
**Corso di Laurea Magistrale in Ingegneria Informatica e**

**dell'Automazione**

# *Analisi atomicità delle transazioni in un cluster MongoDB*

**A cura di:**

**Francesco Baioni - S1101137**

**David Caprari – S1097622**

A.A. 2022/2023

# Table of Contents

Introduzione ..... 3
Cluster MongoDB ..... 4
Architettura ..... 4
Dati ..... 5
Script di inizializzazione ..... 6
Applicativo Desktop ..... 7
GUI ..... 7
Backend ed interfaccia al driver ..... 7
Funzioni previste ..... 8
Inserimento ..... 9
Ricerca ..... 10
Aggregazione ..... 10
Cancellazione ..... 11
Replica Update ..... 12
Sharded Update ..... 14
Phantom Update ..... 16
Note e Conclusioni ..... 18

# Introduzione

Nel seguente lavoro, si è deciso di approfondire lo studio delle transazioni in MongoDB, analizzandone le proprietà anche in condizioni di concorrenza, replicazione e frammentazione.

A questo fine è stato impostato e sviluppato un cluster locale composto da 10 istanze di MongoDB. Su di esso è stata progettata una piccola base di dati ad hoc in grado di supportare gli esperimenti da svolgere.

Esperimenti e accesso al cluster sono stati implementati con la realizzazione di una semplice applicazione basata su librerie WinUI3. All'interno di quest'ultima, che utilizza il driver MongoDB appropriato, sono stati sviluppati diversi metodi per l'interfaccia al database, tra cui diversi algoritmi per testare l'atomicità delle transazioni in alcuni esperimenti.

Infine, in questo elaborato è racchiusa anche una minima documentazione del software progettato, con argomentazione delle procedure legate agli esperimenti.

# Cluster MongoDB

Il cluster è stato sviluppato e realizzato con MongoDB 6.0.3, seguendo la documentazione ufficiale[1].

Al fine di valutare l'atomicità in diverse condizioni si è scelto far in modo di avere a disposizione un cluster che fosse in grado di gestire sia replicazione che frammentazione orizzontale del dato. Questo ha reso possibile diversi test basati sui due concetti fondamentali della distribuzione di una base di dati.

## Architettura

Il cluster si compone di nove istanze di server MongoDB, dette anche mongod, e una istanza proxy per la gestione esterna del database distribuito, detta mongos. Vista la relativa difficoltà nell'ottenere un'architettura hardware che possa supportare il numero appena definito di istanze, si è scelto di implementare il tutto su di una singola macchina sfruttando diverse porte della stessa.

Il cluster è quindi composto da:

- Tre server di configurazione. Questi utilizzano le porte 27018-27019-27030, con il server presso la porta 27018 definito come primario di replicazione. Sebbene nella documentazione sia richiesto almeno un server di configurazione, è stato deciso, in ottica di replicazione, di averne multipla copia. Si definisce così un Replica Set di configurazione.
- Tre server dati a comporre lo Shard A presso le porte 27021-27022-27023. Istanze classiche di mongod, mantengono frammenti dei dati del cluster. Il server 27021 funge da replica primaria del Replica Set.
- Tre server dati a comporre lo Shard B presso le porte 27024-27025-27026. Anche queste istanze classiche di mongod mantengono frammenti dei dati del cluster con server 27024 primario di replicazione.
- Un'istanza mongos presso la porta 27030 di proxy per il database, che ci consente di accedere esternamente agli Shard A e Shard B come se fossero un'unica base di dati centralizzata.

Dal punto di vista implementativo, la versione di MongoDB utilizzata richiede nei file di configurazione la definizione di un ordinamento tra i server che compongono i Replica Set. In questo modo viene definito un server primario e dei secondari in cui la principale differenza è legata a meccanismi di approvazione delle scritture[2]. Questo si traduce col fatto che nel nostro caso, l'istanza di mongos legata alla porta 27030 si occuperà di diffondere le modifiche al database soltanto attraverso le porte 27021 e 27024. I server legati a queste porte effettueranno poi la replica nei server secondari. Lo stesso vale nel caso dei metadati mantenuti nei server di configurazione, dove il server principale della replica è l'istanza legata alla porta 27018.

[1] https://www.mongodb.com/docs/.
[2] Come citato nel manuale di replicazione: https://www.mongodb.com/docs/manual/replication/.

