritto in memoria sotto forma di byte, le performance di replicazione sarebbero state più elevate, ma tutti i nodi del set dovrebbero avere una piattaforma consistente (quindi non sono permessi sistemi operativi diversi).

Tutti i nodi secondari eseguono in maniera asincrona le nuove operazioni inserite nell'oplog, in questo modo riescono sia a tenere traccia di tutte le operazioni effettuate, sia a sapere quali ancora devono essere eseguite.

Ciascuna operazione che risiede nell'oplog è **idempotente**, quindi produce sempre lo stesso risultato sia se è effettuata una volta sia se viene effettuata più volte.

Bisogna stare attenti anche alla dimensione di questa collezione, nel caso in cui siano previsti alti carichi di lavoro (es. modifica di molti documenti con una sola query), bisogna avere una collezione abbastanza grande da contenere tutte le operazioni individuali. Questo perché in una capped collection, una volta che i limiti di dimensione vengono raggiunti, vengono sovrascritti i documenti più vecchi che, se non sono già stati scritti in memoria, verrebbero persi.

E' quindi necessario a garantire l'idempotenza e soprattutto è necessario ridurre al minimo il "replication lag" dovuto ad un numero troppo alto di scritture non ancora propagate sui nodi secondari.

Ingrandire la dimensione della collezione degli oplog è sempre possibile fino ad un limite di 50GB mentre, per ridurre il replication lag, è previsto un meccanismo introdotto con la versione 4.2 di MongoDB chiamato "Control Flow", che permette di impostare un limite al numero di secondi necessari per applicare tutte le operazioni effettuate su un nodo PRIMARY, e propagate su un nodo SECONDARY.

Nel caso in cui questo limite venga raggiunto verrà limitato il numero di operazioni di scrittura effettuato nel nodo PRIMARY (non permettendo a tutti i client di poter scrivere) così da permettere ai nodi secondari di poter elaborare gli oplog rimanenti.

Il concetto di "replication lag" si verifica solo nel caso in cui si ha una scrittura di tipo "majority" [ref. 1.6.1] che vedremo successivamente.

1.4.2 Sincronizzazione

Una delle operazioni più importanti eseguite all'interno di un Replica Set, è quella di mantenere tutti i nodi sincronizzati, in modo da poter avere sempre una versione aggiornata dei dati.

Un nodo nuovo, appena entrato in un Replica Set già configurato, effettua una sincronizzazione iniziale (detta **initial sync**) e copia tutti i dati da un nodo (si può anche specificare quale) a se stesso. Durante questa fase vengono clonati tutti i database dentro dal nodo sorgente, ad eccezione del database "local", e successivamente vengono scaricate tutte le modifiche recenti (oplog) e sincronizzate tramite un secondo tipo di sincronizzazione detto **Replication**.

Ciascun nodo di un replica set possiede una view dei dati chiamata "majority-view" dove risiede una versione dei dati accertata da tutto il set; questo tipo di view dà la sicurezza ad un nodo di avere un dato consistente, infatti l'informazione è presente nella maggior parte dei nodi del set. La "majority-view" viene aggiornata ogni volta che viene effettuata una scrittura di tipo "majority"

[ref. 1.6.1], essa è utile perché leggere da questa view assicura il client (in determinate condizioni) di non effettuare letture inconsistenti.

I nodi secondari replicano i dati in continuazione dopo il loro **initial sync**; una volta selezionato un nodo sorgente (detto **SyncSource**, vengono scaricati i database e le collezioni asincronamente.

Il **SyncSource** viene scelto secondo determinati criteri atti a selezionare il nodo più aggiornato e performante della rete.

Tutti e due i tipi di repliche vengono eseguite in maniera concorrente su più thread.

1.4.3 Elezioni

I Replica Set utilizzano il meccanismo delle elezioni per determinare quale nodo debba diventare PRIMARY nel caso in cui esso diventi irraggiungibile dal resto del Replica Set, oppure, nel caso in cui un nuovo nodo venga aggiunto al set.

I nodi di una replica set mandano continuamente un ping (detto **Heartbeats**) agli altri nodi nel set ogni due secondi per capire se sono raggiungibili; se il ping non torna entro 10 secondi il nodo viene segnalato offline e si inizia il dibattito per le nuove elezioni.

Dalla versione 4.0 MongoDB usa nei Replica Set un protocollo che assegna a ciascun nodo un valore intero che ne indica la priorità durante le elezioni; più alta è la priorità del nodo, maggiore sarà la sua eleggibilità nel rispetto degli altri nodi.

Tutti i nodi dichiarati con priorità 0 non possono essere votati, quindi non possono diventare PRIMARY, e non hanno la possibilità di richiedere nuove elezioni.

L'algoritmo delle elezioni fa in modo di creare una sorta di gerarchia tra i nodi in base al loro livello di priorità, in questo modo non appena il nodo PRIMARY diventa irraggiungibile il nodo SECONDARY con il livello più alto di priorità richiederà delle nuove elezioni.

Il nodo SECONDARY col livello di priorità più alto sarà il primo a chiamare le elezioni rispetto agli altri, anche se, un nodo con un livello di priorità più basso può essere eletto primario per un breve periodo di tempo finché un nodo con un livello di priorità più alto torna disponibile.

Un replica set può avere al massimo 50 membri di cui solo 7 votanti.

Un nodo può essere definito votante o non votante e può al massimo dare un voto durante le elezioni, se un nodo è definito non votante deve avere una priority uguale a 0.

Un nodo SECONDARY con priorità zero continua a mantenere una copia dei dati, accetta le operazioni di lettura ed è in grado di votare nelle elezioni.

1.4.4 Sicurezza

Per natura MongoDB è un database decentralizzato, ciò significa che tutti i nodi che compongono il sistema hanno la necessità di comunicare tra loro.

Se tutti i nodi si trovassero in un private cloud, oppure in una rete locale sicura, la comunicazione tra i nodi non sarebbe soggetta a rischi di intercettazione dei dati, ma in un network pubblico ciò è possibile.

I socket di connessione sono criptati utilizzando i protocolli TLS/SSL con supporto fino alla versione V1.3.

1.5 Sharding

Uno dei punti forza che ha caratterizzato il successo di MongoDB è la possibilità di scalare orizzontalmente il sistema in modo comodo, veloce e sicuro.

Questo DBMS è stato ideato e costruito con il concetto di **shard**, un contenitore per un subset di dati di una collezione di un database. Ogni shard deve essere composto da un replica set (anche di un singolo nodo), così da garantire sempre consistenza e tolleranza ai guasti.

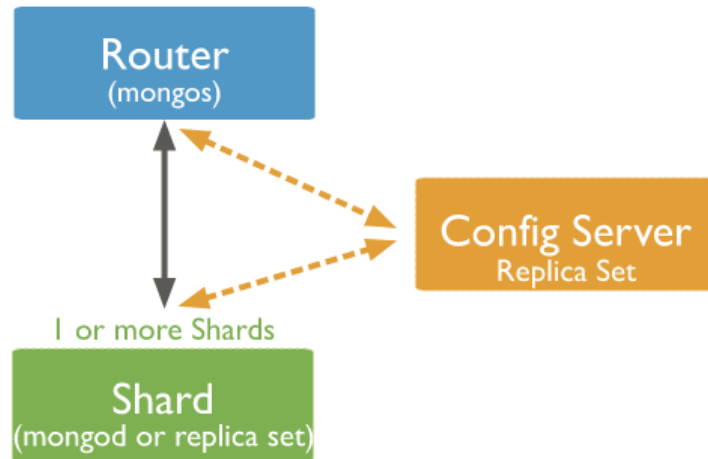


Figura 1.6: Un esempio di shard su MongoDB

Nel momento in cui si decide di effettuare lo sharding di un database, il sistema assume la forma riepilogata in figura 1.6 con le seguenti componenti:

- **mongos:** L'entry point del cluster, tutte le query verranno eseguite su questo processo che, in base alle circostanze, si occuperà di reindirizzare le query allo Shard contenente i dati richiesti (target query) oppure, se ciò non è possibile, effettuerà una richiesta a tutti gli Shard del sistema (broadcast query). Più di un processo mongos può essere utilizzato simultaneamente nel sistema.
- **Config Server:** I server che ospitano i file di configurazione del cluster, necessari al sistema per avere informazioni sulla posizione dei documenti nei vari shard.

MongoDB richiede che sia gli *Shard* sia i *Config Server* siano configurati su dei Replica Set così da garantirne la disponibilità in caso di guasti e non avere un unico point of failure.

1.5.1 Primary Shard

In ogni sistema esiste un Shard detto **Primary Shard** che fungerà da contenitore per tutte le unsharded collections (collezioni che non vogliono essere distribuite su più shard) all'interno del database. Il Primary Shard di un database viene selezionato automaticamente dai mongos scegliendo quello con meno dati fra quelli presenti nel sistema al momento della creazione della richiesta, se non diversamente specificato.

Le ragioni per cui debba esistere un Primary Shard sono molteplici, si pensi ad esempio al database di una multinazionale, con utenza principalmente italiana e con i vari shard sparsi in giro per l'Europa. Se si ha la necessità di avere una bassissima latenza, si potrebbe optare per creare una unsharded collection per tutta l'utenza italiana, situata nello shard ospitato in Italia; a sua volta questo shard potrebbe avere i vari nodi del Replica Set sparsi nei vari datacenter delle principali

città italiane così da ridurre al minimo possibile, il luogo dove ha origine la richiesta dalla posizione fisica del dato.

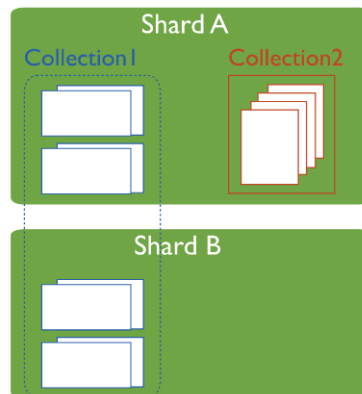


Figura 1.7: Un esempio di Primary Shard su MongoDB. Lo Shard A in questo caso essendo il Primary contiene interamente la Collection2.

1.5.2 Shard Keys

L'attributo di un documento utilizzato da MongoDB per partizionare le collezioni è detto Shard Key. Solo gli indici della collezione (singoli o composti) possono essere usati a questo scopo e una volta selezionata, né la shard key né il suo valore nel documento possono essere cambiati (è quindi **immutabile**) ed inoltre, non è possibile ricomporre la collezione una volta effettuato lo shard.

1.5.3 Chunks

MongoDB utilizza la Shard Key per dividere i documenti in gruppi logici, distribuiti tra i vari Shard, chiamati **chunk**.

I Config Server contengono sempre una mappa dei chunk aggiornata, utile a capire in quale chunk e in quale Shard andare a salvare un documento; la collezione che contiene questa mappa è chiamata "chunks".