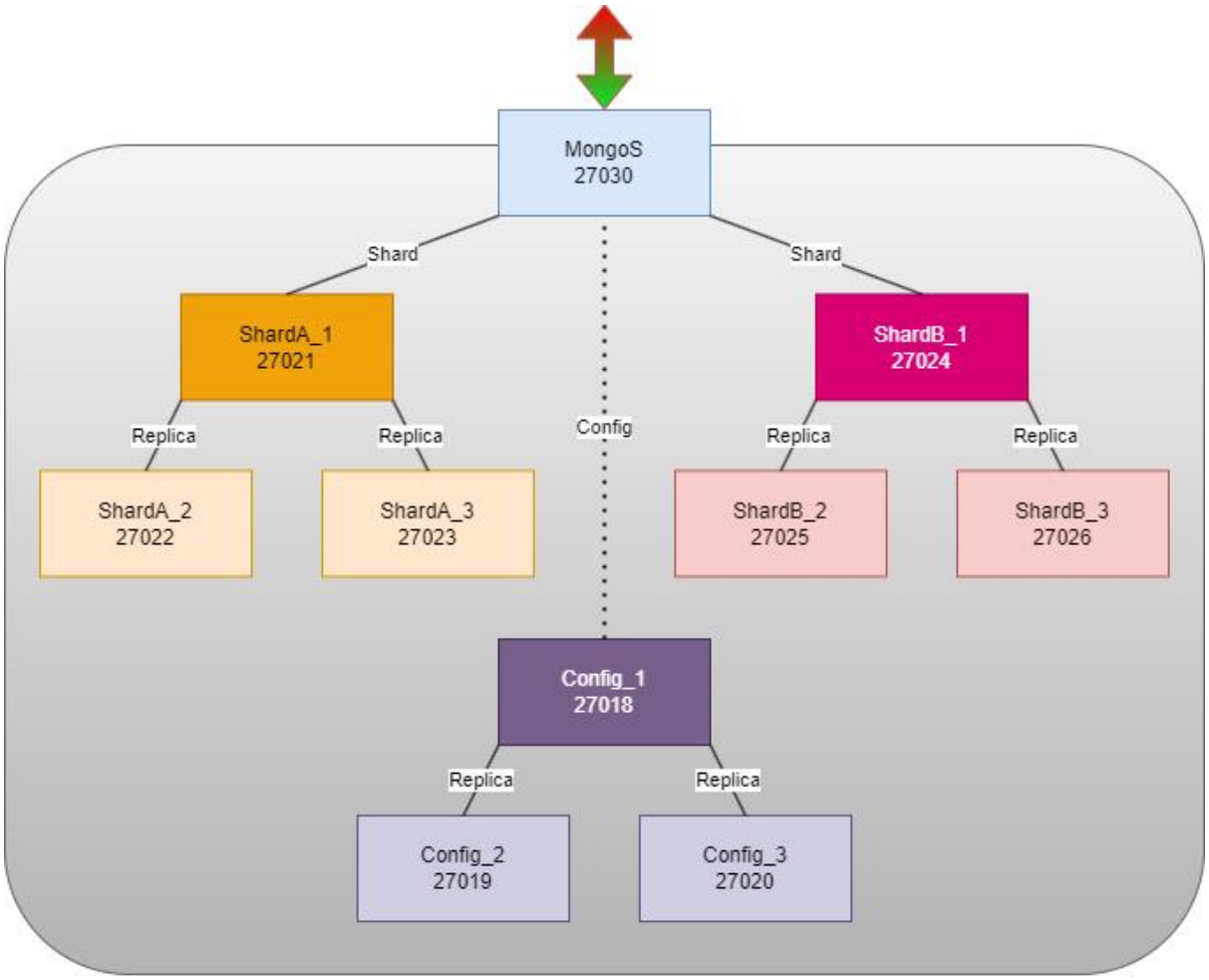


*Figura 1: L'architettura del cluster MongoDB. La linea tratteggiata sta a riportare come i server di configurazione mantengano metadati di tutto il cluster, soprattutto ma non unicamente, a partire dalle modifiche fatte attraverso l'interfaccia mongos presso la porta 27030.*

L'architettura così definita consente l'accesso diretto al dato interrogando la porta 27030 ma allo stesso tempo mantiene le singole interfacce dei server che compongono il cluster: sarà molto importante per gli esperimenti che verranno definiti avere la possibilità di interrogare i singoli shard server al fine di ottenere i frammenti ivi contenuti. Nel caso di una collezione frammentata su diversi shard, saranno infatti disponibili al livello dei singoli server solamente i frammenti contenuti in quella località. Questa proprietà ci consentirà di testare efficacemente i meccanismi di replicazione della scrittura sia internamente ai RS che agli Shard.

## Dati

Con il fine di avere a disposizione una minima quantità di dati semi-strutturati da utilizzare per effettuare gli esperimenti proposti, è nata la necessità di definire una piccola base di dati che si adeguasse correttamente a quanto ci si era proposto.

Per questo è stato creato all'interno del cluster un database chiamato *NGD_Project* nel quale è stata definita la collezione *sharded_coll,* ad indicare la possibilità di frammentazione di quest'ultima.

I dati all'interno della collezione risultano essere quindi documenti molto semplici con un minimo insieme di campi:

1. ObjectID: l'identificativo dell'oggetto che MongoDB assegna in maniera automatica ad ogni singolo documento. Per quanto possa essere possibile la modifica manuale di questo campo, si è deciso di non attuarla.
2. Name: nome definito manualmente dell'oggetto. Nel caso riportato sono stati scelti come nomi i valori *"x"*, *"y"* e *"z"*. Gli oggetti con nome X e Y sono mantenuti salvati all'interno della base di dati, mentre l'oggetto "z" è stato speso per dimostrare operazioni di inserimento e cancellazione che verranno presentate successivamente.
3. Shard_value: MongoDB richiede, affinché sia possibile la frammentazione dei dati all'interno di un cluster, che sia definito un indice rispetto al quale effettuare la frammentazione. In questo caso, l'indice definito si basa su diversi valori numerici di questo campo. In particolare, il valore 0 corrisponde ad uno sharding su Shard B, il valore 1 su Shard A. Non c'è una motivazione rilevante dietro questa scelta.
4. Value: in ottica key:value, si è scelto di avere un oggetto che a livello logico possa essere identificato dal suo campo Name e che contenga l'informazione numerica rilevante in questo. A questo sono quindi associati valori numerici che saranno alla base di diversi test effettuati in multiple operazioni.
5. Field: campo richiesto dalla necessità di avere un attributo che possa essere impostato in maniera comune agli elementi della base di dati. In particolare, si è scelto di settare questo attributo per tutti gli elementi a valore 0.

Localmente, l'appena introdotto oggetto X è memorizzato nello Shard B, l'oggetto Y nello Shard A. Ogni nodo replica principale dello shard trasmette il suo contenuto anche alle repliche secondarie. Questo a significare che, ad esempio, X è disponibile nelle istanze di mongod alle porte 27024, 27025, 27026 e, visto che si utilizza un proxy come vista esterna, anche all'istanza di mongos alla porta 27030.

Date alcune proprietà definite nel cluster, i dati così memorizzati vengono mantenuti all'interno della struttura dati anche nel momento in cui si arresta completamente l'architettura. In questo modo, effettuandone il restart, si hanno ancora a disposizione i dati come vi erano stati depositati inizialmente. Questo ha fatto sì che il processo di sharding della collezione possa essere effettuato inizialmente e non più ripreso, a meno di cancellazioni e nuove modifiche, e che quindi fosse possibile semplificare agevolmente lo script di startup.

## Script di inizializzazione

Per avviare l'intera configurazione è richiesto uno script che lanci le diverse istanze, ognuna con le sue opportune configurazioni. In particolare, per ogni istanza viene richiesto un file di configurazione che indichi porta del server, località dei dati e dei log, ruolo in sharding e replicazione, binding all'indirizzo IP e alle porte di interfaccia. Nel caso della configurazione del proxy mongos, viene richiesto lo sharding di configurazione e gli indirizzi IP dei server, oltre a quanto già definito per i mongod. In aggiunta a questi, in fase di startup, vanno inizializzate le connessioni tra istanze attraverso degli script javascript: questi consentono l'inizializzazione dei collegamenti logici che rendono funzionali Shard e RS.

Lo script di inizializzazione, quindi, avvia le istanze seguendo l'ordine delle porte assegnate ai server: avvia prima i server di configurazione, li configura lanciando lo script assegnato, avvia poi le istanze dello Shard A lanciando il suo script di configurazione, prosegue avviando lo Shard B col suo script e termina lanciando il proxy con il suo script di configurazione. Il tutto viene realizzato implementando opportuni timer, in modo che sia dato sufficiente tempo all'avvio asincrono delle istanze e dei loro processi.

Al termine, il cluster, dato il numero di istanze, processi di gestione e tool richiede una quantità di memoria RAM variabile che va dai 4GB ai 9GB. L'intero progetto, comprensivo anche dell'UI è stato testato su due macchine con 16GB e 32GB di RAM. Non si esclude che destinando al progetto una quantità minore di memoria non si possa incorrere in fault e stati di errore non previsti.

Un bug della versione di mongod utilizzata si traduce in una casuale perdita di funzionamento dell'interfaccia alle singole istanze del driver C#, utilizzato nel progetto come collegamento all'interfaccia grafica. Questo si traduce nella casuale incapacità di testare gli esperimenti che richiedono una transazione all'interfaccia esterna (mongos 27030) e la verifica dell'avvenimento della transazione nel singolo server interno (ad es. 27025). Una soluzione, banale ma efficace, risiede nel riavviare l'intero cluster e riprovare la singola operazione.

## Applicativo Desktop

La seguente sezione descriverà l'applicativo desktop realizzato per fungere da interfaccia grafica e contenitore delle procedure di accesso in lettura e scrittura al database.

Non essendo parte fondamentale di questo lavoro, il lettore troverà che, in questo stato, l'applicativo e sua interfaccia grafica risultano molto semplici e carenti di alcuni sistemi di agevolazione d'utilizzo come controllo degli errori e prevenzione di crash. Si è preferito concentrarsi sugli aspetti teorici portati dal corso di studio per il quale questo progetto è stato richiesto.