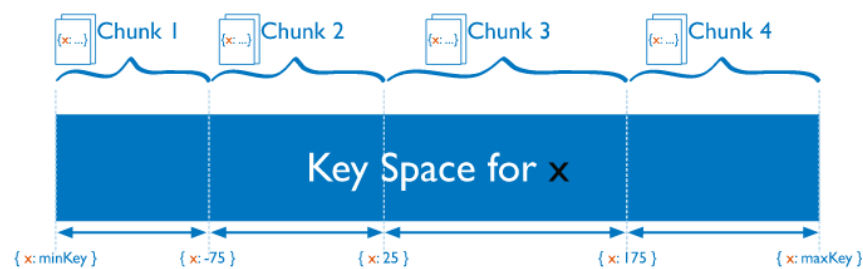


Figura 1.8

La figura 1.8 mostra come avviene la suddivisione logica dei chunk su una generica Sharded Key x, ciascun chunk ha un lower bound (inclusivo) e un upper bound (esclusivo) e il range che comprende la Sharded Key indica il chunk di destinazione del documento. Tutti i documenti nello stesso chunk risiedono fisicamente nello stesso shard, ma la suddivisione dei chunk nei vari shard viene gestita automaticamente in modo da rendere la distribuzione il più bilanciata possibile.

Il numero massimo di chunk che è possibile definire su una sharded key può definire il numero massimo di shard in un sistema.

Per questo il motivo per cui la **cardinalità** della Sharded Key è un aspetto importante da considerare; ad esempio, se un Sharded Key ha cardinalità pari a 1, quindi ad essa è associato sempre un unico valore, tutti i documenti della collezione risiederanno un singolo chunk dal momento che, con un unico valore è impossibile suddividere i documenti in più gruppi, e di conseguenza, è inutile scalare orizzontalmente perché solo un singolo Shard andrebbe utilizzato. D'altro canto, un'alta cardinalità di una sharded key non assicura un'equa distribuzione dei dati nello sharded cluster, infatti, altri fattori da considerare nella scelta di una Sharded Key sono la **frequenza** (assoluta) e la **deviazione standard** del valore di una chiave.

La frequenza (assoluta) di una Sharded Key è il numero di volte in cui una chiave assume un dato valore. Se la maggior parte dei documenti ha lo stesso valore, tutti i documenti andranno nello stesso chunk che, essendo indivisibile non potrà utilizzare tutti gli shard messi a disposizione dal sistema. Un modo di arginare il problema di una possibile Sharded Key su un indice con frequenza alta è quello di dichiararla con un *compound index*, in questo modo un indice con una bassa frequenza insieme alla chiave con alta frequenza possono aiutare il sistema ad un equo bilanciamento dei documenti attraverso i vari chunk nello sharded cluster.

La deviazione standard di una chiave è una stima della variabilità dei valori della chiave all'interno dei documenti, se i valori sono raggruppati, quindi con una bassa variazione fra loro, allora la deviazione sarà bassa; viceversa dei valori molto dispersi fra loro, con un'alta variazione, avranno anche un'alta deviazione. Una deviazione alta permette una distribuzione dei documenti su più chunk invece, una deviazione bassa può portare a indirizzare tutti i documenti nello stesso chunk.

Particolarmente pericoloso è il caso di chiavi i cui valori siano **monotonici** crescenti/descrescenti, in questo caso i documenti verrebbero salvati sempre nell'ultimo/primo chunk a causa dei loro upper ($+\infty$)/lower ($-\infty$) bound

Questo caso particolare è molto frequente (il campo *_id* oppure date/timestamp) e, per venire in contro a queste esigenze, MongoDB ha recentemente messo a disposizione una strategia di indexing sugli Shard chiamata **Hashed Sharding**.

Se un chunk supera la dimensione massima consentite (dichiarabile anche a runtime), esso viene etichettato come **jumbo** chunk. Se il sistema non riesce a smistare i documenti prima della creazione di un jumbo chunk, allora probabilmente si ha un problema di alta frequenza sulla chiave e ciò potrebbe limitare i vantaggi di scalare orizzontalmente un database.

1.5.4 Hashed Sharding

La tecnica dell'hashed sharding prevede l'utilizzo di un hashed index per partizionare i dati su uno sharded cluster.

La funzione di hash predisposta ha come obiettivo quello di garantire un'equa distribuzione dei dati nel cluster in modo da evitare chunk troppo grandi affiancati da chunk troppo piccoli.

L'aspetto negativo di questa tecnica è caratterizzato dall'impossibilità di eseguire efficientemente delle range-based query, cioè query che hanno come obiettivo quello di ottenere dei valori (della chiave) in uno specifico intervallo. Ad esempio, se eseguo una query su un range in un indice che non utilizza un hashed string, per essere indirizzato mongos, grazie alla collezione "chunks", saprà dirmi dove i documenti con valori all'interno dell'intervallo risiedono e in questo caso, si andrà a fare una query ai soli shard che li contengono. Questo tipo di query è detta **target query** mentre, se utilizzo una funzione di hash per smistare i documenti, i chunk saranno suddivisi per valori di hash e non per i valori della chiave; di conseguenza ho la necessità di cercare i valori in tutti gli shard effettuando una **broadcast query**.

1.6 Tipologie di letture e scritture

Il modo in cui MongoDB effettua le partizioni dei documenti aggiunge un'ulteriore difficoltà sia durante le operazioni di scrittura che durante le operazioni lettura.

Ciascun client ha diverse esigenze, c'è chi è più improntato all'efficienza e ai tempi di risposta di una query e chi invece ha la necessità di avere i dati sempre consistenti, per questo motivo MongoDB prevede varie tipologie di letture e scritture.

1.6.1 Tipologie di scrittura

Tutte le scritture sono eseguite sul nodo PRIMARY e propagate nel resto del Replica Set.

Durante un'operazione di scrittura è possibile impostare un'opzione chiamata **writeConcern** per indicare al sistema il comportamento da attuare affinché una scrittura possa essere considerata eseguita sul replica set.

Questa opzione è costituita da 3 valori:

- **w**: il numero di nodi in cui il dato deve essere replicato prima di essere considerato "scritto" nel database.
- **j**: l'intenzione di scrivere sui log di Wired Tiger (write ahead logging) l'operazione che si sta per eseguire
- **wtimeout**: il tempo limite (in millisecondi) da aspettare nel caso in cui **w** sia maggiore di zero.

Il campo **w** del writeConcern può avere diversi tipi di valori:

w:0

Questo valore rende la scrittura più debole ed insicura. In pratica la scrittura non attende nessun tipo di ACK dai nodi nel Replica Set (nemmeno dal PRIMARY) in merito alla propagazione del dato appena scritto.

Ad ogni modo, se una scrittura **w:0** ha anche il campo **j:true**, la scrittura prima di essere considerata eseguita deve prima ricevere un ACK dal gestore della memoria secondaria del nodo PRIMARY in merito alla sua avvenuta registrazione nei log di sistema.

w:1

Questo è il valore di default, impone che scrittura attenda conferma dal nodo PRIMARY dell'avvenuta scrittura del dato all'interno di essa.

w:n

Questo valore fa sì che la scrittura attenda conferma da un numero $n > 1$ di nodi atti a ospitare i dati, prima di tornare l'esito dell'operazione.

I nodi con priorità 0 sono usati per questo tipo di scritture in quanto, se non hanno diritto di voto, mandare un ACK e servire le letture ne rimane l'unico scopo.

w: "majority"

Il valore "majority" fa in modo che la scrittura attende il tempo necessario affinché il dato scritto, sia stato propagato su numero di nodi pari alla cardinalità dell'insieme, che costituisce la maggior parte dei nodi all'interno del replica set.

L'insieme a cui facciamo riferimento è il più piccolo di questi 2:

- l'insieme di tutti i nodi votanti nel set.
- l'insieme di tutti i nodi votanti che possono contenere dati nel set.

Nel primo insieme il nodo PRIMARY viene escluso (in quanto non può votare durante un'elezione) mentre nel secondo insieme viene incluso, viceversa avviene per i nodi ARBITRARY in quanto essi non hanno come scopo quello di immagazzinare i dati.

La seguente figura mostra un esempio temporale di scrittura di tipo "majority":

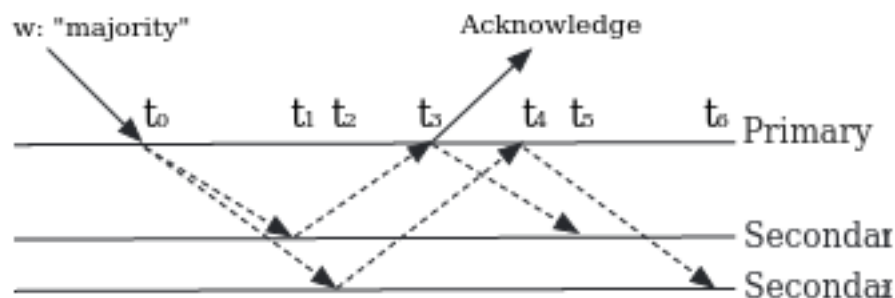


Figura 1.9: Timeline di una scrittura di tipo "majority" su un replica set con 3 membri

In questa situazione l'insieme che costruisce la maggior parte dei nodi equivale a (abbreviando **PRIMARY** e **SECONDARY**):

$$\min = \left\{ \begin{array}{l} |SEC1, SEC| \\ |PRIM, SEC1, SEC2| \end{array} \right\} = \min \{2, 3\} = 2$$

- **t0**: in questo istante il nodo PRIMARY scrive il documento sul disco aggiorna il suo oplog.
- **t1**: il nodo SECONDARY1 scrive il documento sul disco e manda un ACK
- **t2**: il nodo SECONDARY2 scrive il documento sul disco e manda un ACK
- **t3**: il nodo PRIMARY riceve l'ACK di SECONDARY1 e riporta al client l'esito positivo della scrittura. Il nodo PRIMARY aggiorna la sua "majority-view" e comunica al nodo SECONDARY1 di fare lo stesso.
- **t4**: il nodo PRIMARY riceve l'ACK di SECONDARY2 e gli comunica di aggiornare la "majority-view".
- **t5**: SECONDARY1 ha aggiornato la "majority-view".
- **t6**: SECONDARY2 ha aggiornato la "majority-view".

1.6.2 Tipologie di lettura

Di default tutte le operazioni di lettura sono effettuate sul nodo PRIMARY, tuttavia, è possibile effettuare letture anche da nodi SECONDARY oppure dal nodo con latenza inferiore.

Durante una lettura è possibile settare un'opzione chiamata **readConcern** per controllare il livello di isolamento in fase di lettura in modo da avere delle buone performance in funzione delle necessità del client.

MongoDB è stato molto criticato per questa scelta implementativa, infatti alcuni di questi livelli se usati durante le transazioni portano ad anomalie come letture sporche o letture inconsistenti.

Local

Una lettura di tipo "local" è utilizzata di default per leggere i dati dai nodi in un Replica Set; questa tipologia indica che la lettura viene effettuata senza verificare che il dato è stato scritto sul resto dei nodi.

Ciò può essere un problema nel momento in cui la lettura non viene effettuata da un nodo PRIMARY dove si ha la certezza che il dato sia sempre aggiornato, infatti in un nodo SECONDARY (con qualsiasi livello di priorità), non è detto che la versione salvata dei dati sia la più recente (ricordiamo che la propagazione dei dati è effettuata in maniera asincrona).