## GUI

L'interfaccia grafica per l'utente è stata creata attraverso il framework windows WinUI3 ed ha permesso di sviluppare effettivamente l'applicativo. Per evitare problematiche legate alla compatibilità della distribuzione del software e viste le esigenze di progetto, l'applicazione viene fornita come progetto per Microsoft Visual Studio 2016 o Visual Studio 2022. Per avviare l'interfaccia è quindi sufficiente caricare il progetto contenuto nel repository fornito con il software appena citato e attraverso lo strumento di test e debug, avviare il progetto in opzione Debug x86.

L'interfaccia grafica viene quindi completata con un riquadro di selezione delle operazioni sulla sinistra, un riquadro per la visualizzazione degli esiti delle operazioni sulla destra ed infine un riquadro di stato del driver in basso.

Quest'ultimo in particolare non è stato raffinato completamente e a livello implementativo non fornisce indicazioni totalmente precise e pratiche; funge piuttosto da flag per lo stato di connessione del driver e per indicare agli sviluppatori il termine o meno delle operazioni.

## Backend ed interfaccia al driver

Il backend dell'applicazione, legato astrattamente al paradigma di programmazione Model View Controller, è realizzato in codice C#. L'applicazione sfrutta quindi il driver MongoDB per C# messo a disposizione dalla versione di Microsoft Visual Studio correntemente utilizzata.

Il driver impiegato mette a disposizione diverse funzioni, la totalità di quelle che ci si aspetterebbe per uno strumento del genere, sulla falsa riga di quelle disponibili per la CLI di MongoShell.
Tra quelle presenti in questo progetto, le più utilizzate fanno capo a istruzioni di connessione, ottenimento dei documenti, ricerca e aggiornamento, tutte con le minime variazioni richieste dal caso[3].

[3] In base alla richiesta, ad esempio, un aggiornamento asincrono o meno, multi-documento o meno.

Inoltre, in tutta questa sezione si è scelto di utilizzare il termine “operazioni” piuttosto che “transazioni” in quanto, sebbene il secondo indichi in maniera più precisa come si comporti generalmente un’interfaccia al database e il fatto che ciò che viene eseguito a livello della base di dati coincida semanticamente con il termine, alcune delle procedure che sono state implementate per questo lavoro sfruttano una serie di transazioni a diversi livelli, da qui la scelta linguistica.

## Funzioni previste

Avendo quindi a disposizione una minima interfaccia grafica, si è deciso di verificare attraverso l’applicativo appena creato alcune delle funzioni base del DBMS come la ricerca, l’aggregazione, l’inserimento e la cancellazione di alcuni documenti all’interno di una collezione gestita da MongoDB. Il tutto, in fase primordiale di progetto, come approccio per verificare il funzionamento dei moduli grafici, dell’interfaccia e del driver, nonché come agevolazione al fine di fare pratica con il linguaggio di interrogazione.

In aggiunta, come obiettivo dell’analisi, si sono sviluppate altre tre funzioni più avanzate che hanno permesso di studiare il comportamento delle transazioni in MongoDB, ovvero: update all’interno dello stesso RS, update tra due Shard separati e transazione concorrente multi-shard.

Verranno presentate le operazioni in paragrafi dedicati.

## Inserimento

L'operazione di inserimento richiama la funzione *InsertOne()* del driver MongoDB e fa esattamente quanto indicato nella sua denominazione: effettua l'inserimento di un documento BSON (Binary JSON) all'interno della collezione indicata.

I valori aggiuntivi indicati nella casella di testo fornita vengono passati attraverso un parser che si occupa di compilare correttamente il campo BSON richiesto.

Nel caso in esame, a livello di frammentazione del documento, quest'ultimo viene depositato nello shard che ha, al momento dell'inserimento, memoria più libera. A causa di alcuni file di configurazione del cluster e del database utilizzato che sono stati salvati randomicamente nello Shard A al momento dell'inizializzazione, a meno dell'inserimento di un discreto numero di documenti, il lettore noterà che ciò che inserirà, tenderà ad essere memorizzato nello Shard B. In ogni caso, è presente un meccanismo di load rebalancing, correttamente configurato, che MongoDB mette a disposizione. Affinché sia possibile però attivarlo, deve essere presente uno sbilanciamento tra frammenti superiore ai 100MB, non semplice da raggiungere se si lavora con la collezione testuale introdotta precedentemente.

Se ne conclude, verosimilmente, che il documento inserito non verrà riallocato a meno che la riallocazione non venga effettuata manualmente o che si sia consci di star inserendo un elevato numero di documenti.

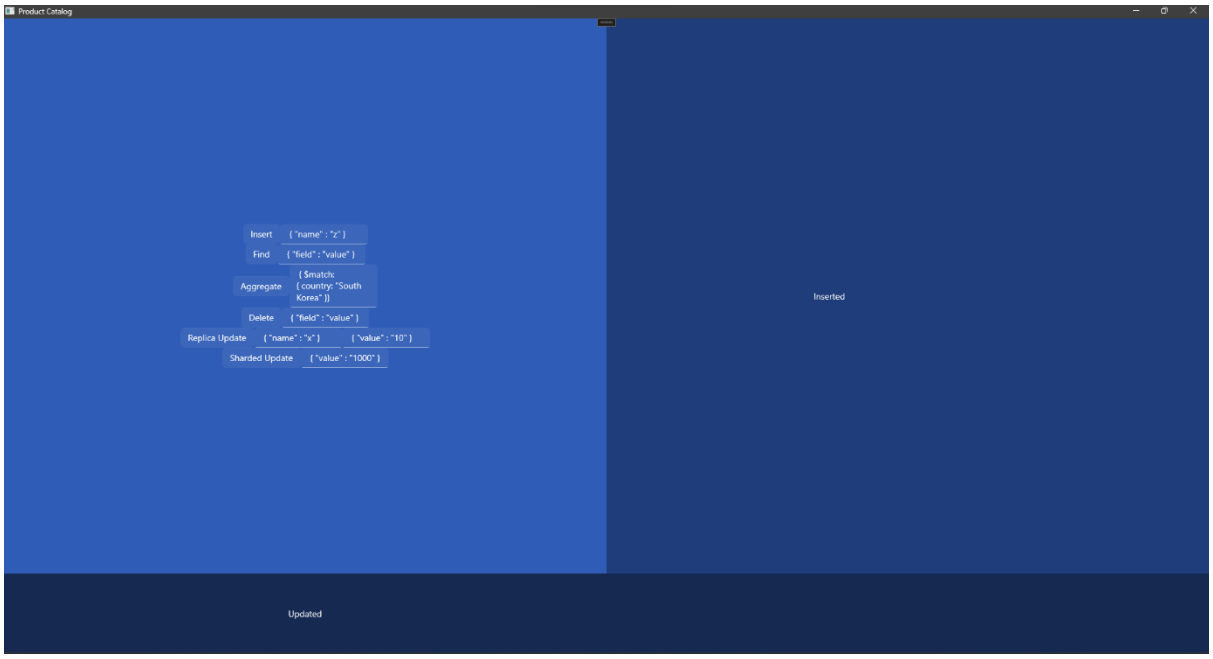


*Figura 2: Dimostrazione dell'inserimento di un documento con "name":"z". Per una migliore fruizione dell'immagine si consiglia la presentazione in PDF allegata a questa relazione o direttamente l'immagine sorgente presente nella documentazione.*

## Ricerca

L'operazione di ricerca, dopo aver definito un filtro su cui effettuarla basato sul campo ad essa collegato, effettua la ricerca all'interno del database ed estrae il primo risultato ottenuto.

Riporta quindi l'oggetto nel campo risultati sulla destra.

Si è scelto di riportare il solo primo oggetto per semplificare la procedura, che altrimenti avrebbe avuto bisogno di una corretta gestione di un indice di documenti su cui poi effettuare cicli per la visualizzazione. Questa più estesa implementazione è stata resa nella funzione di Phantom Update che verrà introdotta successivamente, da qui la scelta di non complicare questa.

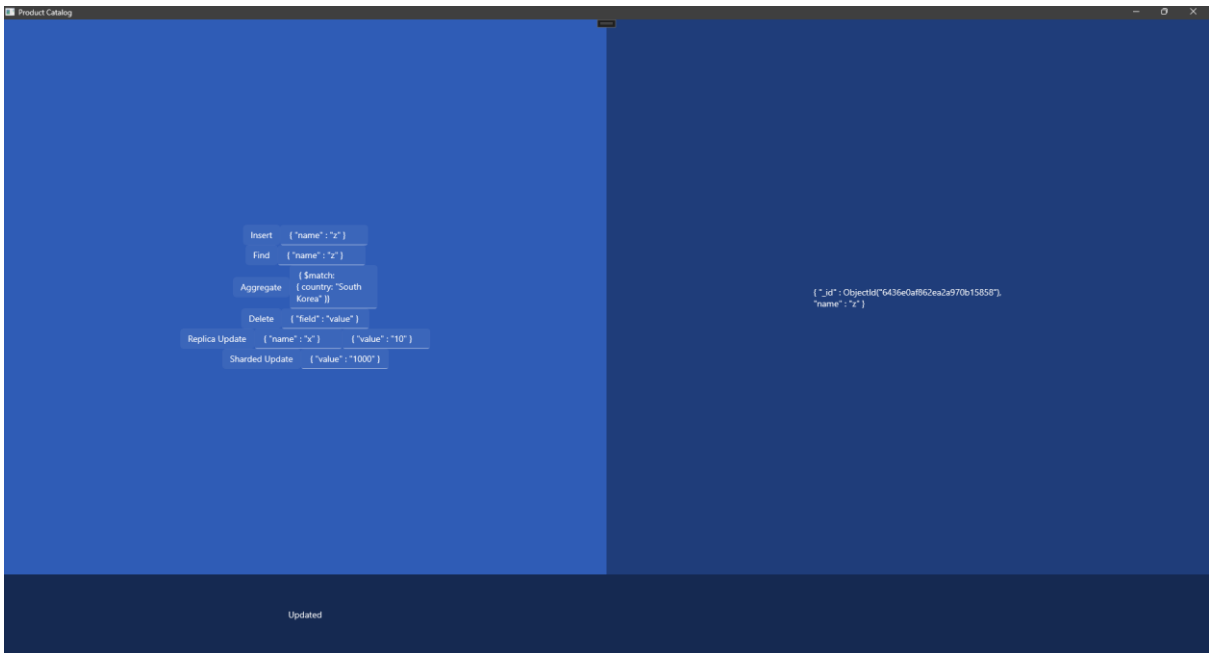


*Figura 3: Dimostrazione della ricerca di un documento con "name":"z". Per una migliore fruizione dell'immagine si consiglia la presentazione in PDF allegata a questa relazione o direttamente l'immagine sorgente presente nella documentazione.*

## Aggregazione

L'operazione di aggregazione, dopo la definizione di una Pipeline corretta all'interno del campo testo collegato, effettua aggregazione degli elementi per i campi definiti sfruttando la funzione *Aggregate()* del driver.

Nel caso in esempio, in Fig.4, l'aggregazione viene effettuata su di un solo elemento con campo nome coincidente con "x", quindi in risposta si ha il documento X.

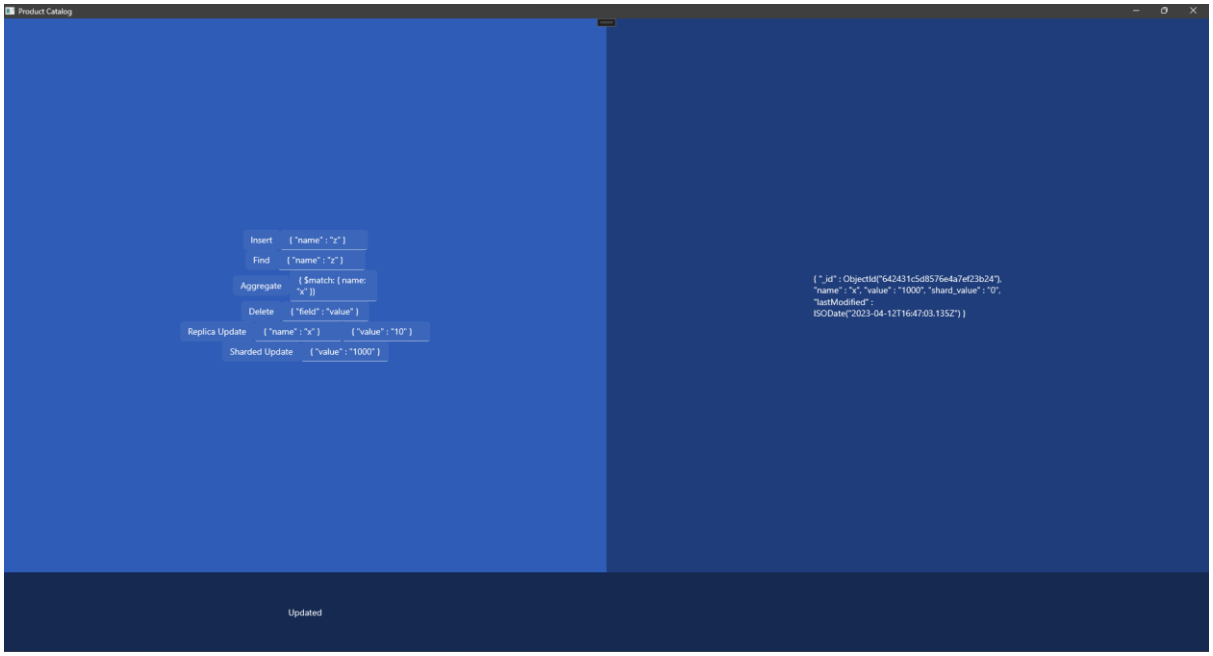


*Figura 4: Dimostrazione dell'aggregazione di un documento in base a "name":"x". Per una migliore fruizione dell'immagine si consiglia la presentazione in PDF allegata a questa relazione o direttamente l'immagine sorgente presente nella documentazione.*

## Cancellazione

L'operazione di cancellazione elimina il record di un documento dal database. Si basa quindi su di una *Find()* come indicata nella sezione di ricerca, dopodiché, ottenuto il record, lo si cancella con una *EraseMany().*

Viene preferita la EraseMany() ad una singola Erase() poiché la prima supporta i RS e quindi modifica correttamente tutte le copie del documento nella base di dati distribuita.