Available

Una lettura di tipo "available" è utilizzata di default per leggere i dati dai nodi secondari. Nelle collezioni che non sfruttano lo shard, questo livello di lettura si comporta nello stesso modo di "local"; in uno sharded cluster invece, questo livello garantisce una lettura consistente dei dati, dal momento che non legge né dal nodo principale, né chiede al config server dove è situato il dato aggiornato. L'accesso diretto al nodo senza assicurarsi che il dato risieda lì, oltre al problema di una lettura inconsistente dà la possibilità di una lettura di un documento orfano.

Un documento orfano si ottiene quando si incorre in qualche problema (arresto anomalo) durante la migrazione di un chunk su un altro shard, quindi sullo shard di partenza rimangono dei documenti senza più riferimenti da parte di MongoDB. Un documento orfano non implica la perdita di un documento, semplicemente è una vecchia copia che non è stata correttamente cancellata.

Majority

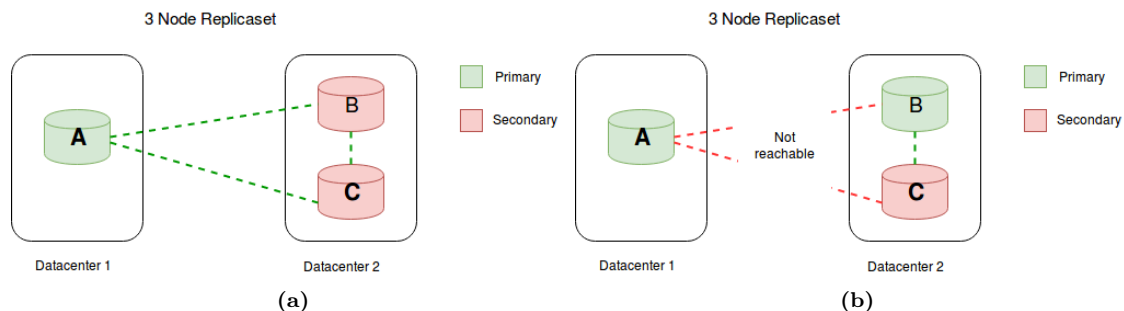
Questo livello di lettura è sicuramente uno dei più importanti in quanto garantisce che il dato letto è stato riconosciuto (ha quindi ricevuto degli ACK) dalla maggior parte dei nodi del replica set (si ha quindi garanzia che i dati non subiranno un roll back).

Dal punto di vista delle performance questo tipo di lettura non è più oneroso rispetto agli altri dal momento che ogni nodo mantiene in memoria la "majority-view" dei dati più aggiornati.

Linearizable

Questo readConcern è stato introdotto nella versione 3.4 di MongoDB per risolvere un potenziale problema di lettura inconsistente.

In pratica supponiamo di avere un replica set di 3 nodi come quello in figura 1.10a



immaginiamo di avere due client *C1* e *C2* che eseguiranno alcune operazioni su documento *x* in una generica collezione nel nostro replica set:

- **t0:** Il client *C1* legge *x* e gli viene tornato il valore *x1*
- **t0:** I nodi **B** e **C** non vedono più il nodo **A** perciò vengono bandite nuove elezioni [Fig. 1.10b]
- **t1:** Il nodo **B** viene eletto come nuovo nodo PRIMARY contemporaneamente al nodo **A** che non si è ancora accorto di non essere più visto dal resto del Replica Set.
- **t2:** Il client *C2* si collega al nodo *B* per scrivere il valore *x2* su *x*.
- **t2:** Viene tornato a *C2* l'ACK della scrittura.
- **t3:** Il client *C1* ancora connesso ad **A** legge, con readConcern "majority" *x*.
- **t3:** La lettura torna il valore *x1*.

Il client *C1* ha appena effettuato una lettura inconsistente sul documento *x* nonostante abbia usato un readConcern "majority", la scrittura fatta dal client **C2** non è visibile al client *C1*.

In questo caso viene violato un concetto cardine dei sistemi concorrenti chiamato **linearizzabilità**; questo concetto prevede che le operazioni su un sistema hanno effetto atomicamente in un certo punto tra la sua invocazione e la sua conclusione. Nel caso dei database, una volta completata la scrittura essa non può avere effetti dopo il suo completamento (comunicati tramite ACK), tutte le letture successive alla scrittura dovranno tornare sempre lo stesso valore.

Per far fronte a questo problema nasce il readConcern "linearizable", che impone alla lettura di controllare se il nodo PRIMARY riesce a vedere la maggior parte dei nodi prima di tornare il risultato dell'operazione di lettura.

Ovviamente questo tipo di lettura ha dei costi superiori agli altri "readConcern".

1.6.3 Failover

Il failover è il processo che permette a un nodo SECONDARY di essere promosso a PRIMARY in caso si verifichi un generico fallimento (i.e. il nodo PRIMARY diventa irraggiungibile).

Quando un nodo SECONDARY viene promosso a PRIMARY Se vengono effettuate letture di tipo "local" o "available", si corre il rischio di dover gestire un rollback dei dati, se, al momento della lettura, si fosse appena verificato un failover.

2. Le transazioni in MongoDB

Come in tutti i DBMS consistenti MongoDB supporta le transazioni implementando a sua volta le primitive di **Commit** e **Rollback**, gode anche delle particolari proprietà "ACID".

Il concetto cardine è che tutte le operazioni sui singoli documenti sono atomiche. Le transazioni che hanno la necessità di operare su più documenti (in una o più collezioni), sono chiamate **multi-document transactions**; il loro supporto per garantire atomicità e isolamento è stato appena introdotto.

Il **Gestore delle transazioni** di MongoDB è stato notevolmente migliorato nella versione 4.2, infatti, come mostrato in figura 2.1, ha appena completato il supporto per le multi-document transactions sugli sharded cluster.

MongoDB 3.0	MongoDB 3.2	MongoDB 3.4	MongoDB 3.6	MongoDB 4.0	MongoDB 4.2
New Storage engine (WiredTiger)	Enhanced replication protocol: stricter consistency & durability	Shard membership awareness	Consistent secondary reads in sharded clusters	Replica Set Transactions	Sharded Transactions
	WiredTiger default storage engine		Logical sessions	Make catalog timestamp-aware	More extensive WiredTiger repair
	Config server manageability improvements		Retryable writes	Snapshot reads	Transaction manager
	Read concern "majority"		Causal Consistency	Recoverable rollback via WT checkpoints	Global point-in-time reads
			Cluster-wide logical clock	Recover to a timestamp	Olog applier prepare support for transactions
			Storage API to changes to use timestamps	Sharded catalog improvements	
			Read concern majority feature always available		
			Collection catalog versioning		
			UUIDs in sharding		
			Fast in-place updates to large documents in WT		

Done

In Progress

Planned

Transaction EPIC

Figura 2.1: Tabella di marcia presentata quando è stata rilasciata la versione 4.0 di MongoDB.

2.1 Le transazioni

MongoDB mette a disposizione dei vari linguaggi di programmazione, dei driver (librerie) per interfacciarsi ed eseguire le operazioni sul database. Esse non supportano più di una transazione per sessione e abortiscono in automatico se una sessione viene chiusa durante l'esecuzione senza **commit** di una transazione.

MongoDB inoltre prevede un tipo di sessione particolare chiamata causalmente consistente (causally consistent session) per dare supporto a quelle operazioni che logicamente dipendono da quelle precedenti alla loro esecuzione. Ad esempio, un'operazione di scrittura che cancella tutti i documenti in base a una specifica condizione seguita da un'operazione che legge i documenti per verificarne la corretta rimozione hanno una relazione causale. Questo tipo di sessioni garantiscono la consistenza causale solo se sia le operazioni di scrittura sia le operazioni di lettura sono di tipo "majority".

2.2 Gestore della concorrenza

Il controllo di concorrenza di MongoDB è in grado di servire diverse applicazioni contemporaneamente utilizzando un meccanismo di controllo basato sui **lock**. L'approccio scelto dallo storage engine *Wired Tiger* è l'optimistic concurrency control, che rimanda il controllo dell'esistenza del lock sulla risorsa, solo quando esso è necessario. Non appena una transazione prova ad effettuare una scrittura su una risorsa con un lock, si verifica un conflitto e il gestore della concorrenza

abortisce e successivamente, prova a rilanciare l'esecuzione di una delle transazioni coinvolte nel conflitto.

2.2.1 I tipi di lock

Il gestore della concorrenza di MongoDB usa dei lock gerarchici in grado di bloccare sia a livello di database, sia a livello di collezione sia a livello di singolo documento. Questo tipo di lock è detto **lock multi-granulare**. Esistono quattro tipologie di lock:

- **S**: Lock di tipo condiviso (Shared), utilizzato per le letture.
- **X**: Lock di tipo esclusivo (eXclusive), utilizzato per le scritture.
- **IS**: Esprime l'intenzione di bloccare in modo condiviso uno dei nodi che discende dal nodo corrente. (Intent Shared)
- **IX**: Esprime l'intenzione di bloccare in modo esclusivo uno dei nodi che discende dal nodo corrente. (Intent eXclusive)

Per esempio, se una transazione dovesse ottenere un lock di tipo *X* su una collezione, sia il corrispondente lock sul database, sia il lock globale dovrebbero essere di tipo *IX*. Per gestire in modo efficiente la coda dei lock, nel momento in cui viene garantito su una risorsa, tutte le altre richieste compatibili che risiedono in coda sono anch'esse esaudite. Per esempio, consideriamo il caso in cui un lock di tipo *X* è stato appena rilasciato, ipotizziamo che la coda delle transazioni in attesa (quindi anche con transazioni in conflitto) sia la seguente:

$$IS \rightarrow IS \rightarrow X \rightarrow X \rightarrow S \rightarrow IS$$

normalmente, un lock manager che segue una metodologia FIFO dovrebbe concedere solo i primi due *IS* lock a sinistra; il lock manager di MongoDB invece, concede tutti i lock di tipo *IS* e *S* nella coda dal momento che essi, sono compatibili tra loro. Non appena saranno rilasciati, concederà alle transazioni, dei lock di tipo *X*, anche se nel frattempo, ci sono nuove transazioni in coda che richiedono lock di tipo *IS* o *S*.

Tutti le altre richieste verranno sempre spostate in testa alla coda così da escludere la possibilità di **starvation** su una richiesta.

L'acquisizione di un lock da parte di una transazione non può impiegare (di default) più di cinque millisecondi. Se tale limite dovesse essere superato la transazione abortirebbe. Questo meccanismo è stato attuato per limitare il più possibile gli eventuali danni di un insieme di transazioni in deadlock.

La figura 2.2 mostra un classico scenario di transazioni che generano un conflitto di scrittura.