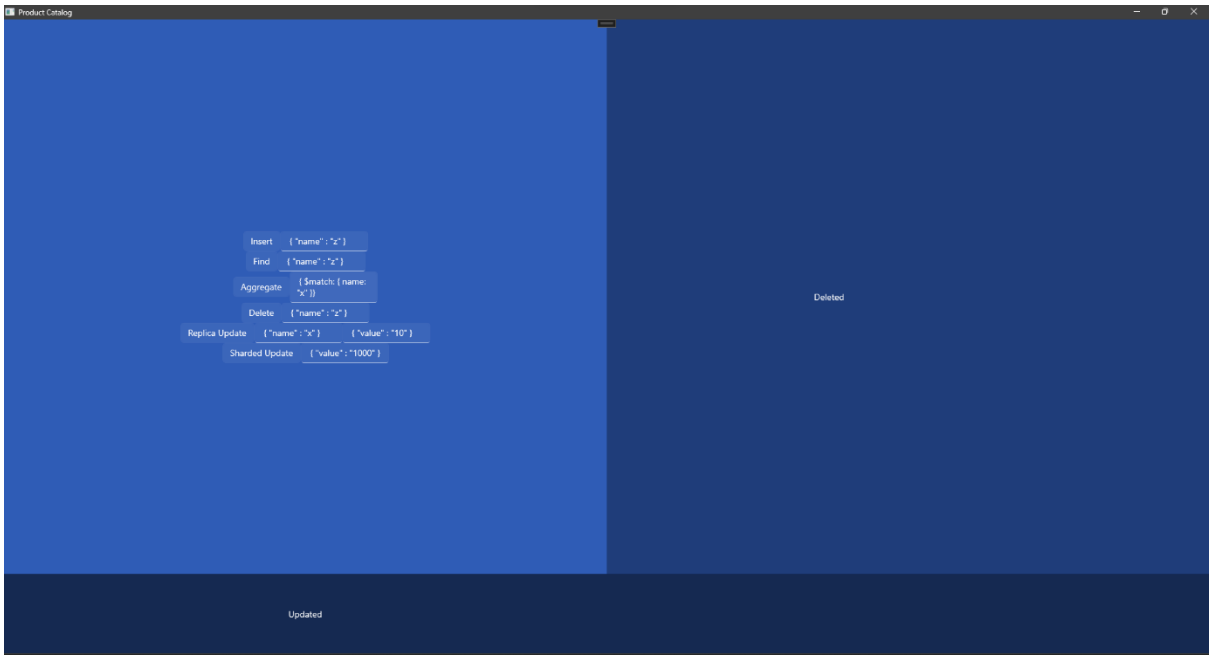


*Figura 5: Dimostrazione della avvenuta cancellazione di un documento in base a "name":"z". Per una migliore fruizione dell'immagine si consiglia la presentazione in PDF allegata a questa relazione o direttamente l'immagine sorgente presente nella documentazione.*

## Replica Update

Prima delle tre operazioni più elaborate. Si basa sul voler ricreare alcune delle condizioni di concorrenza e atomicità definite nella guida alle transazioni presente nella documentazione ufficiale di MongoDB[4]. In particolare, si vuole verificare che le proprietà delle Multi-Document transactions si comportino come atteso.

In presenza di Replica Set, le versioni di MongoDB superiori alla 4.2, come quella utilizzata in questo progetto, mettono a disposizione il supporto a transazioni che modifichino concorrentemente e atomicamente diverse istanze di dati. In particolare, per quanto riguarda questo esperimento, ci si concentra sull'atomicità che viene richiesta nella modifica al singolo documento replicato.

Nel dettaglio, l'operazione definita, modifica con una *UpdateMany()* il documento X attraverso l'interfaccia di connessione al proxy presso la porta 27030; documento che si ricorda essere memorizzato all'interno dello Shard B. A questo punto, il proxy propagherà la necessità di modifica al nodo principale dello Shard B, 27024, il quale a sua volta propagherà la modifica ai secondari 27025 e 27026. Quello che il nostro esperimento verifica è che sia impossibile accedere al dato presente presso il nodo 27025 fino a che l'operazione di UpdateMany non viene completata e che quindi non venga mai registrata incoerenza tra i dati delle varie repliche.

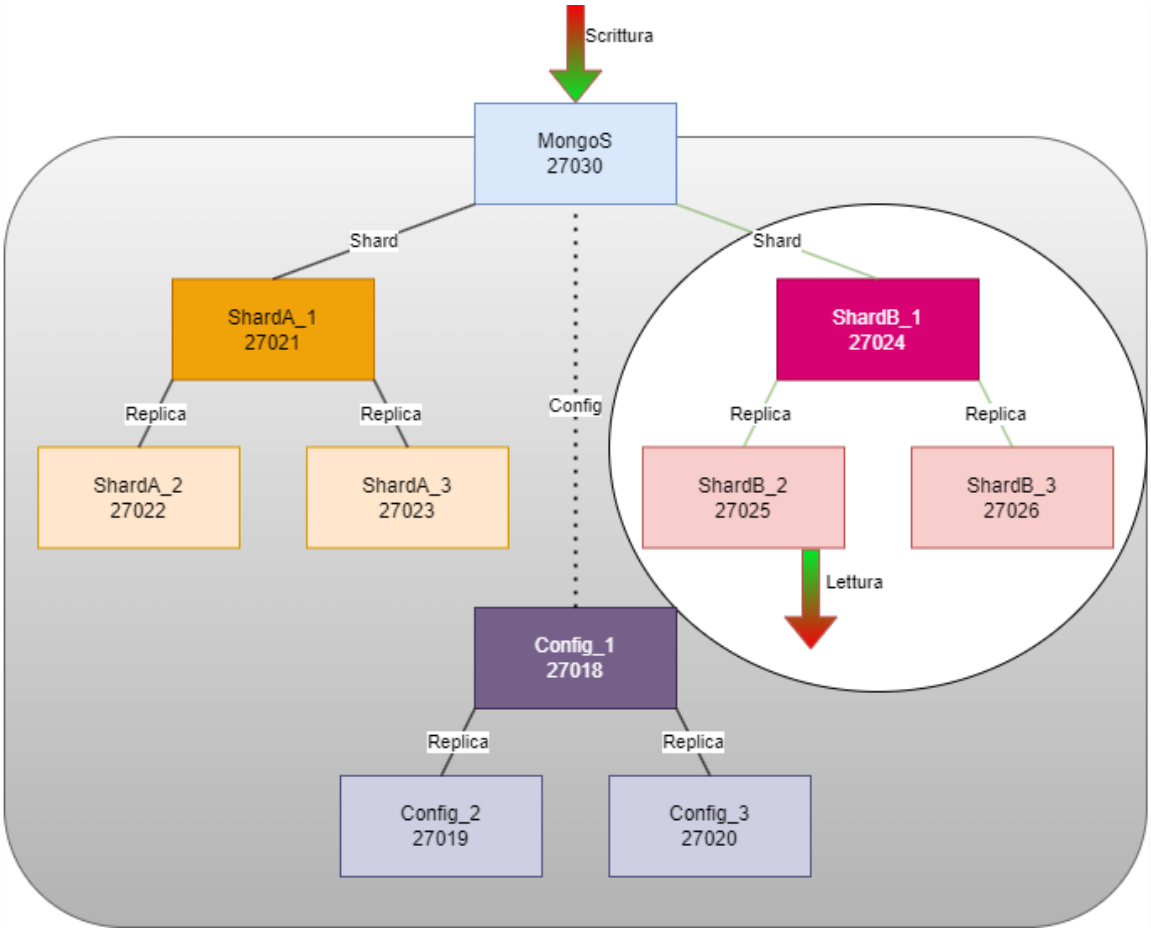


*Figura 6: Riporta la struttura dell'operazione. Viene richiesta una scrittura alla porta 27030, viene concorrentemente richiesta una lettura dei dati che verranno modificati alla porta 27025 che però rimane in attesa del termine della prima.*