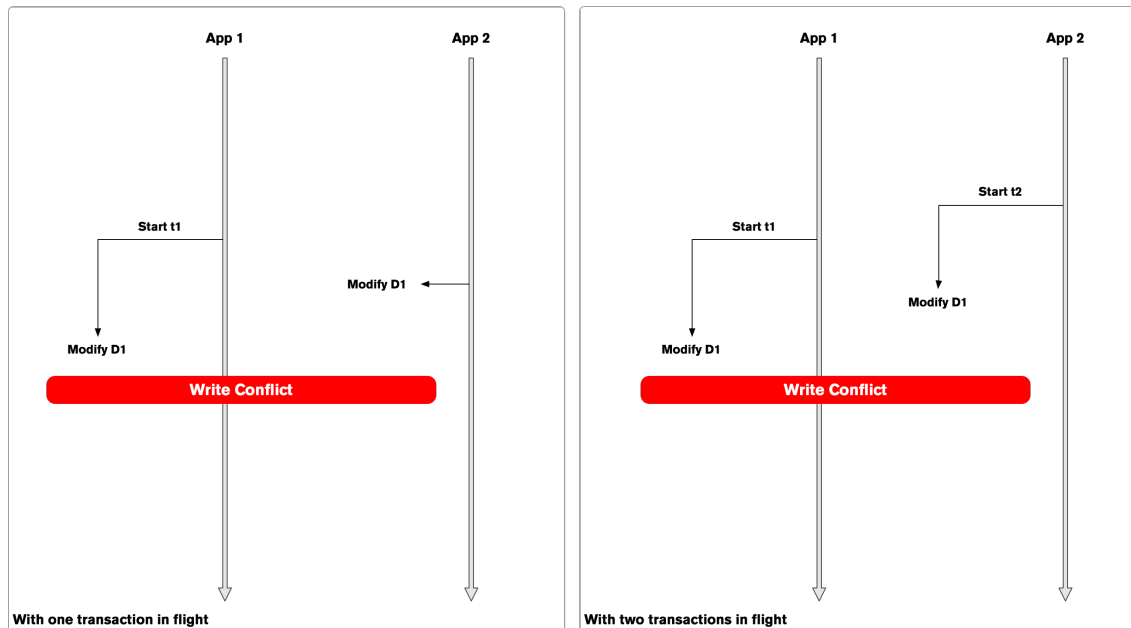


Figura 2.2: Esempio di classico caso in cui si verifica un conflitto di scrittura.

In questo caso la transazione *t1* è soggetta a un conflitto di scrittura dal momento che, un'altra operazione modifica lo stesso documento (D1) nell'esatto istante che percorre l'inizio della transazione e l'operazione di modifica. Il conflitto si verifica a prescindere se l'App2 fa ricorso ad una transazione.

2.2.2 Isolamento

Isolamento e Concorrenza non sono 2 principi che vanno molto d'accordo, è sempre necessario trovare un compromesso in base alle esigenze della propria applicazione; esistono situazioni in cui la presenza di anomalie non influisce sull'operato delle applicazioni (analisi statistiche), quindi si preferisce un alto livello di concorrenza e un basso livello di isolamento, ed altre dove è esattamente l'opposto (transazioni bancarie).

MongoDB per permettere all'applicazione di scegliere il proprio livello di isolamento mette a disposizione i [read/write]Concern a livello di transazione. In questo modo sarebbe possibile specificare delle tipologie di letture/scritture valide per tutte le stesse operazioni all'interno della transazione.

Transaction-level readConcern

La tipologia di lettura va specificata all'inizio di una transazione e può essere di tipo:

- **snapshot:** questa tipologia di lettura garantisce l'isolamento dei vari snapshot e può essere utilizzata solo per le multi-document transactions; effettua una lettura da uno snapshot dei "majority-committed data" se la transazione che ha effettuato scritture precedenti ha già fatto la commit e sono di tipo "majority". Questa tipologia è utile nelle transazioni su più documenti in uno sharded cluster perché sincronizza le varie "majority-committed" view dei replica set.
- **local:** ritorna il dato più recente disponibile nel database ma potrebbe subire un roll-back per una scorretta replicazione. Nelle transazioni su uno sharded cluster la lettura di tipo

"local, non può garantire che venga effettuata dallo stesso snapshot per tutti gli shard. (quindi le letture su più documenti potrebbero essere incosistenti)

- **majority**: ritorna il dato dalla "majority-view" del replica set. Anche qui si potrebbe avere un problema di lettura inconsistente visto nella sezione 1.6.2.
- **linearizable**: non può essere usato nelle causally consistent session e ha lo stesso comportamento analizzato visto nella sezione 1.6.2.

Transaction-level writeConcern

Tutte le scritture in una transazione devono essere della stessa tipologia, settare una tipologia di scrittura alla transazione obbliga, non appena viene effettuato il `commit`, a effettuare tutte le scritture con lo stesso criterio.

Le tipologie di scrittura supportate tutte quelle precedentemente definite, in particolare:

- **"w:1"**: se viene effettuato un `commit` di una transazione con questo livello di scrittura tutte le letture successive di tipo "majority" o "snapshot" non hanno garanzie di consistenza.
- **"w: majority"**: per le transazioni in uno sharded cluster, la view dei dati "majority-committed" non viene sincronizzata tra i vari shard (il livello "snapshot" invece sì).

2.3 Gestore delle affidabilità

Il gestore delle affidabilità di MongoDB serve a garantire atomicità e persistenza delle transazioni, si comporta nel seguente modo in base al comando transazionale lanciato:

- **commit**: Quando una transazione invoca la primitiva di `commit`, tutti i cambiamenti fatti dalla transazione vengono salvati, propagati e resi visibili fuori dallo scope della transazione. Tutti i cambiamenti effettuati all'interno di una transazione non sono visibili da altre operazioni fuori dalla transazione finché essa non chiama la `commit`.
- **abort**: Quando una transazione invoca la primitiva di `abort`, tutte le modifiche "temporanee" effettuate vengono ignorate senza essere mai propagate all'interno del database.

Abbiamo già discusso di come Wired Tiger effettua le scritture nel file di journal [ref 1.3], infatti, durante le transazioni in uno sharded cluster, l'utilizzo dei file di journaling è un prerequisito necessario.

2.3.1 Gestione dei guasti

Il modello scelto per la gestione dei guasti è il *fail-stop*, nel momento in cui il sistema rileva un guasto (di sistema o di dispositivo), vengono immediatamente sospese le transazioni e, in base al tipo di guasto, viene scelto il tipo di ripristino da adottare.

Se dovesse esserci un guasto di sistema, una ripresa a caldo è possibile grazie ai checkpoint effettuati da WiredTiger (ogni 60 secondi) e soprattutto, grazie ai file di journal che permetteranno di ripetere ciascuna operazione effettuata dopo il checkpoint.

La creazione di uno snapshot, utile nel caso di guasti di sistema, è invece più delicata dal momento che essa, prevede il blocco dell'intero sistema. Il caso più semplice è quello in cui si ha un solo nodo, o comunque tutti i dati risiedono in un unico volume dove è molto semplice effettuare uno snapshot di esso. Se invece si dovesse effettuare uno snapshot di un intero Sharded Cluster, bisogna seguire la seguente procedura specifica al fine preservare la consistenza dei dati del sistema:

1. **Disabilitare il balancer**, il componente che si occupa di effettuare la migrazione dei chunk. Se dovesse esserci una migrazione durante uno snapshot si otterrebbero dei dati orfani o, nella peggiore delle ipotesi, duplicati.
2. **Lock dei membri secondari** dei Replica Set, in questo modo non si mette offline un intero sistema ma solo una parte di esso. Il lock è necessario solo se il journaling non si trova nello stesso volume dei dati, in questo modo il nodo smetterà di effettuare le operazioni di repliche previste nell'Oplog così da permettere l'esecuzione di un backup senza dati parziali. Questa procedura va effettuata per tutti i membri del cluster (Config Server compreso).
3. **Backup dei dati.**

Nella versione 4.2 è stato migliorato il ripristino su volumi criptati, ciò significa che il sistema è autonomamente in grado di gestire le chiavi utilizzate garantendo, in base alle circostanze, la rigenerazione dell'**inizializzazione vector** (necessario insieme alla chiavi a criptare i dati) per evitarne un suo riutilizzo.

2.4 Un confronto con PostgreSQL

Effettuare dei benchmark su un database, a volte, può essere molto complicato; paragonare due database che usano approcci differenti (relazionale vs documentale) è ancora peggio. Spesso però capita di chiedersi quale DBMS è migliore rispetto ad un altro e in che circostanza conviene usare chi.

MongoDB è un database ancora in pieno sviluppo, le sue performance con le transazioni non sono il suo punto di forza (le multi-document ACID transactions sono appena state introdotte nella versione 4.0) e ciò lo rende un database adatto a situazioni in cui non si debba fare un uso estensivo di queste.

Ritengo che sia comunque interessante, un confronto di performance con un altro database relazionale: PostgreSQL, entrambi testati su transazioni che rispettano le proprietà ACID utilizzando lo stesso livello di isolamento.

I seguenti risultati sono un estratto di uno studio condotto da Álvaro Hernández Tortosa intitolato PERFORMANCE BENCHMARK POSTGRESQL / MONGODB [1].

In entrambi i database sono state svolte diverse transazioni con le seguenti operazioni:

- Selezione randomica di un record e join (\$lookup) con un altro insieme di record.
- Inserimento (o aggiornamento) di dati randomici anche con riferimenti ad altri record.

Si è inoltre impostato un range di dati da modificare ridotto così da generare frequenti conflitti nelle transazioni concorrenti che provano ad aggiornare lo stesso record.

La seguente tabella evidenzia inoltre i livelli di isolamento offerti sia da MongoDB sia da PostgreSQL ed evidenzia come PostgreSQL offra una scelta più ampia sul livello di isolamento da voler utilizzare rispetto a MongoDB.

Possiamo logicamente dedurre da questa tabella che usare le transazioni in MongoDB ci garantisce l'assenza di anomalie durante le transazioni ma il prezzo da pagare per avere un livello di isolamento *Serializable* è alto, infatti le performance durante le transazioni calano a picco.

Dal grafico in Figura 2.4 notiamo come PostgreSQL sia notevolmente superiore (4-15 volte più veloce) in termini di performance rispetto a MongoDB durante le transazioni; ciò nonostante MongoDB garantisce un livello di isolamento superiore a PostgreSQL (che nei benchmark opera a livello *READ COMMITTED*).

MongoDB	Postgres	Isolation level	Dirty reads	Lost updates	Non-repeatable reads	Phan-toms
without transactions		Read Uncommitted				
		Read Committed				
		Repeatable Read				
transactions		Serializable				

Figura 2.3: Un confronto dei livelli di isolamento offerti dai due DBMS. Qui si nota come MongoDB offre una scelta di tipo "tutto o niente" rispetto a PostgreSQL che invece offre più opzioni.

Ad ogni modo pur effettuando con PostgreSQL le transazioni a livello *SERIALIZABLE* il risultato non cambia.

Il risultato non si discosta troppo dal precedente (PostgreSQL è sempre 4-14 volte più veloce).