[4] https://www.mongodb.com/docs/v6.0/core/transactions/

La verifica viene effettuata lanciando, appena richiamata l'UpdateMany sul 27030, una transazione di lettura del documento X presso la porta 27025. Attraverso l'utilizzo di timestamp è possibile notare come l'operazione di ricerca, che in qualsiasi altra condizione richiede un tempo che va dai 3 ai 9ms, rimanga in attesa del termine dell'UpdateMany per un tempo che invece si aggira nell'intorno dei 55ms.

Tempo di attesa che si riduce intorno ai 15ms se all'UpdateMany viene richiesto di modificare il campo Value del documento con un valore che vi è già presente (ad es. voglio modificare il numero 100 con il numero 100), ovviamente mantenendo l'atomicità della transazione Multi-Documento.

*Tabella 1: Dati relativi alle tempistiche di Ricerca semplice e Replica Update. Ogni operazione riporta i dati di 5 misurazioni sequenziali con campo Value aleatorio. I dati sono ottenuti con processore i7-9750H con 16GB di RAM e SSD nVME.*

| In **[ms]** | **Ricerca** | **Replica Update** | **Replica Update (con stesso valore)** |
|---|---|---|---|
| **I** | 7 | 48 | 18 |
| **II** | 4 | 56 | 15 |
| **III** | 3 | 47 | 12 |
| **IV** | 9 | 66 | 19 |
| **V** | 6 | 53 | 11 |

I dati, coerentemente alla documentazione di MongoDB e del suo driver, riportano come sia presente l'utilizzo di una forma di locking delle risorse che vengono dichiarate esclusive fino a che le operazioni di scrittura non sono completate. Interessante notare come indipendentemente dalla necessità di modificare il dato o meno, come mostrato nel Replica Update con stesso valore, venga imposto il lock alla risorsa. Lock che però non avrà bisogno di attendere il processo di scrittura e che quindi si libererà più velocemente.

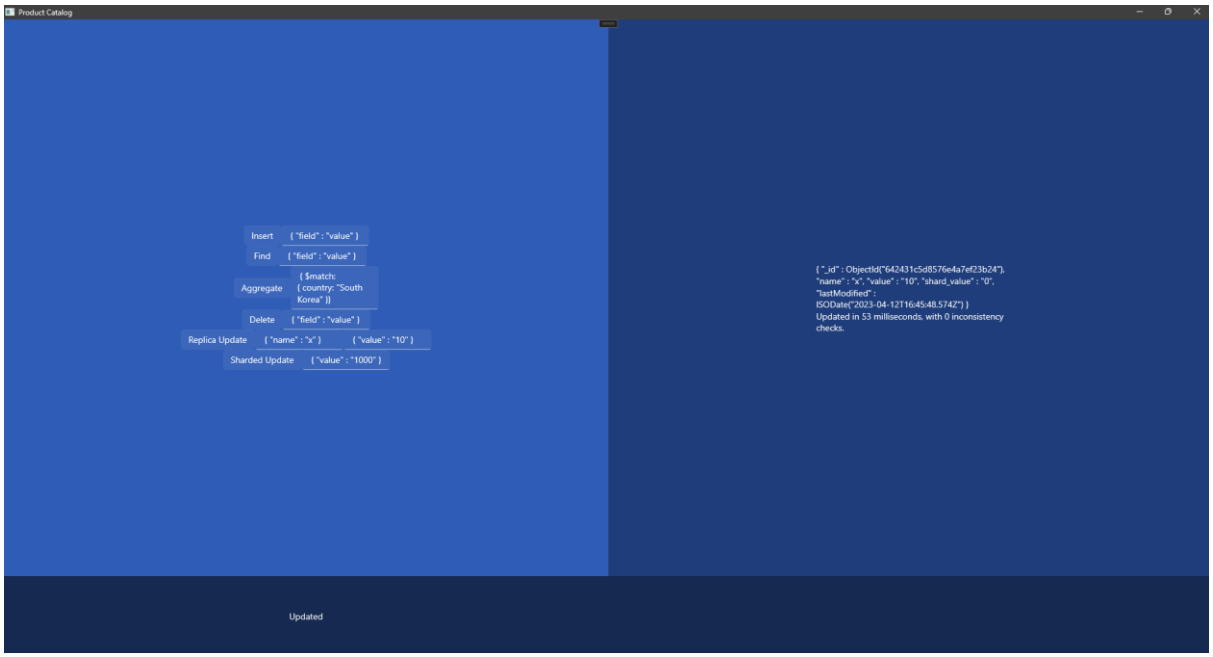


*Figura 7: Dimostrazione della Replica Update, operazione definita in questo paragrafo. Per una migliore fruizione dell'immagine si consiglia la presentazione in PDF allegata a questa relazione o direttamente l'immagine sorgente presente nella documentazione.*

## Sharded Update

Con questa operazione si vuole definire un esperimento, meno significativo del precedente, che si rifà a parte della documentazione sulle transazioni, citata nel paragrafo precedente. In particolare, si cita il caso in cui una transazione multi-shard possa intercettare stati intermedi prima che un'altra abbia effettuato correttamente una serie di scritture. In particolare, si fa riferimento ad una lettura con parametro di livello di isolamento settato a "local" che possa leggere degli stati intermedi di concorrenza prima del commit della transazione principale. Lettura che in questo caso coincide con la ricerca effettuata contattando direttamente la porta legata a un server replica.

Attraverso il driver è stata quindi definita una meta-transazione composta da due transazioni atomiche e distribuite di scrittura sul file X e sul file Y impostando il concetto logico di vincolo tra i due documenti assumendo che il loro campo Value debba rimanere coincidente sia prima che dopo la meta-transazione.

Inserendo un periodo di attesa di un secondo tra le due singole transazioni si nota come, ovviamente, per un tempo vicino a un secondo non venga rispettato il vincolo di integrità se si accede con due transazioni di ricerca con isolamento "local" alle porte 27021 e 27024 ai documenti Y e X. Al termine dell'attesa il vincolo verrà poi ripristinato, coincidendo il completamento e il commit della meta-transazione.

Si verifica quindi, esattamente come la guida ci suggerisce, una possibile inconsistenza nei dati se il programmatore del sistema non si occupa di gestire le autorizzazioni degli utilizzatori legate alle letture sui singoli server o all'esclusività della transazione.