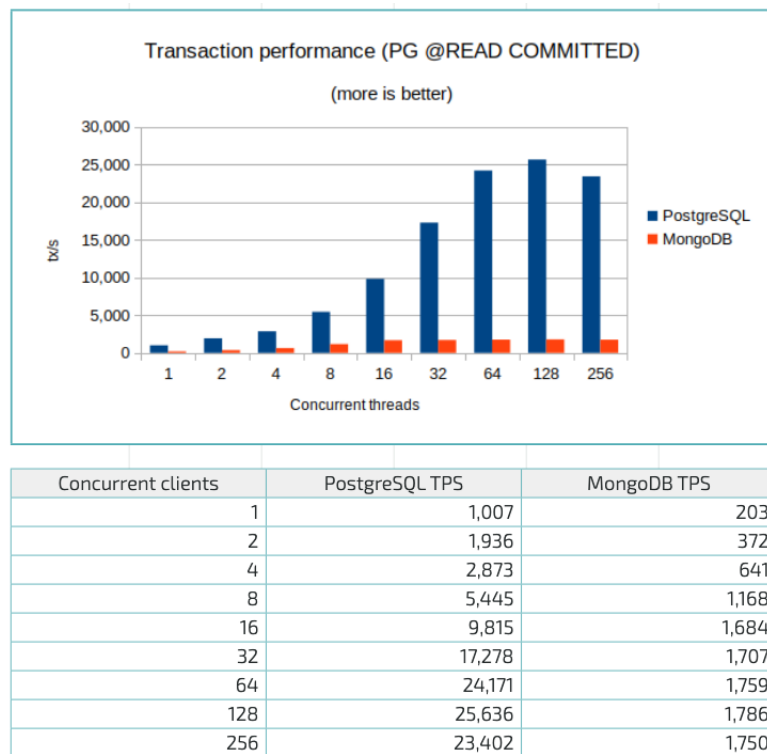


Figura 2.4: Il seguente grafico confronta le performance di MongoDB nelle transazioni contro PostgreSQL con differenti livelli di isolamento.

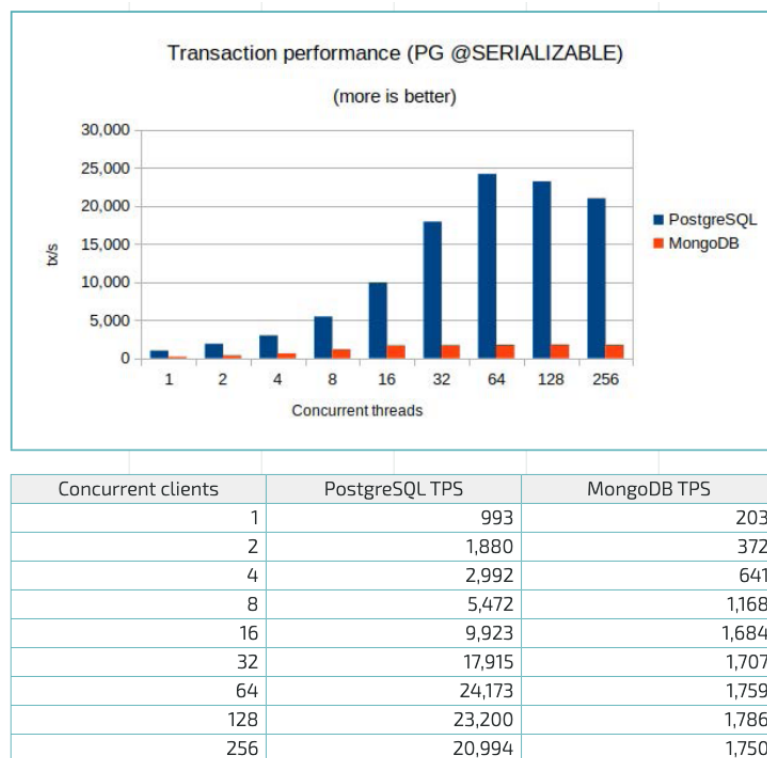


Figura 2.5: Il seguente grafico confronta le performance di MongoDB nelle transazioni contro PostgreSQL entrambi con livello di isolamento *SERIALIZABLE*

3. Setup e test di MongoDB

Mostriamo ora un esempio pratico di un setup completo di MongoDB basato sulla versione 4.2 Server Enterprise.

Il Cluster che andremo a sviluppare composto dai seguenti elementi:

- **Mongos:** l'entry point del sistema, in questo setup ne metteremo solo 1.
- **Config Server:** un Replica Set composto da 3 membri (PSS) con lo scopo di fornire tutte le informazioni sulla suddivisione delle collezioni.
- **ShardA:** un Replica Set composto da 2 membri (PS).
- **ShardB:** un Replica Set composto da 1 membro (P).
- **ShardC:** un Replica Set composto da 1 membro (P).

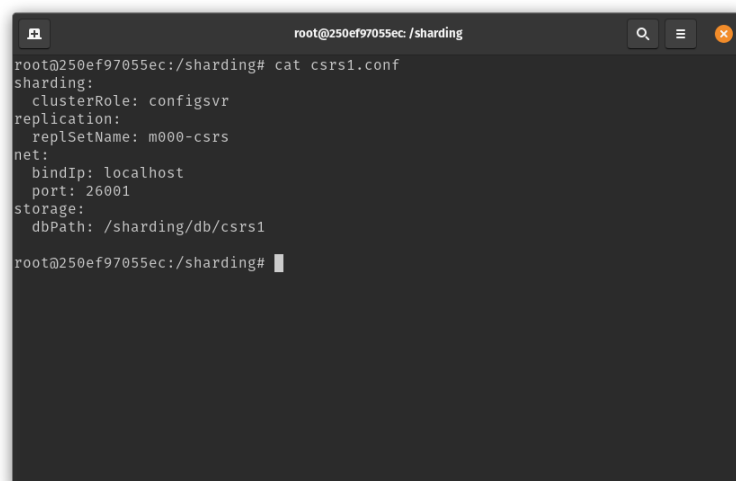
Tutti i processi sono eseguiti sulla stessa macchina in un container docker appositamente creato, disponibile all'interno della repo <https://hub.docker.com/repository/docker/gianlo/readytouse>.

Per semplicità non saranno impostate chiavi al fine di criptare la comunicazione fra i vari shard e verrà creato un solo utente admin con username:password = gianlo:gianlo. I dati dell'esempio sono sia importati da un file JSON disponibile all'indirizzo <https://workspace.occhipinti.dev/uploaded/products.json> sia auto generati con il codice sorgente fornito.

Tutti i codici sono scritti con il linguaggio di programmazione JAVA ed utilizzano i driver messi a disposizione da MongoDB raggiungibili a questo indirizzo <https://mongodb.github.io/mongo-java-driver/>

3.1 Config Server

Per prima cosa procediamo alla creazione del *Config Server Replica Set*, per farlo abbiamo bisogno di avviare MongoDB con dei parametri di configurazione che possono essere passati come argomenti da linea di comando oppure, più comodamente, con un file di configurazione come questo:



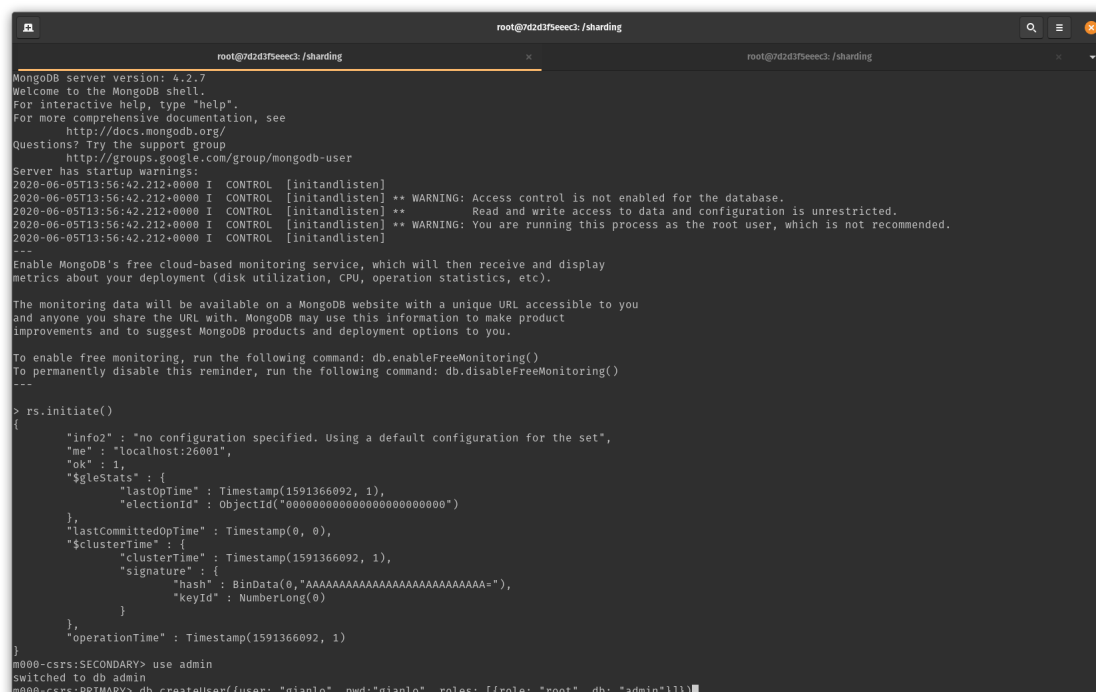
```
root@250ef97055ec:/sharding# cat csrs1.conf
sharding:
  clusterRole: configsvr
replication:
  replSetName: m000-csrs
net:
  bindIp: localhost
  port: 26001
storage:
  dbPath: /sharding/db/csrs1
root@250ef97055ec:/sharding#
```

Figura 3.1: File di config per avviare MongoDB come ConfigServer

Leggendo la configurazione in figura notiamo che con la riga `clusterRole: configsvr` specifichiamo che il server che stiamo per creare dovrà essere un ConfigServer e inoltre, con la dicitura

`replicaSetName: m000-csrs` stiamo inserendo questo server nel replica set (ancora da inizializzare) chiamato `m000-csr`. Avviando il ConfigServer con il comando `mongod -f <file.conf>` viene fatta partire l'istanza del server configurato. Per creare gli altri 2 membri del Replica Set basta copiare il file precedente, assicurandosi di non far coincidere `port` e `dbPath`, e avviare un processo `mongod` passandogli la configurazione appena modificata.