Questo esperimento rimane quindi come monito al programmatore che prevede una sequenza di operazioni semplici senza tener conto del fatto che il sistema potrebbe trovarsi in una condizione non consistente durante lo svolgimento di queste; stato di inconsistenza che potrebbe estendersi nel caso la distribuzione del cluster sia maggiore e magari legata a vera e propria separazione tra i siti geografici dei server.

A questo riferimento, nella guida alle transazioni, vengono indicate diverse misure che il programmatore può assumere per mantenere l'atomicità, tra le quali modificare i parametri di sessione di accesso ai dati, per bloccare magari l'intera base di dati al momento della necessità di esecuzione della serie di transazioni.

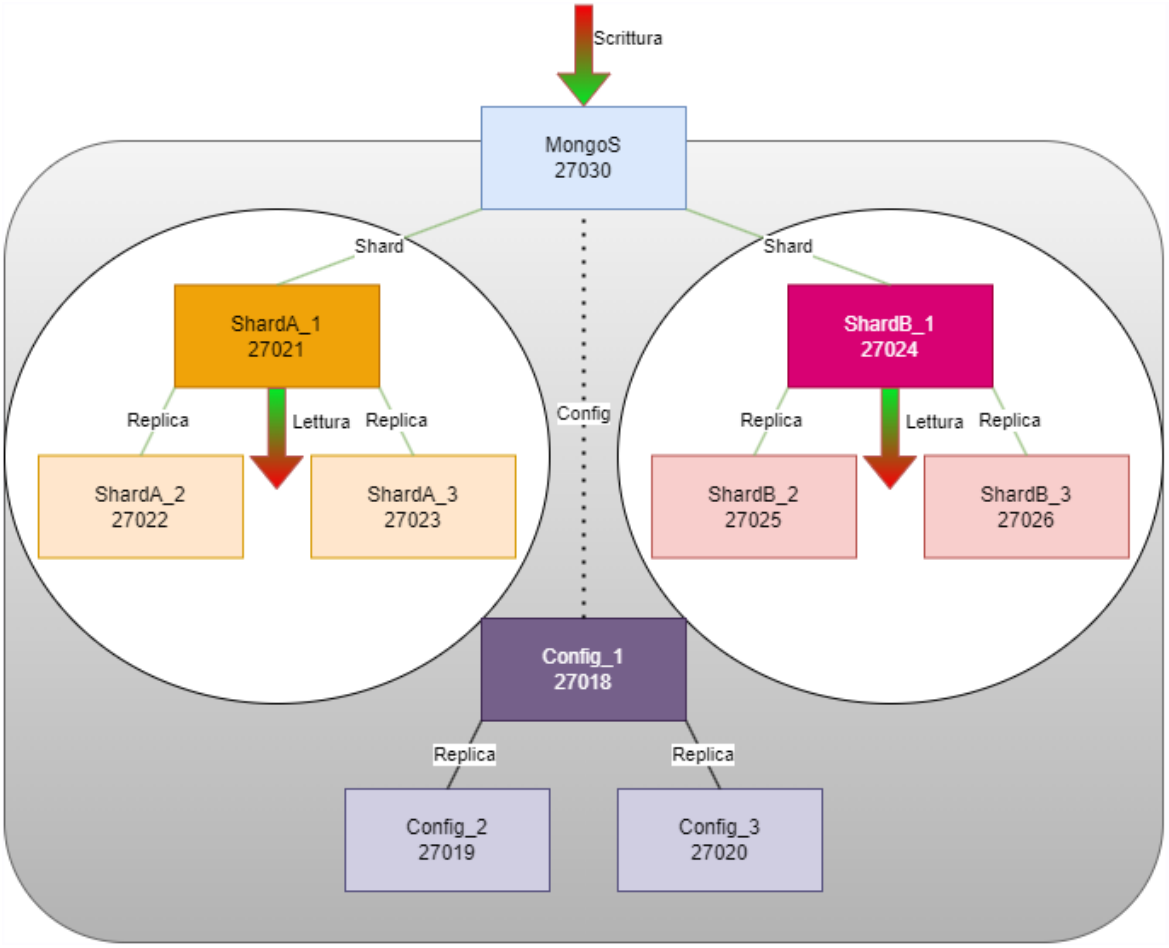


*Figura 8: Schema dell'esperimento. Si richiede una scrittura doppia con intervallo di un secondo alla porta 27030 e si nota lo stesso ritardo di aggiornamento accedendo localmente ai dati modificati passando per le porte 27021 e 27024.*

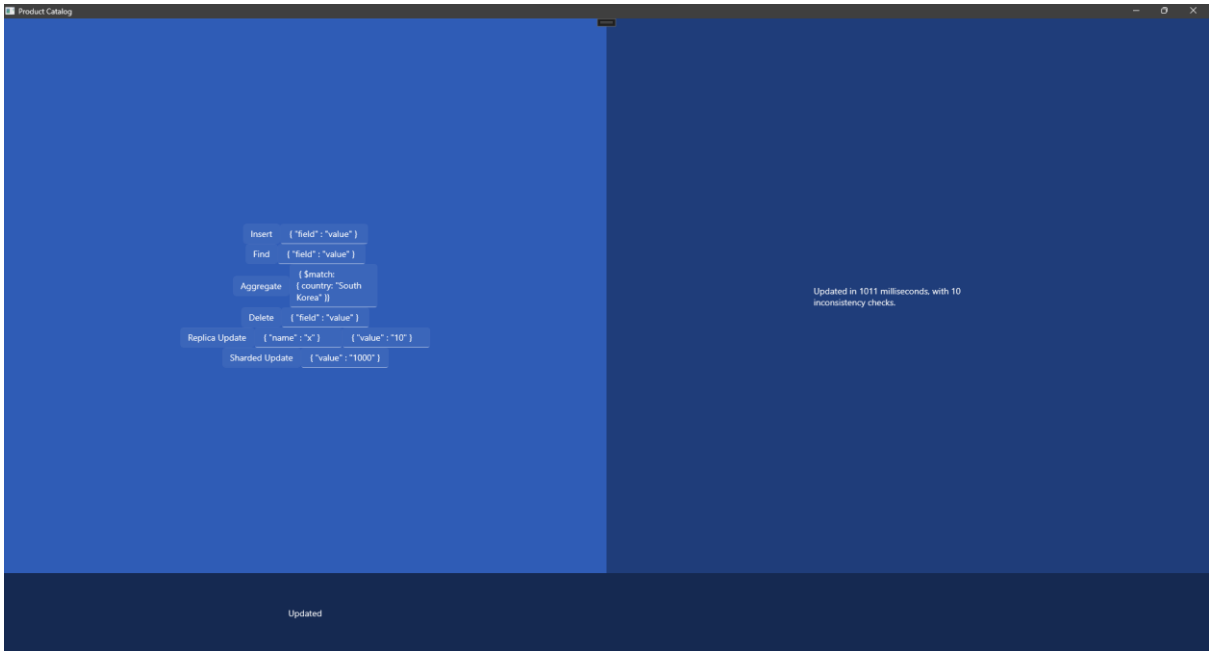


*Figura 9: Dimostrazione dello Sharded Update dove per circa un secondo non è stato rispettato un vincolo di integrità. Per una migliore fruizione dell'immagine si consiglia la presentazione in PDF allegata a questa relazione o direttamente l'immagine sorgente presente nella documentazione.*

### Phantom Update

A completamento del test meno significativo svolto prima, quest'ultimo esperimento utilizza transazioni su proxy esterno per verificare che siano presenti meccanismi di prevenzione dell'anomalia dell'aggiornamento fantasma.

Viene effettuato sul singolo proxy esterno, senza verifiche interne, in quanto negli esperimenti precedenti si è già provata la presenza di meccanismo di locking dell'intero shard (e quindi RS) in occasione di lancio di transazioni di UpdateMany().

In particolare, viene lanciata un'operazione di ricerca di tutte le copie dei documenti che hanno campo "Field" coincidente a 0, quindi vengono richiesti tutti i documenti, repliche comprese. A questo punto, viene invocata una transazione di aggiornamento su tutte quante le istanze di documento in modo che queste settino il loro valore ad un dato indicato nel campo di testo. La transazione di aggiornamento è quindi concorrente alla prima in lettura. Per l'ottenimento dei risultati abbiamo impostato dei timestamp di inizio e termine delle transazioni.

Quello che ci si aspetta è che la transazione di ricerca individui sempre lo stesso valore nel campo "Value" di tutti i documenti. Affinché si finisca però in condizione di possibile aggiornamento fantasma, e quindi almeno un campo "Value" diverso dagli altri, è necessaria la concorrenza delle due transazioni. Per questo, la procedura verifica che inizio o termine della seconda siano interni allo span temporale della prima transazione di lettura.

Verificata la concorrenza, la prova di meccanismi contro l'aggiornamento fantasma, e quindi presenza di locking condiviso delle risorse anche su transazioni di lettura, sta nel fatto che il sistema non incappi mai in una condizione in cui vi sia incongruenza nei campi valori sebbene sia provata la concorrenza.

La verifica di questa proprietà può essere svolta anche interrogando allo stesso modo uno qualsiasi dei server interno ad un replica set con isolamento "local", ottenendo lo stesso risultato. Interessante è il fatto che applicare questa verifica attraverso il proxy verifica l'atomicità tra shard in quanto l'ordinamento di ricerca, e parallelamente di aggiornamento, è ad ordine casuale in base alla rapidità di risposta dei componenti del cluster.

Sebbene non sia stato possibile rinvenire riferimenti diretti all'anomalia dell'aggiornamento fantasma all'interno del manuale sulle transazioni di MongoDB, si dà per scontato che ne siano richiesti meccanismi di prevenzione per un sistema di database distribuito come quello che viene proposto. Proprio in questa ottica, l'esperimento fatto verifica le proprietà di isolamento richieste.

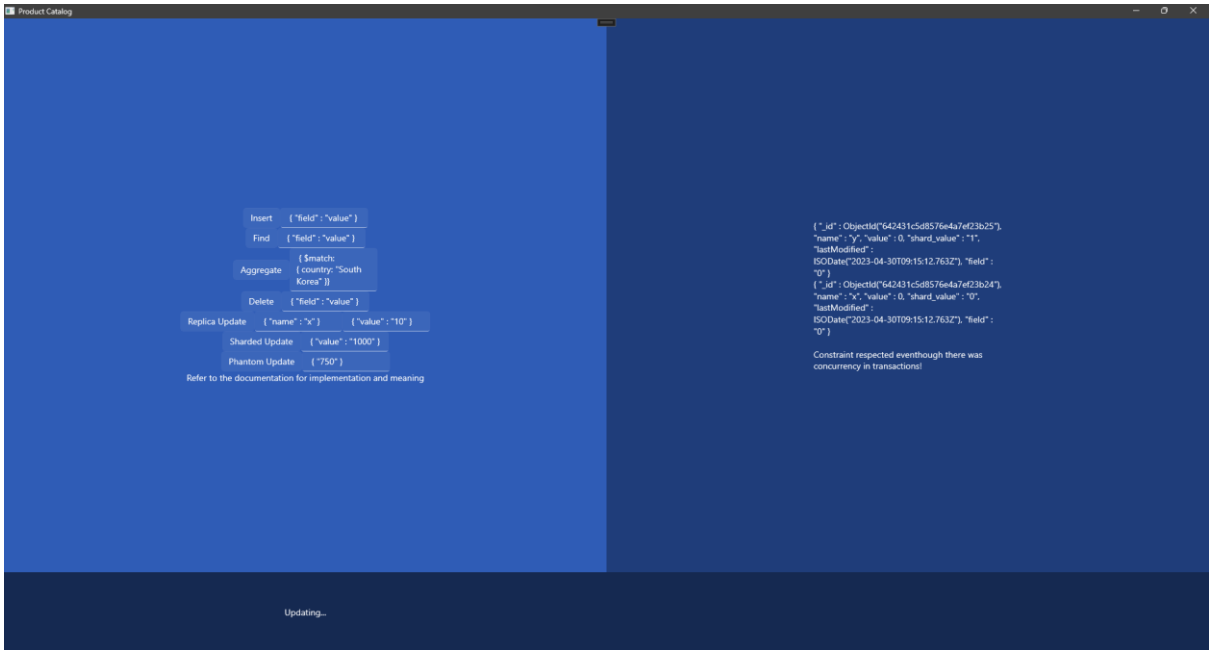


*Figura 10: Dimostrazione della verifica di contromisure per l'aggiornamento fantasma. Per una migliore fruizione dell'immagine si consiglia la presentazione in PDF allegata a questa relazione o direttamente l'immagine sorgente presente nella documentazione.*

# Note e Conclusioni

Alla luce degli esperimenti fatti, concludiamo che MongoDB, nonostante non sia un sistema volto a rispettare le proprietà ACIDe dei sistemi relazionali più diffusi, soprattutto a partire dalla versione 4.2 con il supporto alle transazioni Multi-Document, prevede accorgimenti legati ad atomicità e isolamento concorrente.

In particolare, con gli esperimenti fatti è stato possibile provare l'atomicità delle transazioni a diversi livelli:

- Provando l'atomicità sui RS si estende l'atomicità della transazione su singolo server.
- Provando l'atomicità sui RS si ottiene atomicità multilivello tra interfaccia al database distribuito e interfaccia all'istanza singola.
- Provando l'impossibilità di ottenere aggiornamento fantasma a livello di interfaccia al database distribuito, si ha prova di un livello di atomicità multi-shard legato agli stessi meccanismi di locking previsti nel caso dei RS.

Riguardo all'atomicità delle transazioni multi-shard ci si rifà alla documentazione ufficiale in cui si citano casi particolari nel caso in cui vengano fatti dei revert, casi particolari che non vengono citati in questo lavoro in quanto risultano molto più complessi e richiederebbero architetture realmente distribuite su hardware diversi per avere una migliore percezione degli effetti temporali delle mancate trasmissioni di dati.

Ne concludiamo quindi che con gli esperimenti fatti si abbia prova certa di diversi meccanismi di locking sia condiviso che esclusivo nel caso delle transazioni multi-documento.

Speculando, ci si immagina che un eventuale gestore dell'atomicità condiviso a tutto il cluster, magari attraverso i metadati contenuti nei server di configurazione, sia in grado di provvedere a bloccare con lock opportuni tutte le risorse legate alle operazioni che sono state prese in esame, in modo da mantenere sempre l'atomicità delle transazioni distribuite.