Una volta creati tutti i nodi del Replica Set bisogna inizializzarlo; per farlo basta collegarsi ad un nodo (che diventerà il PRIMARY) con il comando `mongo -port <port>` e digitare il comando `rs.initiate()` come mostrato in figura 3.2.



```
root@7d2d3f5eec3: /sharding
MongoDB server version: 4.2.7
Welcome to the MongoDB shell.
For interactive help, type "help".
For more comprehensive documentation, see
  http://docs.mongodb.org/
Questions? Try the support group
  http://groups.google.com/group/mongodb-user
Server has startup warnings:
2020-06-05T13:56:42.212+0000 I CONTROL [initandlisten]
2020-06-05T13:56:42.212+0000 I CONTROL [initandlisten] ** WARNING: Access control is not enabled for the database.
2020-06-05T13:56:42.212+0000 I CONTROL [initandlisten] **           Read and write access to data and configuration is unrestricted.
2020-06-05T13:56:42.212+0000 I CONTROL [initandlisten] ** WARNING: You are running this process as the root user, which is not recommended.
2020-06-05T13:56:42.212+0000 I CONTROL [initandlisten]
---
Enable MongoDB's free cloud-based monitoring service, which will then receive and display
metrics about your deployment (disk utilization, CPU, operation statistics, etc).

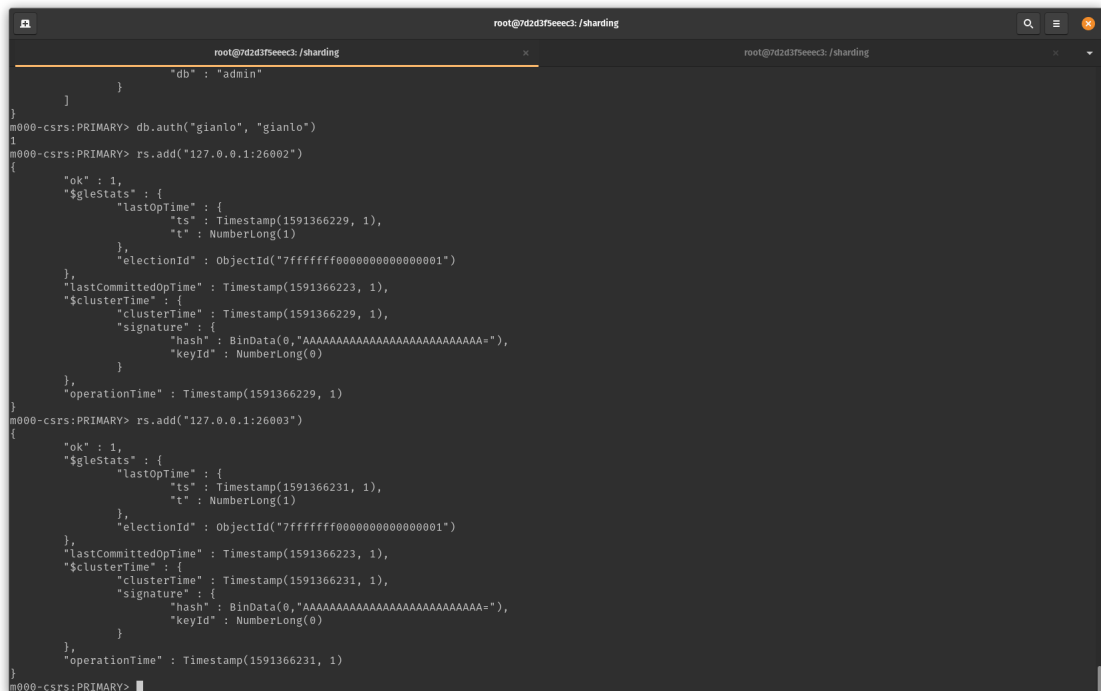
The monitoring data will be available on a MongoDB website with a unique URL accessible to you
and anyone you share the URL with. MongoDB may use this information to make product
improvements and to suggest MongoDB products and deployment options to you.

To enable free monitoring, run the following command: db.enableFreeMonitoring()
To permanently disable this reminder, run the following command: db.disableFreeMonitoring()
---
> rs.initiate()
{
  "info2" : "no configuration specified. Using a default configuration for the set",
  "me" : "localhost:26001",
  "ok" : 1,
  "$gleStats" : {
    "lastOpTime" : Timestamp(1591366092, 1),
    "electionId" : ObjectId("000000000000000000000000")
  },
  "lastCommittedOpTime" : Timestamp(0, 0),
  "$clusterTime" : {
    "clusterTime" : Timestamp(1591366092, 1),
    "signature" : {
      "hash" : BinData(0,"AAAAAAAAAAAAAAAAAAAAAAAA"),
      "keyId" : NumberLong(0)
    }
  },
  "operationTime" : Timestamp(1591366092, 1)
}
m000-csrs:SECONDARY> use admin
switched to db admin
m000-csrs:PRIMARY> db.createUser({user: "gianlo", pwd:"gianlo", roles: [{role: "root", db: "admin"}]})
```

Figura 3.2: Inizializzazione di un replica set. (Sreenshoot effettuato in fase di apprendimento iniziale di MongoDB su una versione non enterprise del server)

Dopo aver inizializzato il Replica Set del Config Server creiamo un utente, con i permessi di root, sul database admin lanciando il comando `db.createUser(user: "<username>", pwd: "<password>", roles: [role: "root", db: "admin"])` sul nodo PRIMARY; questo sarà l'utente che utilizzeremo per i test successivamente.

Infine aggiungiamo i nodi precedentemente creati eseguendo comando `rs.add(<ip>:<porta>)`, per ogni nodo da aggiungere.

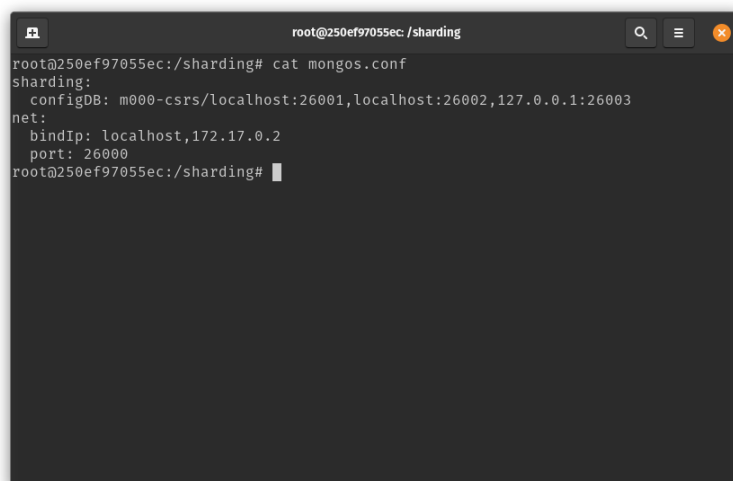


```
root@7d2d3f5eec3: /sharding
    "db" : "admin"
  }
}
m000-csrs:PRIMARY> db.auth("gianlo", "gianlo")
1
m000-csrs:PRIMARY> rs.add("127.0.0.1:26002")
{
  "ok" : 1,
  "$gleStats" : {
    "lastOpTime" : {
      "ts" : Timestamp(1591366229, 1),
      "t" : NumberLong(1)
    },
    "electionId" : ObjectId("7fffffff0000000000000001")
  },
  "lastCommittedOpTime" : Timestamp(1591366223, 1),
  "$clusterTime" : {
    "clusterTime" : Timestamp(1591366229, 1),
    "signature" : {
      "hash" : BinData(0,"AAAAAAAAAAAAAAAAAAAAAAAAAAAA="),
      "keyId" : NumberLong(0)
    }
  },
  "operationTime" : Timestamp(1591366229, 1)
}
m000-csrs:PRIMARY> rs.add("127.0.0.1:26003")
{
  "ok" : 1,
  "$gleStats" : {
    "lastOpTime" : {
      "ts" : Timestamp(1591366231, 1),
      "t" : NumberLong(1)
    },
    "electionId" : ObjectId("7fffffff0000000000000001")
  },
  "lastCommittedOpTime" : Timestamp(1591366223, 1),
  "$clusterTime" : {
    "clusterTime" : Timestamp(1591366231, 1),
    "signature" : {
      "hash" : BinData(0,"AAAAAAAAAAAAAAAAAAAAAAAAAAAA="),
      "keyId" : NumberLong(0)
    }
  },
  "operationTime" : Timestamp(1591366231, 1)
}
m000-csrs:PRIMARY>
```

Figura 3.3: Aggiunta dei membri al Replica Set.

3.2 Mongos

Mongos è il processo che gestirà ed eseguirà tutte le query nel sistema, esso non è altro che un processo, simile a mongod, che non crea nessun server fisico ma funge solo da interfaccia per il sistema creato. Analogamente a mongod anche mongos accetta sia dei parametri, sia un file di configurazione, come mostrato in figura 3.4.



```
root@250ef97055ec: /sharding
root@250ef97055ec:/sharding# cat mongos.conf
sharding:
  configDB: m000-csrs/localhost:26001,localhost:26002,127.0.0.1:26003
net:
  bindIp: localhost,172.17.0.2
  port: 26000
root@250ef97055ec:/sharding#
```

Figura 3.4: File di configurazione di mongos.

Notiamo immediatamente che mongos ha la necessità per essere avviato di conoscere interamente il Replica Set del Config Server, questo ovviamente per motivi legati al partizionamento dei dati;

se la connessione dovesse venir meno al nodo PRIMARY mongos, conoscendo tutti i nodi nel Replica Set, riuscirebbe a cercare il dato in un altro nodo.


Avviamo mongos con il comando `mongos -f <file.conf> -fork` per lanciare il processo come demone in background. Un sistema può avere più processi mongos ad esso collegati.

3.3 Shard

Ora è arrivato il momento di configurare i vari shard del cluster. Abbiamo detto che uno Shard non è altro che un Replica Set, in un sistema con più Replica Set, perciò, la configurazione di uno shard è del tutto analoga a quella vista per il ConfigServer ad eccezione della riga `clusterRole: configsvr` che questa volta non va messa.

Una volta che sono stati avviati tutti i server e istanziati tutti i Replica Set, per aggiungerli bisogna collegarsi al processo mongos con il comando `mongo -port <mongosPort> -u <username> -p <password> -authenticationDatabase "admin"` e per ogni replica set digitare il comando `sh.addShard("<replicaSetName>/<ip:port>...<ip:port>")`.

Una volta aggiunti tutti gli Shard lo status dello Sharded Cluster si può verificare lanciando il comando `sh.status()`.



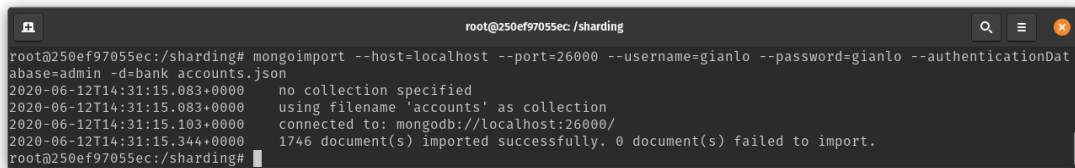
```
--- Sharding Status ---
sharding version: {
  "_id" : 1,
  "minCompatibleVersion" : 5,
  "currentVersion" : 6,
  "clusterId" : ObjectId("5eda85172c9b11548ca8e6ce")
}
shards:
{ "_id" : "m000-repl1", "host" : "m000-repl1/localhost:27011,localhost:27012", "state" : 1 }
{ "_id" : "m000-repl2", "host" : "m000-repl2/localhost:27021", "state" : 1 }
{ "_id" : "m000-repl3", "host" : "m000-repl3/localhost:27031", "state" : 1 }
active mongoses:
  "4.2.7" : 1
autosplit:
  Currently enabled: yes
balancer:
  Currently enabled: yes
  Currently running: yes
  Collections with active migrations:
    config.system.sessions started at Fri Jun 12 2020 13:58:51 GMT+0000 (UTC)
  Failed balancer rounds in last 5 attempts: 0
  Migration Results for the last 24 hours:
    16 : Success
databases:
{ "_id" : "bank", "primary" : "m000-repl1", "partitioned" : false, "version" : { "uuid" : UUID("2f3c118b-c522-4d30-9ff2-3f0273ce1484"), "lastMod" : 1 } }
{ "_id" : "config", "primary" : "config", "partitioned" : true }
config.system.sessions
  shard key: { "_id" : 1 }
  unique: false
  balancing: true
  chunks:
    m000-repl1    1008
    m000-repl2     8
    m000-repl3     8
too many chunks to print, use verbose if you want to force print
MongoDB Enterprise mongos>
```

Figura 3.5: Stato di uno Sharded Cluster.

3.4 Import dei dati

Analogamente ai DBMS relazionali che spesso, tra le varie opzioni, offrono la possibilità di esportare ed importare database tramite un file SQL, in MongoDB esiste un comando chiamato `mongoexport` per esportare tutti i dati (lo schema non esiste) di una collezione in un file JSON; analogamente esiste un comando `mongoimport` per importare i dati da file JSON, CSV o TSV.

La figura 3.6 mostra un esempio dell'esecuzione di questo comando.



```
root@250ef97055ec /sharding
root@250ef97055ec:/sharding# mongoimport --host=localhost --port=26000 --username=gianlo --password=gianlo --authenticationDatabase=admin -d=bank accounts.json
2020-06-12T14:31:15.083+0000 no collection specified
2020-06-12T14:31:15.083+0000 using filename 'accounts' as collection
2020-06-12T14:31:15.103+0000 connected to: mongodb://localhost:26000/
2020-06-12T14:31:15.344+0000 1746 document(s) imported successfully. 0 document(s) failed to import.
root@250ef97055ec:/sharding#
```

Figura 3.6: Import di una collezione su MongoDB.

Attualmente `mongoexport` e `mongoimport` fanno parte del pacchetto programmi scaricato con MongoDB; nella prossima release (versione 4.4) non faranno più parte di MongoDB ma saranno in un insieme di tool accessori chiamato "MongoDB Database Tools" da scaricare separatamente.

3.5 Sharding di una collezione

Sharding Key

Effettuare lo shard di una collezione è un'operazione che richiede una certa analisi preliminare, infatti, sappiamo che la selezione della *Sharding Key* è un'operazione molto delicata in quanto irreversibile.

Per prima cosa bisogna analizzare la collezione di cui vogliamo effettuare lo shard andando alla ricerca degli attributi "fissi" (presenti in ogni documento) e guardando il loro comportamento.

Nel nostro caso, i documenti precedentemente importati, hanno in comune il seguente schema:

- **_id**: l'identificativo univoco di ogni documento nella collezione (monotono crescente).
- **sku**: (Stock Keeping Unit), un numero randomico associato ad ogni prodotto.
- **name**: il nome del prodotto.
- **type**: il tipo del prodotto ("Bundle", "Movie", "Music" ...).
- **regularPrice**: il prezzo del prodotto.
- **salePrice**: il prezzo del prodotto durante uno sconto (cambia a seconda dello sconto del momento).
- **shippingWeight**: il peso del prodotto (non presente per tutti i prodotti).

Notiamo subito che **shippingWeight** non potrà mai fare da sharded key in quanto non è sempre presente, il campo **salePrice** invece, ha la necessità di essere cambiato, il che va contro il concetto di immutabilità di cui necessitiamo. Rimangono **name**, **type**, **sku** e **regularPrice** (**_id** lo consideriamo come ultima spiaggia), a questo punto non ci resta che analizzare la frequenza e la deviazione standard di questi campi.

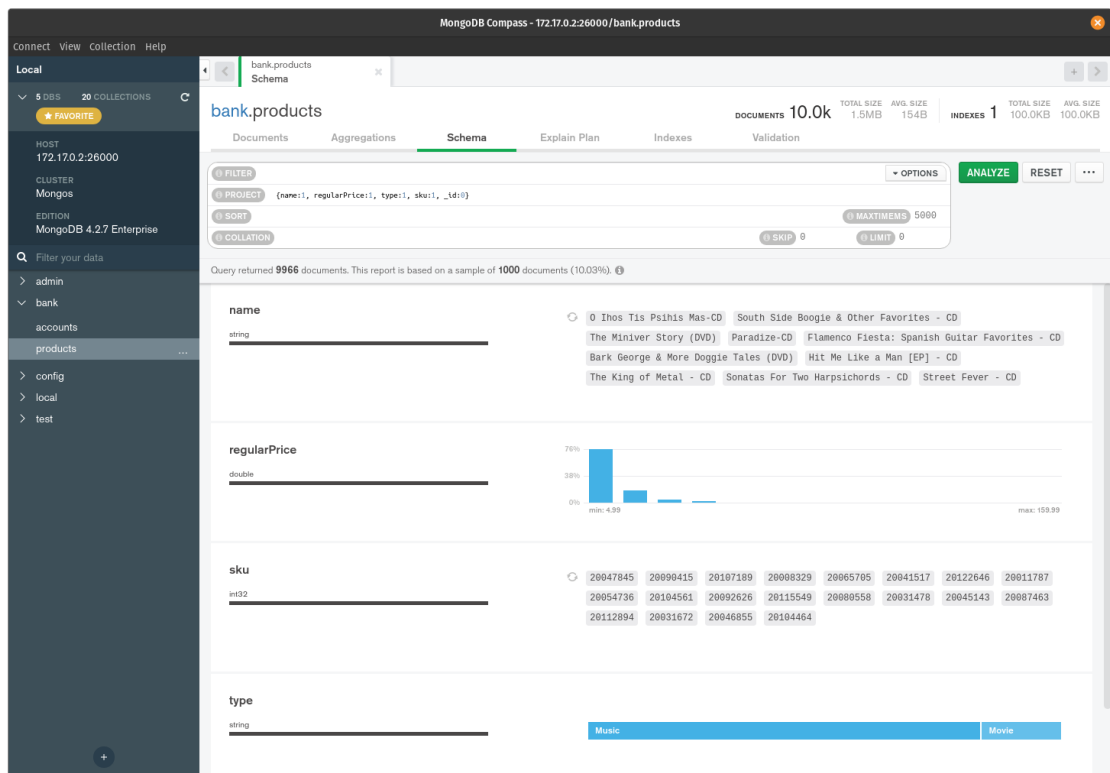


Figura 3.7: Analisi di uno schema di una collezione su un campione di 1000 documenti.

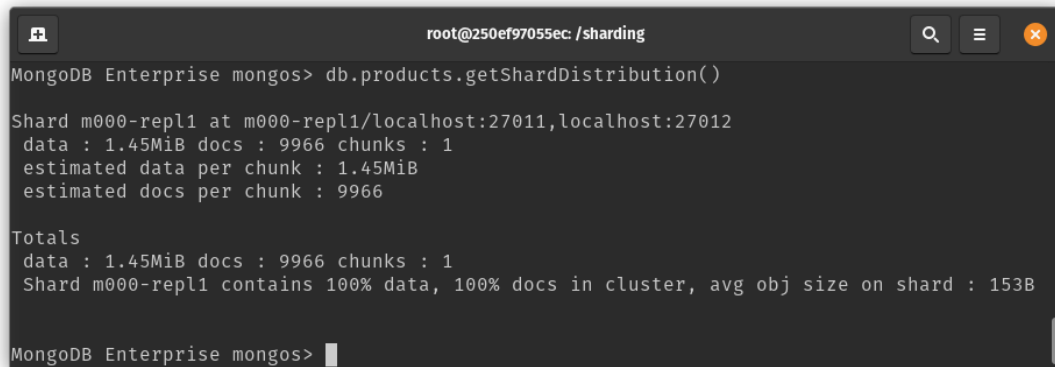
La figura 3.7 mostra la distribuzione dei valori su un campione di 1000 documenti (quindi dati non del tutto affidabile), notiamo subito che **type** e **regularPrice** mostrano un’alta frequenza solo su determinati valori. Provando ad effettuare più volte la stessa analisi notiamo che la frequenza non si discosta molto da quella precedentemente ipotizzata perciò escludiamo i due indici dalla nostra selezione.

La scelta rimane fra il campo **name** e il campo **sku**, per non levare al sistema l’opportunità di cambiare nome ad un prodotto, e per non giocare l’unica indicizzazione possibile sulle stringhe con il campo **name**, optiamo per effettuare lo sharding con una sharding key sull’attributo **sku**.

Creiamo un indice sull’attributo **sku**, attiviamo lo sharding del database e poi effettuiamolo shard della collezione secondo l’indice appena creato.

3.6 Chunk

Stampando la distribuzione dei chunk con il comando `db.<coll>.getShardDistribution`, come mostrato in figura 3.9, notiamo che tutti i documenti sono stati inseriti in un unico chunk.



```
root@250ef97055ec: /sharding

MongoDB Enterprise mongos> db.products.getShardDistribution()

Shard m000-repl1 at m000-repl1/localhost:27011,localhost:27012
data : 1.45MiB docs : 9966 chunks : 1
estimated data per chunk : 1.45MiB
estimated docs per chunk : 9966

Totals
data : 1.45MiB docs : 9966 chunks : 1
Shard m000-repl1 contains 100% data, 100% docs in cluster, avg obj size on shard : 153B

MongoDB Enterprise mongos> 
```

Figura 3.8: Distribuzione dei chunk dopo un import dei dati.

Questo succede perché non è stata superata la dimensione massima di un chunk (di default 64MB) e di conseguenza il sistema non ha provato a suddividere i documenti in più gruppi. Rigarderemo la distribuzione dei chunk dopo aver inserito nel database più dati.

3.7 Inserimento

In genere un inserimento in MongoDB è molto veloce, questo perché i database su mongo sono dei file con accesso casuale dove l'unica operazione da fare, è un append del dato che stiamo creando in un file con accesso casuale, seguito da una indicizzazione che si occuperà di inserire in un B-Tree un indice nella posizione corretta.

Il codice sorgente utilizzato per effettuare le transazioni è il seguente:

```
1 String user = "gianlo";
2 String source = "admin";
3 char[] password = new String("gianlo").toCharArray();
4
5 MongoCredential credential = MongoCredential.createCredential(user, source, password);
6
7 MongoClient mongoClient = MongoClient.create(
8     MongoClientSettings.builder()
9         .applyToClusterSettings(builder ->
10             builder.hosts(Arrays.asList(new ServerAddress("localhost", 26000))))
11         .credential(credential)
12         .build());
13 MongoCollection<Document> products = mongoClient.getDatabase("bank")
14     .getCollection("products")
15     .withWriteConcern(WriteConcern.MAJORITY);
16
17 List<Document> list = new ArrayList<>();
18 for(int i = 0; i < DOCUMENTS; i++){
19     list.add(new Document("name", "Movie " + i)
20         .append("sku", 14000000 + (int) (6000000 * Math.random()))
21         .append("type", "test")
22         .append("regularPrice", 10 + (int) (Math.random() * 30))
23         .append("salePrice", (int) (Math.random() * 30))
24         .append("shippingWeight", 1.0 + Math.random() * 100))
25 }
```

```

25     );
26 }
27
28 System.out.println "[" + THREADID + "] Generated " + list.size() + " elements");
29 long start = System.currentTimeMillis();
30
31 products.insertMany(list);
32
33 long end = System.currentTimeMillis();
34
35 long time = end-start;
36
37 System.out.println "[" + THREADID + "] Insert many took " + time + "ms");

```

Notiamo come nella linea 15 la funzione `withWriteConcern` permette di specificare la tipologia di scrittura che si vuole utilizzare per inserire i dati nella collezione.

La scelta di generare prima tutti i dati per poi inserirli tutti insieme, nella linea 31, è dovuta al fatto che si vogliono monitorare i tempi di inserimento effettivi, senza considerare l'overhead dovuto alla generazione effettiva dei dati.

I driver di MongoDB per java includono un `readConcern` chiamato "JOURNALED", non appartenente alla specifica di MongoDB, che aspetta il tempo necessario affinché la scrittura, da parte di WiredTiger sul file di log, sia stata effettuata su tutti i nodi dello Shard di destinazione del dato.

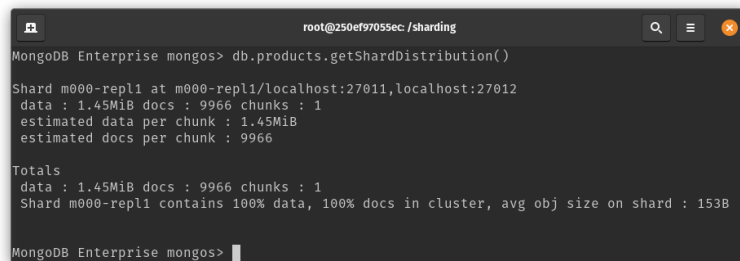
Vediamo ora un confronto nei tempi di inserimento di 10.000 documenti in base alla tipologia di inserimento che si vuole effettuare:

- **"JOURNALED"**: richiede 45865ms \approx 45s
- **"MAJORITY"**: richiede 21485ms \approx 21s
- **"W:1"**: richiede 3033ms \approx 3s
- **"W:0"**: richiede 316ms \approx 0s

Notiamo immediatamente come la scelta della giusta tipologia di scrittura, in base all'esigenza, influisce significativamente sulle performance finali.

3.8 Distribuzione dei chunk

Ora che il nostro database ha circa 140.000 documenti (pari a 17MB) vediamo più nel dettaglio come sono ripartiti i chunk:



```
root@250ef97055ec /sharding
MongoDB Enterprise mongos> db.products.getShardDistribution()

Shard m000-repl1 at m000-repl1/localhost:27011,localhost:27012
data : 1.45MiB docs : 9966 chunks : 1
estimated data per chunk : 1.45MiB
estimated docs per chunk : 9966

Totals
data : 1.45MiB docs : 9966 chunks : 1
Shard m000-repl1 contains 100% data, 100% docs in cluster, avg obj size on shard : 1538

MongoDB Enterprise mongos>
```

Figura 3.9: Ripartizione dei chunk.

Notiamo immediatamente che tutti gli shard del sistema contengono minimo 10 chunks e che esiste un'equa ripartizione del numero di documenti. Per avere un'idea più chiara di come sono suddivisi i chunk andiamo a fare una query alla collezione "chunk", che risiede nel Config Server e stampiamo tutti i chunk della collezione `bank.products`.



```
root@250ef97055ec /sharding
MongoDB Enterprise mongos> db.chunks.find({ns: "bank.products"}).pretty()
{
  "_id" : "bank.products-sku_MinKey",
  "lastmod" : Timestamp(3, 0),
  "lastmodEpoch" : ObjectId("5ee670d91146aacc32baa59e"),
  "ns" : "bank.products",
  "min" : {
    "sku" : { "$minKey" : 1 }
  },
  "max" : {
    "sku" : 14000000
  },
  "shard" : "m000-repl3",
  "history" : [
    {
      "validAfter" : Timestamp(1592217127, 10028),
      "shard" : "m000-repl3"
    }
  ]
}
{
  "_id" : "bank.products-sku_14000000",
  "lastmod" : Timestamp(12, 6),
  "lastmodEpoch" : ObjectId("5ee670d91146aacc32baa59e"),
  "ns" : "bank.products",
  "min" : {
    "sku" : 14000000
  },
  "max" : {
    "sku" : 14262617
  },
  "shard" : "m000-repl1",
  "history" : [
    {
      "validAfter" : Timestamp(1592218037, 5403),
      "shard" : "m000-repl1"
    }
  ]
}
```

Figura 3.10: Lista dei chunk della collezione `bank.products`.

La figura 3.10 da un esempio di come, i 31 chunk distribuiti tra i vari shard, sono suddivisi. Non

esiste un ordine di grandezza, i documenti sono ripartiti equamente, in modo da garantire ad ogni chunk una numerosità simile a quella degli altri chunk nel sistema.

3.9 Transazioni

L'ultima parte del nostro setup mostra come fare eseguire una lettura e un update all'interno di una transazione. Ipotizzando il classico esempio di concorrenza su un conto corrente quindi, per garantire una corretta esecuzione tutte le operazioni devono essere effettuate in mutua esclusione, diamo uno sguardo alle seguenti linee di codice:

```
1 MongoClient mongoClient = MongoConnect.getConnection();
2
3 MongoClient
```

```

44     clientSession.close();
45 }
46 }

```

Questo è un esempio garantisce atomicità e isolamento che non soffre di letture sporche o inconsistenti; ciò è dovuto alla tipologia di lettura impostata nella linea 12 "SNAPSHOT" che, come abbiamo visto nella sezione 2.2.2, richiede una tipologia di scrittura "MAJORITY" come in linea 13. L'invocazione delle primitive di COMMIT e ABORT è rispettivamente alla linea 40 e 42.

Dal codice sorgente possiamo notare come le Driver API lavorano dietro le quinte al posto nostro, infatti non c'è nessuna indicazione nel codice in merito a errori di scrittura che potrebbero verificarsi (conflitti con altre transazioni o ACK non ricevuti). Questo perché la libreria, se non diversamente specificato, ha incorporata una retry-logic che prova a rieseguire le scritture in automatico finché l'operazione non ha successo.

La figura 3.11 mostra l'andamento delle performance durante l'esecuzione di più thread che contemporaneamente eseguono transazioni sulla stessa risorsa.

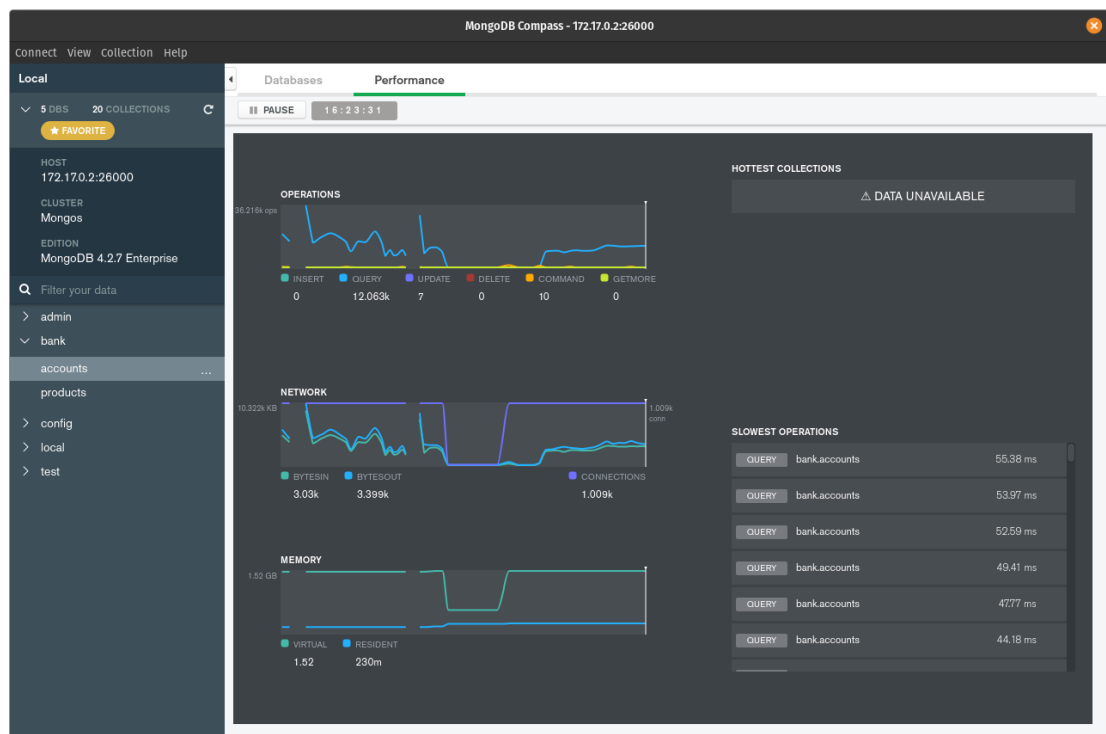


Figura 3.11: Performance di MongoDB durante l'esecuzione di 12.000 transazioni.

Notiamo come transazioni di questo tipo, rallentano il sistema in maniera significativa arrivando a richiedere, in condizioni estreme, 55ms per essere eseguite.

Se avessimo avuto la necessità di bloccare la risorsa anche durante la lettura, così da essere sicuri di ottenere la mutua esclusione in quel momento, l'unico modo possibile è quello di scrivere un valore su un attributo fittizio in modo da far ottenere alla nostra transazione il lock fittizio per la risorsa.

```

Document account = db.foo.findOneAndUpdate(eq("iban", "IT0000",
set("pseudoRandom", new ObjectId())));

```

In questo modo la transazione, per scrivere un valore randomico nel campo `pseudoRandom`, deve acquisire il lock sul documento con `iban` uguale a "IT000".

4. Conclusioni

In definitiva, abbiamo analizzato nel dettaglio come tutte le componenti di MongoDB funzionano, ma è davvero il miglior DBMS? **No**, dipende dal caso d'uso. Se la necessità è quella di utilizzare un database distribuito con alte performance in scrittura e, soprattutto, si sa a priori su cosa andare ad effettuare le query allora MongoDB è perfetto. In fin dei conti ha anche il vantaggio di essere schemaless: poter cambiare, in via di sviluppo (succede sempre!), lo schema senza andare a fare un enorme refactoring sulle tabelle. Ma allora perché non è utilizzato ovunque? Perché abbiamo visto come, ad esempio, sia davvero inefficiente nelle transazioni, se si ha la necessità di avere un alto livello di isolamento le sue performance calano a picco. Inoltre, MongoDB non è l'unico database a offrire Consistenza e supporto alle Partizioni, esistono anche soluzioni come Redis (relazionale) o HBase (ex handoop, colonnare) quindi la scelta è ampia e soprattutto può adattarsi alle esigenze di qualsiasi applicativo. MongoDB comunque è al quinto posto nella lista dei database più popolari [4], rimanendo al primo posto tra i database documentali. Probabilmente ciò è dato dal famoso MongoDB Connector for Business Intelligence (BI) che, senza dilungarci troppo, permette a linguaggi (SQL) o programmi esterni di effettuare delle query in sola lettura, quindi si può adattare comodamente un po' a tutte le situazioni esistenti.

Bibliografia

- [1] Álvaro Hernández Tortosa aht@ongres.com. *PERFORMANCE BENCHMARKPOSTGRESQL / MONGODB*. URL: https://info.enterprisedb.com/rs/069-ALB-339/images/PostgreSQL_MongoDB_Benchmark-WhitepaperFinal.pdf.
- [2] Elena Botoeva et al. “Formalizing MongoDB Queries”. In: *Proceedings of the 11th Alberto Mendelzon International Workshop on Foundations of Data Management and the Web, Montevideo, Uruguay, June 7-9, 2017*. A cura di Juan L. Reutter e Divesh Srivastava. Vol. 1912. CEUR Workshop Proceedings. CEUR-WS.org, 2017. URL: <http://ceur-ws.org/Vol-1912/paper5.pdf>.
- [3] MongoDB. *Aggregation Pipeline Optimization*. URL: <https://docs.mongodb.com/manual/core/aggregation-pipeline-optimization/>.
- [4] DB-Engines Ranking. *DB-Engines Ranking*. URL: <https://db-engines.com/en/ranking>.